UNIVERSITY COLLEGE LONDON

A Practical Hardware Implementation of Systemic Computation

by

Christos Sakellariou

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

in the

Faculty of Engineering

Department of Computer Science

December 2013

Declaration of Authorship

Abstract

Faculty of Engineering, Department of Computer Science

UNIVERSITY COLLEGE LONDON

Doctor of Engineering

by Christos Sakellariou

It is widely accepted that natural computation, such as brain computation, is far superior to typical computational approaches addressing tasks such as learning and parallel processing. As conventional silicon-based technologies are about to reach their physical limits, researchers have drawn inspiration from nature to found new computational paradigms. Such a newly-conceived paradigm is Systemic Computation (SC). SC is a bio-inspired model of computation. It incorporates natural characteristics and defines a massively parallel non-von Neumann computer architecture that can model natural systems efficiently.

This thesis investigates the viability and utility of a Systemic Computation hardware implementation, since prior software-based approaches have proved inadequate in terms of performance and flexibility. This is achieved by addressing three main research challenges regarding the level of support for the natural properties of SC, the design of its implied architecture and methods to make the implementation practical and efficient.

Various hardware-based approaches to Natural Computation are reviewed and their compatibility and suitability, with respect to the SC paradigm, is investigated. FPGAs are identified as the most appropriate implementation platform through critical evaluation and the first prototype Hardware Architecture of Systemic computation (HAoS) is presented.

HAoS is a novel custom digital design, which takes advantage of the inbuilt parallelism of an FPGA and the highly efficient matching capability of a Ternary Content Addressable Memory. It provides basic processing capabilities in order to minimize time-demanding data transfers, while the optional use of a CPU provides high-level processing support. It is optimized and extended to a practical hardware platform accompanied by a software framework to provide an efficient SC programming solution. The suggested platform is evaluated using three bio-inspired models and analysis shows that it satisfies the research challenges and provides an effective solution in terms of efficiency versus flexibility trade-off.

Acknowledgements

I sincerely believe I could not have been luckier when I first emailed Peter Bentley about five years ago on the prospect of working under his supervision towards a doctoral degree. I am extremely thankful to him for his tremendous level of support during these five years. I hugely appreciate all his help along this journey and clearly acknowledge that this thesis would not have been possible without his guidance and encouragement. I enjoyed the fact the he is a person that I could talk to freely and count on being listened every single time. I am grateful to consider him as a friend from whom I learned a lot.

I also wish to thank the other researchers of my research group. Special thanks go to Erwan Le Martelot for sharing with me early drafts of his thesis when I started my studies. Further thanks go to Hooman Shayani, Arturo Araujo and Marjan Rouhipour for sharing parts of their work on Systemic Computation that greatly helped me move forward. I would also like to thank other colleagues at UCL: my second supervisor Prof. Steve Hailes and Prof. Izzat Darwazeh for their guidance during the reviews of my progress reports. I also would like to express my thanks to my industrial sponsor Toumaz Technology and especially to my industrial supervisor Alison Burdett for her invaluable support on the initial application towards my doctorate degree and the financial support along its duration. I further wish to acknowledge the EPSRC, the EngD VEIV doctoral centre and the department of Computer Science in UCL for their financial support and for hosting me along my studies.

I would like to thank all my friends that supported me during happy and rough times during these last five years (thank you so much guys for endless coffees and nights in and out!!): my good friends from Greece Psilos, Spirakos, Thomas and Giannakis, my good Toumaz friend Andrikos, my Physics friends Thiseas and Thanos, my soton friend Giannis, my UCL friends Melinos, Dimitris, Theo, Tim and Alexis, my weekend ciccilones friends Pippo, Adrian, Vero and Andreea, my Reading friends Stamo and Dia, my neighbours Bogo and Stella, my childhood friends Sakis, Nantia, Zetta, Stelios and Panagiotakis, my Imperial friend Dimos, all my dear cousins and so many more that need to be here and aren't due to my long-lasting sleep deprivation.. :)

Of course I want to thank my family for their continuous support before, along and beyond my studies. Special thanks to my parents, Aspasia and Thomas, and my little sister Kleiw for their unconditional love. Further thanks to all my extended family (grandparents, uncles, aunts and cousins) for helping me become who I am. Last but not least, I'd like to express my gratitude to my loving partner Marilia for always being there for me along these years, following and supporting me on every step of the way.

Table of Contents

Abstract 3

Acknowledgements 4

List of Figures 10

List of Tables 14

List of Listings 16

List of Algorithms 16

Chapter 1 Introduction 17

1.1 Introduction to Natural Computation 17

1.2 Introduction to Systemic Computation 20

1.3 Systemic Computation in Practice 21

1.4 Hypothesis 23

1.5 Objectives 25

1.6 Publications 26

1.7 Thesis Organization 27

Chapter 2 Background 29

2.1 Approaches to Natural Computation 30

2.1.1 Software Approaches 31

2.1.2 Alternative Paradigms 32

2.2 Hardware-based Approaches to Natural Computation 33

2.2.1 Conventional Hardware Approaches 33

2.2.2 Unconventional Hardware Approaches 38

2.2.3 Other silicon-based designs 55

2.2.4 Hardware Approaches Summary 62

2.3 Systemic Computation 63

2.4 Prior Systemic Computation Implementations 68

2.4.1 Original SC Implementation 68

2.4.2 High-level SC Implementation 70

2.4.3 GPU SC Implementation 71

2.5 Summary 72

Chapter 3 Designing a Suitable Hardware Architecture for SC 77

3.1 Potential Architectures 77

3.1.1 Virtual SC 78

3.1.2 Fundamental Processing Element 79

3.1.3 Reconfigurable Predetermined Processing Elements Array 80

3.1.4 SC2HDL 82

3.1.5 GPUplusFPGA 83

3.1.6 Summary 84

3.2 Design Analysis of the SC Architecture 85

3.2.1 Local Knowledge & Scope Definition Method 86

3.2.2 Scopes Support 86

3.2.3 Valid Triplet Generation & Schemata Matching 87

3.2.4 Threshold Matching 89

3.2.5 Systems Representation & Coding Method 89

3.2.6 The Compiler 91

3.2.7 Interactions Order 91

3.2.8 SC Architectural Features Summary 93

3.3 HAoS Instruction Set 94

3.4 HAoS Architecture 98

3.5 The Control Unit 99

3.6 The SC Core 100

3.6.1 The Core Memory Elements 101

3.6.2 The Random Selection Logic 102

3.7 Programming HAoS 108

3.8 Initial Testing 113

3.8.1 Functional Verification 113

3.8.2 Implementation Statistics 117

3.9 Summary 117

Chapter 4 The HAoS Programming Platform 119

4.1 HAoS-CPU Communication Interface Investigation 119

4.2 CPU Subsystem Integration 127

4.3 Optimizations and Enhancements 128

4.3.1 Refining the Random Selection Logic 128

4.3.2 Minimizing the Schemata-Matching Overhead 129

4.3.3 Further Addressing I/O Efficiency 130

4.3.4 Further Addressing User-Friendliness with a Functional Model 132

4.3.5 Further Addressing Programmability 133

4.3.6 Refinements Results 137

4.4 Addressing Scalability for Single-Chip Implementations 138

4.5 HAoS Model Development Methodology 140

4.6 Summary 143

Chapter 5 Verification and Evaluation 144

5.1 A Genetic Algorithm Optimization of the Binary Knapsack Problem 145

5.1.1 The Binary Knapsack Problem 145

5.1.2 Applying a Genetic Algorithm to the Binary Knapsack Problem 146

5.1.3 Systemic Analysis 148

5.1.4 SC Binary Knapsack Model Implementation 150

5.1.5 Results 153

5.1.6 Analysis 154

5.2 Simulation of a Biochemical Process with HAoS: the MAPK Signalling Cascade 157

5.2.1 The MAPK Signalling Cascade 157

5.2.2 Systemic Analysis 158

5.2.3 SC MAPK Signalling Cascade Model Implementation 162

5.2.4 Results 165

5.2.5 Analysis 167

5.3 Modelling the Effect of Chromosome Missegregation and Typical Cancer Therapy Approaches in Tumour Evolution with HAoS 168

5.3.1 The Cancer Model 170

5.3.2 Systemic Analysis 174

5.3.3 SC Cancer Model Implementation 187

5.3.4 Results 193

5.3.5 Analysis 201

5.4 Summary 210

Chapter 6 Conclusion 212

6.1 Summary of Work Revisiting the Objectives 212

6.2 Contributions 216

6.3 Critical Evaluation ... 217

6.4 Future Work ... 220

6.5 Closing Words ... 223

References ... 226

Appendix A. SC Example Test Programs Source Code ... 245

Appendix B. CPU Subsystem Integration Details ... 248

Appendix C. RSL Optimisations Details ... 250

Appendix D. Revising the TCAM Design ... 253

Appendix E. Provided HAoS API ... 255

Appendix F. File Types used along HAoS Model Development ... 256

Appendix G. Cancer Models SC Source Code ... 257

Appendix H. HAoS Binary-To-ASCII Conversion Resulting Text File Format ... 262

List of Figures

Figure 1.1. Comparison in flexibility and efficiency of prior software SC implementations 22

Figure 1.2. Breakdown and organisation of thesis investigation 25

Figure 2.1. Example organisation of a computer cluster 36

Figure 2.2. Anatomy of a WSN node. 40

Figure 2.3. Hardware Evolution using a Genetic Algorithm. 45

Figure 2.4. The three organizational layers of the POEtic tissue. 47

Figure 2.5. Overview of the POEtic tissue architecture. 48

Figure 2.6. Organisation of the Ubichip architecture. 50

Figure 2.7. SpiNNaker CMP chip organisation. 56

Figure 2.8. Organic System Architecture. 59

Figure 2.9. SC notation and systems representation 64

Figure 2.10. Illustration of a sum operation on a pool of data systems using SC notations. 65

Figure 2.11. SC calculation of PRINT((A1-A2)*(A3-A4)). 67

Figure 2.12. SC calculation of the linear expression ((((A1-A2)*A3)+A4)/A5). 68

Figure 2.13. System representation, schemata decoding scheme and scope table of the original SC version. 69

Figure 2.14. Visualisation of a SC model using SCoPE 70

Figure 2.15. Overview of the task and data flows in GPU SC 72

Figure 3.1. Virtual SC architecture simplified block diagram 79

Figure 3.2. SC Fundamental Processing Element architecture simplified block diagram80

Figure 3.3. SC Reconfigurable Predetermined Processing Elements Array architecture simplified block diagram 81

Figure 3.4. GPUplusFPGA architecture simplified block diagram 83

Figure 3.5. The suggested hybrid design 85

Figure 3.6. Typical RAM and TCAM usage 88

Figure 3.7. HAoS TCAM usage 89

Figure 3.8. HAoS Systems Representation 90

Figure 3.9. HAoS Top-Level Architecture 98

Figure 3.10. HAoS Program Control Flow 100

Figure 3.11. HAoS Core basic building blocks 101

Figure 3.12. The TCAM contents 102

Figure 3.13. Parallel Bit Count Example 103

Figure 3.14. Shift-and-Add implementation of the parallel bit-count mechanism 104

Figure 3.15. HAoS Divider based on a modular approach. 105

Figure 3.16. 16-bit Fibonacci and Galois type LFSRs 106

Figure 3.17. Human-readable HAoS Assembly Code 110

Figure 3.18. HAoS Sample Output 111

Figure 3.19. The three verification example SC programs 116

Figure 4.1. Revised Triplet Memory Map and Write-Detection Mechanism 131

Figure 4.2. HAoS driver flow diagram 135

Figure 4.3. HAoS programming toolchain and software framework 136

Figure 4.4. Linearity on area utilization as the number of maximum supported systems is increased 140

Figure 4.5. HAoS model development methodology 141

Figure 5.1. 16-Element Binary Knapsack Problem 146

Figure 5.2. Representation of the optimal solution for the Binary Knapsack Problem . 147

Figure 5.3. Standard genetic operators: Binary Mutation, One-Point and Uniform Crossover 148

Figure 5.4. The binary knapsack SC model. 150

Figure 5.5. Binary knapsack problem experimental results across a range of number of systems 154

Figure 5.6. Binary knapsack experiments HAoS normalised performance compared to the sequential and GPU implementations 155

Figure 5.7. Simplified biochemical description of the MAPK signalling cascade ignoring phosphate groups 158

Figure 5.8. The HAoS MAPK model in SC graphical notation. 161

Figure 5.9. Traversing the MAPK signalling cascade 166

Figure 5.10. MAPK cascade experiment HAoS normalised performance compared to the SPiM and SCoPE simulators 168

Figure 5.11. The reference cancer model. 171

Figure 5.12. The three genetic configurations, employing different gene chromosomal linkage 171

Figure 5.13. Initial SC cancer model 175

Figure 5.14. Revised SC cancer model with tissue and two-step cell death and division processes 176

Figure 5.15. Therapy-Enabled SC cancer model 178

Figure 5.16. Time-Enabled (Tic-Toc) SC cancer model 181

Figure 5.17. Optimized SC cancer model 186

Figure 5.18. SC cancer model data systems and their contents 189

Figure 5.19. Non-therapy cancer models comparison for gene distribution A 195

Figure 5.20. Non-therapy cancer models comparison for gene distribution B 196

Figure 5.21. Non-therapy cancer models comparison for gene distribution C 197

Figure 5.22. Therapy-enabled cancer models comparison near cancer diagnosis for gene distribution B... 198

Figure 5.23. Therapy-enabled cancer models comparison near cancer diagnosis for gene distribution C... 199

Figure 5.24. Non-therapy cancer time-supporting models results differences for gene distribution A 203

Figure 5.25. Non-therapy cancer time-supporting models results differences for gene distribution B... 204

Figure 5.26. Non-therapy cancer time-supporting models results differences for gene distribution C... 205

Figure 5.27. Differences on apoptosis-to-division ratio between the therapy-enabled time-supporting cancer models for gene distribution B.. 206

Figure 5.28. Differences on apoptosis-to-division ratio between the therapy-enabled time-supporting cancer models for gene distribution C.. 207

Figure 5.29. Cancer growth experiment HAoS normalised performance against the dedicated c++ implementation 209

Figure 5.30. Performance Evaluation Results Summary 211

Figure 6.1. Comparison in flexibility and efficiency provided by the HAoS programming platform... 225

Figure B.1. The AXI4-Lite based HAoS-MicroBlaze communication link 248

Figure B.2. Top-level On-Chip HAoS Platform Block Diagram 249

Figure C.1. Critical path of initial HAoS design based on static timing analysis 250

Figure C.2. The Revised RSL module 251

Figure D.1. The SRL16E-based building block of the base HAoS TCAM 253

Figure H.1. Annotated HAoS ASCII program example 262

List of Tables

Table 1.1. Properties that differentiate natural from conventional computation 19

Table 2.1. Approaches to Natural Computation and alternative paradigms 30

Table 2.2. Features of the original SC implementation 69

Table 2.3. Detailed evaluation of the reviewed hardware-based approaches 75

Table 3.1. HAoS Compression Code 91

Table 3.2. Implementation-specific features of HAoS and prior implementations 93

Table 3.3. HAoS Instruction Fields 95

Table 3.4. HAoS Instruction Set 95

Table 3.5. Simulated SC test programs 114

Table 3.6. HAoS Prototype (64 systems) Implementation Statistics on Virtex-6 LX240T FPGA 117

Table 4.1. Commonly used interfaces for off-board communication and their nominal raw bandwidths 121

Table 4.2. Sustainable Bandwidth Results for various practical configurations implementing common communication interfaces on FPGAs 123

Table 4.3. Indicative Collection of Available Soft Processors 125

Table 4.4. Benchmark timing improvements reflecting various architectural optimizations 137

Table 4.5. HAoS platform implementations statistics as the number of maximum systems increases 139

Table 5.1. Summary of the Knapsack SC model functions 152

Table 5.2. The HAoS MAPK model interactions in SC calculus notation. 162

Table 5.3. Summary of the MAPK SC model functions 164

Table 5.4. Performance of the HAoS, SPiM (v1.13) and SCoPE simulators based on simulation duration 167

Table 5.5. Time-Enabled (Tic-Toc) cancer SC model interactions 182

Table 5.6. Timeless and Approximate Time cancer SC models interactions 184

Table 5.7. Optimized Approximate Time cancer SC model interactions 186

Table 5.8. Biological representation of the systems of the SC cancer models 188

Table 5.9. Absolute and normalized average execution times for all simulated cancer scenarios 200

Table 5.10. Statistical comparison of the time-enabled cancer models to the reference one in terms of mean error (ME), standard deviation (STD) and correlation (COR).... 208

Table E.1. Summary of the provided functions (simplified) in the HAoS API 255

Table F.1. Summary of file types used along HAoS model development 256

List of Listings

Listing 3.1. HAoS Source Code Example 109

Listing 3.2. Selected output from the 3 example SC test programs verifying the functionality of HAoS by simulation 115

Listing 5.1. Binary Knapsack SC model source code 151

Listing 5.2. Verification environment output for Binary Knapsack SC model 153

Listing 5.3. MAPK Signalling Cascade SC model source code 163

Listing A.1. Addition in Multiple Scopes (Test1 Example) Source Code 245

Listing A.2. Subtraction-escape and then multiply and print (Test2 Example) Source Code 246

Listing A.3. Context Adapting (Test3 Example) Source Code 247

Listing G.1. Time-Enabled Cancer Model SC Source Code 257

Listing G.2. Timeless Cancer Model SC Source Code 258

Listing G.3. Approximate-time Cancer Model SC Source Code 259

Listing G.4. Optimized Approximate-time Cancer Model SC Source Code 260

List of Algorithms

Algorithm 5.1. The reference cancer model algorithm 173

Algorithm 5.2. The reference cancer model algorithm written in a “systemic way” 191

Chapter 1

Introduction

1.1 Introduction to Natural Computation

It has been claimed that everything computes [1], [2]. Biological systems appear to be superb at performing something that resembles computation, although they accomplish that by using methods fundamentally different from those used to perform conventional computation [3], [4]. Supercomputers strive to simulate a microsecond of protein folding [5], yet biology scales from molecules to cells, then to organisms, then to species and so on to more complex structures.

Complex tasks, like DNA synthesis and sequencing, have been shown to outpace Moore's law [6]. Although the semiconductor industry has been making continuous leaps in the past half century, silicon-based approaches seem weak in delivering more raw power, as the physical limitations of this technology appeared quite some time ago [7]. While engineers are left to devise workarounds to these issues (cache memory, branch prediction, out-of-order execution, multi-core chips), modern computers seem to be inefficient and too slow to model biological processes. This incompetence is not surprising, since, although the advances in microprocessor technologies have been numerous, the fundamental design principles have remained unchanged for almost a century. The vast majority of computing devices today follow the design pattern revealed in 1945 by John von Neumann [8]. This is a completely centralized partitioning, comprising of a set of main building blocks: the Central Arithmetic (CA) unit, the Central Control (CC) unit, the Memory (M) and Input/Output (I/O) devices. Von Neumann believed that those "distinctions suggest themselves immediately" [8] and until today the majority of those in the scientific community and the consumer industry agree with this. However nature does not.

While computation in a conventional electronic computer is the outcome of a program, which is a set of defined instructions that are sequentially executed, the rules are quite different in nature. Nature seems to work in a massively parallel fashion instead. Natural systems, viewed in different levels of abstraction, have a common characteristic. A massive number of subtasks are carried out at any given moment in order to accomplish an operation. This operation can be any biological process. For example, under different contexts, the process of photosynthesis from a leaf in a tree involves chemical reactions among complex biochemical systems in order to convert carbon dioxide in organic compounds [9]. This process is happening through all the leaves of the tree, and it is vital for the tree itself. The tree comprises a system of leaves, branches and roots and each is playing its role in accomplishing the survival of the tree. At the same time, photosynthesis is not only vital for the tree but for all flora in the ecosystem, and in turn for any living organism on the planet relying on oxygen for survival.

The human brain is another example. It is composed of billions of neurons which continuously interact [10] with each other. The brain is just one of the organs that build the nervous system, which in turn along with other systems compose the human body. Groups of people form societies and all the societies, joined, build mankind.

Numerous examples like the ones mentioned above can be given: a herd of deer, an ant colony, our planetary system, the immune system, a school of fish and even the Dow Jones Index. All of them are composed from fundamental building blocks but also, combined with others, constitute more complex structures. The underlying processes seem to work without any centralised control method but with the coexistence and interaction of their structural elements.

The observation of the success of nature in coping with such complex systems had a significant impact in modern science, giving birth to several biologically inspired research fields [11]: Evolutionary Computing (EC) [12], Artificial Neural Networks (ANN) [13], Artificial Immune Systems (AIS) [14], Swarm Intelligence (SI) [15], Particle Swarm Optimization (PSO) [16], Cellular Automata (CA) [17], L-systems[18], Artificial Life (ALife) [19], DNA computing [20] and Quantum Computing (QC) [21] are some of them. According to [11], these fields form three groups: the first five are inspired by nature, the next two (CA and L-systems) simulate and emulate nature by means of computing, while the rest use natural materials for computation. They are all influenced/inspired by nature, serving computation and modelling purposes and hence they constitute a super-group: Natural Computation [11], [22], [23].

Characteristics, embedded in natural systems, have been a rich source of inspiration for the scientific community since it is commonly accepted that nature can outperform any manmade device on factors like complexity, homoeostasis, self-organization, self-replication, self-adaptation and fault-tolerance.

Table 1.1. Properties that differentiate natural from conventional computation

Property Type	**Natural Computation**	**Conventional Computation**
Computational	Stochastic	Deterministic
	Asynchronous	Synchronous
	Parallel	Serial
	Distributed	Centralized
	Continuous	Batch
	Approximate	Precise
	Embodied	Isolated
	Local Knowledge	Global Knowledge
	Circular causality	Linear causality
Behavioural	Self-organised	Explicitly Organised
	Fault tolerant	Fault intolerant
	Open-ended	Limited
	Complex	Simple
	Autonomous	Human-reliant
	Homoeostatic	Heterostatic
	Robust	Brittle

Various opposing properties that highlight the distinction between natural and conventional computation as they are separated in [22], [24] are given in Table 1.1. Therefore, natural computation in general is/has [22]:

- Stochastic: The behaviour of natural systems is non-deterministic and their interactions are randomised.
- Asynchronous: Mostly[1], behaviour is not synchronized. There are no clock signals which determine the timing of every behaviour akin to our processors.
- Parallel: Interactions are usually concurrent among all systems.
- Distributed: Computation is spread and allocated across several systems to achieve the result.
- Continuous: Natural systems are designed to keep working for as long as possible; their behaviours are designed to work continuously for the lifetime of each organism.
- Approximate: The notion of an exact number or quantity is meaningless.

[1] There can be approximate synchronization to solar or lunar cycles or seasons.

- Embodied: A natural system and its environment constantly affect each other.
- Local Knowledge: Knowledge is not stored in a centralised archive or library. An interaction can occur among two systems at the same hierarchical level which are within range of each other, implying scopes of interaction.
- Circular causality: Two interacting natural systems affect each other during interaction.
- Self-organised: Natural systems define their own organization and architecture without external interventions.
- Fault-tolerant: Natural systems are tolerant to partial failures and usually able to also self-heal.
- Open-ended: Systems in nature are able to adapt and constantly evolve.
- Complex: Natural systems are organized over numerous hierarchical levels in a bottom-up manner. Starting with basic elements at the lowest level, they built successively more complex systems at the higher levels.
- Autonomous: Natural systems are self-reliant and independent of any external authority.
- Homoeostatic: A natural system preserves the inner stability of its state(s) by internal feedback mechanisms.
- Robust: Natural systems can handle and adapt to unforeseeable situations.

Influenced by the importance of those properties, a novel computation model was conceived by Bentley [24]. The new model, systemic computation (SC), was proven to be Turing complete [25] and attempts to embody the much sought characteristics of biological systems found in nature as listed in the left column of Table 1.1[2]. Turing completeness was proved by implementing a rule 110 cellular automata algorithm [17], [26], [27], stating the equivalence of SC to any other computation model.

1.2 Introduction to Systemic Computation

Systemic computation, further discussed in section 2.3, has its roots on the work of Jean-Louis Le Moigne's [28] on General System Theory [29]. The core notion that was adopted by systemic computation can be found in the second percept [28] of Le

[2] In this work the focus will be on the computational properties.

Moigne's systemic method – interaction – as opposed to the corresponding percept on Descartes' analytical method [30] – reduction. Reductionism can be traced back to ancient Greece [31]. It states that a complex system is a sum of its parts but it is known to have limitations [32]. Holism on the contrary argues that a complex system is more than the sum of its constituents.

Systemic computation adopts a holistic analysis approach of systems embracing the significant importance of the interactions of their fundamental elements and their environment. Its intention is to resemble natural computation in order to simulate biological processes effectively. To accomplish this, it follows the conventions listed below [24] :

- Everything is a system.
- Systems may comprise or share other nested systems.
- Systems can be transformed but never destroyed.
- Interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and also all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems.
- Computation is transformation.

According to these conventions, it is implied that in order to perform any computation in SC, two main tasks are always involved:

- Identify the interacting systems and
- Transform the interacting systems according to the interaction determined by the contextual system in the scope that this interaction is defined.

1.3 Systemic Computation in Practice

While the Systemic Computation paradigm has been designed to feature all the properties of Natural Computation, as they are given in Table 1.1, a practical platform to support SC has yet to be devised. Its highly unconventional nature makes the implementation of such a platform very challenging, since it radically differs from the notion of computation, as we have grown to perceive it. The validity of the concept has been proven in previous work [22], but in order to take advantage of its potential, applying SC in a practical and efficient way is essential.

Three SC implementations have been developed so far (Original Sequential SC Implementation [24], High-level SC Implementation [22] and the GPU SC Implementation [34]). However, since their conventional design does not denote a natural way of implementing the SC paradigm, they are just low and high level simulations of a systemic computer, with only the latest implementation succeeding in obtaining satisfactory results in terms of speed. As shown later, in section 2.4, these software approaches are largely inappropriate to implement a SC platform, mainly due to the conventional sequential nature of their underlying architecture which is incompatible with the SC paradigm. As illustrated in Figure 1.1, there is no current implementation that combines flexibility with efficiency. Consequently there is a clear need for a new SC programming platform that is both efficient and flexible.

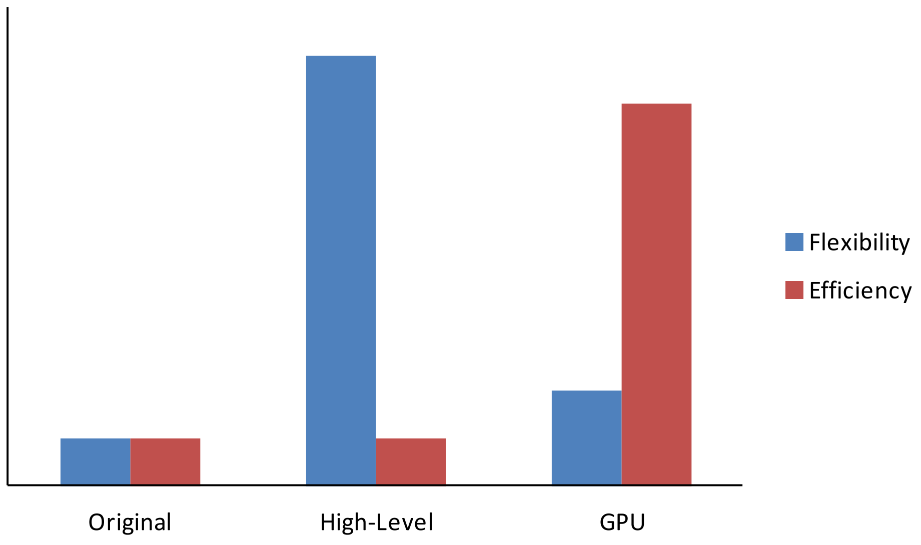


Figure 1.1. Comparison in flexibility and efficiency of prior software SC implementations. A practical hardware-based implementation is expected to provide a balanced SC programming solution

As previous work has demonstrated the incompatibility of conventional hardware for SC, it seems likely that the most practical, viable and usable platform which addresses this need would be a novel hardware-based implementation.

It is thus vital to investigate the trade-offs of available implementation platforms in order to identify the substrate that a practical SC platform can be based on and then explore how the practical features of conventional computation can be combined with the unconventional properties of natural computation and architectural features of SC. In order to properly evaluate such a controversial design, it is required to identify the requirements that define a practical platform for SC and the degree that each of them can be satisfied.

1.4 Hypothesis

The hypothesis of this thesis is:

It is possible to implement a practical Systemic Computation hardware architecture that is viable and useful.

The thesis will provide evidence to support this hypothesis through an investigation of the viability and utility of a SC practical implementation. Yet, the unconventional nature of SC may itself be proven to be partially incompatible with the practicality aspect of the implementation, as practicality partially implies a conventional way of thinking and undertaking well-studied and proven techniques to accomplish a feasible and usable means to perform Systemic Computation.

In essence, this collision of the definitions of unconventionality and practicality, in a computational context, formulates the main investigation that this work attempts to tackle. It is suggested that investigating the features, advantageous and disadvantageous, that modern hardware implementation platforms offer while exploring potential suitable architectures for Systemic Computation, will result in a satisfactory compromise combining the benefits of the inherent natural properties of SC with the usability and utility provided by a practical platform.

This work will investigate the *viability* of a practical SC implementation and the trade-offs between encompassing naturals properties against the feasibility and constraints of the hardware taking into consideration flexibility, performance and scalability. The supported programming model should provide a user-friendly interface to the underlying architecture, which should be optimized in terms of speed and area while being able to easily scale in size.

A practical SC *hardware implementation* is required because software approaches do not seem to be able to efficiently handle the complexity or properly address the implied non-conventional architecture of the SC paradigm (see sections 2.1, 2.4.1 and 2.4.2), since they solely rely on conventional processors. The *utility* of such a custom hardware design will be demonstrated by showing that natural processes can be modelled in a more native way by addressing these limitations and mapping more efficiently and accurately the SC architectural features.

The outcome of this work should be a *practical* hardware implementation in order to be easily reproduced and also be, at least partially, compatible with conventional architectures, in terms of communications. This will enable reusability and enhance flexibility in order to achieve a broader user community which in return can improve the architecture and expand its functionality. This implies that the suggested implementation should also address availability (meaning that a user should be able to relatively easily access the selected enabling technology). Thus it should be based on a mature technology, possibly using Commercial Off-The-Shelf (COTS) [33] components, with a rich knowledge base which is broadly used both in academia and industry.

This work provides evidence to support its hypothesis by proving a proof of concept via a realisation of a novel SC hardware implementation. Building on the discussion of the three previous sections, it accomplishes this by focusing on three main research challenges:

Chg1: How can a hardware platform support the natural properties that are central to SC?

Specifically this challenge focuses on the inherent to SC natural properties of Table 1.1. An ideal platform would be able to support a hardware implementation that would be stochastic, asynchronous, parallel, continuous, distributed, approximate (in a high level) and embodied while it would show circular causality and have only local knowledge. Incorporating these properties, the SC implementation would be self-organized, fault-tolerant, (at least virtually) open-ended, complex, autonomous, homoeostatic and robust.

Chg2: How can a hardware platform support the underlying architecture of SC?

Specifically this challenge focuses on the compatibility of the platform with the inherent features of the implied SC architecture: systems, scopes, contexts and interactions among systems should be able to be represented in a manner that allows efficient modelling of systems interactions.

The first two research challenges refer to the *viability of a SC implementation*. It is suggested that investigating the trade-offs of implementing and attempting to combine the desired natural properties with the architectural features of SC will sufficiently explore how viable such an implementation is. The third challenge addresses the *utility and practicality* of the suggested design and the way of realizing it:

Chg3: How can a hardware platform meet the first two challenges while also being practical and efficient?

Specifically this challenge focuses on the support of features to result in a practical platform: the resulting solution should be user-friendly, taking into consideration flexibility and adaptability, and efficient in terms of performance and required resources which in extent will prove its utility.

Thus, this thesis proves its hypothesis by addressing its three research challenges. We break down the investigation of the hypothesis and the three sub-challenges into a set of objectives, listed in the next section. This is illustrated in Figure 1.2.

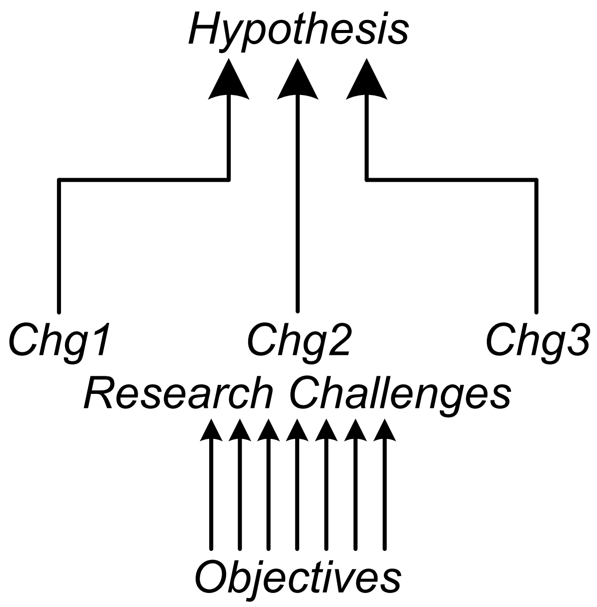


Figure 1.2. Breakdown and organisation of thesis investigation. A set of objectives address three main research challenges which provide evidence to support the hypothesis

1.5 Objectives

The main objectives for this research work can be identified as:

1. Review the work done on Natural Computation to date with a focus on hardware-based approaches.
2. Review and assess the work done on Systemic Computation (theory and implementations) to date.

3. Investigate the suitability of available hardware implementation platforms for SC by evaluating them in terms of their ability to support the natural properties of SC (*Chg1*), the implied SC architecture (*Chg2*), and practicality/efficiency (*Chg3*) and select the most appropriate.
4. Analyse the SC architectural features and create a prototype hardware implementation designed to support the SC architecture.
5. Create a complete and standalone practical SC programming platform with the ability to meet the three challenges.
6. Analyse and address the limitations of the hardware prototype by means of optimizations and enhancements taking into consideration the research challenges.
7. Evaluate the ability of the prototype SC platform to meet the research challenges by simulating natural models against alternative solutions.

1.6 Publications

The work presented in chapter 3 has been awarded the Best Paper Award in the international Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2011) and was published in the Lecture Notes for Computer Science (LNCS) proceedings of the workshop. Overall this thesis resulted in the refereed publication of two international conference papers, two international journal papers, a book chapter and a research poster, listed below:

- C. Sakellariou and P. Bentley, "Introducing the FPGA-Based Hardware Architecture of Systemic Computation (HAoS)", in *Mathematical and Engineering Methods in Computer Science*, Lecture Notes in Computer Science (LNCS) vol. 7119, Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, and D. Antoš, Eds. Springer Berlin / Heidelberg, 2012, pp. 179−190.
- C. Sakellariou and P. Bentley, "Describing the FPGA-Based Hardware Architecture of Systemic Computation (HAoS)", *Journal of Computing And Informatics*, vol. 31, no. 3, pp. 485–505, 2012.
- C. Sakellariou and P. Bentley, "Extending the Hardware Architecture of Systemic Computation to a Complete Programming Platform", in *IEEE International Conference on Evolvable Systems (ICES 2013) - IEEE Symposium Series on Computational Intelligence (SSCI 2013)*, Singapore, April 2013.

- C. Sakellariou and P. Bentley, “Demonstrating the performance, flexibility and programmability of the Hardware Architecture of Systemic Computation modelling cancer growth”, submitted to *International Journal of Bio-Inspired Computation*, *Special Issue on Bio-inspired Hardware*, 2013.
- C. Sakellariou and P. Bentley, “Computing Nature at the Intersection with Chemistry: Innovative Architectures”, Book Chapter to appear in *Genesis Engines: Computation and Chemistry in the Quest for Life's Origins*, Springer, 2013.
- C. Sakellariou and P. Bentley, “Building a Bio-Inspired Computer: The Hardware Architecture of Systemic Computation (HAoS)”, in *Frontiers of Natural Computing Workshop*, York, 2012.

1.7 Thesis Organization

The thesis comprises six chapters, four lists (including list of figures, tables, listings and algorithms), an extensive reference list and eight appendices. Chapter 2 reviews the literature on the field of Natural Computation, critically focusing on hardware-based approaches, and describes the SC theory, as it was introduced by Bentley [24]. It illustrates how SC can perform computation and presents the three prior SC implementations: Original SC Implementation, High-level SC Implementation and the GPU SC Implementation. Furthermore, it identifies the most appropriate SC hardware implementation platform among the various hardware-based approaches to Natural Computation. Chapter 3 introduces the first FPGA-based Hardware Architecture of Systemic computation (HAoS), discusses the functionality of its structural elements, justifies the design decisions which result in this prototype design, outlines the applied optimizations and details a programming example. It also gives implementation statistics of the suggested design on the intended FPGA development board and explains the verification methodology used to confirm its functionality. Chapter 4 investigates suitable approaches for the implementation of the communication interface between HAoS and the CPU, revisits parts of the design providing enhancements taking into consideration performance, I/O efficiency, user-friendliness and programmability. The HAoS base design is combined with an embedded soft processor to provide a standalone platform while a methodology for HAoS models development is suggested. Chapter 5 verifies and evaluates the functionality of the platform by illustrating how HAoS can be used to simulate three natural models of increasing complexity: a genetic algorithm optimization implementation solving the binary knapsack problem, a well-studied biochemical process involving enzyme-based protein activation and a more challenging biological model simulating the effect of genetic anomalies and typical treatment approaches to cancer growth. The provided models are given as SC model development examples and the acquired results are compared against previous SC implementations and other conventional programming approaches. The time complexity of the HAoS schemata matching mechanism is also evaluated. Finally, chapter 6 summarizes the thesis, states its contributions, provides a critical evaluation and discusses future work.

Chapter 2

Background

The notion of natural computation [35] resulted in significant advances on research in the field of natural computing [11], [23]. Adleman after successfully solving a seven-point Hamiltonian Path problem [36] using DNA sequences in 1994 [20], concludes his article:

"*Biology and computer science —life and computation— are related. I am confident that at their interface great discoveries await those who seek them.*"

Leonard Adleman [20]

This chapter discusses various methods attempting to approach natural computation, starting with a broad perspective and increasingly focussing on work more closely related to the topic of this thesis. Section 2.1 provides an overview of some of the major works in this area, and specifically in terms of software-based approaches and alternative paradigms. Section 2.2 gives a critical review of literature related to hardware approaches to natural computation. Conventional and unconventional ways and some related hardware designs are presented in this section and initially assessed regarding their compatibility with the Systemic Computation concept (a thorough analysis is given in the next chapter). Their potential to become the basis of, or inspire the features and requirements of a SC hardware implementation is discussed, as implied by the three research challenges (section 1.4 - supporting natural properties and the SC architecture and being able to facilitate an efficient and practical implementation). Section 2.3 elaborates on the SC paradigm as it was introduced by Bentley [24]. Finally, the three prior SC implementations are overviewed in section 2.4.

Part of the work presented in this chapter has been accepted for publication in [37]. Also, part of this work has been previously submitted for the degree of Master of Research as part of the Doctor of Engineering degree in UCL.

2.1 Approaches to Natural Computation

The Natural Computation research field is quite broad. Kari and Rozenberg, attempting to provide a complete review on the field in [23], separate its individual fields and computing paradigms in three groups using the role of nature as the differentiating factor: nature as inspiration, nature as implementation substrate and nature as computation. Especially for the last group, computation can both refer to quantitative algorithms and qualitative approaches that investigate natural processes taking into consideration communications and interactions [23].

Table 2.1. Approaches to Natural Computation and alternative paradigms

Software Approaches

Conventional	Exception handling
	Recovery blocks
	N-version programming
	Expert systems
	Multi-agent systems
Nature-inspired	Evolutionary algorithms
	Artificial neural networks
	Swarm intelligence[3]
	Artificial immune systems
	Artificial Life
	Fractal Geometry (Cellular automata-L-systems)

Computational Paradigms

Maths, physics & technology inspired	CGPs[4]
	π-calculus
	Asynchronous π-calculus
	Stochastic π-calculus
	Ambient calculus
	Petri nets
	Statecharts[5]
	Bigraphs
	Ons algebra
Nature-Inspired	BioAmbient calculus
	Membrane computing
	Brane calculi
	CLS[6]
	Bio-graphs
	Systemic computation

Hardware Approaches

Conventional	Nature-inspired
Multi-core / Multi-CPU	Ubiquitous computing
High-availability cluster	Reaction-diffusion computing
Beowulf cluster	Speckled computing
GPU	FPGA / ASIC
Grid / Cloud computing	Evolvable hardware
Pure Peer-to-peer	POEtic / Ubichip
Load-balancing cluster	Wireless Sensor Network
	Collision-based computing
Super-computers	Molecular (DNA) computing
	Organic Computing
	Bacterial Computing
	Quantum computing

3 Ant colonies (ACO), Particle Swarm Optimization (PSO)

4 Constrained Generating Procedures

5 Just a flow graphical tool, not a computational paradigm

6 Calculus of Looping Sequences

The various approaches to date on natural computation, which for this work are separated into software-based methods, alternative paradigms to computation and hardware-based methods are illustrated in Table 2.1. An extensive literature review can be found in [11], [22], [23]. While the focus of this thesis lies on the hardware-based approaches (discussed in the next sections), a summary (informed by [22]) regarding the related software-based approaches and alternative concepts of Table 2.1 is provided below.

2.1.1 Software Approaches

Computation based on software approaches is quite common since they provide great flexibility and ease of use. Conventional software approaches, usually, do not take into consideration natural properties while conventional hardware approaches consider them by simulating them.

Conventional approaches address issues like reliability, robustness and autonomy. Exception handling [38] provides a mechanism of controlling the execution flow in case of foreseeable special cases. Recovery blocks [39] (the same programmer writes multiple versions of some parts of a program – blocks of code) and N-version programming [40] (multiple versions of a whole program are written by different programming teams) exploit code redundancy in order to overcome failures and minimize errors. Expert [41] and multi-agent [42] systems are used to perform autonomous tasks, the former by performing an analysis on a given problem and providing answers, the latter by diverging information and/or interests.

Computer scientists, inspired from nature, expanded on unconventional methods, adapting their programs to create or simulate natural properties like self-organization, self-adaptation and fault-tolerance. A Genetic Algorithm [12] (described in section 2.2.2 in the context of Evolvable systems) is a global heuristic search method and provides distributed, parallel, local and autonomous computation. Artificial Neural Networks [13] is a field inspired by biological neural mechanisms and shows distributed knowledge and self-organization. Swarm Intelligence [15] mimics concepts, inspired by insect civilizations, and based on their collective behaviour obtains self-organization and self-adaptation. Those properties are also observed in Artificial Immune Systems [14], which derive inspiration by (mostly) the adaptive and (less) the innate responses of biological immune systems, and Artificial Life [19] which is a field of study (and an associated form of art [43]) that employs a synthetic approach to the study and creation of life [11] (typical subjects of this study are termites, flocks, herds, evolution and artificial chemistries). Fractal Geometry [11] deals with non-Euclidean objects of non-integer dimensions which are characterized by self-similarity and infinite detail. In a computational context, Fractal Geometry includes the fields of Cellular Automata [44], [45], systems that are discrete in both time and space, showing properties like self-replication and autonomy, and L-systems [18], a formalism to simulate the development of multi-cellular organisms [11] employing parallel rewriting systems (able to modify an existing word and generate new ones by applying various rewriting rules to its characters in parallel) [23].

2.1.2 Alternative Paradigms

In order to further understand and exploit natural processes, new paradigms of computation were developed, since conventional languages were not well suited for effectively simulating nature [22]. Inspiration was derived by conventional sciences (maths, physics and technology) and from nature.

CGPs [46] are finite state machines that can analyse complex systems by reducing them (breaking them down) in mechanisms and constraints of interactions. π-calculus [47] and its extensions (asynchronous π-calculus [48] and stochastic π-calculus [49]) are process calculi used or adapted for biological systems simulation. Ambient calculus [50] is also a process calculus which was developed to describe concurrent systems that include mobility. Petri nets [51] are a graphical tool, with a corresponding mathematical theory, that describes concurrent processes. Originally they were targeting chemical processes. Statecharts [52] are commonly used to describe the data and control flow of state machines in communication and, in general, hardware systems. Bigraphs [53] provide a well defined form of concurrent computations and a graphical notation, that exploits topographical and communication ideas, which is well suited for a number of the aforementioned calculi. Systemic computation can be seen as equivalent to bigraphs while the two paradigms share a similar graphical formalism. Ons algebra [54] is an algebraic formalism attempting to reach the foundations of physical rules development by using, in a metaphorical way, only two elementary particles, the particle of time and the particle of space.

BioAmbients Calculus [55] was designed to allow modelling of biological systems, having biological compartments as a central idea. Membrane computing [56] deals with distributed and parallel computing models of systems (P systems), that use the analogy of the organization of a cell being compartmented by membranes, creating this way hierarchies. Brane calculus [57] identifies the importance of the membrane itself and gives it an active role on computation. The Calculus of Looping Sequences [58] and its variants is another formalism that allows the description of biological membranes, biomolecular systems and proteins interaction. Bio-graphs [59] were designed to model biological systems in a molecular level and include a corresponding graphical notation.

Since the hypothesis of this work focuses on a hardware-based implementation of SC, the reader is redirected to the work of Le Martelot [22] for a detailed description and a critical review which compares SC with the software approaches and the computational paradigms listed in Table 2.1. The various hardware approaches are described in the next sections.

2.2 Hardware-based Approaches to Natural Computation

While computer applications become ever more computationally demanding, the traditional Von Neumann [8] architecture, after serving humanity for more than half a century, appears inadequate [2], [60], [61] when extremely complex tasks are involved (brain function modelling, protein folding). Although refurbished designs keep consumers happy, hardware designers and researchers realised that alternative approaches should be followed for ground-breaking efficiency and performance improvement.

As explained in the previous sections, researchers found inspiration in nature. This is reflected in various hardware-based approaches. In this section, both conventional (subsection 2.2.1) and unconventional (subsection 2.2.2) hardware-based approaches are described[7]. In the context of this thesis, unconventional approaches do not conform to the conventional von Neumann architecture or use standard technologies (and are usually inspired by nature). Relevant silicon-based designs are discussed in subsection 2.2.3. Each paragraph is concluded with a short discussion on the compatibility of each approach to a practical SC implementation.

2.2.1 Conventional Hardware Approaches

Conventional hardware-based approaches to natural computation include multi-core chips, supercomputers, computer clusters, peer-to-peer networks and GPUs. They are usually based on some variation of the von Neumann architecture, except GPUs which fall in this category since they are widely used in consumer desktops and laptops, and are

[7] It is noted that because of different definitions for various technologies or for clarity reasons there is some overlap between the technologies and methods described in next sections.

attempts to provide more processing power using various design approaches explained below.

Chip Multi-Processors

Chip Multi-Processor (CMP) [62] systems were the response of the semiconductor industry, to the consumer market, when around 2003 the clock frequency of uniprocessor systems reached the limits imposed by the physics of their underlying technology. CMPs take advantage of the limited parallelism that multiple processors provide, often being able to execute more than one instruction thread simultaneously each. The limitation of CMPs to provide natural computation is evident, as their sequential architecture is incompatible with any natural property (except maybe parallelism, but that is true only when they are compared with their predecessor uniprocessor architectures).

CMPs are based on the conventional von Neumann architecture [62]. They are based on the most widely used hardware implementation approach to computation, since their deterministic sequential processors are highly flexible and easily programmable. Their technology is more mature than any other. As such, there is a plethora of tools, specifically designed for them. However, their flexibility comes at the expense of performance, as their generic architecture cannot compete with custom designs, optimized for specific applications. The nature of their architecture makes them incompatible with almost any natural property (maybe except parallelism, since they provide limited support), therefore they are unsuitable for a SC implementation. As shown later in sections 2.4.1 and 2.4.2, CMPs were used for the first two SC implementations, revealing the inefficiency of such an approach, as these implementations could only simulate a systemic computer. Although the high-level SC implementation provided programming flexibility, performance limitations make it inadequate for modelling complex systems.

Supercomputers and Computer Clusters

While CMPs are targeted to the consumers, supercomputers [63] are used for computationally super-demanding tasks, such as modelling climate change, nuclear reactions and molecular interactions [63]. They were introduced by Seymour Cray in the 1960s. Modern supercomputer designs often consist of a cluster of Multiple Instruction stream - Multiple Data stream (MIMD) multiprocessors, which have Single Instruction - Multiple Data (SIMD) processors as building elements. The SIMD processors execute the same instruction on different sets of data while the MIMD processors function asynchronously, enabling the underlying SIMD units to perform different operations on different sets of data. According to Flynn's "very high speed computing systems" (supercomputers) taxonomy [64], which dates back to 1966 there are also MISD and SISD supercomputers. According to a more modern dichotomy there are SIMD, multiprocessor (all the processors under the same instance of operating system - OS) and cluster computers (each processor under a different instance of the OS).

Supercomputers may appear as a possible SC implementation, although gaining access to a modern supercomputer can be limited and that would mean that only privileged users could use the SC paradigm. The code is usually specially written for such processors, in order to be highly optimised, resulting in limited portability.

Computer clusters are a supercomputer type. While relying on sequential processors for instruction execution, they follow a network structure resulting in a parallel architecture that shows signs of fault-tolerance and distributed operation, as a failure in one of the nodes will not terminate the operation of the cluster. The Commodity-of-the-shelf (COTS) clusters [65] can be built from consumer parts but rely heavily on software to deliver performance. A collection of representative projects on COTS can be found at [66].

Other classifications of clusters are based upon their functionality. High-availability clusters [67] use duplicates to survive individual computer failures. Computers provide feedback to each other to detect failure. However, the detection scheme is susceptible to failures as well. Beowulf clusters [35] use a one-server-multiple-clients organization to achieve high performance but suffer from the centralized control that resembles the multiprocessor architecture. Load-balancing clusters [68] adopt the server-client approach as well but, additionally, they distribute the workload among them through software. Server farms are load-balancing clusters where all the nodes are servers. Grid computing is also a cluster based approach, although the nodes seem autonomous. The user gains access to the processing power supply just by joining the network. Cloud computing is very similar to grid computing. The main difference is that cloud computing provides on-demand resource (and services) provisioning.

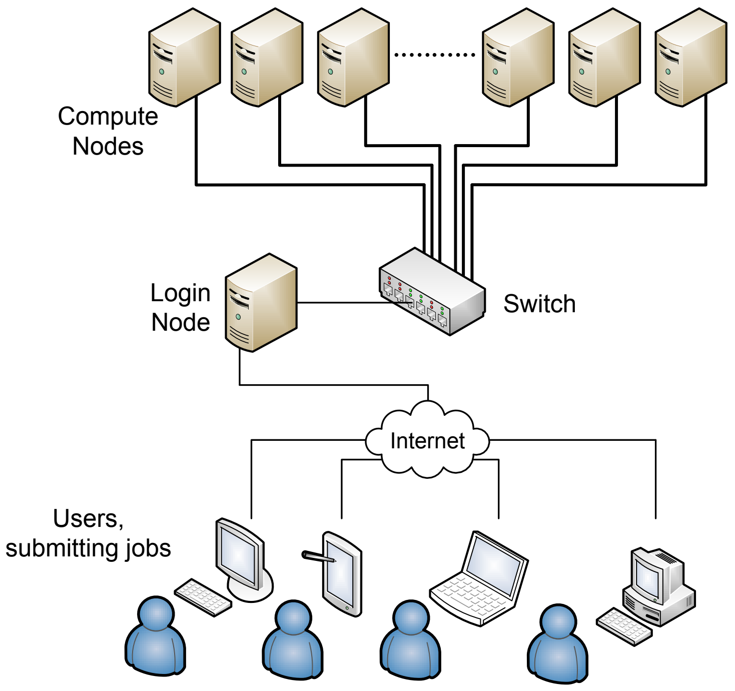


Figure 2.1. Example organisation of a computer cluster

Supercomputers provide vast levels of parallelism [63]. They provide great performance for specific applications tailored to their specialized hardware architecture. Computer clusters, which are a type of supercomputer, appear to be strong candidates for a SC hardware implementation as they show a level of asynchronous (in the cluster level) and distributed computation, by forming a network of cooperating conventional synchronous computers. Load-balancing clusters, in particular, show a low level of self-organization (by using specialized software for task distribution), while high-availability clusters provide fault-tolerance by having duplicate nodes for the same task. Computer clusters are usually easily scalable, since nodes can join the network dynamically.

It appears that the vast number of computational resources, provided by a supercomputer, would be sufficient for a SC implementation. However, their availability can be very limited and their building blocks are based on conventional architectures using centralized control making them incompatible with the SC paradigm, since it provides limited support for natural properties. This limited support mainly derives (in computer clusters) from their organization in a network pattern. Thus, this feature may be employed by the SC implementation.

Pure Peer-to-Peer

Peer-to-Peer (P2P) networks [69] originally referred to networks that consisted of identical nodes, lacking administrative elements. Pure P2P networks refer to networks of peers that exchange resources and execute operations in a decentralized manner. All nodes can act both as a server and a client. Increasing the number of peers in a pure P2P network increases its efficiency.

Pure P2P networks show the higher level of support for natural properties among the evaluated conventional approaches. The peers can be regarded as autonomous nodes in the network, relying in local knowledge and being organized in a decentralized manner. A P2P network can provide asynchronous (at the network level), parallel and distributed computation while it can show a high-level of robustness. A P2P network, as the previous conventional approaches, is constituted by conventional synchronous hardware. As such, it can be programmed, using traditional programming languages. The nodes of a P2P network can correspond to SC systems, while they can interact through exchanging information and performing computation. The notion of scopes could be embedded in the communication (e.g. by the number of maximum hops). Thus, a P2P SC implementation would be viable. Care would be required designing the networking architecture following this approach as the numerous peers' communications in such a platform would probably pose a performance bottleneck.

Graphics Processing Units (GPUs)

Using GPUs for general-purpose computation has lately become a trend since they offer affordably significant gains in terms of speed for computationally intensive tasks [70]. Responsible for the speedup is their architecture that exploits applications parallelism. Originally GPUs targeted only image rendering operations, yet the revolutionary change was made when manufacturers made GPUs programmable and thus GPUs entered the computing mainstream [71]. Evident for their success is the fact that GPU design was adapted in order to improve their programmability and enhance their general-purpose computation capabilities [34]. General-purpose GPU (GPGPU) languages [70] were developed, reflecting the need of support for user-defined applications.

GPUs offer a great level of parallelism [70] at a (relatively) low cost. The vast parallel power given by the multiple stream processors of a GPU is a property highly sought by a SC implementation. In contrast with the previous approaches, they do not use the conventional von Neumann architecture. However, the use of a CPU is obligatory to provide centralized control. GPUs do not provide inherent support for other natural properties (except for a limited form of local knowledge, at the level of its internal parallel processing units). The development of GPGPUs provided flexibility to GPU users. Further advancements in GPU architectures and performance are certain, since the main use of GPUs lies in the gaming industry which is ever-more demanding. The great success of the general-purpose use of GPUs indicates that more advanced and optimized programming languages will be created, while more tools will become available in the future to ease application development. The first GPU SC implementation [34] is described in section 2.4.3. Its performance proves that GPUs can support the SC architecture efficiently. Scalability issues can be resolved by using the GPGPU functionality on a computer cluster. Thus, a future SC hardware implementation could exploit the great performance potential of a GPU cluster.

2.2.2 Unconventional Hardware Approaches

Nature has lately been the source for inspiration for designers since natural systems, while being extremely complex, simply work. Usually they show high levels of stability while remarkable properties like self-organisation, self-replication and fault-tolerance are inherent to them. The next subsections describe emergent and promising technologies, which do not follow conventional approaches and broadly-used paradigms, and usually draw inspiration from nature.

Ubiquitous Computing

Ubiquitous computing [72] aims at a different human-computer interaction paradigm than the one of the desktop user. Numerous interconnected devices (pocket-size tabs and page-size pads) while providing various services appear effectively invisible to the user. Waiser uses the term “embodied virtuality” [72] to describe the idea of computing ubiquity. Pervasive computing [73] is another term similar to ubiquitous computing. Traditional input devices, wireless mobile devices and smart devices form the pervasive computing model that aims to build sensitive and adaptive digital environments. An example would be a wireless health monitor, like the one presented in [74], which could communicate the health status of a patient on-line with a hospital server that can detect abnormalities.

Ambient Intelligence (AmI) [75] extends at ubiquitous and pervasive computing and takes under consideration intelligent systems, context awareness and objects interactions to build human-responsive environments that facilitate everyday life. An indicative example would be the smart house. It is notable that AmI initially attracted criticism [76] since its anticipatory and adaptive nature raised societal and cultural concerns.

Ubiquitous computing can be implemented by emergent technologies as Speckled Computing introduced in 2004 by Arvind and Wong [77]. Specks are semiconductor grains which are connected wirelessly to form a vast parallel sensing and processing network (Specknet). Numerous specks can be sprayed on any surface to convert them to computational resources. The prototype in [77] consisted of programmable specks over Zigbee radio.

Although more objects get interconnected nowadays, moving the Internet of Things [78] closer to reality, further progress needs to be done in order for practical implementations to be incorporated to everyday life. As Shadbolt concludes in [75], numerous independent electronic devices form an ubiquitous clutter in the majority of living rooms, which is far from the disappearance of computers in the background.

The Ubiquitous Computing paradigm is compatible with natural properties as asynchrony, parallelism and should be able to provide distributed, continuous and embodied computation. Ambient Intelligence and speckled computing should provide systems that show self-organising, autonomous and homoeostatic behaviour. Ubiquitous computing is an emergent field of research with great potential [72]. However, a practical SC implementation could not be based on it since the technology is not yet mature and basic practical requirements like programmability and design-friendliness are not satisfied.

Wireless Sensor Networks

Wireless sensor networks (WSNs) [79] are an outcome of advances in wireless networking, micro-fabrication and integration. They comprise numerous sensor nodes which are heavily resource-constrained since they are usually required to function for long terms on a finite on-board battery. Typically, sensor nodes, commonly referred as motes, operate autonomously and are equipped with a low-end microprocessor and limited amount of memory for local processing. Communication bandwidth is also usually limited. Network abstractions have to be designed in order to reduce power consumption and improve performance. Limited support is provided for software development.

Initially, WSN research was military based. This led WSNs to be defined as large-scale, ad-hoc, multihop networks of tiny, fixed-location (after initial placement), homogeneous motes [80]. This definition changed with civilian WSNs applications (environmental and species monitoring, agriculture, production, delivery and healthcare [80] – a more specific collection of applications like vital sign monitoring, power monitoring and rescue of avalanche victims among others can also be found in [80]). Mobile and heterogeneous motes can form WSNs as well. The classification of a given WSN can vastly vary depending on its specific application. This is shown by the number of different network topologies (star, ring, bus, tree, fully connected, mesh), communications protocols, routing mechanisms, power management schemes, network structures and multiple developed standards [81].

Sensors for various measurands [81] (pressure, temperature, humidity and position to name a few) have been developed. The sensing elements can either be fixed on a mote or able to be replaced by others (of the same or different type). The anatomy of a commercial WSN node is illustrated in Figure 2.2 (taken from [82]). Compatibility among sensors (of various types and different manufacturers) and the rest infrastructure on a mote, along with communication interfaces to network those devices, is ensured by the IEEE 1451 Family of Standards [83].

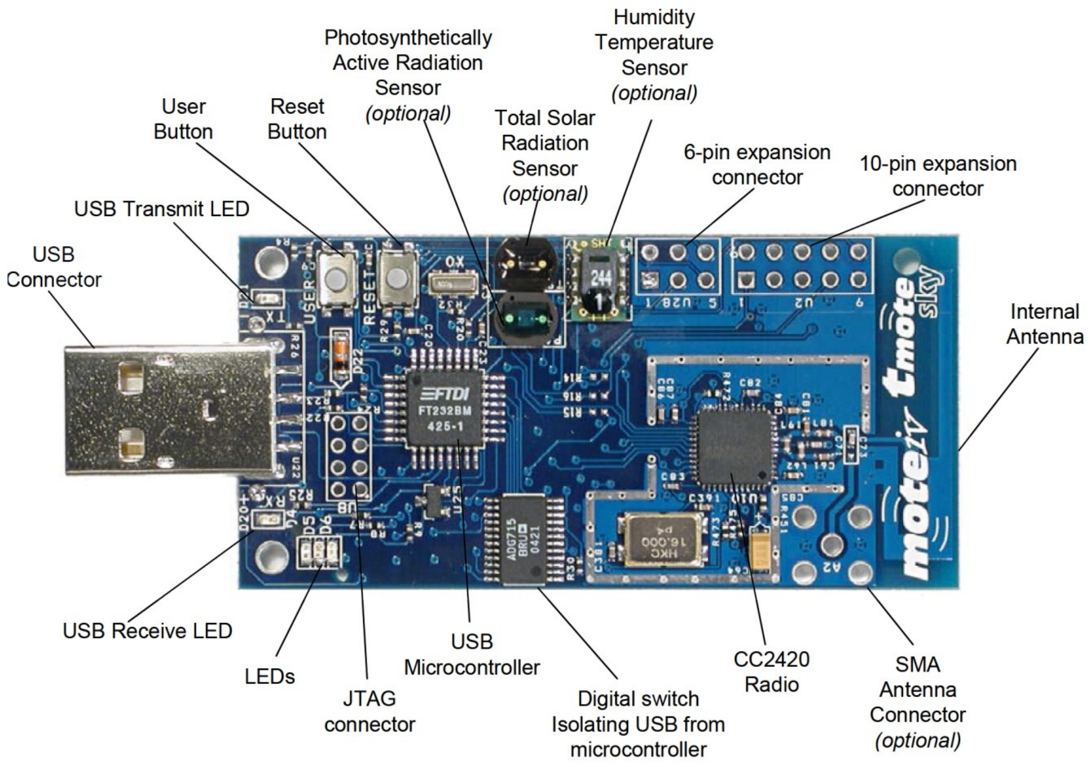

Photosynthetically Active Radiation Sensor *(optional)*
Humidity Temperature Sensor *(optional)*
User Button
Reset Button
Total Solar Radiation Sensor *(optional)*
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector *(optional)*

Figure 2.2. Anatomy of a WSN node. From [82]

WSNs, comprise spatially distributed can provide an autonomous, parallel, distributed and asynchronous (to some extent) form of computation. They can be responsive to the environment and extract information from it through their sensing elements. The network itself defines a system of nodes, each with some limited processing power performance (since they are heavily resource-constrained), yet combined they can form a powerful, asynchronous (at the system level), distributed and highly parallel computing machine

[84]. The system can be easily programmed, since a microprocessor is always part of the node configuration. Groups of autonomous nodes can show a level of self-organization. Using the wireless link, the network can be easily expanded, while using the inputs of the embedded sensors, it can show homoeostatic behaviour. These statements reveal the compatibility of WSNs with the requirements of a systemic computer architecture.

The idea of WSNs as a possible hardware implementation platform for SC was introduced in [84]. The author suggests that motes can be treated as systems, while their resources can be treated as subsystems. Sensor inputs can provide environmental feedback, which can be used either to evolve the systemic structure or as a fitness function in a genetic algorithm [12], which is used to adapt the architecture in case of damage or unforeseeable changes and to sustain functionality and optimize performance. Systems, in the form of binary data, would be exchanged between motes, while the network would dynamically be expanded or shrunk as new motes join it or fail. As motes usually run some lightweight operating system, extensions to the existing communication protocols, probably layered over the underlying communication stack, would need to be designed in order to accommodate the systemic functionality. Some of the tasks to be considered are the maintenance of the scope tables, systems interaction within a mote, the mutual system exchange protocol between motes and supported transformation function set [84].

It is concluded from the above that a WSN SC implementation would be viable. It was shown that fault-tolerance could be accomplished with the aid of sensor input feedback. Self-organization can also be accomplished, subject to cleverly written middleware communication layers. Decentralized and leaderless computation is highly compatible with the SC paradigm. The wireless link provides some scalability. Thus, WSNs are strong candidates as a SC implementation platform in the future.

Field-Programmable Gate Arrays (FPGAs)

Although combining the high performance of a hardware implementation with the flexibility of a circuit that can be programmable may have been conceived as early as 1967 [85], the idea was commercialized and patented [86] around two decades later by Freeman, co-founder of Xilinx. FPGAs are reconfigurable integrated circuits. Generally, hardware description languages, such as VHDL and Verilog, are used to provide the source code which is then translated to a binary bitstream through specialized software, which in turn is downloaded to the FPGA and programs it (enabling, disabling and configuring accordingly its reprogrammable components) to behave as the target circuit.

As opposed to Application-Specific Integrated Circuits (ASICs), which are fixed-function circuits tailored for definite operations, they provide more flexibility, shorter time to market and lower costs (when accounting for fabrication costs) and sometimes power consumption. Lately, the semiconductor evolution and the advantages mentioned above lead system designers to prefer FPGAs on an increasing amount of commercial products.

The inner structure of modern FPGAs is similar for different vendors. They consist of a great number of programmable logic cells and a reconfigurable interconnect network. Commonly, logic cells include a Look-Up-Table (LUT) that can implement any logic function (subject to the number of inputs of the LUT – typically 4 or 6), some memory elements (a number of flip-flops) and some simple logic (a full adder and carry propagation logic). The design is usually hierarchical, with a number of logic cells forming logic blocks[8]. A set of modern FPGAs, called Platform FPGAs, also provide other functional blocks, like multipliers, blocks for digital signal processing (DSPs) and big chunks of RAM memory to optimize designs. Some high-end models even include embedded processors, high-speed communication interfaces and/or simple analog features. Special Input/Output cells (I/O pads) are used at the chip boundaries.

The versatile nature of FPGA-based systems led to their use in a plethora of fields. A collection of applications for FPGAs is given in [87] and includes among others: multi-mode implementations, various algorithms implementations (especially ones that can exploit the provided fine-grain parallelism), multi-FPGA systems, mathematics applications (as modular multiplication), physics applications (as real-time recognition in high-energy physics), genetic optimization algorithms and genetic database searches, stereo matching for stereo vision and Laplace equation solvers. A digital neuron model for evolving spiking neural networks is presented in [88]. One of the applications with great potential is logic emulation [87]. It provides considerable acceleration compared to software simulation, lowering the time and cost of custom chip (ASIC) prototyping. A complete and functional implementation[9] of a circuit can be available in seconds, once the design has been adapted to be mapped on the FPGA.

FPGAs can also be added to standard computer systems as attached processing units, coprocessors or even internal processing units, in the form of add-on cards, on-board or

[8] The naming varies among different vendors: logic cells are called Configurable Logic Blocks (CLBs for Xilinx) or Logic Array Blocks (LABs for Altera)

[9] Performance validation and timing constraints cannot be assessed using logic emulation

on-chip respectively. Add-on cards are used in the NetFPGA project [89] that enables researchers to build high-performance networking systems in hardware. Extensive work has also been done in the field of neural networks. As each of their basic elements needs to be configured for a given problem [87], FPGAs provide an optimal implementation platform. Modern FPGA families allow part of the circuit to be reconfigured during normal operation (Dynamically Reconfigurable or Run-Time Reconfiguration) which gave birth to Evolvable Hardware, described in the next subsection.

FPGAs can exploit fine-grain and coarse-grain parallelism because of their adaptive nature [90]. The reconfigurability of the hardware liberates the designer to implement new architectures, optimized for specific applications. This flexibility has shown that properties as fault-tolerance [91], self-replication and self-repair [92] can be accomplished on FPGAs. Asynchronous circuits have also been successfully simulated on FPGAs [91]. Therefore, considering that FPGAs is a mature technology and that they provide an intermediate trade-off between flexibility and performance, pose a strong candidacy for a SC hardware implementation. Again, a cluster of FPGAs, probably utilizing a crossbar [90] or a systolic chain [93] connecting the FPGAs, would be a viable solution to accommodate any size of SC programs, thus design expandability could be accomplished. The implementation could either comprise a systemic processor, that would be able to run systemic programs, or following a totally different design approach, a different circuit could be downloaded on the FPGA, according to the systemic program, which would be highly optimized for the specific program. The latter approach would require a SC-to-HDL translator program (a high-level SC synthesis tool) to be written.

FPGAs are unique in the sense that they combine the flexibility of software on a hardware medium, since they can be reconfigured and implement a different custom circuit every time. A number of natural properties, mentioned above, can be implemented using this feature. They can provide a medium for parallel and distributed computation, while they can also implement sequential logic. The ability to self-reconfigure is very important since it can be used to provide circuits that are adaptive and robust. Various tools and standard design methodologies exist for FPGA-based design. Thus, it is apparent that FPGAs are highly suitable for a SC hardware implementation.

Evolvable/Evolved Hardware

Evolvable hardware is defined in [94] as "a scientific field that integrates evolutionary computation [12] and reconfigurable hardware devices" while similarly in the context of a recent comprehensive review of the field [95], it is defined as "the design or application of evolutionary and bio-inspired algorithms for the specific purpose of creating physical devices and novel or optimised physical designs" [95]. Evolvable hardware devices reconfigure themselves dynamically in an autonomous manner by interacting with their environment, without human intervention, to sustain functionality and increase performance. Two lines of research are identified in [94] on the subject, the first involves self-reproduction and self-repair of existing circuits while the second utilizes genetic algorithms [12] for autonomous reconfiguration leading to altered circuits. Some indicative applications are human-competitive analog design, Micro-Electro-Mechanical System (MEMS) fine-tuning and evolvable antennas for space missions [94]. Hardware evolution has been applied to digital, analog and mechanical systems resulting sometimes in human-competitive designs [94].

A central notion on evolutionary computation is a genetic algorithm (GA) [12]. A GA is a search technique which tries to find a solution to a problem (exploring a search space) in an incremental way. There is no need for a priori knowledge about the problem. The process involves the preparation of a pool of candidate solutions (chromosomes), the definition of an evaluation (or fitness) function and the search process. A solution is selected to continue to the next evolution stage depending on its comparison with the output of the evaluation function. During this process, an evolution cycle, giving a new generation, is executed iteratively until some termination criteria are met[10]. Solutions can be evaluated by simulation (extrinsic evolution) or by physical realization (intrinsic evolution). Each cycle involves generating a new chromosome, evaluating it according to the fitness function and selecting the chromosomes to form the next generation (usually the ones with higher fitness function, for example, roulette wheel selection). Typical methods of generating new chromosomes, further explained later in section 5.1.2, are selective reproduction (genetic material from each parent create an offspring), crossover (exchange of genetic material between chromosomes) and mutation (a bit, or group of successive bits, is randomly chosen and flipped). The evolution process described above, applied in the field of evolvable hardware, is illustrated in Figure 2.3 [96].

[10] A fitness threshold value is reached or a loop count limit is reached

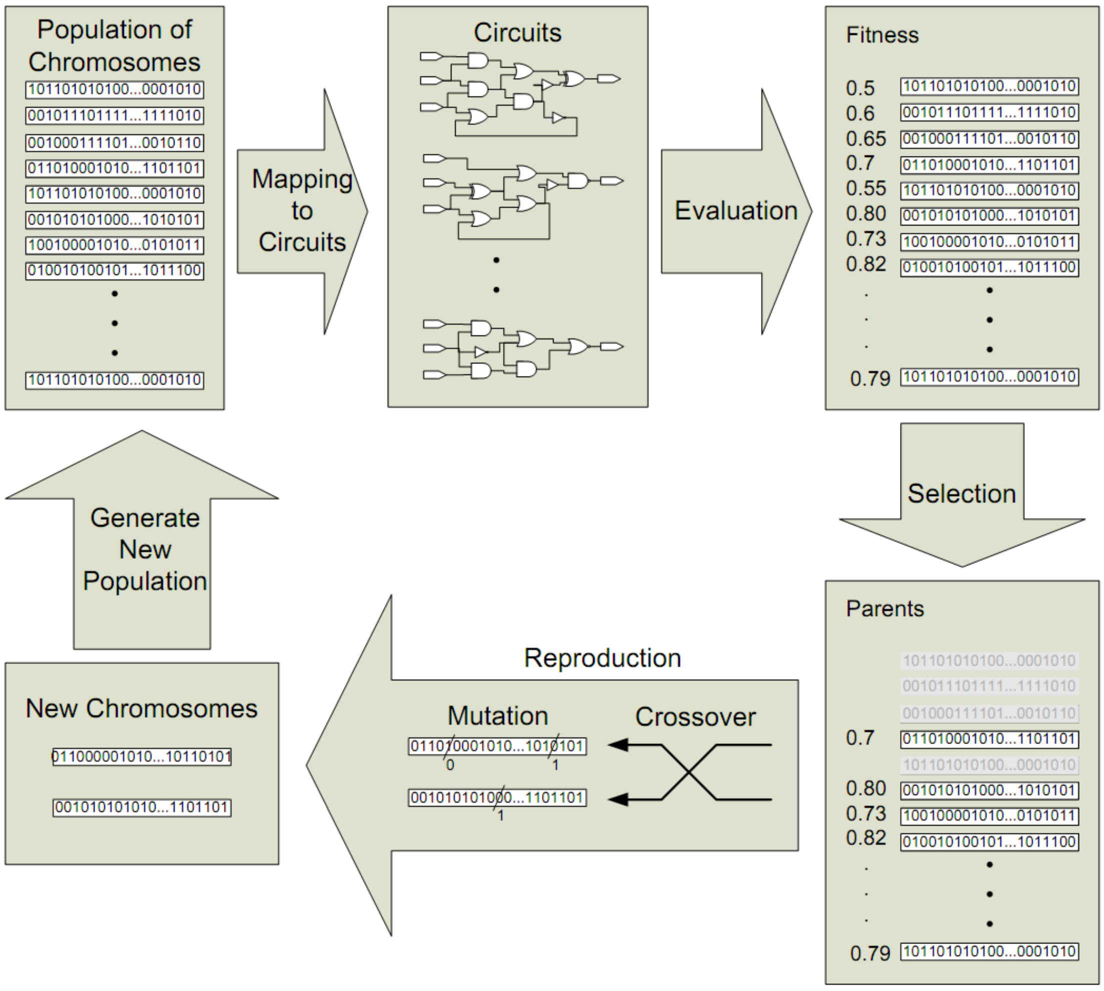


Figure 2.3. Hardware Evolution using a Genetic Algorithm. Reproduced with permission from [96].

For evolvable hardware, the bits in the configuration bitstream of an FPGA are regarded as the chromosomes for GAs. If the fitness function is defined to map the behaviour of the target circuit, then the GA, by continuously downloading altered configurations to the FPGA, will ultimately produce a design that will match in some degree the required functionality. A collection of research work on the field of evolvable hardware using GAs is given in [94] and includes among others: a myo-electric hand control chip, simple arithmetic circuits capable of built-in self-test, a clock-timing adjusting technique and an evolvable image filter.

Under the evolvable circuits category, apart from the GA-based designs, [94] provides a collection of bio-inspired projects that target fault-tolerant, self-replicating and self-repairing evolvable circuits like the Embryonics project [92], a multi-cellular universal Turing machine [97] and one of its applications, the BioWatch [98], defining a cellular and molecular architecture of a giant artificial organism. The Embryonics project drew inspiration from Ontogenesis, which is one of the three axes of bio-inspiration [99], discussed in the next section in the context of the POEtic/PERPLEXUS projects, which combined all three axes to provide circuitry designed to develop and adapt its functionality through evolution, growth and learning [92].

An important difference is noted in [100] between evolved and evolvable circuits. An evolved circuit is the outcome of continuous refinement, by using evolutionary methods, but the architecture remains static once a satisfactory solution-design is identified. Evolvable systems, on the contrary, can dynamically and autonomously be self-reconfigured possibly throughout their existence [100]. They should be able to adapt their structure according to environment changes, thus they are more tolerant to faults and failures and more probable to optimize their performance according to these changes.

Evolvable hardware shows natural properties such as fault tolerance, self-repair and self-replication. It provides autonomous circuits that can potentially be parallel and provide distributed computation. Therefore, evolvable systems would be a potential SC hardware implementation platform. However, the definition of a representative fitness function would not be trivial for such a complex design, using the GA approach.

POEtic/PERPLEXUS Projects

The three major axes of bio-inspiration, in analogy to nature, are Phylogenesis, Ontogenesis and Epigenesis according to the POE model [99] of bio-inspired computing. The phylogenetic axis involves the evolution of the species through time based on alterations of the genetic code. The ontogenetic axis refers to the development (or growth) of a single multi-cellular organism. This is accomplished through cellular division (a mother cell, or zygote, divides, the resulting cells divide as well and the process continues – each new cell contains a copy of the whole genetic material, or the genome) and cellular differentiation (new cells acquire different functionality depending on surroundings). Cells are continuously destroyed and generated in an organism. Self-healing is based on this property [101]. The epigenetic axis involves the learning processes during the lifetime of an individual organism and allows it to increase in complexity as it grows.

The “Reconfigurable POEtic tissue” project [101] (or POEtic) targeted all three POE axes. The goal of the project was the development of a multi-cellular, self-contained, flexible and physical computational substrate, inspired by the evolutionary, developmental and learning phases in biological systems, designed to interact with its dynamic environment, develop, adapt its functionality and self-repair [94], [101]. The POEtic tissue was designed as a structure consisted of three layers [101] and it is illustrated in Figure 2.4 (the layers are represented here next to each other):

- The Genotype Layer: Corresponds to the phylogenetic model processes. Each cell contains the genome of the tissue. It consists of a set of operators, which defines all the functions a cell can execute, and a differentiation table, which is used to determine which operators each cell will use.
- The Mapping Layer: Corresponds to the ontogenetic model, implementing cellular differentiation and growth. Self-repair functionality is also involved in the layer. The selection of the operators to be used occurs in this layer as well.
- The Phenotype Layer: Corresponds to the epigenetic model, modifying the operation of the organism during its operation. It consists of an execution unit, a set of application-specific resources, and a communication unit to handle the connectivity of the cells.

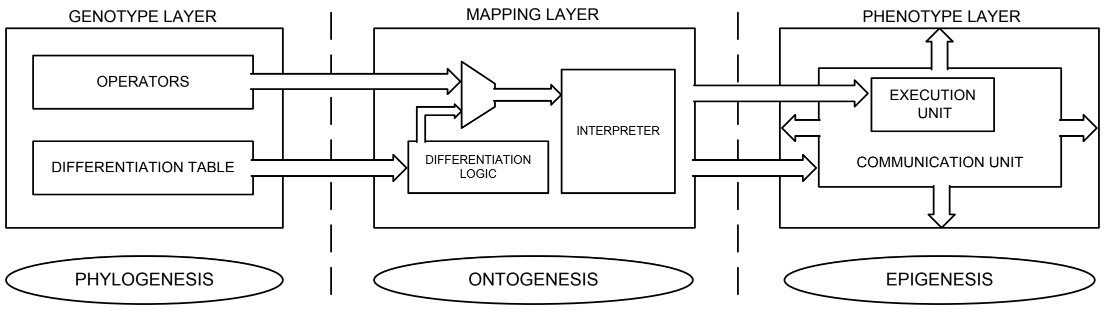


Figure 2.4. The three organizational layers of the POEtic tissue. Based on [101]

Upon a given problem, the user can chose the required layers to be implemented. Cells are implemented on a molecular substrate (programmable logic) to provide adaptability. The chosen architecture is compatible with the three axes of biological organization [101] and includes an input/output interface that permits each cell to modify its environment.

A practical POEtic system architecture is described by [102]. The tissue is divided in three main components illustrated in Figure 2.5 [103] :

- The environment subsystem, which manages the interactions with the environment (using sensors and actuators) and implements processes related to the phylogenetic axis. A microprocessor, which provides centralized control at the organism level and executes evolutionary algorithms, is part of this subsystem.

- The organic subsystem which manages the behavioural operation and learning methods of the tissue by determining how ontogenetic and epigenetic processes are physically realized. It consists of two layers: a 2-dimensional array of basic programmable elements, the molecules, which can be configured to 8 different modes of operation and enable various functionalities, and a dynamic routing algorithm implementation for the creation of connection paths between molecules.
- The system interface, which provides the communication channel between the two subsystems and mechanisms (interface bus, one active “master” environment subsystem for multichip configurations, automatic coordinate propagation) that permit the tissue to be scalable without constraining the number of POEtic chips that can be employed. From a user perspective, a multi-chip POEtic tissue has got one environment and one organic subsystem.

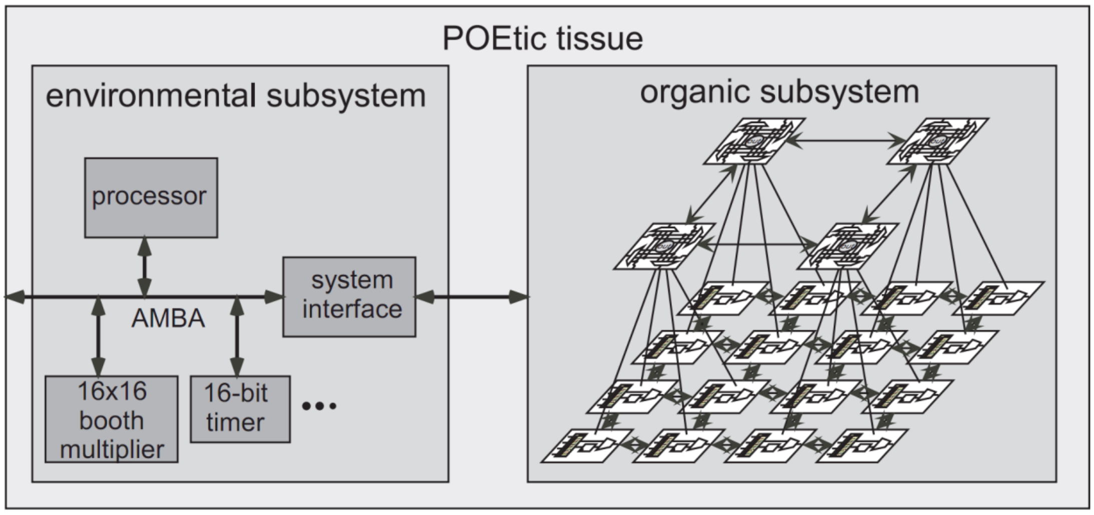


Figure 2.5. Overview of the POEtic tissue architecture. Reproduced with permission from [103].

Moreno *et al.* [102] demonstrated that real-time emulation of large-scale spiking neural network models can be accomplished using the aforementioned design. Other applications of the POEtic tissue include self-repairing hardware [104] (utilizing the dynamic routing mechanisms of the environment subsystem), circuits that show fault-tolerance [105] (in the form of error detection and recovery through dynamic routing, reconfiguration and on-chip reprogramming), [106] (using hardware Gene Regulatory

Networks) and an interactive artistic installation, called the POEtic-Cubes [107] (autonomous robots controlled by POEtic chips).

The successor to POEtic was the PERPLEXUS project [108]. The goal of PERPLEXUS was "to develop a scalable hardware platform made of custom bio-inspired reconfigurable devices that will enable the simulation of large-scale complex systems and the study of emergent complex behaviours in a virtually unbounded wireless network of computing modules" [109]. At the heart of these ubiquitous computing modules, ubidules, is a custom-designed reconfigurable chip, the ubichip [108], capable of implementing bio-inspired mechanisms such as growth, learning and evolution. The ubidule can be customized to use a set of peripherals (such as USB, SD card, Wi-Fi), to satisfy the requirements of a given application, as modularity was a key design consideration. The overall architecture is illustrated in Figure 2.6. The project targeted, but was not bounded, to three applications: neurobiological modelling, culture dissemination modelling and cooperative collective robots.

The limitations of the POEtic architecture were identified [103] and improved [108] in the PERPLEXUS framework:

- The POEtic dynamic routing algorithm required long-distance combinatorial links. The new algorithm better exploited existing paths, used an 8-neighborhood approach to reduce congestion risk and allowed path destruction, allowing unused connections removal.
- Further scalability: the wireless link combined with the Address Event Representation (AER) scheme [111], which involves encoding/decoding a sequence of events to/from a sequence of addresses to overcome communication issues, caused by massively interconnected components, provides virtually unbounded scalability.
- The partial self-reconfiguration in the POEtic chips allowed partial replication of the circuit while they needed to be pre-programmed (preconfigured configuration paths and reconfiguration units loaded by the microprocessor). PERPLEXUS allows real self-replication employing the THESEUS mechanism [112], through self-inspection (recovering the configuration bitstream, the genome, from the replicator) and built-in reconfiguration-aiding units.
- Neural networks friendliness: The structure of the reconfigurable cells, called Macrocells, in the ubichip, was defined around four 4-LUTs which could be

configured as any four 4-input function or as a 4-bit ALU. The ALU, which was provided with a neural-oriented instruction set, allowed the implementation of basic neural processing elements and could be scaled to form a neural SIMD multiprocessor.

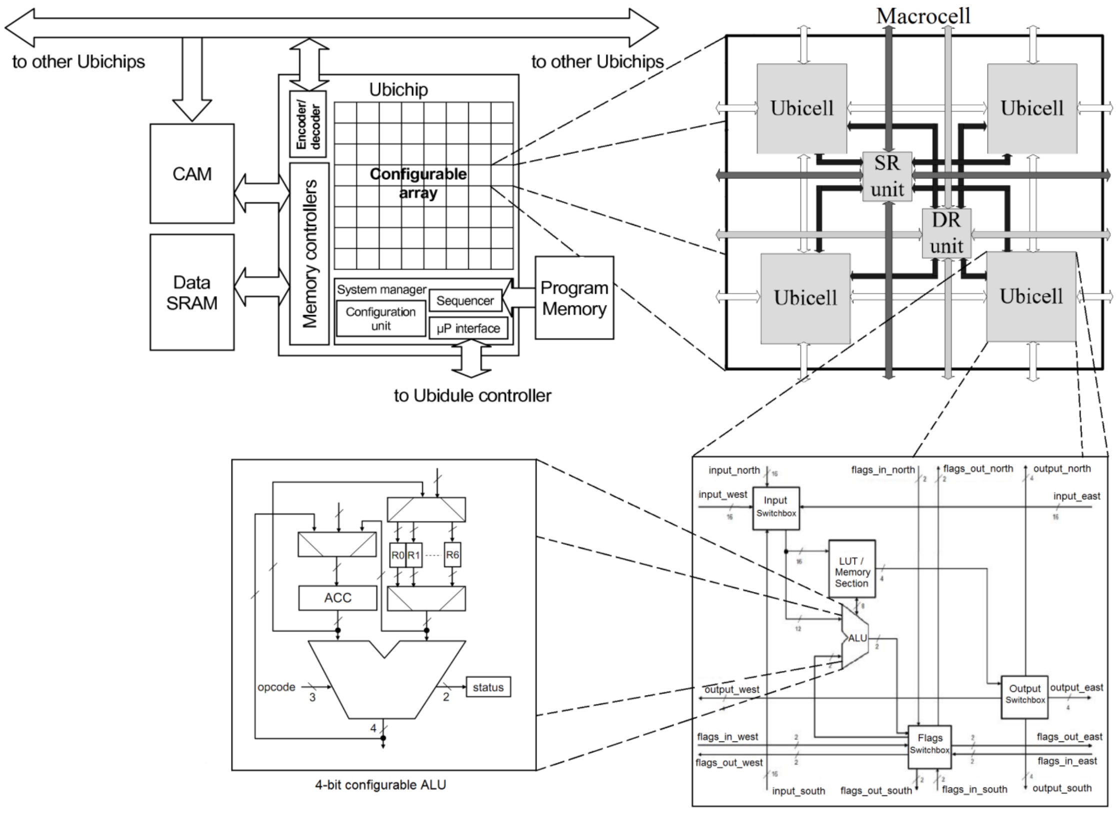


Figure 2.6. Organisation of the Ubichip architecture. Each ubichip contains an array of reconfigurable cells called Macrocells. Each Macrocell consists of a pair of self-replication (SR) and dynamic routing (DR) units associated with four ubicells. The ubicells are composed of three switchboxes (for input, output and flag signals) establishing configurable communication paths with their neighbours and a dedicated LUT/Memory section for each 4-bit configurable ALU. Reproduced with permission based on [108][110].

The POEtic and PERPLEXUS projects were collaborative attempts on implementing hardware that can mimic natural properties on all three bio-inspirations axes. They provide the most complete solutions in terms of circuits that embody a lot of the natural properties of Table 1.1. They can provide vastly parallel autonomous systems, which can be self-organised and tolerant to faults. Their architecture is distributed and partially decentralized, as the cells show self-configuration abilities, yet a microprocessor is used to provide control at the system level. They presented a refined, scalable and bio-friendly solution. The architecture defines an array of reconfigurable blocks which may be used individually as fine-grain logic functions or collectively as a parallel SIMD machine.

Thus, the Ubichip would be a suitable platform for a SC hardware implementation. However, the PERPLEXUS project was not continued after the introduction of the architecture, so access to the final ASIC chip, including 100 Macrocells [110], would be limited. An alternative solution would be to implement the design on an FPGA in order to further take advantage of the additional design flexibility. Taking into consideration that an array of 4 Macrocells required the equivalent of 25K logic cells [110][157] only around 30 Macrocells would fit in a mid-range modern FPGA device[11]. Assuming that each Macrocell would represent ideally 4 systems (mapping one system per Ubicell), we would get less than 100 systems if we accounted for the additional requirements of the SC architecture (scopes and matching functionality). This would imply that we would need a network of FPGA devices to prototype any practical application following this approach, increasing the cost of our research project. In addition, the ideal SC hardware architecture would have to compete in terms of performance with alternatives approaches, e.g. a modern GPU-based system or a high-end conventional CPU. Time-multiplexing has been used in [102] to enhance the performance of the architecture while emulating in real-time a 10000-neuron spiking neural network but this resembles the way conventional CPUs implement parallelism. Nevertheless, the compatibility of the Ubichip with the SC paradigm is evident.

Organic/Autonomic Computing Paradigm

Organic Computing (OC) is a research field which explores the feasibility of controlled emergence [113]. The objective of OC is the technical usage of principles observed in natural systems. Organic systems are independent, flexible, adaptive and autonomous while they show natural properties like self-organization, self-configuration, self-healing, self-protection, context-awareness and self-explanation (in order to inspect the results of self-organization). Organic systems follow the observer/controller paradigm, which observes the functional system and the environment and controls the parameters of the functional system according to the observations, while a guard system prevents illegal actions.

A collection of promising ongoing research projects on OC can be found at [114]. An indicative project is “Digital on-demand Computing Organism (DodOrg): Stability and Robustness” which is overviewed in the next section.

[11] Assuming 75% utilization in the mid-range Xilinx Virtex-6 LX240T FPGA device with 240K equivalent logic cells [161]

It is noted that Autonomic Computing (AC) [115], which was introduced by IBM initially targeting IT systems, shares the same requirements and objectives with OC. The two terms are used both in conjunction (AC/OC) and interchangeably.

AC shows a high level of compatibility with SC, in terms of the natural properties the two paradigms target. AC only provides design aims by describing a vision. SC also has a corresponding architecture. AC research projects target software and hardware implementations. A SC hardware implementation could possibly draw inspiration from designs provided by AC/OC projects if they shared the same implementation platform.

Computing with Unconventional Materials

Almost any electronic circuit nowadays is silicon-based. Researchers lately identified the need to find its successor. As every broadly-used technology in the past (relays, valves, transistors [11]), it will reach its limitations and will eventually need to be replaced. Their research focuses on computation implementations on new physical substrates, exploiting computational properties of various physical, chemical and biological media. It comes under the broader field of non-classical, unconventional computation [3].

Computing based on unconventional material and methods shows great potential for future developments. The majority of the approaches, described below, show promising results and usually provide great performance gains. As most of them are either nature-inspired or nature-based, they show inherent natural properties, so they can provide massively parallel, distributed, autonomous and asynchronous computation. However, they have a limited, if any, set of specialized applications and show several limitations (for instance in flexibility, programmability and availability) when a practical hardware implementation is concerned.

Thus, a SC hardware implementation based on an unconventional medium would not be a viable approach (in the context of this thesis). It has to be noted that, since this section sums up the current research trends on alternative media, it is possible that at least one of those will become conventional in the future.

DNA or Molecular Computing

DNA computing [20] involves data encoded as biomodules, such as DNA strands, and uses molecular biology tools to imitate operations on those data. The structure of the genetic material provides vast data-parallelism, thus problems that can be adapted to this method can be efficiently solved. As mentioned in section 2.1, Adelman was the first to solve an NP-complete [116] problem in the lab [20], by using DNA molecules and biomolecular techniques to manipulate DNA. Based on this experiment, it was concluded that any problem in NP (set of problems that can be verified in polynomial time) could be efficiently solved with DNA computing [11]. A collection of DNA computing applications (like graph coloring, protein conformation, matrix multiplication and cryptography) is given in [11].

Quantum Computing

Atoms and molecules do not follow classical mechanics laws. Quantum physics explains these non-classical behaviours of atomic-scale objects. Information representation in quantum computers [11], [21], [23] is in the form of quantum bits, or qubits, in analogy with bits in conventional computers. A qubit can hold any superposition of the two classical states, 0 and 1. Thus, a set of n ordered qubits (a length-n quantum register) can hold information equivalent with any superposition of 2^n quantum states. Measurements and manipulations alter the contents of a qubit and can be modelled as matrix multiplications. Quantum gates are used for qubits manipulation, which translates to quantum state transformations. Each type of gate implements a basic quantum algorithm. Quantum computers are able to provide tremendous speed-up in solving problems compared to their classical counterparts. Typical quantum applications are cryptography, database search and combinatorial optimization problems [23]. Various methods have been used for practical quantum computer implementations [23]: superconductors, liquid-state nuclear magnetic resonance techniques and ion-traps to name a few, with the latest practical designs reaching the capacity of 512-qubits [117].

Chemical Computing

Dittrich [118] defines chemical computing as computing with real molecules (real chemical computing), as well as programming electronic devices using principles taken from chemistry (chemical computing metaphor). Following this definition, molecular computing is entwined with chemical computing. Along with molecule-based approaches, this field includes computation achieved with chemical mediums like light-sensitive chemical waves [119] (applied to image processing with the possibility of realizing associative memories), a fluorescein dye [120] (capable of performing a full scale of elementary addition and subtraction operations) and protein molecules which are able to perform a variety of logical or computational operations [121]. The chemical computing metaphor has inspired new architectures [118], such as computers based on reaction-diffusion media [122]. Reaction-diffusion computers are regarded as massively parallel devices, where tiny portions of the chemical media act as elementary processors and information is stored and manipulated by means of local disturbances of concentrations. A set of logic gates and simple combinatorial logic based on chemical compounds is presented in [123].

Bacterial Computing

A data storage and retrieval method, based on sequence alignment of the DNA of living organisms, was introduced in [124]. Building upon that, the DNA computing paradigm was extended in bacteria, to give birth to bacterial computing [125]. Bacteria can be genetically programmed to execute various operations, forming bacterial computers, which can be autonomous, responsive and self-reproducing [125]. The highly parallel nature of this approach (each bacterium is a basic processing unit) allowed the solution [125] of a Hamiltonian Path Problem [36], similar to the one solved by Adleman using DNA computing. In vivo computing is a similar research field [23] with studies on the computational capabilities of gene assembly in unicellular organisms.

Other Unconventional Media

A set of other computation media are reported in the literature. Collision-based computing involves mobile self-localizations, travelling in space and executing computation when colliding to each other [126]. An example implementation, introduced in [126], uses fusion gates as collision points which were inspired by the above-mentioned reaction-diffusion paradigm. In [127], a non-conventional paradigm is introduced, where the logic values are carried by independent stochastic noise processes (electronic noises) implying greatly reduced energy consumption. In [128], the authors use computer controlled evolution to manipulate liquid crystals to evolve logic gates. Other unconventional materials for implementations with computational purposes proposed in the literature include molten metals and soft solids [129], carbon nano-tubes and carbon nano-wires [130].

SC based on Unconventional Media

The implementation approaches which are based on unconventional material are more compatible to natural properties than any other. The reason is really simple. The implementation media that they use are natural. The disadvantage with these approaches is that their underlying technologies are not mature. There are no design methodologies, supporting tools and generic input/output interfaces yet. They would require specialized knowledge from fields usually away from computer science and would entail access to a modern scientific lab. This in turn would imply a more limited user space and an elevated cost of development. Thus, while all of unconventional material approaches seem greatly promising, they could not be considered for a practical SC implementation at the moment.

2.2.3 Other silicon-based designs

Silicon is arguably the most widely-used substrate for designs with computational purposes. While current research attempts to identify other promising materials with superior physical and chemical characteristics (an example that currently attracts increased interest would be the carbon allotrope graphene [131]), suitable for integrated circuits implementation, (processed) silicon is still the preferred material due to its tolerance to high temperatures and electrical powers.

Using silicon as their base substrate, a vast number of relevant research papers attempt to break conventional design patterns and, using various approaches, try to incorporate natural properties. In this section, an indicative set of them was chosen to be overviewed in order to designate relevant design techniques, from which inspiration can be drawn provided that a silicon-based approach will be selected for the SC hardware implementation.

SpiNNaker

The SpiNNaker Massively Parallel Computing System [132], [133] was mainly designed for neural networks modelling. It will consist of a vast number of processing cores (scheduled to exceed 1 million, distributed across 57600 chips with 18 cores each), arranged in independently functional and identical power-aware ARM-based chip multiprocessors to achieve parallel, robust and distributed computing [134]. Each core is self-sufficient in terms of storage (it has a local "Tightly-Coupled Memory" (TCM) [132]), while there is a shared off-chip memory, among the cores – connected to them through a DMA controller with the help of an asynchronous Network-on-Chip (NoC), in the CMP level. The off-chip memory is virtually local to each processor since it is segmented into discrete regions and each processor has exclusive access to one region, a specific address range, only. The organisation of each 18-core (16 application cores, 1 monitor and 1 spare) SpiNNaker CMP chip is shown in Figure 2.7 [133].

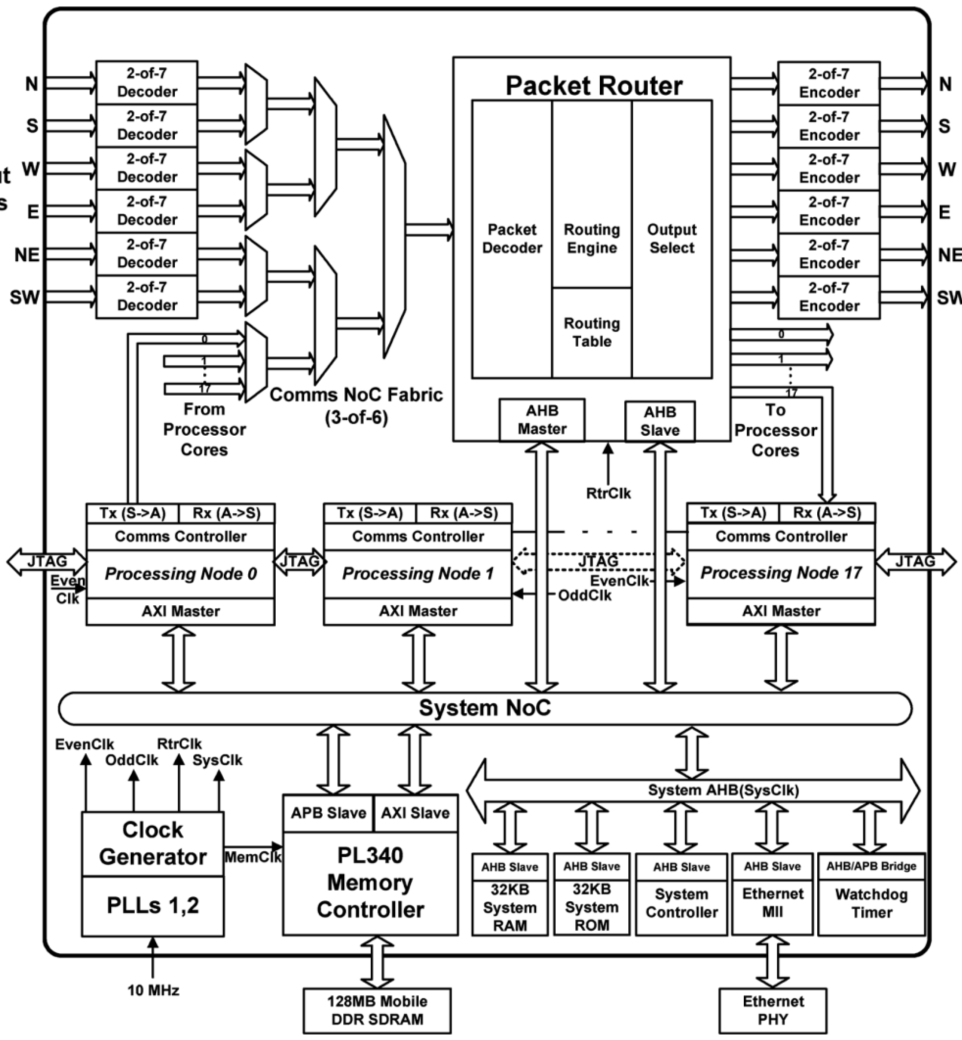


Figure 2.7. SpiNNaker CMP chip organisation. Reproduced with permission from [133].

The system was provided with sufficient hardware resources redundancy, thus the processing and communication infrastructure can show a high-level of fault tolerance. A configurable asynchronous packet-switching routing network, based on a custom designed on-chip multicast router, was used to support the high degree of interconnection at the chip and system levels. Communication between processors was based on Address-Event Representation [111] (as in the PERPLEXUS project). Generating an interrupt, which is issued to the processor when it receives a new packet, allows different clock domains for each processor eliminating the need of synchronization, thus making the system virtually asynchronous (Globally Asynchronous Locally Synchronous - GALS). The system can be reconfigured on the communications side, by changing the routing table of the on-chip router, and on the processing side, by changing the running code (altering the data part of the TCM). Its configuration is made through an on-chip Ethernet link by a Host system (a personal computer) while board-to-board communication is realized with high-speed serial (3Gbps SATA) connections with their communication interfaces implemented on Spartan-6 FPGAs [135].

The SpiNNaker project envisions a library-based development system which allows the high-level description of a model and uses an automated design flow to create silicon implementations, which are predesigned custom chips. This approach is interesting from a SC point of view. The idea could not be directly mapped on a SC hardware implementation, but a SC language (similar to those introduced in [24], [136]) could be used in an automated design flow to create highly-optimized hardware SC implementations dynamically on reconfigurable media.

The SpiNNaker architecture defines is a high-performance, low-power application-specific platform optimized for neuroscience applications [133]. Essentially being a massively parallel computing machine made from conventional CPUs, SpiNNaker addresses mainly communication and power consumption challenges. As such, the architecture may be suitable for exploration of unconventional computing paradigms that require raw parallelism, thus making the platform a good candidate of a hardware SC implementation. While the underlying architecture of the building nodes of this power-aware "computer cluster in a box" would not be compatible with much of the required natural properties of SC, regarding SpiNNaker as a whole might be useful in modelling processes with asynchronous processing (yet locally synchronous) elements interacting in a parallel fashion. However, the SpiNNaker platform is still on a development phase, with prototypes gradually increasing the number of available cores an order at a time[12] as part of the ongoing Biologically Inspired Massively Parallel Architecture (BIMPA) research project[13]. Thus, the completed architecture may be a suitable candidate for a future SC implementation, especially if its benefits could be combined with the added flexibility provided by reconfigurable hardware to better map the underlying architectural features of SC.

[12] The project defines 10N milestone machine designations (where 10N stands for approximately 10^N supported cores). 101, 102 and 103 machines have been sampled where 104, 105 and finally the 106 machine are yet to be implemented.

[13] A scalable custom 64-FPGA machine, Bluehive [248], targeting also Neural Network Simulation was developed under the BIMPA project, as an FPGA-based alternative architecture to be used for evaluating the spiNNaker platform

Molen

Molen [137] is a reconfigurable processor, following the tightly coupled co-processor paradigm. It features a general-purpose fixed processor core (GPP) enhanced by user-defined commands executed on reconfigurable hardware. Molen addresses issues like opcode space explosion, modularity and limitations on the number of parameters for operations mapped on FPGA [137]. It identifies blocks of software code that can be efficiently mapped on reconfigurable hardware and replaces them with their hardware equivalent executed on reconfigurable media. This is accomplished by the use of special microcode (termed ρμ-microcode), which differentiates from traditional microcode, since instead of being executed on fixed hardware, it is executed on custom hardware that itself designs to operate on [137]. The reconfigurable co-processor, which is consisted of the ρμ-microcode unit and the custom computing unit (CCU), is configured by the general-purpose core. Therefore, it can be tailored to a different application each time.

Molen exploits GPP-FPGA co-execution. It embeds application-specific functionality without altering the GPP architecture. The architecture is essentially based on a conventional CPU with the ability to off-load computation to the reconfigurable fabric of an FPGA. While the nature of the sequential part of the design would be unsuitable to perform SC background tasks in a parallel fashion (further explained in section 3.2.3), the ability to enable user-defined hardware-supported instructions would be quite useful (and is in fact suggested in section 3.1.3). Another interesting feature in this design is the micro-programmable nature of the CCU reconfiguration that increases flexibility and allows automation.

DodOrg

DodOrg (Digital On-demand Computing Organism) [138] is a bio-inspired self-organizing architecture, which exploits parallel heterogeneous systems. It is an adaptive system which is bound to natural self-x [138] properties (like self-adapting, self-healing and self-configuring). DodOrg is organized in three levels: the cell, the organ and the brain.

At the cell level, organic processing cells (OPCs) with various resources (microprocessor, DSP core, FPGA, FPFA[14]), announce their suitability (based on monitoring system metrics like performance, network load and energy consumption) for processing tasks. At the organ level, virtual organs are created using "organic" middleware, which implements decentralized closed control loops, in order to accomplish decentralized and fault-tolerant task distribution. Organs are formed by a number of neighbour cells with cooperating tasks, which exchange accelerator and suppressor messages to handle task execution (this technique implements a variation of the observer/controller paradigm). At the brain level, a software architecture uses input and feedback from the environment to implement the targeted application, which is a real-time control system for robot-based manufacturing. This hierarchy can be further extended to groups of organisms (self-organizing robot swarm) [138], forming dynamic societies.

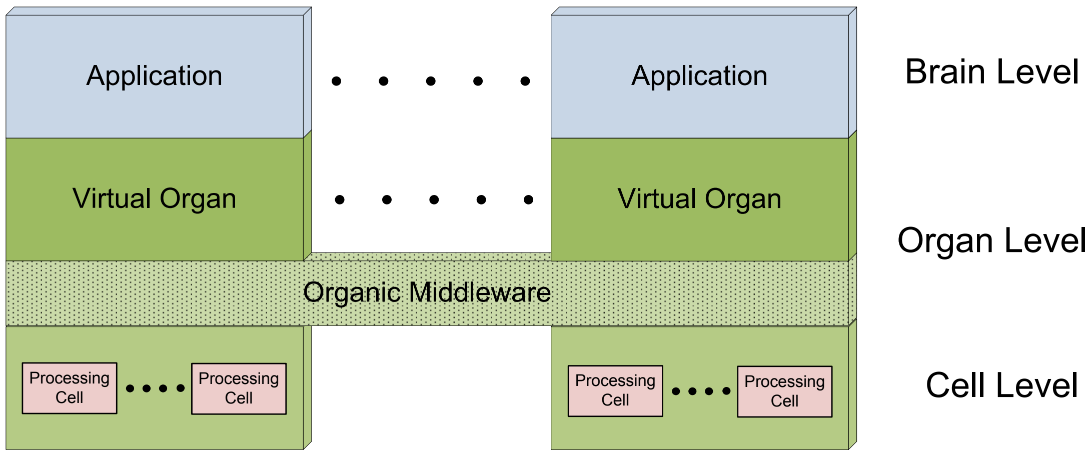


Figure 2.8. Organic System Architecture. Suggested in [138]

DodOrg is an indicative example of an organic computing hardware implementation. It is interesting, from a SC viewpoint, since the two paradigms, as stated earlier in the OC section, share very similar aims and target nearly the same fundamental natural properties. The similarities extend also in the hierarchical approach DodOrg adopts to organize its control system, which are compatible with the systems hierarchy in SC.

The project defines the organization of an architecture supporting many bio-inspired properties. While this layered approach (see Figure 2.8) is compatible with the SC paradigm, the project focuses more on an organic control robot and specifically on robot-

[14] Field-Programmable-Function-Arrays. Introduced as part of the Chameleon [246] System-on-Chip, FPFAs are word-level reconfigurable datapaths consisting of multiple processor cores. Each core includes 5 custom ALUs.

based manufacturing. Decentralized hardware components are communicating over the organic middleware and the individual autonomous robots can form a self-organising robot swarm. Evidently, this specific level of abstraction is not suitable for a system modelling low-level natural processes, as required by SC. In terms of organisation, DodOrg moves towards the software domain as it scales up (middleware at the organ level, software at the organism level). However, this approach may not be as distant from the final SC implementation, since some high-level tasks (systems on the highest hierarchy levels that realize advanced instructions) may need to be in the software domain, in order to increase flexibility and programmability.

IBM Cell processor

The IBM Cell processor (or Cell Broadband Engine – Cell BE) [139] is a single-chip multiprocessor based design which aims at high performance by exploiting parallelism at all levels of the system: data-level (SIMD support), instruction-level, thread-level, memory-level and compute-transfer-level. Workload is offloaded from the main processor (PowerPC architecture), which mainly handles control tasks, onto the (eight) Synergistic Processor Elements (SPEs – dual-issue in-order SIMD cores), thus the system is heterogeneous. The SPE architecture focuses on data processing (wide datapaths, more and wider registers, single use privilege level). The SPEs interconnect network consists of four data ring buses, thus multiple concurrent transfers can be handled. Computation and data transfer operations are executed concurrently, while concurrent memory accesses from different cores are allowed to exploit memory-level parallelism. The Cell BE is widely known for being used in a games console, yet it has also been used in HDTVs, home servers, game servers and even, as a building element, in supercomputers [140].

Larabee and the Intel MIC

The Intel Many Integrated Core (MIC) architecture [141] uses multiple in-order (program execution stalls until the operands of an instruction are available) x86 CPU cores extending previous work during the Larrabbee [142] project. The choice of in-order CPUs is justified by the fact that one of the main design considerations was to achieve a great level of parallelism[15]. It uses a bi-directional ring network to handle inter-chip communication between the various cores. Scalability is accomplished with

[15] Out-of-order architectures have improved performance since they explore instruction parallelism but their die utilization factor is higher than their performance factor (1.5x-1.7x on performance corresponds to 2x-3x on size [142]). Thus, those architectures are better suited for single-stream performance aware designs.

multiple short-linked rings. Routing is simplified by following a simple convention: a message is accepted by an agent (logic block connected to the ring network) from one direction on odd clocks and from the other direction on even clocks.

Larrabbee initially targeted visual computing, essentially being a hi-end GPU with extended programmability features, since it adopted a familiar programming model (with some alterations) based on the traditional x86 architecture. While Larrabbee never became a commercial product, its derivative, the MIC architecture, targets high-performance computing and promises great gains for highly parallel applications, largely reusing existing parallel code. The first MIC PCI-Express prototype board featured 32 in-order Aubrey Isle CPUs while its next revision, branded as the Xeon Phi, offers up to 61 cores with 244 threads, 256-bit vector units supporting 512 SIMD-instructions, on a single chip [143].

SARC

The Scalable computer ARChitecture (SARC) project [144] is a research project with aim to develop a general-purpose scalable integrated architecture, explore design and compilers creation automation and develop new programming models compatible with future architectures. According to [145], the SARC architecture will be a multi-node heterogeneous architecture, very similar to the Cell BE. The main difference is that SARC will consist of multiple cores and, instead of identical SPEs, application hardware accelerators, each of which can be optimized for a different application.

SyNAPSE

SyNAPSE [146] is the acronym for Systems of Neuromorphic Adaptive Plastic Scalable Electronics. SyNAPSE is a research project that aims to "investigate innovative approaches that enable revolutionary advances in neuromorphic electronic devices that are scalable to biological levels" [146]. It identifies the limitations of traditional approaches to computation and seeks to break the programmable machine paradigms by using neuromorphic [147] devices, which are based on adaptive analog circuitry principles. The final deliverable of the project is a multi-chip neural system of $\sim 10^8$ neurons and instantiate it into a robotic platform, which then should be an autonomous entity and show indications of abilities like perception, cognition and response [147].

CPU-GPU Hybrids

The advantages and disadvantages of CPUs and GPUs are outlined in section 2.2.1. While graphics applications became more intensive, communication between the two components was provided with more bandwidth and lower latency (AGP to PCI-E connections). Current design trends for consumer applications involve the integration of CPUs and GPUs on a single chip. AMD recently presented the AMD Fusion architecture, calling the CPU-GPU hybrid Accelerated Processing Unit (APU) [148].

This can be an important design, since if APUs (and later Intel-based hybrids), become the conventional architecture of the near future, they will have native on-chip parallelism, becoming more compatible with a vast number of applications, including SC.

2.2.4 Hardware Approaches Summary

The explosive growth of technology in the last century enabled the conception, design and fabrication of what we consider today conventional computer architectures. However, from the very early stages of this revolution, pioneers in the field realised that there is more than one ways to approach the definition and implementation of computation. This is evident by the late work of one of the architects of the conventional computer architecture, von Neumann, who after devising the sequential and centralized design [8] which (with various optimizations and enhancements) became the basis for virtually every contemporary computing device, started exploring the potential and relation of biology to computation (and specifically between the computer and the human brain [149]). Similarly one of the designers of the hugely successful ARM processor - Steve Furber - now leads SpiNNaker [133].

While the conventional approaches have addressed the constantly increasing computational needs for commercial, research and even more specialized purposes, with designs and architectures also evolving and getting optimized and tailored to adapt to these changing demands, they eventually reached their limitations, resulting in new approaches and computational concepts being emerged. This section discussed how conventional approaches attempted to provide more computational power and how unconventional approaches, using nature as both inspiration and alternative implementation substrate attempt to address natural features as parallel, decentralized and distributed computation to name a few.

GPUs, chip-multiprocessors and supercomputers provide parallelism with different levels of granularity, from the chip level to the cluster level while peer-to-peer networks come closer to the natural computing paradigm providing a decentralized network of cooperating nodes. Ubiquitous computing and wireless sensor networks define parallel and distributed systems of usually self-contained and adaptive interconnected devices, with the ability to show self-organization and fault tolerance. FPGAs combining the great flexibility, from the ability of being reprogrammed, with the performance, provided by the fine-grained parallelism on the hardware level, became a useful tool for numerous projects overviewed above to implement evolutionary and bio-inspired designs. Alternative materials (molecules, chemical compounds, bacteria) show great potential in a computational context but, still being at a proof-of-concept stage, are not ready yet for a broad range of practical applications making silicon the norm when it comes to digital circuitry.

The various approaches and paradigms of this chapter are presented in a Systemic Computation context, taking into consideration that SC was designed to incorporate the various natural properties in a more complete way. The next sections give more insight in the SC paradigm and its three implementations prior to this work.

2.3 Systemic Computation

Systemic computation is designed to be a model of natural behaviour and, at the same time, a model of computation. This approach was based on the generally accepted, but still intuitive notion that natural systems are able to perform some form of computation [24]. It is a computational model with characteristics similar to biological systems and processes.

The link between biology and computer science under the SC prism can be found in the last convention of SC (*computation is transformation* - section 1.2), enabling us to identify a common denominator between them [22]. In SC, everything is regarded as a system. This implies the notion of the inherent hierarchy in nature and enables SC analysis in different levels of abstraction. Also, SC is designed to operate using any system, meaning that, provided that the interaction pattern is the same, systems of different levels of abstraction can perform the same calculation. Systems can never be destroyed, reflecting the fundamental principle of conservation of energy (first law of thermodynamics [150]). As a result, systemic computations imply metabolism and ecology, since new systems need to be transformed and unwanted computation remnants need to be removed, meaning that the "waste" of one program will have to be recycled as "food" for another [24].

The interaction of two systems can be described by the systems themselves and a third "contextual" system which denotes how/if the interacting systems are transformed after their interaction [24]. The scope here, as in nature, is an important factor. The scope of a system defines the neighbourhood (which can be other than spatial) in which the system can interact with other systems in a certain way. SC attempts to capture the characteristics of natural scopes by enabling partial or fuzzy memberships and scope alteration after system interaction.

In order to represent a system in a modern computer, the choice of a binary format is compulsory. For the first systemic implementation [24], Bentley used binary strings to describe systems. Other descriptions [24] (π-calculus, bigraphs, brane calculus, Petri nets, calculus of looping sequences and other emergent technologies [22], [24] like speckled computing, DNA computing, membrane computing) were also considered but they could not provide practical implementation platforms compatible with traditional digital resources.

Bentley [24] used the notions of schemata and transformation function to describe interacting systems and the way the systems are transformed through interaction. Thus, each system comprises of three parts, two schemata and one function (see Figure 2.9), also called a triplet. Both schemata may change after an interaction, which implies circular causality (each system may affect the other). The model may support interactions among more than two systems, since an n-ary interaction may be reduced to n-1 binary interactions [24].

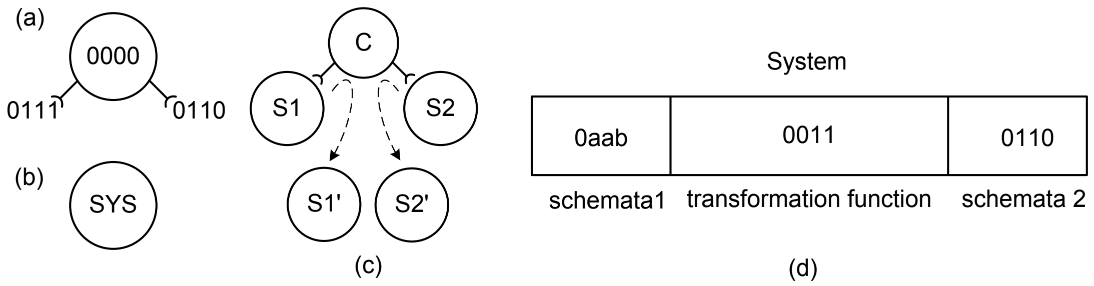


Figure 2.9. SC notation and systems representation: (a) a data system revealing its binary contents; its transformation function is zero (b) alternative notation for a data system called SYS (c) Systems S1 and S2 interact according to the function of the context C; the notation may optionally include the resulting systems S1' and S2' (d) The 3 elements of a system.
Reproduced with permission from [24]

A system in SC is represented as illustrated in Figure 2.9. The two interacting systems (schemata 1 and 2) are positioned in the receptors and set the possibility (through matching against the schemata of other systems) of the system to interact with them in its context. The transformation function determines the outcome of the interaction. Data systems do not define an interaction, thus their transformation function is always zero. The key notion of interaction here differentiates SC from conventional approaches since it is not a sequential operation, as a set of instructions executed in a conventional computer, but rather the sum of events that occur in a massively parallel and stochastic fashion – implied by the constant simultaneous transformations of systems.

A simple demonstration of the computation of the sum over a pool of data systems is given in [34]. Given a set of inert systems that can interact, but not act as context, with a transformation function which replaces data in the one of the systems with the sum of data of the two systems and zeroes the other system, and provided that enough time is available, only one system will remain containing the sum of all systems while all the rest will be zero. The operation is illustrated, using SC notations, similar to bigraphs, in Figure 2.10.

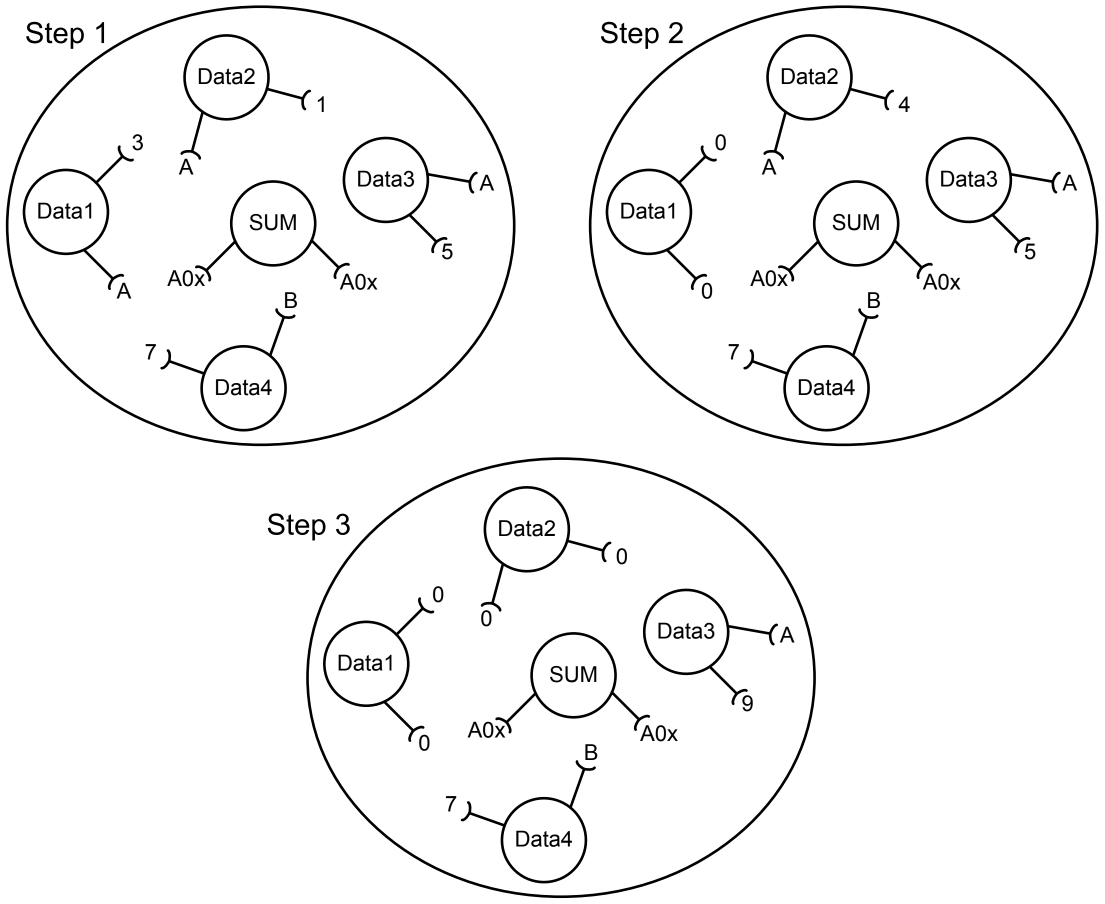


Figure 2.10. Illustration of a sum operation on a pool of data systems using SC notations. Based on [34]

In more detail, the only contextual system SUM (the only one with a non-zero transformation function) defines a way that other systems may interact. The definition of this interaction involves providing a valid transformation function (in this case addition) and also identifying two systems that will interact according to its schemata. In order to qualify as possible interacting systems, these systems will need to match the templates defined in the schemata of the context system. In this case (Figure 2.10), both schemata of the SUM context define a template requesting a data system (its transformation function should be zero - implied by the zero in the template A0x) of type A (its left schema should correspond to type A - implied by the A in the template A0x) while it can have any value on its right schema (denoted by the "don't care" x value in the template A0x). In the first interaction (at step 1 of Figure 2.10), two A systems (here systems *Data1* and *Data2*) will be chosen and one of them (*Data2*) will hold the sum while the other (*Data2*) will be reset (as shown at step 2). The resulting system (*Data2*) will interact with the third type A data system (*Data3*), the resulting sum will be again stored in one of them (*Data3*) and the other (*Data2*) will be reset. The type B data system (*Data4*) will never be part of an interaction in this example as it does not match any of the schemata of the context SUM (since it only defines interactions between data systems of type A).

The progression of a simple program which performs a nested parallel calculation is shown in Figure 2.11 (A-C) [24]. The program calculates the expression ((A1-A2)*(A3-A4)) and prints it. At first, the initial systems belong to scopes in different hierarchy levels. Next, the subtract-escape context systems "-e" transform the pairs (A1, A2) and (A3, A4) of data systems by means of subtraction (in their respective scopes c1 and c2) and change their scope one level higher in the hierarchy (effectively one of the interacting systems "escapes" from the scope it belongs to), leaving calculation "waste" in the initial scopes (c1 and c2), as no system can be destroyed. It is noted the (A1-A2) is correctly performed, (instead of A2-A1). This is accomplished by a mechanism called schemata matching, described in section 2.4.1, which identifies an appropriate interacting system to each interacting position. A1 is selected here as the first interacting system (the minuend) and A2 is selected as the second (the subtrahend).

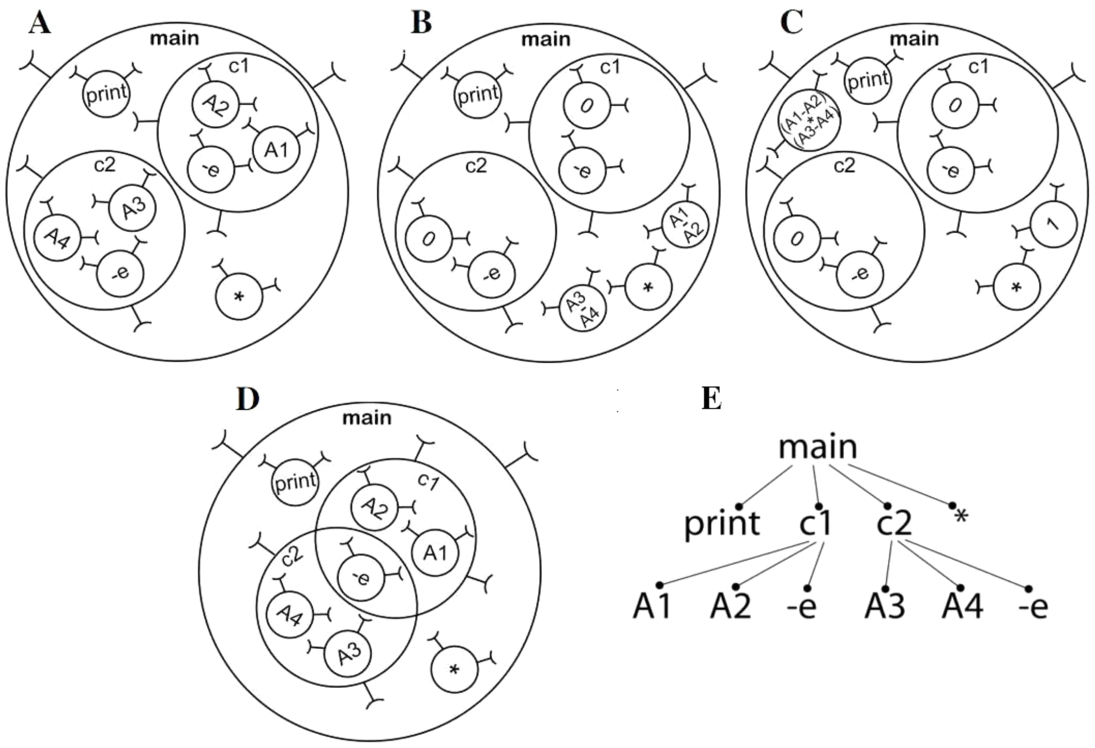


Figure 2.11. SC calculation of PRINT((A1-A2)*(A3-A4)). Reproduced with permission from [24], [34]

Eventually, systems are transformed by the multiply function. Overlapping scopes which share systems can be used for more compact representations of the same calculation (as shown in Figure 2.11D [24]). The parallel nature of SC dictates that all the systems interact continuously (function "print" will print the correct result upon completion of the program but it will also print intermediate results at earlier stages). Thus, the tree of scope memberships (Figure 2.11E) enables the correct calculation of complex expressions. An example of how overlapping scopes can be used to accomplish linear execution of such an expression is given in Figure 2.12 [24], with intermediate results escaping to their outer scope until the expression is fully evaluated.

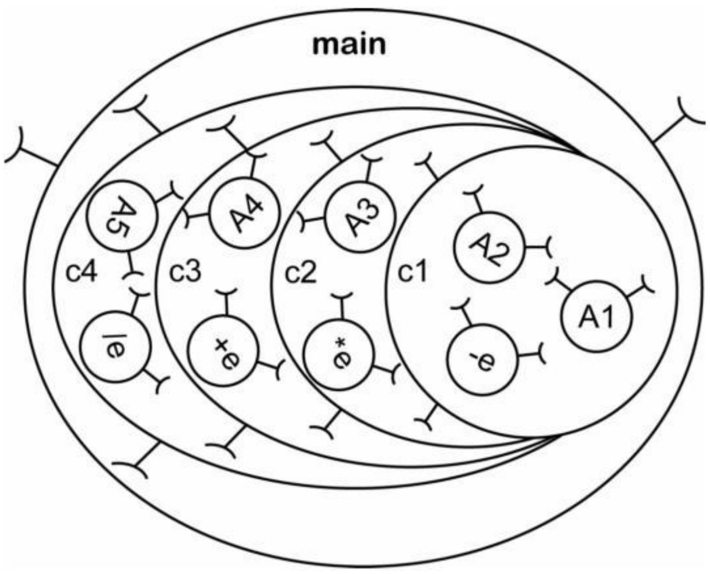


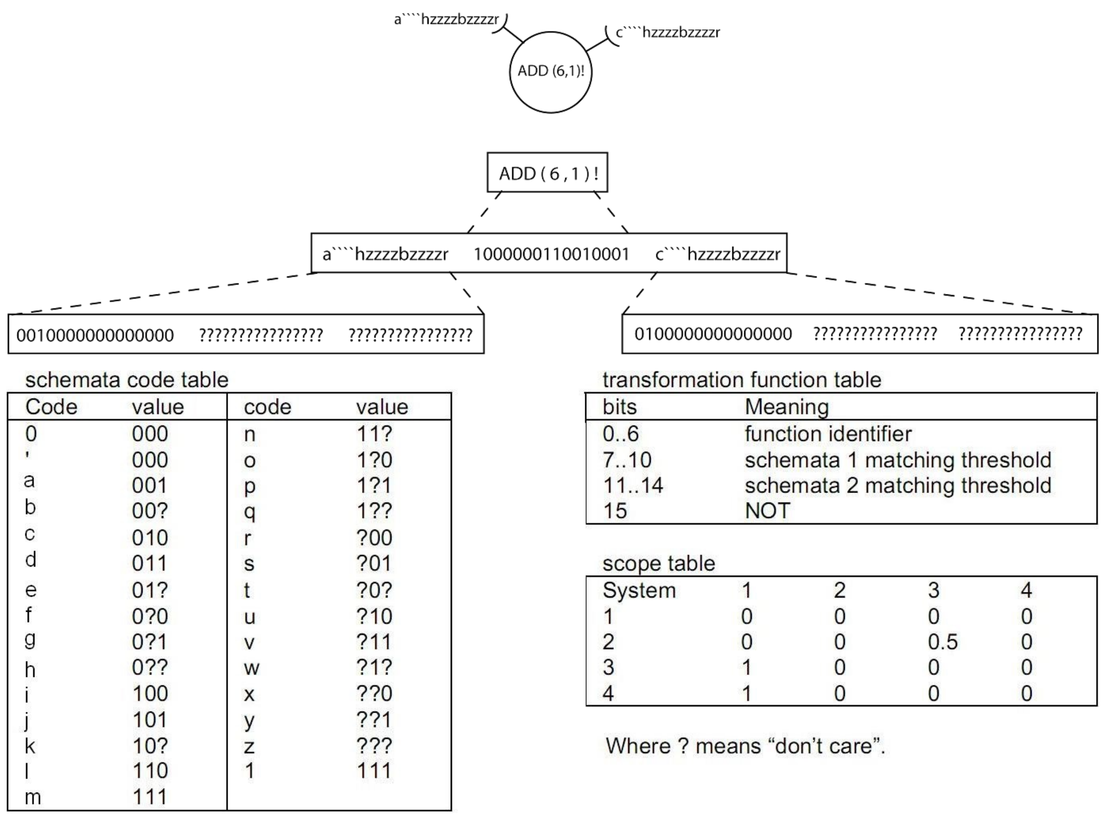
Figure 2.12. SC calculation of the linear expression ((((A1-A2)*A3)+A4)/A5). Reproduced with permission from [24]

2.4 Prior Systemic Computation Implementations

In [24], Bentley, along with introducing SC, provided a corresponding virtual computer architecture and its first (software) implementation. This attempt included a basic instruction set, an assembly language, a compiler and its resulting machine code. However this implementation was merely a simulation of a systemic computer, although it was a satisfactory proof-of-concept. To date, there are two more SC implementation attempts. The first provides a complete SC platform (language, compiler, virtual machine and visualization tools) [136]. However, it is also a SC simulation, although based on high-level language. The second [34] is yet another PC-based implementation, utilizing the inherent parallelism of graphics processors (GPUs) with considerable gains (of the order of one hundred) in terms of speed compared to previous attempts. The performance improvement is justified since this is the first implementation with a hardware constituent (GPU cores) and the first step towards a real systemic computer.

2.4.1 Original SC Implementation

The original implementation was a low-level simulation of a systemic computer, compatible with consumer processors. A more detailed description along with various SC applications can be found in [24]. The various features of the design are presented below.



schemata code table

Code	value	code	value
0	000	n	11?
'	000	o	1?0
a	001	p	1?1
b	00?	q	1??
c	010	r	?00
d	011	s	?01
e	01?	t	?0?
f	0?0	u	?10
g	0?1	v	?11
h	0??	w	?1?
i	100	x	??0
j	101	y	??1
k	10?	z	???
l	110	1	111
m	111		

transformation function table

bits	Meaning
0..6	function identifier
7..10	schemata 1 matching threshold
11..14	schemata 2 matching threshold
15	NOT

scope table

System	1	2	3	4
1	0	0	0	0
2	0	0	0.5	0
3	1	0	0	0
4	1	0	0	0

Figure 2.13. System representation, schemata decoding scheme and scope table of the original SC version. Reproduced with permission from [24]

As illustrated in Figure 2.13, characters ` to z where used to encode triplets of string systems. Partial matches were accomplished by enriching the binary {0,1} set with an wildcard (? – matching both a 0 and a 1), while the matching precision could be adjusted by using thresholds.

Table 2.2. Features of the original SC implementation

Feature	Original SC implementation
Word-length	16-character word length (systems consist of 48 characters)
Coding Method	characters of alphabet 29
Transformation Function Set	Thirty basic functions
Schemata Matching Method	Partial matching against thresholds
Interactions Order	Random (Biased – Prioritizes recently changed systems)
Scope Definition Method	Global Scope Table

Matching was based on the Hamming distance (number of different characters) between the schemata of the context and the systems. The transformation function, along with an identifier (analogous to the opcode of conventional architectures instructions), and the two matching thresholds (one for each system), also includes a NOT operator to set the matching polarity.

The membership of a system, set by the index of the respective column of the scope table, in the scope of another system, set by the index of the respective row, was set by the value at the corresponding position of the table (0 : not in scope, 1: fully in scope, 0 < value < 1 : partially in scope).

Figure 2.13 illustrates the graphical representation of a context system, with ADD as its transformation function. The schemata are decoded based on the decode table to match the schemata of other systems with thresholds of 6 and 1 for systems 1 and 2 respectively. If the schemata is matched, the addition is not executed since the polarity is negative (NOT is true). According to the scopetable, systems 3 and 4 are in the scope of system 1. System 2 is partially in system 3.

2.4.2 High-level SC Implementation

The extensive work of Le Martelot on SC [136], [151], [152], [153], [154], [155], [156] (which can be found collectively in [22]) provides outcomes in formalization, a complete platform, natural-inspired models implementation, analysis of native SC features and a description of the developed visualization tools. The implementation platform, called "Systemic Computation Platform and Environment" (SCoPE), includes a full definition for the SC programming language, a compiler and a virtual machine, the SC runtime environment and the visualization framework (see Figure 2.14) [22].

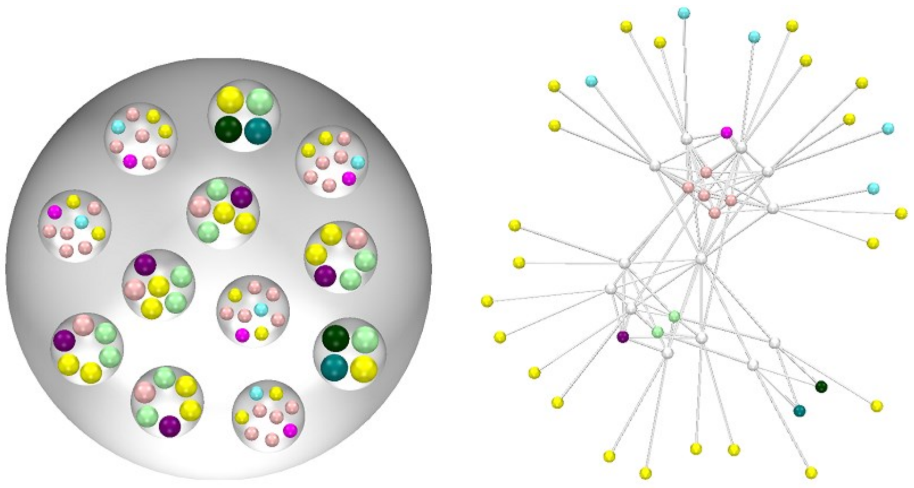


Figure 2.14. Visualisation of a SC model using SCoPE. Reproduced with permission from [22]

Some differences are identified in this implementation as opposed to the original one [22]: Recursive scopes, with a system containing itself, are supported. Fuzzy scopes are not supported, since they would add overhead in the implementation for a feature that was not critical and, thus, overlapping scopes are not supported either. Partial and threshold matching are not supported for the same reason. There is always a supersystem

– at the top of the hierarchy – called the universe. An active context can only change during the current interaction only in cases that this action will provide significant simulation gains. Also, unaltered interacting systems have a higher probability to interact next.

The main difference is that this implementation is higher level, fully programmable and more flexible than the original one. While the transformation function set, the string length and the alphabet are fixed in the original version, they all can be customized by the user for each model simulation in SCoPE. The flexibility is clearly reflected in the corresponding SC language which uses the original one as ground and expands its functionality and ease of use. Naturally, this flexibility comes in expense of execution speed. User-defined functions are implemented as C++ plugins and loaded as dynamic libraries at simulation initialisation and called for every function reference in the code. Also, the scopes are not held in a global table, but every system stores locally, along with its triplet, all the systems it contains and it is contained in.

2.4.3 GPU SC Implementation

The third SC implementation is GPU-based [34]. A GPU-based approach is completely justified since the fundamental property of SC, parallelism, is an inherent GPU architectural characteristic. While the first two implementations where just simulating SC, the third one is much closer to an actual SC architecture since there is now native hardware support. GPUs are well-suited for applications with numerous threads running in parallel over a set of shared data. Here, the shared data are the systems.

The GPU implementation follows the original SC model in terms of specification. Only implementation-specific minor differences (optimisation technicalities) differentiate them. The parts of the original algorithm that could be parallelized were identified and they are executed in the GPU cores, called devices, while the sequential parts are left to be executed in the CPU, called the host [34]. So, this is a hybrid approach which utilizes the advantages of both the sequential and the parallel domain.

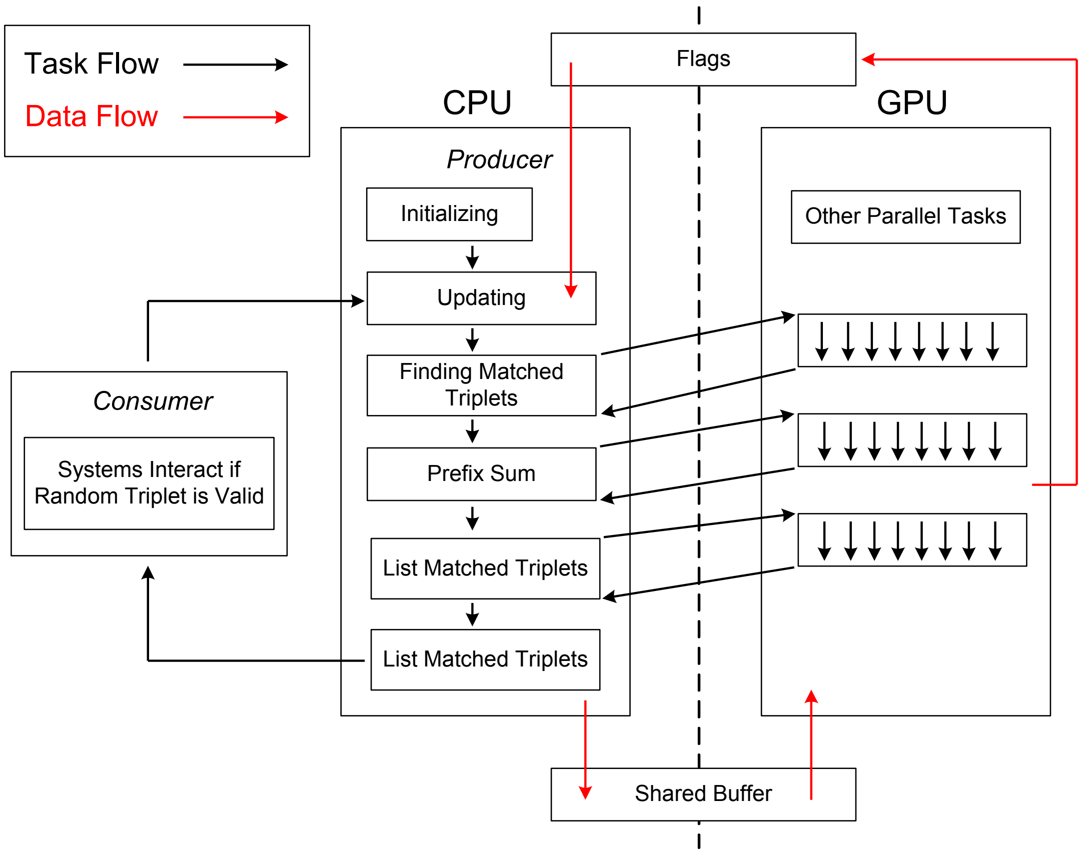


Figure 2.15. Overview of the task and data flows in GPU SC

The architecture developed in [34] is as follows: Two threads, which reflect the two main parts of SC, run in parallel in the host. These are called the producer and the consumer. The producer finds triplets with systems that match the schemata of a contextual system and belong to the same scope, called valid triplets. It consists of six successive steps which run sequentially on the host. Three of them are offloaded to the GPU. The consumer consumes the valid triplets, by executing the transformation function of the context, with the two interacting systems as the arguments. Valid triplets are chosen completely randomly (without prioritization). Triplet validity is rechecked before the interaction, since a previous transformation might have changed the systems scheduled to interact. An overview of the GPU SC implementation is illustrated in Figure 2.15.

2.5 Summary

This chapter provides a detailed discussion on various approaches to Natural and Systemic Computation. It summarizes software approaches and alternative computational paradigms and further critically focuses on conventional and unconventional hardware approaches on Natural Computation with an initial assessment of their compatibility with SC. Moreover, an overview of the SC architecture, as it was introduced in [24], is given and the work performed to date on SC is reviewed and assessed. This work involves three software implementations, a simple proof-of-concept sequential design with limited functionality and performance, a high-level fully-parameterizable sequential design with limited performance but extensive modelling capabilities and a hybrid design with increased performance, taking advantage of the vastly parallel computational ability of a GPU, but limited features.

It is highlighted throughout the chapter that widely-adopted computational paradigms and techniques are inherently incompatible with Natural Computation while mainly unconventional approaches are generally best suited to model nature in a more native way. Systemic Computation has been designed to be compatible with those differentiating properties that can be noticed in computation happening in nature, thus a SC implementation is expected to model those natural systems natively. Software implementations of such an unconventional paradigm, being sequential in nature, fail to properly map SC so they just simulate a systemic computer.

A custom hardware design, exploiting the freedom of tailoring its architecture away from conventional approaches, is expected to more closely match the underlying SC architectural properties. Thus, an investigation should be performed at first to determine the most appropriate hardware implementation platform for such a design to be realized on. Numerous alternative platforms, having been presented critically in a SC context above, can now be compared and indicate the most suitable among them for a practical SC implementation.

The ideal implementation platform should ideally be compatible with the natural properties of Table 1.1. However, there are some limitations, as many of the hardware approaches in Table 2.1 represent emergent fields of research and would not be suitable for a practical implementation. In order to identify the most appropriate among them, the suitability of each approach for a SC hardware implementation must be evaluated.

Therefore, the features incorporated in a practical SC hardware implementation should be identified. After examining the SC paradigm and its corresponding architecture and taking into consideration the hypothesis of this work and its research challenges, focusing on the utility and viability of a hardware system computer, it was concluded that these features, in the hardware domain, should be:

- Compatibility with as many as possible of the natural properties central to SC (research challenge *Chg1*).
- Compatibility with inherent architectural features of the SC (research challenge *Chg2*).

And addressing the practicality and efficiency of the implementation (research challenge *Chg3*):

- Efficiency of Input/Output Functionality: sufficient to result in a standalone platform.
- Programmability: an (at least basic) instruction set should be provided.
- Design friendliness: the implementation platform should be supported by standard design methodologies, tools and documentation to accelerate the design period, decrease error-proneness and enable efficient design verification.
- Technology Maturity: the implementation platform should be based on a mature technology in order to be able to provide a practical implementation. Furthermore, if a rich literature exists on designs based on the technology, inspiration can be derived from it while existing design methods can be improved to increase performance and efficiency.
- Scalability: the implementation platform should be able to be efficiently scaled to support modelling of large-scale natural systems.

Along with the hardware-related requirements, there are also some design considerations in the software domain:

- Compiler Support: a compiler should either be available or created to enhance programmability.
- Support for more advanced instructions/functions: in order to enhance flexibility and programmability.
- Backwards-compatibility with at least one of the earlier SC versions: this would allow reusability of functional code (including a compiler).

An ideal hardware implementation platform would satisfy all the above-mentioned requirements and considerations. However, as discussed in this chapter and summarized in Table 2.3, finding a platform that fully satisfies all of them is not realistic.

Table 2.3. Detailed evaluation of the reviewed hardware-based approaches against the implementation requirements implied by the research challenges. No dot represents the absence of support for the requirement, while three dots indicate full support

			Hardware Implementation Platforms																
			Chip Multi-Processor	High-availability Cluster	Beowulf Cluster	Load-Balancing Cluster	Grid/Cloud Computing	Pure Peer-to-Peer networks	GPUs	Ubiquitous Computing	Wireless Sensors	FPGAs	POEtic/PERPLEXUS	SpiNNaker	Evolvable Hardware	DNA Computing	Quantum Computing	Bacterial Computing	Chemical Computing
Implementation Platform Requirements	SC Natural Properties (*Chg1*)	Computational	Stochastic								•				•	•••	••	•••	•••
			Asynchronous		•	•	•	•	••		••	••	•	•	•		•••	••	•••	•••
			Parallel	•	••	••	••	••	••	••	••	••	•••	••	•••	•	•••	•••	•••	•••
			Distributed		••	••	••	••	••	•	••	••	••	••	••	•	•••	•	•••	•••
			Continuous						•		••	••					•••	•	•••	•••
			Approximate														•••	••	•••	•••
			Embodied								••						•••		•••	•••
			Circular causality														•••		•••	•••
			Local Knowledge				•	•	••	•	••	•	•	•	•	•	•••	•	•••	•••
		Behavioural	Self-organised				•		••		••	•	••	•	•	••	•••	•	•••	•••
			Fault tolerant		••				••		••	•	•	•	•	•	•••		•••	•••
			Open-ended														•		•	•
			Complex		••	••	••	••	••		••	••	••	••	••	••	•••	•	•••	•••
			Autonomous						•		••		•	•	•	••	••	•	••	••
			Homoeostatic						•		•	•	•				•	•	•	•
			Robust				•		••		••		•			••				
	SC Architecture Features (*Chg2*)		Systems	•	•	•	•	•	•	•••	••	•••	•••	••	••	•••	•	•	•	•
			Scopes	•	•	•	•	•	•	•••	••	•••	•••	••	••	•••	•	•	•	•
			Contexts	•	•	•	•	•	•	•••	••	•••	•••	•••	•••	•••	•	•	•	•
			Interactions	•	•	•	•	•	•	•••	••	•••	•••	•••	•••	•••	•	•	•	•
	SC Implementation Features (*Chg3*)		I/O Efficiency	•••	•••	•••	•••	•••	••	•••	••	•••	•••	•••	•••	•••				
			Programmability	•••	••	•	•	•	•••	••	•	••	•••	••	••	•••				
			Design-Friendliness	•••	•••	•••	•••	•••	•••	•••	•	•••	•••	•	••	•••				
			Technology Maturity	•••	•••	•••	•••	•••	•••	•••	•	•••	•••	••	•	••				
			Scalability	••	••	•••	•••	•••	•••	••	•••	••	••	•••	•••	••	•••	••	•••	••

According to Table 2.3, three entries are more suitable for a practical SC implementation than the others according to existing technologies, approaches and platforms. These are:

- Wireless Sensor Networks: WSNs appear to be one of the most suitable hardware implementation platforms. The variety of supported natural properties, the compatibility with the SC architecture, the maturity of the technology and the satisfactory scalability offered by the wireless link are the advantages. Their main disadvantage is the underlying fixed conventional architecture of their processor and their restriction in terms of resources and computational power.
- FPGAs: FPGAs have the unique advantage of external reconfiguration and self-reconfiguration. The limitation is really left to the designer to exploit all their potential to implement various natural properties. There is also the advantage of the big number of FPGA-based projects on the field of Natural Computation, where useful ideas can be adopted and extended. Scalability issues may be addressed using an FPGA cluster or with the addition of wireless connectivity on the FPGA board.
- GPUs: GPUs are among the most suitable implementation platforms without great support for natural properties since it has already been proven that they provide great performance. Additionally, their performance is certainly going to improve as new GPU models are released, since GPU development is driven by the games industry. However, the solution we have at present is a compromise that parallelises only some parts of the SC process. GPUs are dependent upon a CPU for centralized control.

The only implementation platform among the three that does not solely depend on the existence of a conventional von Neumann architecture CPU in the system is the FPGA platform. FPGAs are the only platform that provides the flexibility to design and implement a custom and dedicated hardware design from the very beginning until the system level, in order to highly optimise it for the selected application (in our case the Systemic Computation), and at the same time not compromise on performance. In addition, taking into consideration that FPGAs would be practical in terms of the number of systems they can support and also able to provide an easily-accessible standalone platform leads us to decide that:

The selected hardware platform for the first practical hardware-based implementation of systemic computation is the FPGA platform.

The next chapter further discusses the SC architecture properties and presents the first (FPGA-based) hardware architecture of Systemic Computation.

Chapter 3

Designing a Suitable Hardware Architecture for SC

This chapter focuses on the investigation of a hardware design to support the underlying architecture of SC (research challenge *Chg2*, section 1.4) and suggests the first Hardware Architecture of Systemic computation (HAoS), taking into consideration the natural properties of SC (research challenge *Chg1*, section 1.4) and at the same time attempts to provide an efficient, practical and user-friendly solution (research challenge *Chg3*, section 1.4).

Various potential architectures are initially reviewed (section 3.1), while then the SC architecture properties are analyzed and discussed in the context of a hardware implementation (section 3.2). This discussion leads to the suggested design (sections 3.4 - 3.6) which is presented along with the proposed extendable instruction set (section 3.3) and a basic programming model (section 3.7). This base design is initially verified in section 3.8 and is used in the next chapter as the basis of the complete HAoS programming platform.

Part of the work presented in this chapter has been published in [158],[159] and [160].

3.1 Potential Architectures

The optimal solution for the hardware implementation of SC would be highly flexible and at the same time highly efficient. The user would be able to write SC programs in an unrestricted manner (following just the SC language rules). The key SC notion of parallelism should be implemented for both functional and background system tasks (systems update and storage, scopes update and storage, systems comparison and communication). A number of candidate architectures (given that the implementation platform is an FPGA), taking into account the implementation feasibility and viability, were considered before concluding to the final design. These are overviewed below:

- Virtual SC: offloading functional computation to the CPU.

- Fundamental Processing Element: SC performed solely on the FPGA but providing support for an extremely restricted instruction set that will be implemented by elementary processing elements.
- Reconfigurable Predetermined Processing Elements Array: Providing a more rich instruction set but assuming low reconfiguration frequency.
- SC2HDL: Translate the SC code into hardware, using a tool that takes source code written in the SC language as input and performs translation in a Hardware Description language (HDL)[16], synthesis and Place-And-Route (PAR) in an automated way.
- GPUplusFPGA: preserve the functionality of the GPU version (background parallelizable tasks performed on the GPU) and offload computation to the FPGA (by means of an predefined instruction set realized on hardware or a dynamic instruction set by use of the SC2HDL tool).

3.1.1 Virtual SC

It is evident (see section 2.3) that the use of a conventional CPU is not compatible with SC. However, the power of a custom design which is highly optimized to perform the background SC system tasks, as they are mentioned above, could take advantage of the flexibility and performance provided by a CPU.

Provided that a modern FPGA is used, an estimation of the highest frequency of an on-chip implementation could be claimed to be in the order of 600MHz [161]. For the purposes of this analysis, we may assume that the final design may achieve 1/3 of the maximum frequency (200 MHz). Assuming 10 on-chip flexible processing elements and taking into account any delays caused by off-chip communication, it would be safe to claim that a conventional single-core CPU could cope with the computational load, provided that the communication interface is able to cope with the communication load. While the background system tasks will optimally run on hardware, the functional tasks will be executed on "virtual" on-chip processing elements, simulated by the CPU. Extending this strategy, a modern multi-core multi-threaded processor with 4 cores and 2 threads for each core could provide the computational equivalent of up to approximately a hundred processing elements (assuming a 2-3GHz frequency for each core) by

[16] Most probably VHDL.

consuming on-chip resources just for the communication infrastructure. Nevertheless, it is noted that the communication overhead could become substantial for a large number of systems.

Embedding a sequential processor aside, the disadvantage of this approach is its resemblance with the GPU implementation (see section 2.4.3). Modern GPUs are becoming more powerful, embedding typically hundreds or even thousands[17] of processing elements but yet preserving a more centralized architecture than the one that a custom FPGA design can provide. However, the GPU architecture still needs to be fairly generic in order to support any parallelized task. This can be avoided with a highly optimized FPGA design.

Despite of the above-mentioned disadvantages, the virtual SC architecture, shown in Figure 3.1, could be considered as an entry-level design that is focused on realizing the background system tasks on hardware and emulate the functional subsystem in the CPU.

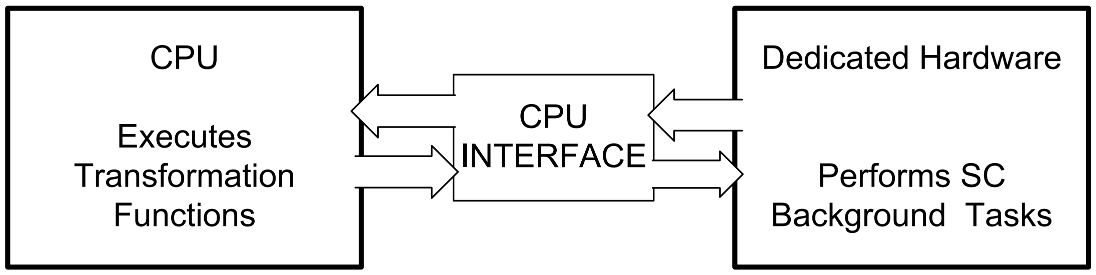

CPU
Executes Transformation Functions
CPU INTERFACE
Dedicated Hardware
Performs SC Background Tasks

Figure 3.1. Virtual SC architecture simplified block diagram

3.1.2 Fundamental Processing Element

In contrast with the previous approach, instead of offloading computation to the CPU, this implementation severely restricts flexibility by keeping functional complexity to a minimum but keeps computation on-chip. The instruction set is limited to basic functions that can be realized in a combinatorial way. Sequential processing elements are avoided as they are not compatible with the SC paradigm. Also, basic functions realizations have lower area requirements and thus a larger number of them may be implemented on-chip. Restricting all SC functionality on-chip makes communication interfaces on the chip boundaries obsolete, with the exception of possible chip-to-chip interfaces that will

[17] The NVIDIA GeForce GTX TITAN GPU has 2688 CUDA cores while the AMD Radeon HD 7990 GPU has 4096 stream processors.

enable expanding the functionality on multiple chips. A simplified block diagram of the architecture is given in Figure 3.2.

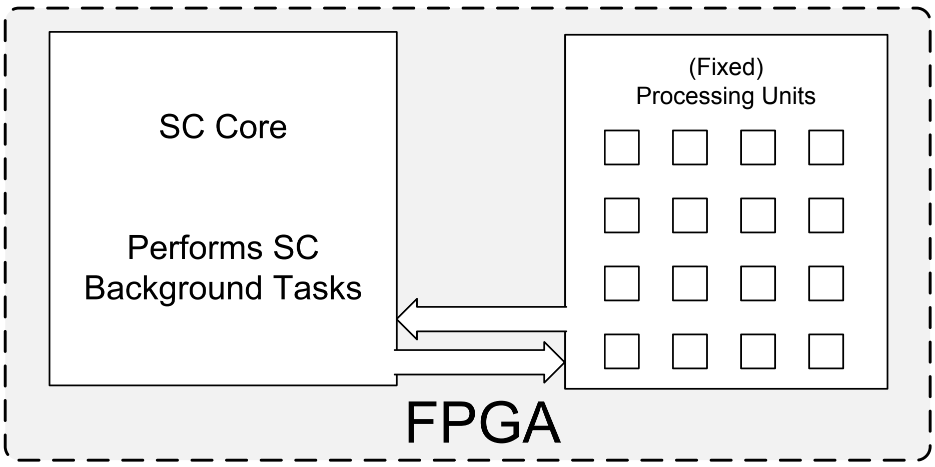


Figure 3.2. SC Fundamental Processing Element architecture simplified block diagram

The limitation of this approach is obviously its restricted functionality. Nevertheless, it could be argued that it could serve as an intermediate step towards the final design. Provided that the background system tasks are developed for the Virtual SC design, they could be reused for this implementation. The functional difference of the two designs is the supported flexibility and the relative location of performing the computation. In essence, the extended functionality of the Virtual SC design is traded with the ability of having a standalone design that complies with the non-sequential rule of the SC paradigm.

3.1.3 Reconfigurable Predetermined Processing Elements Array

Having moved computation on-chip in the previous approach severely limited functionality. To address this issue a reconfigurable predetermined processing elements array[18] can be used, essentially meaning that each transformation function can be pre-mapped to custom logic and loaded on the reconfigurable logic on-demand. Since, after the SC source code is compiled, the initial required functional elements are known, those can be realized in a chosen on-chip area that is configured to perform the required form of computation. Provided that the required instructions do not imply a restrictive area overhead, this design can potentially support any predefined processing element. The set

[18] Analogous to the approach followed in the Molen reconfigurable processor (section 2.2.3) and a similar design providing an Algorithm-on-Demand implementation (relying on a host CPU, thus provided in a co-processor form) suggested in [249].

of supported functions can be optimized for hardware implementation and can either be statically realized on chip or dynamically chosen from a pool of functions, stored either on-chip or off-chip (depending on size) in the form of configuration bitstreams. The architecture is shown in Figure 3.3.

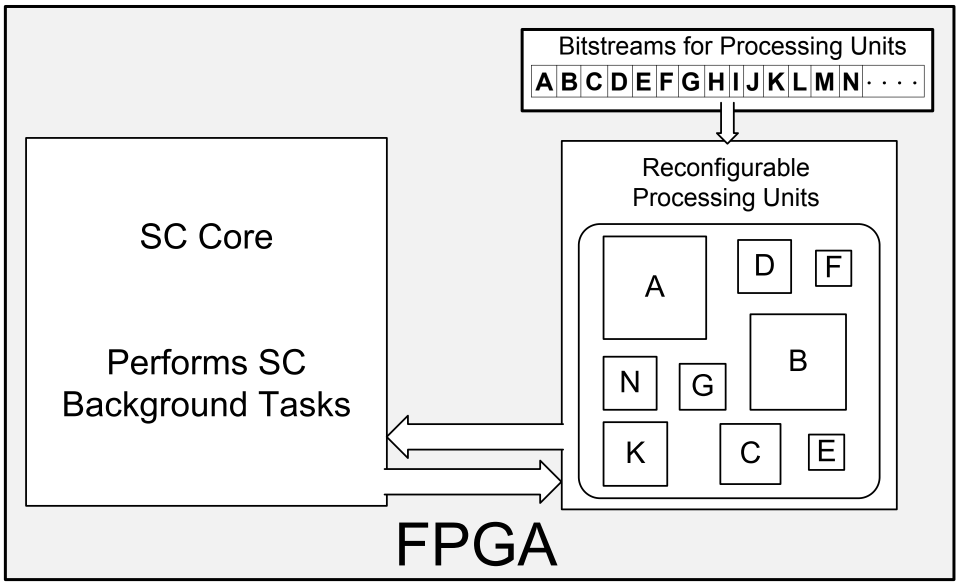

Bitstreams for Processing Units
A B C D E F G H I J K L M N · · · ·
Reconfigurable Processing Units
SC Core
Performs SC Background Tasks
A
D
F
B
N
G
K
C
E
FPGA

Figure 3.3. SC Reconfigurable Predetermined Processing Elements Array architecture simplified block diagram

The limitations of this implementation are the area overhead due to the functional elements and the design complexity. Restricting the supported instruction set and the number of on-chip instances of processing elements could reduce the required on-chip area. The use of partial chip reconfiguration will inevitably lead to following vendor-specific design methodologies that will restrict the design to a vendor-specific implementation (the vendor being the chosen FPGA platform supplier). The reconfiguration time highly depends on the size of the partial bitstream being loaded to the FPGA and the selected configuration mode[19] and it can vary greatly (from the order of 10 μs down to the order of 100 ms) [162]. Thus, a low reconfiguration frequency must be assumed for this implementation to be functional.

Another consideration is that a predefined array implies that a functional element can only be altered to become another already existing block in the provided function pool.

[19] For example, if a Xilinx Virtex FPGA is chosen, the maximum provided configuration bandwidth is 3.2 Gbps for the vendor-specific ICAP mode, and 100 Mbps and 66 Mbps for the more common Serial and JTAG modes, respectively [162].

Assuming a limited number of different types of functions being realized for any given SC program, functions being changed during execution (as shown in [151] with the genetic operator adapter) is not essentially supported. Along with having a prearranged set of functions, a predetermined area of the chip will need to be pre-allocated in order to realize the partial reconfiguration functionality. This implies that, depending on the functions being realized, part of the reserved hardware resources may need to be unused.

This design could potentially be the base of the final SC hardware implementation. The majority of the aforementioned limitations are bound to the restrictions imposed by the chip size. Further improvements will have to address chip-to-chip communication to resolve this issue. Any high level SC instructions will have to be mapped to hardware processing elements by a translation step on the software side that decomposes those instructions to basic instructions supported by the hardware instruction set.

3.1.4 SC2HDL

A SC to HDL tool would translate a SC program to a circuit that would be optimized to execute this program only. Effectively, this approach could maximize the on-chip hardware resources utilization. Also, it could be applied either for a predefined set of instructions (following the first SC implementation) or, targeting a more generic and flexible approach, it could be developed as a high-level SC synthesis tool (following the SCoPE implementation). It is noted that following this approach, every SC program would result in a different custom design.

This implementation would probably be the most flexible and area-aware. The size is again the limiting factor. This approach assumes that the user has acquired a license for the required tools that will manage the backend realization process (synthesis and PAR). The development process will require a conventional microprocessor to download the circuit on the FPGA. Extra care should be taken when developing the SC2HDL[20] tool since, while the FPGA-vendor tools have been developed to make the backend tasks automated, user feedback input is typically required before having a fully-functional implementation.

In order to realize any function, and thus get maximum flexibility, the high-level source code will have to be translated to an HDL. However, none of the SC language versions

[20] SC2FPGA might be more clear in describing the whole flow. SC2FPGA would insist of the SC2VHDL tool for the front-end and vendor-specific tools for synthesis and PAR.

to date provide inherently the ability to describe new functions, but rather, those are described by means of a software high-level language (C/C++) (either predefined in the code or dynamically created as plugins). This implies the need of a C2HDL compiler. A few C2HDL compiler attempts have already been made and the majority of them are commercially available [163], [164]. A viable solution for an SC2HDL tool would be to incorporate one of the available compilers and combine this with the vendor-specific tools. The main disadvantage of this approach would be the limited control of the resulting HDL. A possible solution to this issue would be to develop an intermediate tool that accepts the automatically extracted HDL code and alter it in a way that it is synthesizable and ensure that the backend tools will not encounter any problems, taking always into consideration the development speed of such an approach. Yet, this may be an infeasible task since further (especially automated) optimizations may be prohibited depending on the level of abstraction.

3.1.5 GPUplusFPGA

This is a rather novel approach. Since the power of a GPU performing background SC system tasks has been demonstrated in [165], the idea of reusing the advantages of this implementation is highly appealing. The GPU will still be used as a co-processor, but the place of the sequential processor will be taken by a pool of processing elements implementing on the FPGA. Various attempts can be found in the literature that use both FPGAs and GPUs as coprocessors [166], [167], even combining them on the same board [168], suggesting transferring the GPU logic on the FPGA [169], communicating directly through a PCI-Express switch [170] and translating directly a GPU programming language (CUDA or OpenCL) on FPGA resources [171][172]. Yet, all these attempts rely on a host CPU. Offloading computation on an FPGA that acts as a host for the GPU, illustrated in Figure 3.4, could be a potential solution for the SC hardware implementation but probably it would not be trivial.

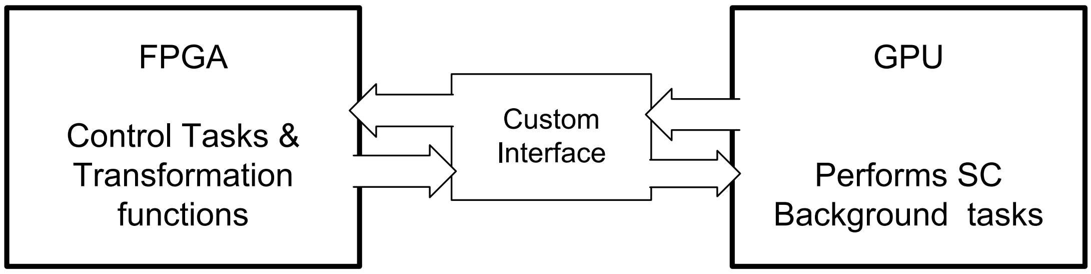

FPGA
Control Tasks & Transformation functions
Custom Interface
GPU
Performs SC Background tasks

Figure 3.4. GPUplusFPGA architecture simplified block diagram

The main disadvantage would be the lack of supported tools for an implementation like this and the need for development of drivers for the GPU. The idea is to emulate the control flow provided by the CPU on a typical CPU-GPU system on the FPGA. Inevitably, the GPU-FPGA interface would have to be sequential and might even cancel the benefits of avoiding using a CPU when considering the development effort, FPGA required resources and limited flexibility due to lack of programming and debugging tools for a configuration like this.

3.1.6 Summary

The hardware systemic processor could potentially be implemented on an FPGA following one or more of the aforementioned ways. It is noted that combining more than one of those approaches (e.g. the first three) can result in a more modular design process. In contrast, the last two approaches can be considered as standalone solutions. Nevertheless, it should be underlined that there are overlapping design elements for all approaches (the hardware realization of the SC background system tasks is required for all approaches except the last one while the SC2HDL tool could potentially be used for the Predefined Elements Array or the GPUplusFPGA approach).

The main advantages and limitations of each approach are outlined in the preceding sections. It can be claimed that the most modular solution of going through the first three approaches is more feasible based on the required work load against the available time frame of this research project. The addition of the SC2HDL tool (or part of it - with support for a predefined set of instructions) can be reconsidered in the future as an expansion to this work. The last approach may not yet be feasible due to the lack of support tools.

A fully non-sequential SC hardware implementation would not be practical since, even if the majority of on-chip logic could be combinatorial, the nature of the memory elements and the interfaces with the off-chip resources will have to be sequential. Also, chip size imposes limitations to the hardware resources that can be utilized in order to implement the required functionality of a SC program. The sequential alternative may need to be used (especially for a single-chip SC hardware implementation) in such cases.

As a result of the analysis above, an appropriate solution would combine the Virtual SC and the Fundamental Processing Element approaches by embedding dedicated processing elements on-chip but also providing the option of work-load offloading to an (internal or external) CPU in order to handle the functional (data-processing) system tasks. The background system tasks are executed natively on-chip. The suggested hybrid design is illustrated in Figure 3.5 and detailed throughout the rest of this chapter.

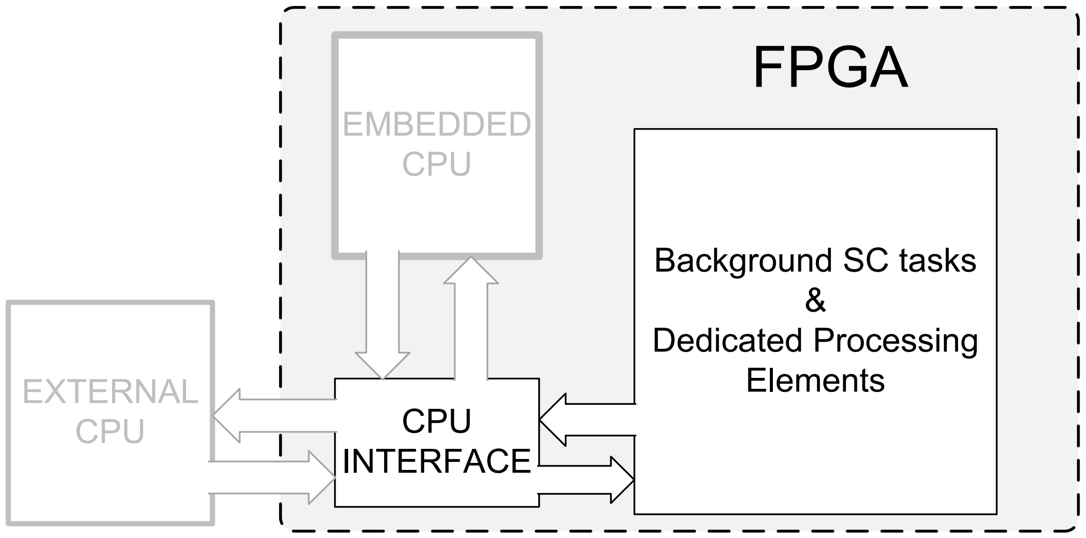


Figure 3.5. The suggested hybrid design

3.2 Design Analysis of the SC Architecture

The proposed Hardware Architecture of Systemic computation (HAoS) attempts to satisfy the basic SC requirements, taking into consideration the desired features of a practical implementation: programmability, design friendliness, technology maturity, I/O functionality efficiency, advanced processing features, compiler support and scalability. It preserves partial backwards compatibility with the original SC implementation in order to take advantage of the available compiler but expands on the supported functionality by adding new features. Thus, SC source code targeting the original implementation (and the GPU-based version) can be natively executed on HAoS.

The SC concept dictates that any three systems are eligible to form a valid triplet. A fully parallel implementation would generate a valid triplet of systems, in a random manner, for all contexts, in all scopes during an iteration of a SC program. In addition, all interactions would happen instantaneously, provided that adequate parallel processing resources were available. Resource limitations forbid a practical implementation of this approach on an FPGA. It is evident that the main two tasks that would ideally be executed in parallel are valid triplet generation (finding triplets of interacting systems) and system transformation (the actual data processing).

This section mainly addresses research challenge *Chg2*, as it discusses various features of the SC architecture, and analyses their respective design decisions. Related natural properties are also discussed (research challenge *Chg1*), where applicable, while as shown below practicality and efficiency are major decision factors (research challenge *Chg3*).

3.2.1 Local Knowledge & Scope Definition Method

One of the assumptions (and supported natural properties) of the SC paradigm is that systems have “local knowledge”. This can both refer to local storage of the internal state of each system and awareness in terms of its membership within other systems' scopes.

However, local knowledge is a feature that cannot be efficiently mapped on on-chip logic. The system bit representation and the scopes it belongs to could potentially be stored in registers which do not reside in the same area of the chip. Yet, storing this information in local registers was not adopted but it was decided that the proposed design should store it in system RAM instead. The use of a RAM in this design is justified by the fact that RAM storage volumes are greater than those provided by registers in modern FPGAs and since no more fabric would need to be consumed for address decoding logic. Moreover, only a finite number of systems can be stored on a single RAM, which defines a neighbourhood for its systems, while the total number of systems can be spread over multiple RAMs. As a result, a potential failure in one of the RAMs would leave the rest of systems of the program unaffected.

Apart from its binary contents, every system can belong to any number of scopes defined by other systems. In SCoPE [22], local knowledge is correctly simulated as each system holds a list of all its parents (the scopes it belongs to). In order to fully support this feature, HAoS would need to locally store the parents' information in registers, which would result in a considerable increase in the number of required on-chip registers as the number of maximum supported systems scaled up. It was decided that it is more important to preserve scalability (research challenge *Chg3*) than fully support local knowledge (research challenge *Chg1*), so the global scopetable approach was selected as the scope definition method, with the parents of each system stored in RAMs.

3.2.2 Scopes Support

In the original SC implementation (see section 2.4.1), scopes are infinitely recursive; they have fuzzy boundaries and may overlap. Recursive scopes may contain themselves and other systems which in turn contain themselves and other systems and so on. Fuzzy scopes enable partial membership of a system into another system while overlapping scopes partially belong to each other.

In this work, a system containing itself is supported but fuzzy or overlapping scopes are not. This decision was made in order to reduce the amount of required storage for the program scopetable. Enabling fuzzy or overlapping scopes would require a fractional value to be stored in the scopetable, increasing the number of bits to represent the scope membership. This would enable a more accurate control of interaction probabilities; however multiplying the size of the scopetable would limit scalability (research challenge *Chg3*). Thus, a one-bit representation was preferred, denoting that a system can either belong to a scope as a whole or not belong at all. The interaction possibility control can either be embedded in the implementation of the transformation function of the executed instruction (as shown in [22]) or by appropriately setting the number of identical systems in the scope (assuming all individual systems share the same interaction possibility).

In SCoPE [22], the notion of the universe is also introduced to define a super-scope which includes all other systems in the SC program. The notion of universe here is analogous to the "main" function of a typical conventional C program. This super-scope is indirectly supported and used in all HAoS programs to include all used systems of a HAoS SC model (yet allowing floating systems - systems that do not belong in any scope). The universe system can also have a physical meaning in future work, as it can be used to describe all the systems that are stored on a single FPGA device. For a multi-FPGA configuration, each universe can correspond to one FPGA device, each with its own scopetable and systems. Then, all universes would belong to a root scope or "multiverse" (forming a multi-FPGA SC program) enabling communication between them with the form of mutual systems exchange.

3.2.3 Valid Triplet Generation & Schemata Matching

One of the main limitations of the software-based implementations was the way valid triplets were generated. The common strategy was to randomly select three systems (one of which acted as context) in a scope and then check triplet validity (by examining if the operand systems matched the schemata of the context). In [34], this task is assigned to the GPU which handles it in parallel, resulting in great performance gains.

The most straightforward hardware implementation for the valid triplet generation mechanism would be a sequential design with an optimized comparator iteratively trying to match the templates defined by the schemata of the context system to all valid systems in the selected scope. This approach would result in minimal area utilization, as the same comparator would be reused for all comparisons, and possibly a very fast circuit as the required combinatorial logic would be minimal too. However, the overall latency of such a sequential design would be increased proportionally to the number of systems and would prove impractical for SC programs involving a big number of systems.

Thus, a parallel schemata matching mechanism is crucial for the design, if we want to minimize latency and handle valid triplet generation optimally (research challenge *Chg3*). In addition, as parallelism is one of the desirable natural properties, employing a parallel design would also address research challenge *Chg1*. Thus, in order to identify which systems may interact during each iteration of the systemic program, a parallel binary matching mechanism, matching the templates of context systems against the schemata of all interaction candidates, would be the most suitable solution. This essentially implies the use of a comparator which, given a binary input, has the ability to match this input (the template) against the contents of an array of elements (storing the systems) in a parallel fashion. In addition, it should also support full and partial matching, meaning that some parts may need to be ignored during comparison.

These requirements for optimal valid triplet generation are fully satisfied by exploiting the inherent parallelism of a Ternary Content Addressable Memory (TCAM). While traditionally used Random Access Memories (RAMs), when provided with an address return the data stored in this address, CAMs compare their input data with the data which they store and provide all matching addresses in parallel. This is illustrated in Figure 3.6. Moreover, CAMs can be efficiently implemented on modern FPGAs, utilizing on-chip memory resources [173].

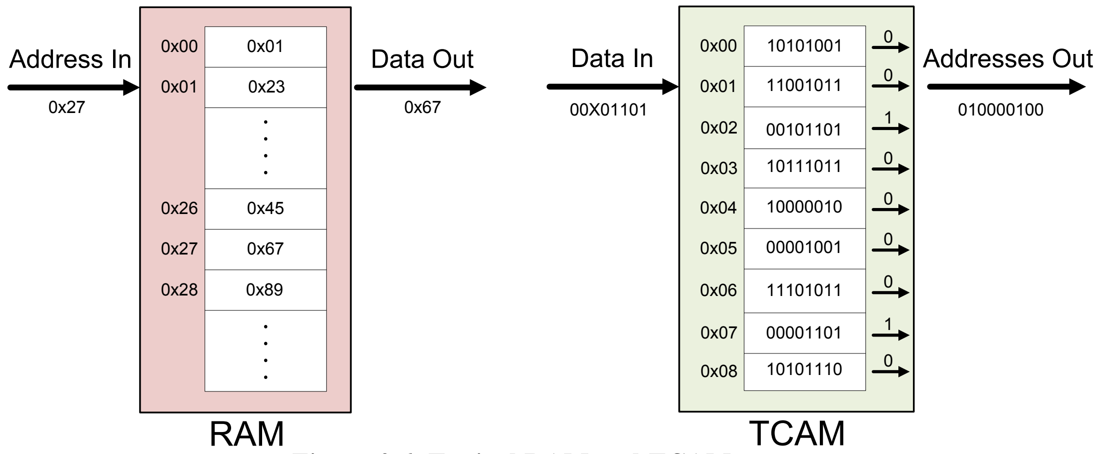


Figure 3.6. Typical RAM and TCAM usage

Prior implementations compared each character of the given template of a context (see Figure 3.8) to the corresponding character of a candidate system separately, yet HAoS, by using a CAM, compares the given template as a whole (all its characters) with all the systems that the program defines and gives all the matching systems in parallel (see Figure 3.7). Moreover, TCAMs have the ability to perform ternary comparisons, meaning that both the input and stored data can include “don't care” bits. As shown in Figure 3.6, data stored in both addresses 0x02 and 0x07 of the TCAM match the input binary data word 00X01101 as its “don't care” bit, written as an “X” wildcard, can match both a 0 and a 1 bit. These features allow parallel partial schemata matching which enables a guaranteed match of systems to the schemata of the given context, provided there are such systems in the scope of the context.

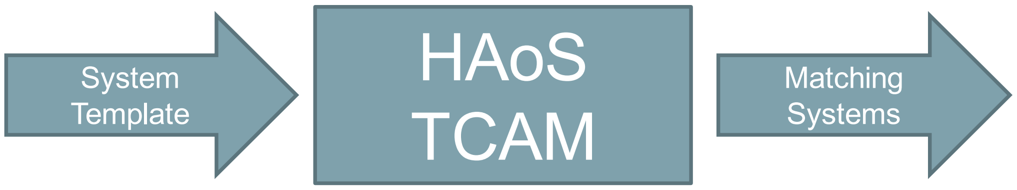


Figure 3.7. HAoS TCAM usage

3.2.4 Threshold Matching

As mentioned in section 2.4.1, the original SC implementation, along with partial matching, defines threshold-based matching in order to control the schemata matching precision by comparing the Hamming distance between the two schemata against the matching threshold. Effectively, this means that systems similar to the ones indicated by the schemata template can be selected to interact (the threshold adjusts the similarity).

Since schemata matching is performed as a parallel operation in this design, as explained above, supporting this functionality would require an array of Hamming distance hardware blocks equal to the maximum number of supported systems instead of using the highly efficient and more compact solution of the TCAM.

Thus, in order to minimize the area requirement of the circuit (research challenge *Chg3*), the TCAM was chosen instead, disabling threshold matching for HAoS. However, the user can use fixed-position wildcards in order to partially adjust the similarity (by setting the position in which the systems may be different but nevertheless match).

3.2.5 Systems Representation & Coding Method

As explained in section 2.3, the SC paradigm defines interactions between any two systems according to the transformation function of a third contextual system. Thus, HAoS supports three types of systems, as shown in Figure 3.8: (a) data systems, comprised of two schemata (with 16 effective bits each) and a zero (32-bit) function part, (b) context systems, comprised of a (32-bit) transformation function and two schemata templates (used for matching with data systems and thus occupying the size of a whole data system, 64-bits, each but with zero transformation functions) and (c) context adapter systems which have the same structure with context systems (but each of their templates can match a data system or a context). Since all the systems have the same size, each bit in a schema of a data system is padded with three zero bits to form a 4-bit element or character.

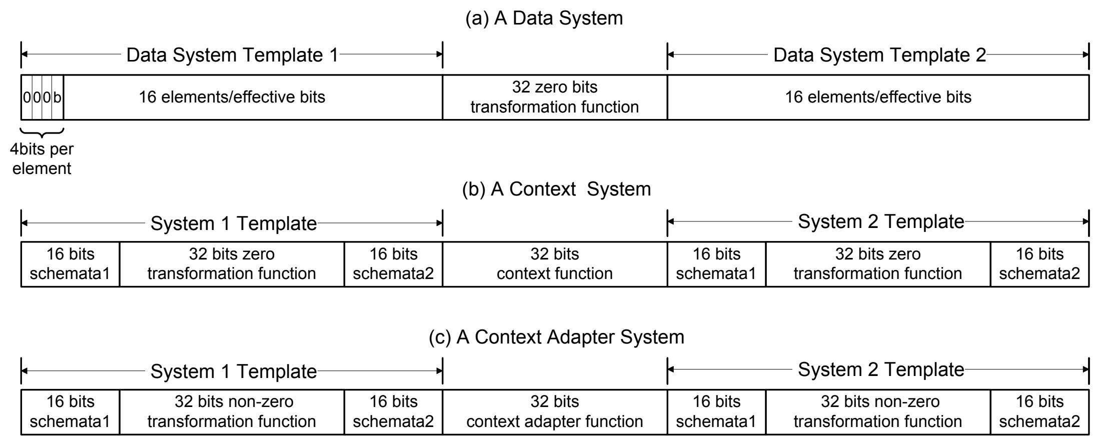


Figure 3.8. HAoS Systems Representation

While the word length of the schemata (system templates) and transformation function is equal (16 one-byte characters) in the original implementation, as shown in Figure 2.13, it was decided that HAoS should adopt a different approach in order to optimize the required size of representing each system. Since a system template has to indicate a prototype for a whole system to match, it should have a size equal to the effective bits (which are the bits used for matching purposes) of this system. If schemata and transformation function had the same size, this would imply a compression scheme (compression of a whole data system, in order to have the same size with the template, by using only the effective bits of each schema) with compression ratio 3:1. This, then would denote that each character (or element) of the template should have at least three bits in order to be compressed into the minimum storage space (one bit). However, four bits per character were selected instead, in order to resolve any byte-alignment issues that a choice of three bits might cause, simplify the control logic and, by providing a 4:1 compression ratio, enable the use of a greater transformation function size. This allows more distinct instruction opcodes and provides more space for future uses (for example a variable part, see Table 3.3).

Table 3.1. HAoS Compression Code. Each HAoS assembly code ASCII character (C) is compressed internally into a 4-element ternary value (Val). Each element is composed of 2 bits representing 0, 1 and (?) ternary bits (tbits)

Val	C	Val	C	Val	C	Val	C	Val	C	Val	C	Val	C	Val	C	Val	C
0000	!	0100	/	0?00	:	1000	@	1100	I	1?00	R	?000	`	?100	i	??00	r
0001	#	0101	2	0?01	;	1001	A	1101	J	1?01	S	?001	a	?101	j	??01	s
000?	%	010?	3	0?0?	<	100?	B	110?	K	1?0?	T	?00?	b	?10?	k	??0?	t
0010	&	0110	4	0?10	=	1010	C	1110	L	1?10	U	?010	c	?110	l	??10	u
0011	*	0111	5	0?11	>	1011	D	1111	M	1?11	V	?011	d	?111	m	??11	v
001?	+	011?	6	0?1?	[	101?	E	111?	N	1?1?	W	?01?	e	?11?	n	??1?	w
00?0	,	01?0	7	0??0	]	10?0	F	11?0	O	1??0	X	?0?0	f	?1?0	o	???0	x
00?1	-	01?1	8	0??1	^	10?1	G	11?1	P	1??1	Y	?0?1	g	?1?1	p	???1	y
00??	.	01??	9	0???	_	10??	H	11??	Q	1???	Z	?0??	h	?1??	q	????	z

In order to support this four bits per character compression scheme a different code table is used (given in Table 3.1) from the one used in the original SC implementation (shown in Figure 2.13). This code table is used to compress the SC assembly code which is generated from the SC compiler (in ASCII format) into a ternary format using 0, 1 and ternary (?) bits (matching both 0 and 1) to give machine code for the HAoS digital architecture. Each ternary bit is represented internally with two binary bits.

3.2.6 The Compiler

The SC compiler of the original SC version was written in C [24]. Targeting practicality and efficiency (research challenge *Chg3*), the compiler was updated to support the extra functionality that HAoS offers (context adapters, signed numbers), efficiently handle memory management for programs with a big number of systems and support the required compression code of Table 3.1. As the compiler program needs to be executed on a machine which is able to run compiled C code, the use of a conventional CPU is inevitable if this architecture is to remain backwards compatible with earlier versions.

Once the SC source code is compiled in HAoS human-readable assembly code, the assembly code which corresponds to the systems which are defined in the HAoS program must be compressed (according to the HAoS compression code) into a representation which is tailored to the underlying hardware architecture (optimized HAoS machine code) and loaded on on-chip memory (see section 3.6.1).

3.2.7 Interactions Order

One of the fundamental properties of natural systems that SC supports is that they are stochastic (see section 1.1), denoting that interactions happen in a random order. All previous SC versions attempt to implement this property by randomly selecting the next interacting systems, but result in just simulating the process. This is because the random selection should be combined with parallel execution of interactions or, even more accurately, there should be no selection but just random parallel interactions to truly implement the stochastic property. However, in practice, the random selection process is inevitable, as it is a means of controlling the systems interaction flow. This implies the requirement for a source of randomness and some associated circuitry (this random selection logic is presented later in section 3.6.2) to implement this process.

This requirement for on-chip randomness denotes the implementation of a random number generation (RNG) scheme. RNG on FPGAs has been extensively addressed in the literature with approaches targeting pseudo-random numbers sequences (PRNG) [174], [175], usually involving some post-processing logic if non-linearity is required, true RNG (TRNG), which relies on some source of natural randomness (as thermal input or jitter from on on-board or on-chip ring-oscillators [176], [177]) and Quasi-RNG (QRNG) which covers some multi-dimensional space uniformly (usually used for Monte-Carlo simulations).

Since typically more randomness quality involves more complex circuitry, a Linear Feedback Shift Register (LFSR) pseudo-random generator is assumed to be sufficient for random selection in HAoS, as it provides a well-balanced solution in terms of utilization, throughput and randomness [178] for non-security applications, especially as a starting point for the suggested prototype implementation. The implementation of the PRNG block may be revised, should increased randomness quality is required, and replaced with one of the more advanced approaches mentioned above.

Prior SC implementations used priority queues that either gave priority to systems that had recently interacted [24] (in order to increase execution efficiency by enabling a diffusion effect for subsequent interactions involving the same system) or had not recently interacted [136] (ensuring a more “fair” interaction allocation since all systems share the same interaction probability). HAoS also uses a pseudo-random number generator to randomly identify valid triplets but this operation is not biased by previous interactions. All matching systems have the same interaction probability (resulting in reduced control logic complexity) while, as explained above, the use of the TCAM ensures maximum matching efficiency. While future work may target parallel processing capabilities, true parallel interaction is currently not supported by HAoS, since writing to the TCAM is limited to one system at a time in order to improve its area and enable ternary comparisons (assuming that parallel interactions would transform the interacting system simultaneously).

A fully asynchronous design might enable the true implementation of the stochastic property, but such an implementation would require that all systems, matching and control circuitry and interconnections would be realized in combinatorial logic which would pose a great area requirement and increase the possibility of timing hazards [179]. However, it is noted that ongoing research is been carried on providing practical asynchronous FPGAs [180][181], conversion methodologies from synchronous designs to their functional asynchronous equivalents [182] and hybrid approaches like Globally Asynchronous Locally Synchronous [183] circuits.

3.2.8 SC Architectural Features Summary

The SC architectural features to be implemented by HAoS, and discussed in the previous sections, are summarized in Table 3.2 along with the corresponding solutions used by prior implementations. The analysis above addresses research challenge *Chg2* by explaining how HAoS will support the underlying architecture of SC.

Table 3.2. Implementation-specific features of HAoS and prior implementations

Feature	Original	SCoPE	GPU	HAoS
Implementation Platform	Software (CPU), written in C	Software (CPU), written in C++	Software (CPU), hardware-accelerated (GPU), written in CUDA	Hardware (FPGA), written in VHDL, supporting software (CPU) extensions (C/C++)
Word-length	16-character word length, 1 byte/character (systems consist of 48 characters)	variable length (1 byte/character), customizable for each program	16-character word length, 1 byte/ character (systems consist of 48 characters)	16 4-bit characters schemata length, 32 1-bit characters function length (systems consist of 64 characters)
Coding Method	characters of alphabet 29 (ASCII characters 0,1 and ' to z)	customizable for each program (default is ASCII characters # and a to z)	characters of alphabet 29 (ASCII characters 0,1 and ' to z)	81 (3^4) (4 ternary bits each, 3 values each tbit) ASCII characters ! to z excluding characters 0 and 1
Transformation Function Set	Thirty basic functions	customizable for each program, functions defined as C++ plugins (DLLs)	Thirty basic and seven hardcoded application-specific functions	Basic, hardcoded application-specific and user-defined functions support

Table 3.2.(Continued) Implementation-specific features of HAoS and prior implementations

Feature	**Original**	**SCoPE**	**GPU**	**HAoS**
Schemata Matching Method	Partial matching against thresholds	Partial matching	Partial matching against thresholds	Parallel Partial matching
Interactions Order	Pseudo-Random (Biased – Prioritizes recently changed systems)	Pseudo-Random (Biased – Prioritizes recently unchanged systems)	Pseudo-Random (Biased – Prioritizes recently changed systems)	Pseudo-Random (Unbiased)
Scope Definition Method	Global Scope Table	Local Scope Simulation (scopes including a system are part of its definition)	Global Scope Table	Global Scope Table

As indicated by the design choices of the last column of Table 3.2, explained throughout this chapter, HAoS attempts to optimize the efficiency versus flexibility trade-off, providing the user with a flexible architecture which takes into consideration performance and programmability in order to provide a practical solution, addressing in this way research challenge Chg3.

3.3 HAoS Instruction Set

It is necessary to provide an instruction set for HAoS, and the solution proposed here is to use an on-chip hardware-supported RISC-like set of simple functions. Furthermore, in order to enhance flexibility, this core instruction set can be further extended by both extra hardware-supported application-specific instructions or software-implemented functions (see section 3.4). It is noted that a HAoS instruction does not share the definition of an instruction found in a conventional ISA but rather expresses the type of transformation that systems undergo when they interact. These interactions happen in a random manner; the execution probability of each SC interaction depends solely in the number and types of systems in the SC program.

The instructions are given by the transformation function (middle) part of a system (see Figure 3.8). Their respective fields are explained in Table 3.3. In this prototype HAoS implementation, the transformation function is given by a 32-bit field. The first (LSB) 22 bits give the function identifier, the next bit (at position ESC_BIT_POS) enables the hardware-supported escaping functionality (to be explained later) which can be executed in parallel with any instruction except the CAPTURE instructions (also to be explained later), the next 8 bits are reserved (they may be used to store variables as part of the instruction) while the MSB enables the NOT functionality which reverses the matching requirement of an instruction (when enabled, the systems that do not match the provided schemata are selected).

Table 3.3. HAoS Instruction Fields

Bits	Meaning
FUNCTIONID-2..0 (21..0)	Function Identifier
ESC_BIT_POS (22)	If Set Then system also escapes from parent scope
FUNCTIONSIZE-2..ESC_BIT_POS+1(30..23)	Reserved (variable part)
FUNCTIONSIZE-1 (31)	If Set then the matching requirement is reversed

The prototype implementation of HAoS supports the instruction set given in Table 3.4. It is noted that this is an example instruction set, as more instructions can be supported according to user requirements. Table 3.4 comprises three sections: the SC Core hardware instructions which are supported natively from HAoS Function Unit (these were supported in software by the prior fixed instruction-set SC implementations [24][34]), SC Extra instructions, which are also implemented on-chip but can be application-specific or realized outside the FU (e.g. on dynamically reconfigurable fabric or DSP blocks) and software-based instructions implemented on the (on-chip or off-chip) CPU (these instructions are defined to have an opcode above a predetermined threshold in order to simplify HAoS control logic).

Table 3.4. HAoS Instruction Set

Mnemonic	Code(hex)	Short Description	Context Adapter Flag	Operation
SC Core HW Functions				
NOP	0000000F	No Interaction	-	(Non-zero to differentiate from data systems)
ESCAPE	0040000F	System escapes from parent scope to all scopes the parent scope belongs to	-	Scopetable manipulation
ADD	00000001	Add schematas of interacting systems	-	sys1.sch2 = sys1.sch2 + sys2.sch2; sys2.sch2 = 0;
SUBTRACT	00000002	Subtract schematas of interacting systems	-	sys1.sch2 = sys1.sch2 - sys2.sch2; sys2.sch2 = 0;
MULT	00000003	Multiply schematas of interacting systems	-	sys1.sch2 = sys1.sch2 * sys2.sch2; sys2.sch2 = 1;
DIV	00000004	Divide schematas of interacting systems	-	sys1.sch2 = sys1.sch2 / sys2.sch2; sys2.sch2 = 1;
MOD	00000005	Modulo of schematas of interacting systems	-	sys1.sch2 = sys1.sch2 % sys2.sch2; sys2.sch2 = 1;
ISZERO	00000006	Check if schemata of system is zero	-	if sys1.sch2 = 0 => SET sys1.sch1[schematasize-1]
AND	00000007	AND schematas of interacting systems	-	sys1.sch2 = sys1.sch2 AND sys2.sch2; sys2.sch2 = sys1.sch2 AND sys2.sch2;

Table 3.4. (Continued) HAoS Instruction Set

Mnemonic	Code(hex)	Short Description	Context Adapter Flag	Operation
SC Core HW Functions				

OR	00000008	OR schematas of interacting systems	-	sys1.sch2 = sys1.sch2 OR sys2.sch2;
				sys2.sch2 = sys1.sch2 OR sys2.sch2;

XOR	00000009	XOR schematas of interacting systems	-	sys1.sch2 = sys1.sch2 XOR sys2.sch2;
				sys2.sch2 = sys1.sch2 XOR sys2.sch2;

COPY	0000000A	Copy parts of interacting systems	00	sys1.sch1 = sys2.sch1;
				sys1.sch2 = sys2.sch2;

			01	sys1.function = (sys2.sch2,sys2.sch1);
			10	sys2.function = (sys1.sch2,sys1.sch1);
			11	sys1.function = sys2.function;

ZERO	0000000B	Zero parts of interacting systems	00	sys1.sch1 = 0; sys1.sch2 = 0;
				sys2.sch1 = 0; sys2.sch2 = 0;

			01	sys1.sch1 = 0; sys1.sch2 = 0;
			10	sys1.sch1 = 0; sys1.sch2 = 0;

			11	sys1.function = 0;
				sys2.function = 0;

CAPTURE	0000000C	System is removed from parent scope and captured to capturing scope	-	Scopetable manipulation
SC Extra HW Functions				

ADDxc	00000011	Add schematas & exchange	-	sys1.sch2 = sys1.sch2 + sys2.sch2;
				sys2.sch2 = sys1.sch2;

ADDuc2	00000012	Add schematas but keep the second unchanged	-	sys1.sch2 = sys1.sch2 + sys2.sch2;
SC Example CPU Functions (Above SC_SW_THRESHOLD=512)				
XESCAPE	00000200	Software emulation of ESCAPE task	-	Scopetable manipulation
XCAPTURE	00000201	Software emulation of CAPTURE task	-	Scopetable manipulation
PRINT	00000202	Print system in standard output	-	-
POWER	00000203	Exponentiation	-	sys1.sch2 = math.pow(sys1.sch2,sys2.sch2)
ROOT	00000204	Arithmetic root	-	sys1.sch2 = math.pow(sys1.sch2,(1.0/sys2.sch2))
KNAPSACK*	00000280	Knapsack Problem Related Functions	-	-

For each instruction, its mnemonic (codename), opcode (in hexadecimal notation), a short description of the interaction they represent based on the Context Adapter Flag (discussed below) and its operation (their effect on the state, data and scope of the interacting systems) are given in the respective columns of Table 3.4. For example, the Multiply instruction has MULT as a mnemonic, its opcode is 0x00000003 while schema

2 of system 1 (*sys1.sch2*) gets the product of the multiplication of the schemata 2 of both systems (*sys1.sch2 * sys2.sch2*) while schema 2 of system 2 is set to 1 (*sys2.sch2 = 1*).

Various systems parts are altered after an interaction according to the Operation column. For some instructions there is the option to define a different type of interaction depending on the type of the two interacting systems. This option is controlled by the Context Adapter Flag - CAF (4th column). The CAF is a 2-bit field which states the types (data or context) of the interacting systems. Each bit corresponds to one of the system templates of a context adapter system (see Figure 3.8c). The LSB corresponds to template 1 while the MSB corresponds to template 2. A set bit in the CAF implies a context system template while a zero bit implies a data system template.

Thus, a context system is essentially a context adapter system with both its system templates representing data systems (CAF = 00). When CAF is 01 or 10, the context adapter system is in mixed mode with a data system interaction with a context system and vice versa respectively, while when CAF is 11 two context systems interact.

Two instructions are SC-specific and perform scopetable manipulation meaning that they alter the relationship or membership [24] of one system to another. These two instructions are ESCAPE and CAPTURE and are both optimized to be executed natively in HAoS.

The ESCAPE instruction moves the escaping system (which, by convention, is the system that matches template 1) one level up in the membership hierarchy by removing it from its parent scope (which is the active scope for the interaction) and then inserts it to all the scopes that the parent scope belongs to (or parent scopes of its parent scope or in short the grandparents). The grandparents are conveniently provided in parallel (as a bus of length equal to the maximum number of scopes with set bits at the positions of the grandparents), as a part of the scopetable (SCOPES OF SYSTEM - see Figure 3.11). The ESCAPE task is further optimized by avoiding looping through all the possible scopes to identify the grandparents but rather only the positions of set bits are selected (using BITPOSSEL, see section 3.6.2) resulting in great performance gain.

The CAPTURE instruction, as the name implies, is the reverse of the ESCAPE task where the captured system is removed from its parent scope and added in the scope of the capturing system(s) which are selected based on matching template 2 of the CAPTURE system. A less efficient software implementation of the scopetable manipulation tasks is also provided to the user as an option (see Table 3.4).

3.4 HAoS Architecture

Having provided an overview of the HAoS design and justified the choices with respect to the research challenges, this section provides more detail of the implementation of the architecture. HAoS consists of the SC core (CORE), the Control Unit (CU), the Functional Unit (FU) and a set of configuration and data registers (REG BANK) for communication with the optional CPU (see Figure 3.9).

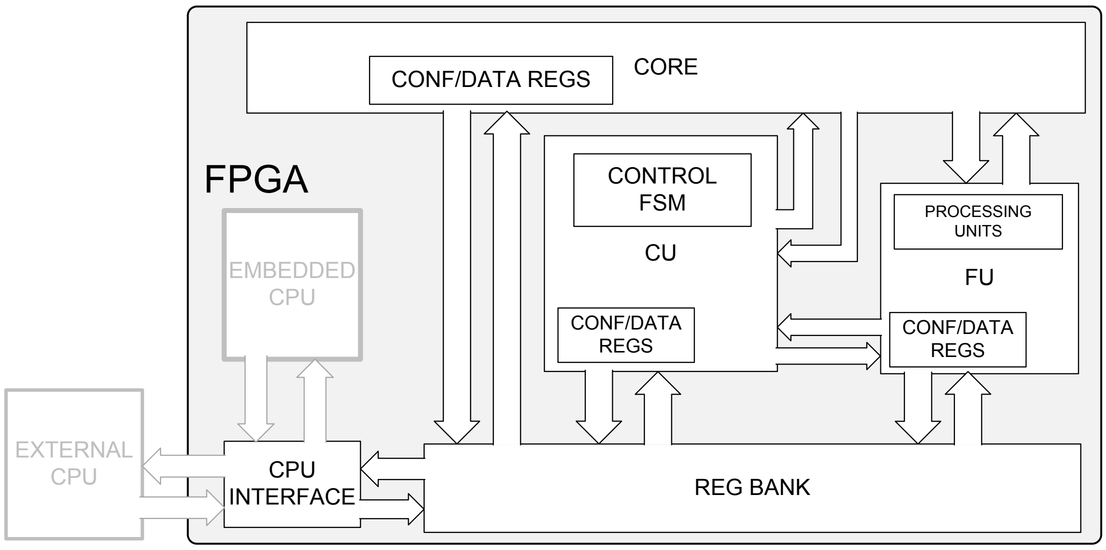


Figure 3.9. HAoS Top-Level Architecture

The CORE contains the optimized logic for the parallel schemata matching and the memory elements. The CU handles the execution sequence of the SC program and the communication with the optional CPU. The REG BANK provides a control and debug interface between the CPU and the local registers of the SC sub-modules. The FU provides basic local processing functionality. A set of simple instructions is supported to avoid expensive data transfers between the REG BANK and the CPU. The prototype implementation includes only one FU, but future implementations can take advantage of the plethora of DSP processing cores which are available on the FPGA, and give the option to be used as a simple ALU each, to provide multiple parallel processing resources.

The CPU is provided to the system in order to make more complex high-level functions available. This functionality was available only in SCoPE [136], since the other implementations had a fixed instruction set. This hardware architecture increases flexibility by letting the user define new instructions, when this is necessary, in an unrestricted way. The SC compiler, which preserves backwards compatibility with the compiler presented in [24], is written in C and translates SC source code in SC assembly. Apart from the extra usability, the CPU in the prototype design is used to load the SC assembly code into the memory elements of the CORE during initialization or in the case of a hardware reset. A possible enhancement would be to provide the option for assembly loading through an external memory card, thus making the CPU link completely optional, depending on the high-level functionality requirements of the user. The CPU can reside either on the FPGA, with the form of a soft IP processor communicating with the design using a shared internal FPGA bus, or be an external conventional processor connecting to the design through a standard communication interface, as illustrated in Figure 3.9. Since the main SC program runs on the FPGA, the CPU is used as a co-processor in HAoS.

A further performance and flexibility boost could be achieved if we take advantage of the reconfigurability capabilities provided by the FPGA (see section 3.1.3). A set of user defined pre-synthesized hardware functions can be stored on an external memory and dynamically loaded when needed. This technique could be applied for applications that do not frequently change the function part of contexts as reconfigurability speeds are quite low and would require the use of an embedded CPU to handle the reconfiguration of a reserved area on the FPGA.

3.5 The Control Unit

The CU handles the flow of the user-defined SC program. As systems can never be destroyed, the program runs in theory indefinitely, although practically it halts when systems become stable and no further interaction is possible[21]. The main control flow for each iteration of the program can be seen in Figure 3.10.

Upon a hardware reset, the SC assembly code is loaded into the core. For each iteration of the SC program, four consecutive steps are performed. A scope is randomly selected, and then a valid triplet of systems is randomly chosen, the selected systems are retrieved from memory, they interact (the actual computation is performed) and then the outcome of the interaction (the computation results) is written back to memory (the random system selection logic is described in the next section). At the end of each iteration, the user is granted access to pause execution. This optional step is mainly provided in order to facilitate the extraction of debug information. All the optimized low-level SC micro-

[21] This implies a closed system. The halting mechanism may be disabled for a SC program with an open system which might receive input or communications from an external source.

routines (for scope and memory manipulation) are available to ensure maximum flexibility.

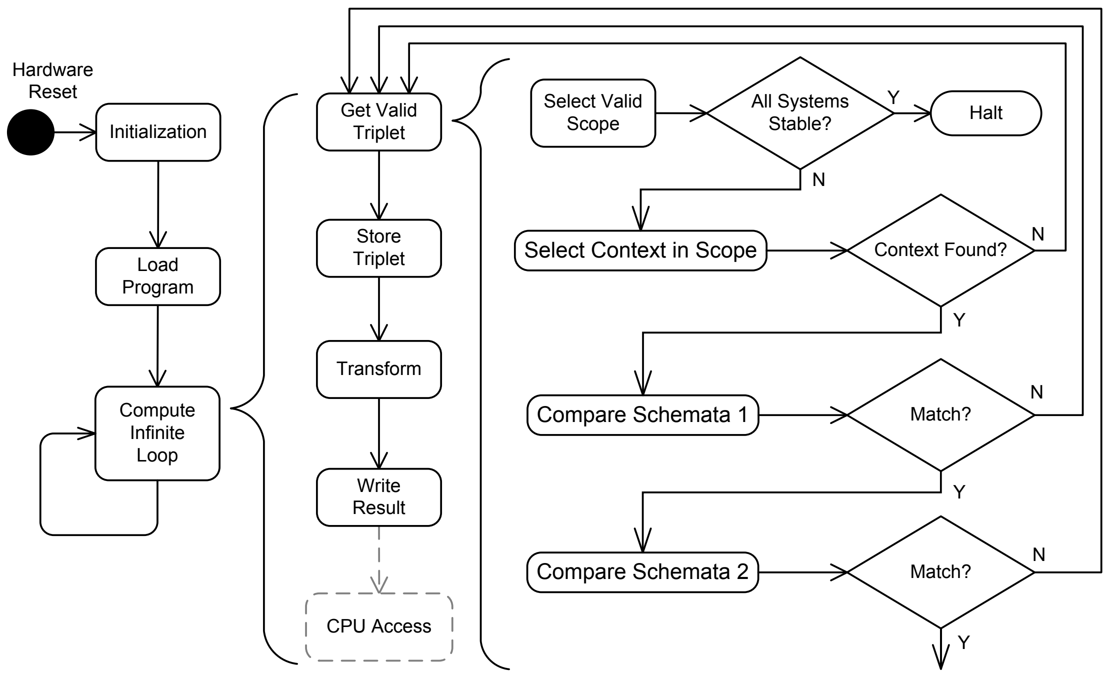


Figure 3.10. HAoS Program Control Flow : HAoS enters an infinite computation loop after the SC program is loaded, which involves finding valid triplets and transforming the selected systems

Various optimizations have been applied in order to ensure optimal performance. When the selected context system gives a mismatch, meaning that both its schemata do not match any two systems in the scope, it gets disabled and becomes an invalid context for this scope to prevent future mismatches (see next section). Moreover, once a scope is selected, if it contains less than three systems or of it does not contain any valid contexts (any contexts that have not recently given a mismatch), it also gets disabled and becomes an invalid scope until a new system is added to it. If all scopes have been disabled, no further transactions can occur and the program halts.

3.6 The SC Core

The CORE is mainly responsible for the efficiency of the design due to the way it handles the task of schemata matching. Its main components are the various memory elements including the TCAM, the system memories, the scopetable memories, the system status registers and the random selection logic, as illustrated in Figure 3.11.

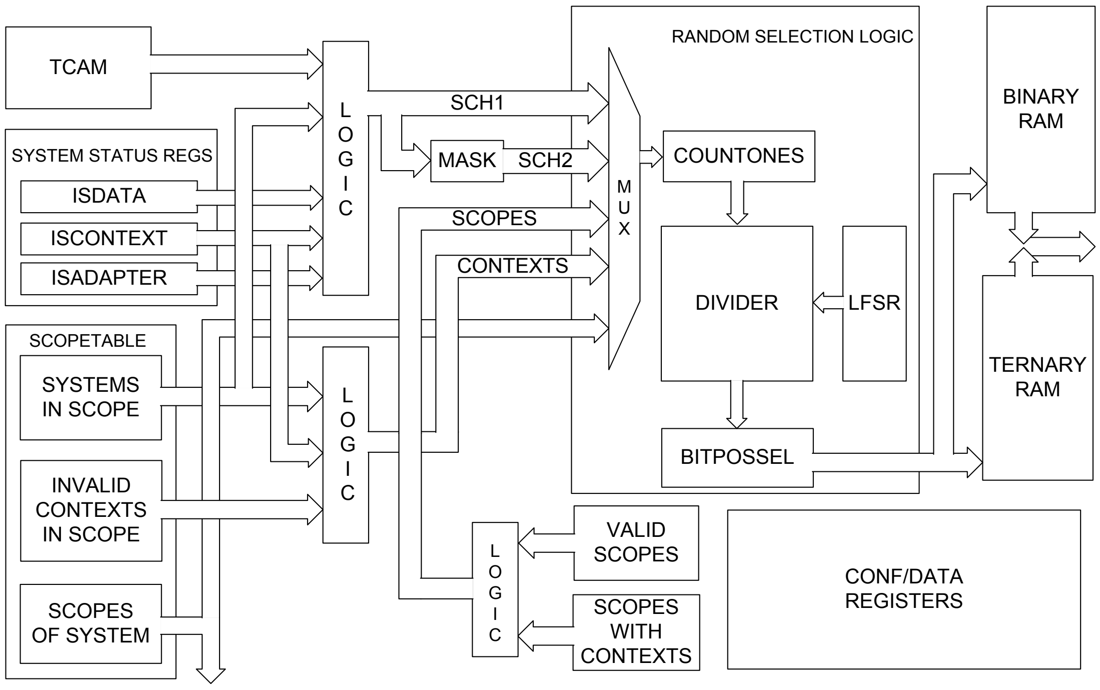


Figure 3.11. HAoS Core basic building blocks

3.6.1 The Core Memory Elements

The full contents of a system are stored in two separate RAMs, one of them holds the binary part while the other stores the ternary part (the “don't care” bits). Since the function part of a system is always binary, it is not stored in the ternary RAM. The various parts of a system are located in the same address in all memories in order to simplify the required address-decoding logic.

The global scopetable information is stored in three RAM-based structures. One of them stores the systems that belong in each scope at the corresponding to the scope address, the second stores the scopes that each system belongs to at the corresponding to the system address while the third stores a mask for all the invalid contexts in a scope. The first two structures, although effectively storing the same information, provide parallel access to two different aspects of the scopetable (systems in scope and parent scopes of a system).

The TCAM is loaded with the regions of the systems, which may be compared (see Figure 3.12), during initialization. For data systems, the function part is always zero, so only the binary representation of their two schemata may be compared while for context systems only their function part (which is double the size of a schema) may be compared. This implies that context systems can interact with other context systems or data systems, which greatly enhances functionality since it denotes that context adapting (where context systems can interact with other systems and be changed) is supported (a feature only supported previously in the highly flexible SCoPE implementation). Context adapter systems may not interact with other systems in HAoS. The restriction of comparing only parts of a system is posed by the fact that the TCAM resource requirements increase really fast when the maximum number of supported systems is scaled up.

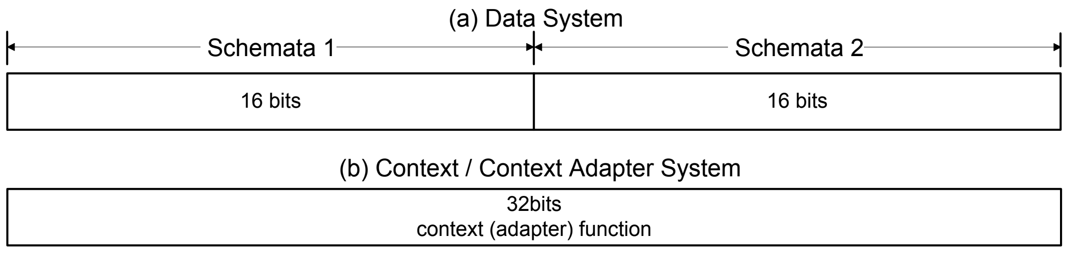

(a) Data System
Schemata 1
Schemata 2
16 bits
16 bits
(b) Context / Context Adapter System
32bits
context (adapter) function

Figure 3.12. The TCAM contents: Filled with the systems' regions that may be compared

3.6.2 The Random Selection Logic

The random selection logic (RSL) accepts a bus as an input and returns the address of a randomly selected set bit. It consists of an optimized module that counts the set bits of the bus, a maximal-length Linear Feedback Shift Register (LFSR) for pseudo-random number generation, a combinatorial divider (which also performs integer division when required in the Transform state - see Figure 3.10) and a module (BITPOSSEL) that given a bus and the rank of a set bit of this bus (the position of the set bit with rank 2 is 3 in 0100*1*101 - when rank starts from 0 and position 0 is the rightmost one), it returns its position (inspired from an optimized implementation found in [184], combining a parallel bit count and branchless selection method). A random number, provided by the LFSR, is divided by the sum of the set bits of the bus. The remainder of this division is used as the rank of the random set bit that is given to BITPOSSEL in order to identify its position.

Counting the Set Bits

The COUNTONES block design implements a counter of the set bits of the input bus (also known as sideways sum or population count [185]) using a divide-and-conquer approach (inspired by a low-level software optimization presented in [184]). The parallel bit-count is performed in log_2N steps for an N-bit wide input bus (where N is a power of two). In each step, the sum of adjacent groups of bits is calculated - the length of the groups of the first level is 2 and is doubled for every successive step. In the final step, the total set bits sum is accumulated on the least significant bits of the bus.

This parallel bit count mechanism is illustrated in Figure 3.13 for an example 16-bit input bus. Adjacent bits are summed to formulate the 2-bit fields in step 1. The resulting pairs of bits are then summed to formulate the 4-bit bit-groups is step 2 which are in turn summed to give the 8-bit sums of step 3. This is repeated until the final step 4, when the final sum of set bits of the input bus (6 in this example) has been accumulated in the LSBs of the output.

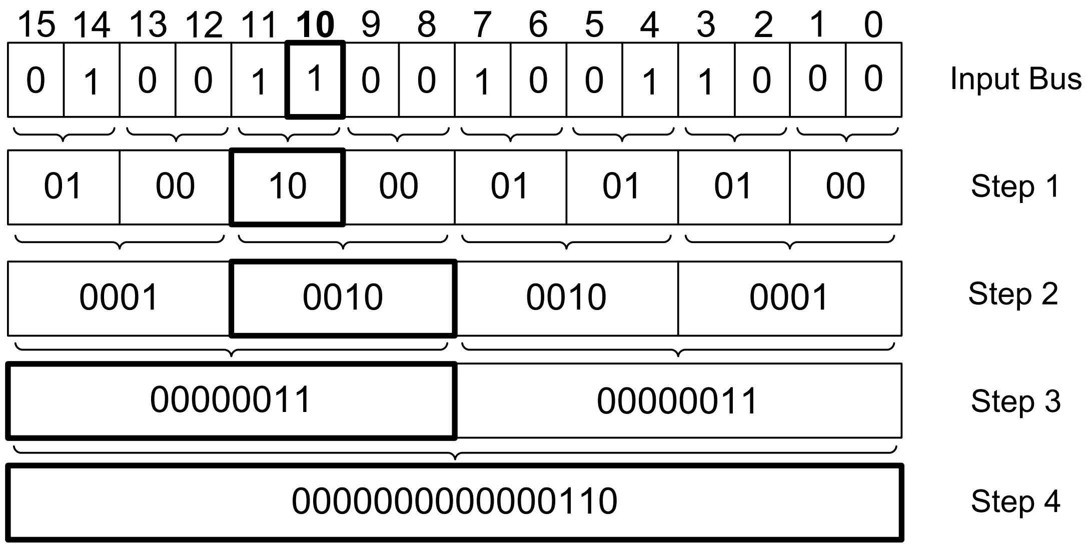
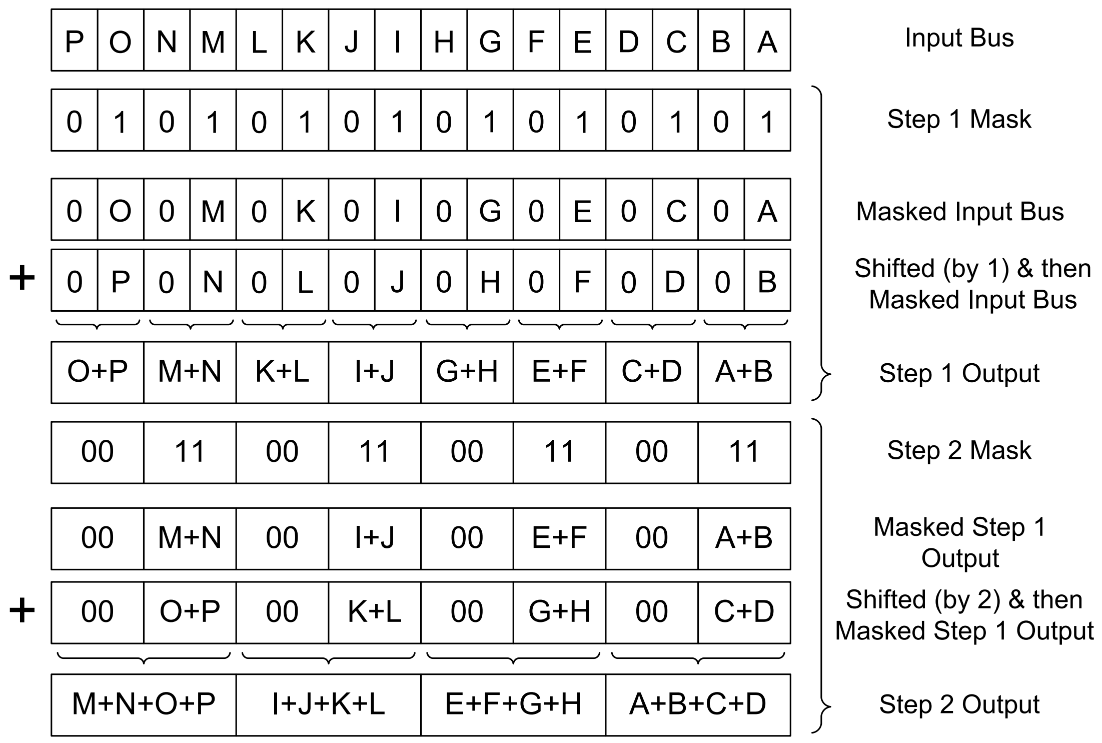


Figure 3.13. Parallel Bit Count Example. Adjacent bit groups are summed in successive steps until the sum of all set bits is accumulated in the LSBs of the output. Using the partial sums enables positioning a set bit given its rank (counting from right to left and starting from 0)

The summation of the adjacent bit groups is implemented by first masking the right group of bits in each bit group pair, then right shifting the bus by a number of bits equal to the length of each bit group for the specific step and then masking the shifted version and adding the two values.

This mask-and-shift approach is illustrated in Figure 3.14 explaining how the adjacent bit-groups are summed in the first two steps of the example of Figure 3.13. Each step has an associated mask (to isolate the target bit-group) and a related shifting constant. For step 1, the mask follows the pattern 0x0505 and the shifting constant is 1. The two versions of the input bus which are added to get the output of step 1 are obtained: one by ANDing it with the mask and the other by shifting it by the shifting constant (1) and then masking it. The same actions are performed in each step. As shown in Figure 3.14, in step 2 the mask follows the pattern 0x0303 and the shifting constant is 2.



P O N M L K J I H G F E D C B A
Input Bus
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Step 1 Mask
0 O 0 M 0 K 0 I 0 G 0 E 0 C 0 A
Masked Input Bus
+
0 P 0 N 0 L 0 J 0 H 0 F 0 D 0 B
Shifted (by 1) & then Masked Input Bus
O+P M+N K+L I+J G+H E+F C+D A+B
Step 1 Output
00 11 00 11 00 11 00 11
Step 2 Mask
00 M+N 00 I+J 00 E+F 00 A+B
Masked Step 1 Output
+
00 O+P 00 K+L 00 G+H 00 C+D
Shifted (by 2) & then Masked Step 1 Output
M+N+O+P I+J+K+L E+F+G+H A+B+C+D
Step 2 Output

Figure 3.14. Shift-and-Add implementation of the parallel bit-count mechanism (only two steps shown). A mask and a shifting constant correspond to each step. Two versions of the input of each step are obtained and added: one by masking it and the other by first shifting it and then masking it

Effectively with this method we position the left bit group in each adjacent bit group pair under the right one in order to perform the addition of their set bits. The approach is scalable to any input bus width. However, the implied adder tree for long input buses will increase the latency of the unit when implemented in a purely combinatorial way. However, this does not impose a problem, as the critical path of the COUNTONES block can be refined using pipelining later, being fine-tuned according to the critical path of the whole design.

The Divider

The hardware divider implements a slightly modified restoring division algorithm. Restoring division algorithms [186] compare part of the dividend with the divisor and when that specified part of the dividend is greater than the divisor (their difference is positive), they set the corresponding bit in the quotient and pass the difference in the next stage. If the difference is negative, the result is restored to the value of the partial dividend prior to the subtraction before being passed in the next stage. In the hardware divider of the RSL, both the comparison and the subtraction are performed by the same logic. Left-shifted versions of the divisor are given as input to the successive (16 in this prototype) stages of the division logic, each stage being responsible for generating one bit of the quotient. The remainder of the division is the (positive) comparison result of the last stage. Where typical restoring division implementations add the divisor back to a negative comparison result in order to restore a negative intermediate result to a positive value (as only positive values are propagated to the next stage), in this design a multiplexer is used instead. A block diagram of the divider (excluding some logic handling signed numbers), its individual stages and their corresponding inputs are given in Figure 3.15.

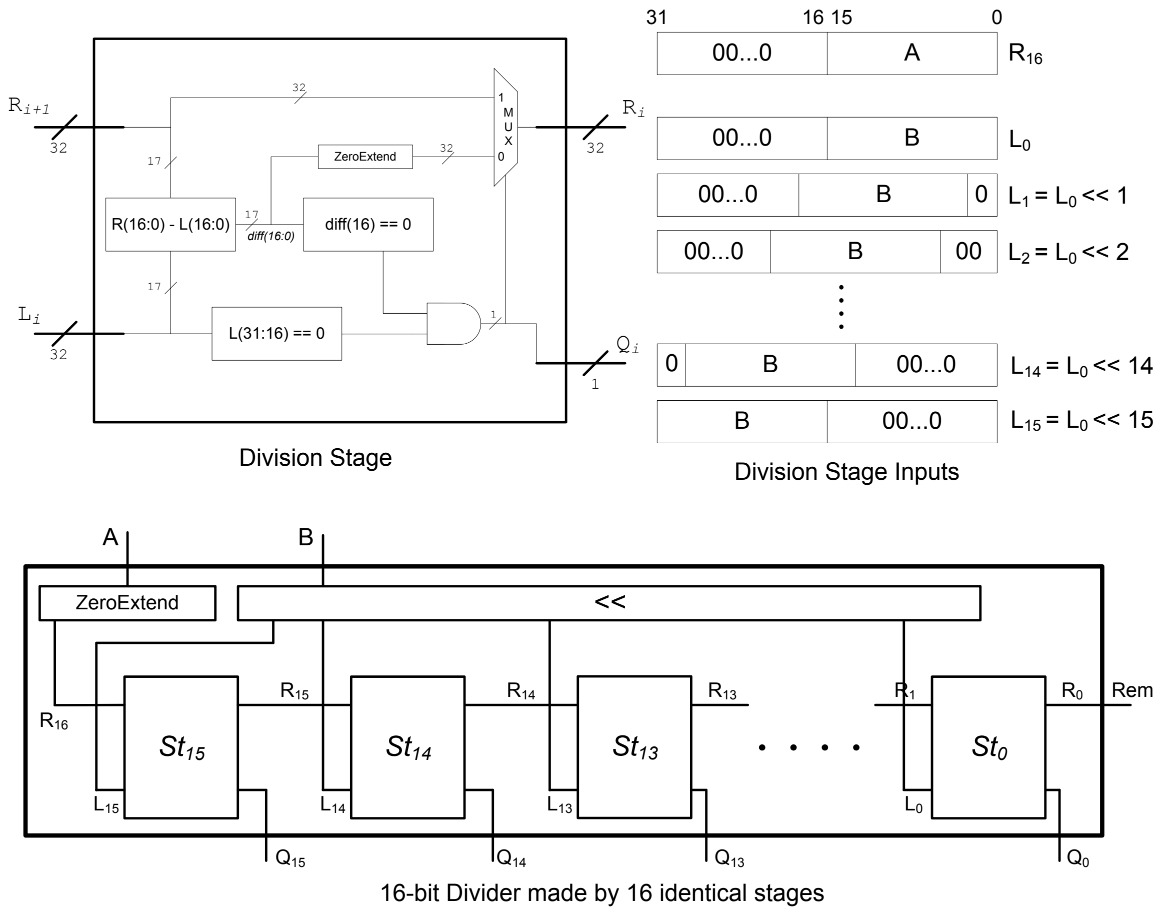


Figure 3.15. HAoS Divider based on a modular approach. Each stage gives one bit of the quotient. The design essentially unrolls the loop of the classic shift-and-subtract method and can be further fine-tuned to balance its latency and throughput

As opposed to restoring division algorithms, non-restoring algorithms waive the restriction of accepting only positive partial remainders, saving the restoring step. However these algorithms need an additional step in the final stage to restore a possible negative final remainder. Both restoring and non-restoring algorithms belong in the digit recurrence family of algorithms [186] which generally rely on subtraction to perform division. Other approaches involving pre-normalization of the division operands (SRT algorithms) and/or use of higher radixes to give multiple quotient bits in each step are also commonly found in the literature along with division algorithms based on multiplication (division by convergence or reciprocation using Taylor series expansion and Newton-Raphson or Goldschmit approximation) [187][188]. The non-restoring algorithm was selected to implement the required divisions in HAoS due to its simplicity, scalability and the ability to easily fine-tune its critical path according to the overall latency of the complete design at a later stage.

Random Number Generation

The choice of using an LFSR for random number generation is discussed in section 3.2.7. The implementation of the LFSR is straight-forward as it typically involves a shift register either with a feedback line to one of its most or least significant bits, resulting by XORing some of its bits, called taps, for external feedback (Fibonacci LFSRs), or XORing the bits in the tap positions with the serial output resulting this way in internal feedback (Galois LFSRs) [174]. The arrangements of the taps correspond in finite field arithmetic to a polynomial mod 2 (its coefficients may be 0 or 1). The LFSR will be maximal-length (with maximum period before the output repeats) for a set of well-defined primitive polynomials [189]. HAoS uses a Fibonacci LFSR and its design ensures the maximal-length property for any number of maximum supported systems as it always implements an appropriate primitive polynomial.

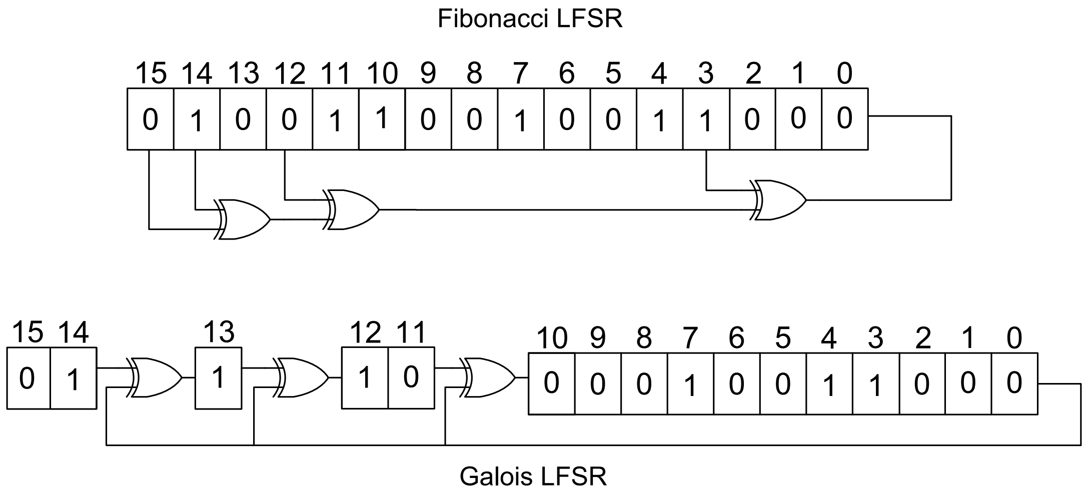


Figure 3.16. 16-bit Fibonacci and Galois type LFSRs

Finding the position of a set bit given its rank

The BITPOSSEL block returns the position of a set bit of its input bus given its rank again based on a divide-and-conquer approach, similar to the one discussed above for counting the set bits. Using Figure 3.13 to explain the operation of BITPOSSEL for an example input bus of 16 bits, we again perform a parallel bit count keeping all the intermediate steps. As seen in Figure 3.13 the input value is 0100110010011000. As previously mentioned, the desired rank is the remainder of the division of a random number from the LFSR with the sum of set bits from COUNTONES. For this particular example, we will suppose that the desired rank is 3 (starting counting from 0).

Since the partial sums are known for each step in Figure 3.13 (the selected bit-groups in the following analysis are highlighted with a bold outline), starting from bottom up (from the last step), we set a virtual pointer (which will hold the position of the desired bit according to the rank in the end) at position 0 and then the desired rank is compared with the total number of set bits. Since the rank is less than the total sum, we move to the previous step (if it was greater or equal, that would imply that there would not be a set bit corresponding to the requested rank). Looking at step 3 of Figure 3.13, three set bits (00000011) exist on the left half of the bus and another three on the right half. If the requested rank is less than the bit sum of the right group, it means that the targeted set bit is part of that group. In this case we would select that group and leave the virtual pointer unchanged. However, since for our example the desired rank is 3, this implies that the requested bit lies on the left half. When the desired rank is greater or equal to the sum of the right (least significant) bit group, we select the left group, we move the virtual pointer at the middle (add to it a value equal to the length of each bit group at the current step - at step 3 the length is 8) and subtract the bit sum of the right half from the rank. So, now the virtual pointer gets the value *old virtual pointer + bit-group length = 0 + 8 = 8* and the rank becomes *old rank - right group sum = 3 - 3 = 0.*

Following the same methodology, moving to the previous step (step 2), the rank (now 0) is less than the bit sum of the right part (which is equal to 2), so the right part is selected and both the rank and the virtual pointer remain unchanged (rank = 0, pointer = 8). In the next step up (step 1 with bit-group length 2), the rank is equal to the sum of the right part (which is 0), so the left part is selected, the rank becomes *old rank - right group sum = 0 - 0 = 0* and the virtual pointer gets the value *old virtual pointer + bit-group length = 8 + 2 = 10.* In the last step examining the input bus, the rank (0) is less than the right bit (1), so that bit is selected and the virtual pointer remains unaltered giving its position (10).

As easily noticed, this bit is the targeted bit with rank 3 and the virtual bit position pointer contains its location (counting from right to left the bit with rank 0 is at position 3, the bit with rank 1 is at position 4 and bit with rank 2 is at position 7).

RSL functionality

The function of the RSL (the result of the selection) is controlled by a multiplexer (MUX) which feeds the RSL with one out of five possible input buses (see Figure 3.11). When we need to choose a system that matches the first schema of the context, the input bus (SCH1) is generated by ANDing the output of the TCAM with valid SYSTEMS IN SCOPE (which of them are valid depends on the type of the context system and is identified based on the SYSTEM STATUS REGS). The same bus is used for the second schema match (SCH2) after masking out the selected system for SCH1 (a system may not interact with itself). When a random scope is needed the input bus (SCOPES) is the result of ANDing valid scopes (scopes with more than two systems) with scopes with contexts (scopes that are not disabled at that time). Finally, when we need to randomly identify a context in a previously selected scope, the input bus of the SRL is generated by ANDing the contexts in the scope (ISCONTEXT status register AND SYSTEMS IN SCOPE) with INVALID CONTEXTS IN SCOPE (in order to mask out previously used contexts that resulted in a mismatch). The fifth input of the MUX serves a low-level optimization for the ESCAPE task, as mentioned in section 3.3.

3.7 Programming HAoS

The HAoS programming model is based on the one of the original implementation [24]. This decision was made in order to retain backwards compatibility with prior implementations and take advantage of the available SC language definition and accompanying compiler. The SCoPE platform [190] was also considered, but it was decided that the original version was more suitable for the prototype HAoS architecture due to its simplicity and more hardware-suitable resulting assembly code. However, some functionality of the SCoPE platform (like high-level function plugins generation) is supported by HAoS to increase its user-friendliness and flexibility. The SC source code (see Listing 3.1) of the simple PRINT((A1-A2)*(A3-A4)) program that was discussed in section 2.3 (Figure 2.11A-C) is given below as a programming example.

The user should first state the transformation functions which are embedded in the context and context adapter systems of the program. This is done by using the keyword “function”, the name of the function and its 32-bit binary opcode (Listing 3.1, lines 4-6).

```
1.  #systemic start
2.
3.  // define the functions
4.  #function SUBTRACTe %b01000000000000000000001000000000
5.  #function MULT      %b11000000000000000000000000000000
6.  #function PRINT     %b01000000100000000000000000000000
7.
8.  // define some useful labels
9.  #label dontcare   %b????????????????
10. #label num1       %b1000000000000000
11. #label num2       %b0100000000000000
12. #label num3       %b1100000000000000
13. #label num4       %b0010000000000000
14. #label scp        %b1111111111111111
15.
16. #label zero       %b00000000000000000000000000000000
17.
18. // and the program begins here:
19. main (scp %d0 %d0) // system 0
20. // system 1
21. minus ([num1 zero dontcare] SUBTRACTe(0,0) [num2 zero dontcare])
22.
23. c1 (scp %d0 %d1)       // system 2
24. data1 (num1 %d0 %d10) // system 3
25. data2 (num2 %d0 %d3)  // system 4
26.
27. #scope c1
28. {
29.    data1
30.    data2
31.    minus    // 10-3=7
32. }
33.
34. c2    (scp %d0 %d2)    // system 5
35. data3 (num1 %d0 %d16) // system 6
36. data4 (num2 %d0 %d4)  // system 7
37.
38. #scope c2
39. {
40.    data3
41.    data4
42.    minus      // 16-4=12
43. }
44.
45. // system 8: 12*7=84
46. times  ([num1 zero dontcare] MULT(0,0) [num1 zero dontcare])
47. output ([num1 zero dontcare] PRINT(0,0) [num1 zero dontcare]) //sys 9
48.
49. #scope main
50. {
51.    c1
52.    c2
53.    times
54.    output
55. }
56.
57. #systemic end
```

Listing 3.1. HAoS Source Code Example: PRINT((10-3)*(16-4))

Then the user can optionally define labels (Listing 3.1, lines 9-16), equivalent to constants of conventional programming languages, which can be used instead of frequently used immediate values. Then, the systems and scopes are defined. Data systems are defined by their name and the values of their schemata while their function part is always zero (Listing 3.1, lines 19, 23-25, 34-36). Context systems define their schematas either by using the data system definition method (Listing 3.1, lines 21, 46-47) or by referencing other data systems.

```
// number of functions
3
// number of systems
10
// scope table
0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
// function definitions
SUBTRACTe 01000000000000000000001000000000
MULT 11000000000000000000000000000000
PRINT 01000000010000000000000000000000
// system definitions
1111111111111111 00000000000000000000000000000000 0000000000000000
@!!!!!!!!!!!zzzz 01000000000000000000001000000000 /!!!!!!!!!!!zzzz
1111111111111111 00000000000000000000000000000000 1000000000000000
1000000000000000 00000000000000000000000000000000 0101000000000000
0100000000000000 00000000000000000000000000000000 1100000000000000
1111111111111111 00000000000000000000000000000000 0100000000000000
1000000000000000 00000000000000000000000000000000 0001000000000000
0100000000000000 00000000000000000000000000000000 0010000000000000
@!!!!!!!!!!!zzzz 11000000000000000000000000000000 @!!!!!!!!!!!zzzz
@!!!!!!!!!!!zzzz 01000000010000000000000000000000 @!!!!!!!!!!!zzzz
```

Figure 3.17. Human-readable HAoS Assembly Code for PRINT((10-3)*(16-4)) Example Program

Their transformation function is defined by referencing one of the declared functions. Context adapter systems are defined as context systems do, but their schemata can also be a context system prototype (having a non-zero function). Two numeric fields (in parentheses) follow the function of a system. These were used in the original version to define the matching thresholds and are preserved here for backwards compatibility. All functions support the (matching polarity) NOT functionality (see section 3.3) by having an exclamation mark following the parentheses. By convention, all functions that include the ESCAPE functionality (see end of section 3.3) have the suffix -e (SUBTRACTe denotes the ESCAPE-enabled SUBTRACT function).

A scope is defined by the "scope" keyword and the systems that belong to it in brackets (Listing 3.1, lines 27-32, 38-43 and 49-55). It is noted that all systems, regardless of their type, have a scope, meaning the ability of including other systems within them. If the scope of a system is not defined in the program, then this system does not contain any other system, and all its corresponding entries in the scopetable are zero.

When the source code of Listing 3.1 is compiled, the annotated HAoS assembly language of Figure 3.17 is generated. As mentioned in the compiler related discussion of section 3.2, the parts of the assembly code that are loaded on HAoS are the global scopetable and system definitions. The scopetable, which contains a number of rows and columns equal to the maximum number of supported systems (equal to 64 for this prototype HAoS implementation), is partially shown in Figure 3.17 as the remaining entries are all zero. Context schemata are compressed according to the mapping given in Table 3.1.

```
SUBTRACTe {HAoS}:
sys1@3(sch1:1,k:0,sch2:10) - sys2@4(sch1:2,k:0,sch2:3) =>
sys1(sch1:1,k:0,sch2:7),sys2(sch1:2,k:0,sch2:0)  <sc:2,cxt:1,it:1>
SUBTRACTe{HAoS}:ESC:sys(3) from scope(2) to scope(s)(pos:0)
<sc:2,cxt:1,it:1>

SUBTRACTe {HAoS}:
sys1@6(sch1:1,k:0,sch2:16) - sys2@7(sch1:2,k:0,sch2:4) =>
sys1(sch1:1,k:0,sch2:12),sys2(sch1:2,k:0,sch2:0)  <sc:5,cxt:1,it:2>
SUBTRACTe {HAoS}: ESC : sys(6) from scope(5) to scope(s)(pos:0)
<sc:5,cxt:1,it:2>

PRINT     {CPU}: sys2@6{12:-:1}, sys1@3{7:-:1} <sc:0,cxt:9,it:3>

PRINT     {CPU}: sys2@3{7:-:1}, sys1@6{12:-:1} <sc:0,cxt:9,it:4>

PRINT     {CPU}: sys2@3{7:-:1}, sys1@6{12:-:1} <sc:0,cxt:9,it:5>

PRINT     {CPU}: sys2@3{7:-:1}, sys1@6{12:-:1} <sc:0,cxt:9,it:6>

PRINT     {CPU}: sys2@6{12:-:1}, sys1@3{7:-:1} <sc:0,cxt:9,it:7>

MULT      {HAoS}: sys1@6(sch1:1,k:0,sch2:12)* sys2@3(sch1:1,k:0,sch2:7)
=> sys1(sch1:1,k:0,sch2:84),sys2(sch1:1,k:0,sch2:1)  <sc:0,cxt:8,it:8>

PRINT     {CPU}: sys2@6{84:-:1}, sys1@3{1:-:1} <sc:0,cxt:9,it:9>

PRINT     {CPU}: sys2@6{84:-:1}, sys1@3{1:-:1} <sc:0,cxt:9,it:10>

MULT      {HAoS}: sys1@6(sch1:1,k:0,sch2:84)* sys2@3(sch1:1,k:0,sch2:1)
=> sys1(sch1:1,k:0,sch2:84),sys2(sch1:1,k:0,sch2:1)  <sc:0,cxt:8,it:11>

MULT      {HAoS}: sys1@6(sch1:1,k:0,sch2:84)* sys2@3(sch1:1,k:0,sch2:1)
=> sys1(sch1:1,k:0,sch2:84),sys2(sch1:1,k:0,sch2:1)  <sc:0,cxt:8,it:12>
```

Figure 3.18. HAoS Sample Output from the Simulation Environment for the PRINT((10-3)*(16-4)) Example Program

Finally, a sample of the output (with extended debugging information obtained by the verification environment - section 3.8.1) of the example program, discussed above, is shown in Figure 3.18. The annotations from Figure 3.18 and Listing 3.1 are given in different font without parentheses in the analysis below for clarity.

At iteration 1 `it:1`, the system defined at position 1 (indexes start at 0), called from now on system 1 for simplicity, is selected as context `cxt:1` in the scope of the system at position 2, called from now scope 2, `sc:2`. System 1 is the second system under "systems definitions" in Figure 3.17 and corresponds to the system which is called `minus` and defined at line 21 of Listing 3.1. Once the context and active scope systems are selected HAoS performs a number of actions, listed below:

- Two data systems are selected according to the schemata of the `minus` context.
- System 3 is selected as the first interacting system `sys1@3` because its definition `data1 (num1 %d0 %d10)`, Listing 3.1, line 24 - also found as `(sch1:1,k:0,sch2:10)` in Figure 3.18 - matches the prototype which is defined by schema 1 of the `minus` context `[num1 zero2 dontcare]`, Listing 3.1, line 21.
- System 4 is selected as the second interacting system `sys2@4` because its definition, `data2 (num2 %d0 %d3)`, Listing 3.1, line 25 - shown as `(sch1:2,k:0,sch2:3)` in Figure 3.18, matches the prototype which is defined by schema 2 of the `minus` context `[num2 zero2 dontcare]`, Listing 3.1, line 21.
- After the subtraction (10-3=7), the result is stored in the first interacting system `=> sys1(sch1:1,k:0,sch2: 7)` while the second interacting system gets value zero `sys2(sch1:2,k:0,sch2:0)` according to the operation of SUBTRACT instruction in Table 3.4.
- However, since the transformation function is SUBTRACTe, the first interacting system, which is system 3 (`ESC:sys(3)`), escapes from the active scope, which for this iteration is scope 2 `from scope(2)` to the scope(s) that the active scope belongs to (see Figure 2.11B). According to the third line of the scopetable (see Figure 3.17), this is scope 0 `to scope(s)(pos:0)` because there is only one bit set in this line, which is the line which corresponds to the scope of system 2, and this bit is at position 0. If more than one bit were set, this would indicate that the active scope would belong to more than one scope, and it would escape to all of them.
- Finally, since system 3 has escaped from scope 2, or scope c1 (Listing 3.1, line 27), this scope now contains only two systems (`data2` and `minus`) which cannot

define a triplet and, thus, no interaction can happen in it. HAoS detects this and disables scope 2.

In a similar way, skipping the detailed analysis, during iteration 2 `it:2`, system 6 `sys1@6(sch1:1,k:0,sch2:16)` interacts with system 7 `sys2@7(sch1:2,k:0,sch2:4)` by means of subtraction, and the result (16-4=12) is stored in system 6 `=>sys1(sch1:1,k:0,sch2:12)` while system 6 also escapes from the active scope 5 `ESC : sys(6) from scope(5)` to scope 0. Scope 5 now contains only two systems, so it is disabled.

Since scopes 2 and 5 have been disabled, interactions can only occur in scope 0. While the two previous SUBTRACTe interactions were executed on HAoS (`{HAoS}:` in Figure 3.18), the next five are PRINT interactions, which just show the contents of the interacting systems, are executed on the CPU (`{CPU}:`). Each system is printed according to `interacting system @ position (schemata2 : transformation function (-if zero) : schemata 1`. Eventually the `times` context (Listing 3.1, line 46) is selected, system 6 interacts with system 3 and the expected product (12*7=84) is stored in system 6 (see Figure 2.11C). From then on, the systems in scope 0 continue interacting for ever without further noteworthy changes to their contents.

3.8 Initial Testing

Before the final design is implemented and tested in silicon, it is possible to verify its functional behaviour and assess its performance by using standard industry EDA tools. The selected FPGA evaluation board to implement HAoS is the Xilinx ML605 board. HAoS was described in VHDL and synthesized targeting the on-board Virtex-6 LX240T FPGA device by using the Xilinx ISE v13.3 design suite. The verification environment was written in SystemVerilog and Mentor Graphics QuestaSim was used for simulation.

3.8.1 Functional Verification

In order to achieve system-level functional coverage closure, a series of SC programs were designed to test and stress the design in various ways. The collection of these SC test programs is given in Table 3.5. As shown, basic (the core transformation functions and scope handling) and more advanced (context adapting, sequential flow emulation, high-level user-defined functions) functionality is verified, targeting mainly research challenge *Chg2* (SC architecture support).

An indicative set of test programs is further explained below and their SC source code is given in Appendix A (the source code for all test programs can be found in the official HAoS webpage [191]). For each reviewed test program, an excerpt from the verification environment output, validating the functional correctness of the design, and their corresponding SC graphical notations are given in Listing 3.2 and Figure 3.19 respectively. The three test programs, selected as verification examples here, are:

Table 3.5. Simulated SC test programs

Systems	Description of the SC Test Program	Functions Used
20	Additions in 4 different scopes	ADD
20	Dummy program testing all basic non-escape transformation functions in one scope. Eventually all systems are killed (zeroed)	All non-escaping core HW functions
20	Dummy program testing basic escape transformation functions. Systems interact and escape from various scopes into the same and then, they are printed	ADDe, MULTe, MODe, SUBe, DIVe, PRINT
20	Systems escape from scope. Then multiplied & result printed	ESCAPE, MULT, PRINT
20	Systems subtract & escape from scope. Then multiplied & result printed	SUBe, MULT, PRINT
9	Incrementing counter example (testing capture functionality)	ADDe, PRINT, CAPTURE, OR
6	Optimized incrementing counter using scopes to control the sequential flow	ADDuc2e, CAPTURE
4	Extra optimized incrementing counter using just one context	ADDuc2
12	Systems escape and multiplied based on NOT functionality	ESCAPE, MULT, PRINT
24	Two systems subtract-escape from different scopes to main scope and then they are recaptured back in the same initial sub-scopes	SUBe, CAPTURE, MULT, PRINT, ESCAPE
24	Same as above but also testing a scope included in itself	same as above
25	Subtraction context systems are transformed to addition context systems by a COPY context adapter	ADD, SUB, COPY
37	Subtraction-escape context systems are transformed to addition systems by COPY context adapter	ADD, SUBe, COPY
39	Subtraction systems are transformed to addition systems by a context adapter and then they are killed (transformed to NOP)	ADD, SUB, COPY
41	Subtraction systems are transformed to addition systems by a context adapter and then they are transformed back to subtraction systems	ADD, SUB, COPY
37	Mixed-mode context adapter transforms subtraction contexts to data systems which interact with other data systems	ADD, SUB, COPY
37	Mixed-mode context adapter transforms subtraction contexts to data systems and then retransforms the data systems back to context systems	ADD, SUB, COPY, ZERO
33	Part of schemata 1 of a context is changed. This change makes it match (previously unmatching) data systems	ADD, ZERO
12	Fibonacci numbers generator (using a special add-and-exchange context)	ADDxce, COPY, PRINT, CAPTURE
58	A 16-element binary knapsack problem solver based on a genetic algorithm[22]	user-defined: INIT, OUTPUT, CROSSOVER, MUTATE

[22] Further explained in section 5.1

- *Test1*: Tests simple interactions in multiple scopes (using systems in 4 different scopes). The expected sums are 1024, 1155, 1187 and 1200 for scopes 0, 1, 2 and 3 respectively.
- *Test2*: Tests interactions involving escaping. Systems resulting after subtraction in scopes c1 (10-3=7) and c2 (16-4=12) escaping to the parent scope main and get multiplied to give 84 as a final result.
- *Test3*: Tests context adapting - transformation of context systems through a context adapter. Here, a COPY adapter transforms subtraction contexts into addition by performing binary copy of their contents, so after all the transformations only addition interactions are possible.

```
Selected Output from Test1:
#431820ns :: PRINT {CPU}: sys2@17{1024:-:1}, sys1@16{0:-:1} <sc:0,cxt:19,it:19>
#489260ns :: PRINT {CPU}: sys2@9{1187:-:1}, sys1@10{0:-:1} <sc:2,cxt:19,it:32>
#611340ns :: PRINT {CPU}: sys2@7{1200:-:1}, sys1@5{0:-:1} <sc:3,cxt:19,it:61>
#789420ns :: PRINT {CPU}: sys2@15{1155:-:1}, sys1@14{0:-:1} <sc:1,cxt:19,it:104>

Selected Output from Test2:
#427580ns ::SUBTRACTe{SCC}: sys1@11(sch1:1,k:0,sch2:16) SUB sys2@12(sch1:2,k:0,sch2:4)
=> sys1(sch1:1,k:0,sch2:12),sys2(sch1:2,k:0,sch2:0)  <sc:10,cxt:9,it:1>
#427580 ns ::SUBTRACTe{SCC}: ESC : sys(11) from scope(10)  <sc:10,cxt:9,it:1>
...
#436620ns ::SUBTRACTe{SCC}: sys1@2(sch1:1,k:0,sch2:10) SUB sys2@3(sch1:2,k:0,sch2:3) =>
sys1(sch1:1,k:0,sch2:7),sys2(sch1:2,k:0,sch2:0)  <sc:1,cxt:9,it:4>
#436620ns ::SUBTRACTe{SCC}: ESC : sys(2) from scope(1)  <sc:1,cxt:9,it:4>
...
#443420ns ::MULT {SCC}: sys1@2(sch1:1,k:0,sch2:7) TIMES sys2@11(sch1:1,k:0,sch2:12) =>
sys1(sch1:1,k:0,sch2:84),sys2(sch1:1,k:0,sch2:1)  <sc:0,cxt:18,it:7>
#468940 :: PRINT  {CPU}: sys2@2{1:-:1}, sys1@11{84:-:1} <sc:0,cxt:19,it:14>

Selected Output from Test3:
#602540ns ::SUBTRACT {SCC}: sys1@13(sch1:3,k:0,sch2:1030) SUB sys2@5(sch1:3,k:0,sch2:110)
=> sys1(sch1:3,k:0,sch2:920),sys2(sch1:3,k:0,sch2:0)  <sc:0,cxt:19,it:1>
# 606220ns ::SUBTRACT {SCC}: sys1@6(sch1:3,k:0,sch2:120) SUB
sys2@12(sch1:3,k:0,sch2:1020) => sys1(sch1:3,k:0,sch2:-900),sys2(sch1:3,k:0,sch2:0)
<sc:0,cxt:21,it:2>
...
#656us ::COPY{SCC}:sys1@22(sch1:61441,k:2,sch2:61441) CP
sys2@23(sch1:61441,k:1,sch2:61441) =>
sys1(sch1:61441,k:1,sch2:61441),sys2(sch1:61441,k:1,sch2:61441)  <sc:0,cxt:24,it:17>
#687us ::COPY{SCC}:sys1@19(sch1:61441,k:2,sch2:61441) CP
sys2@22(sch1:61441,k:1,sch2:61441) =>
sys1(sch1:61441,k:1,sch2:61441),sys2(sch1:61441,k:1,sch2:61441)  <sc:0,cxt:24,it:27>
...
#749260ns ::ADD  {SCC}: sys1@12(sch1:3,k:0,sch2:0) PLUS sys2@13(sch1:3,k:0,sch2:940) =>
sys1(sch1:3,k:0,sch2:940),sys2(sch1:3,k:0,sch2:0)  <sc:0,cxt:21,it:46>
#752940ns ::ADD  {SCC}: sys1@7(sch1:3,k:0,sch2:0) PLUS sys2@11(sch1:3,k:0,sch2:0) =>
sys1(sch1:3,k:0,sch2:0),sys2(sch1:3,k:0,sch2:0)  <sc:0,cxt:23,it:47>
#755980ns ::ADD  {SCC}: sys1@5(sch1:3,k:0,sch2:0) PLUS sys2@12(sch1:3,k:0,sch2:940) =>
sys1(sch1:3,k:0,sch2:940),sys2(sch1:3,k:0,sch2:0)  <sc:0,cxt:17,it:48>
```

Listing 3.2. Selected output from the 3 example SC test programs verifying the functionality of HAoS by simulation. Refer to section 3.7 to be reminded how to extract all the information from the output of the verification environment. In this listing, the parts that verify the functionality of the design according to the expected results, as they are given in the bullet descriptions of the test programs above, are emboldened: (Test1) the expected final result is printed in each correct scope, (Test2) the expected subtractions (`16 SUB 4 => 12` and `10 SUB 3 => 7`) and escapes (`ESC`) lead to the correct multiplication (`7 TIMES 12 => 84`) and the expected result (`84`) is printed (`PRINT`), (Test3) While initially only subtractions (`SUBTRACT`) are performed, the transformation function of addition (`k:1`) is copied (`CP`) in all subtraction contexts (`k:2`), so in the end only additions happen (`ADD`)

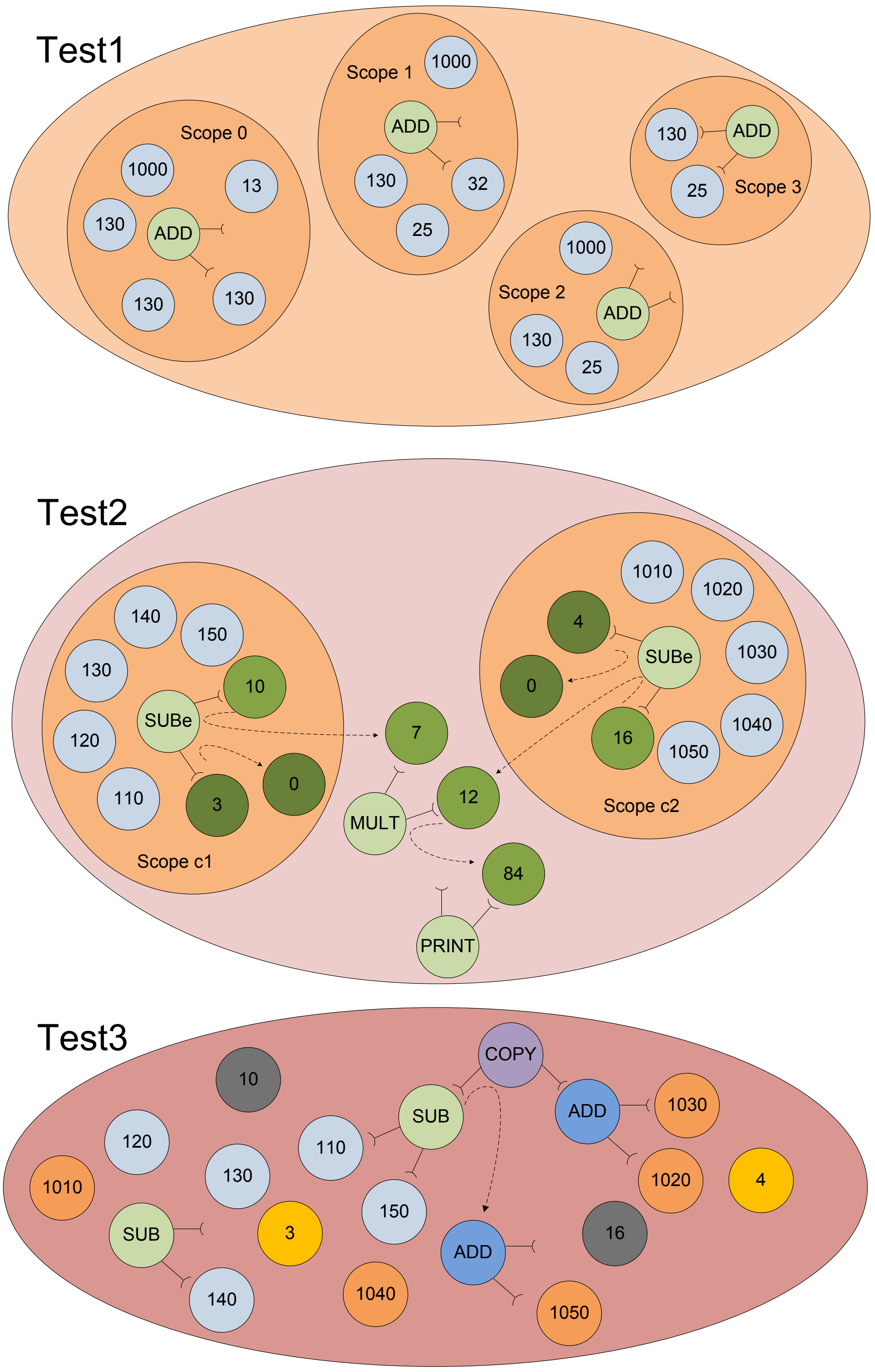


Figure 3.19. The three verification example SC programs

It is also noted that the equivalent of a conventional program (like a counter) can be executed by HAoS (sequential flow emulation). However, as the architecture is designed to model parallel systems, it can just simulate such sequential programs (experimental results show that although the prototype runs at approximately 1% of the speed of a modern Intel i7 core clocked at 2GHz, counts up in average 500 times slower).

Successful simulation results, similar to the ones given in Listing 3.2, have been obtained for all test programs of Table 3.5. Thus, the system passed all the functional verification tests, proving this way the validity of the design.

3.8.2 Implementation Statistics

Xilinx design tools provide accurate area and timing implementation statistics. Thus, we can present precise performance metrics before downloading the design on the FPGA. As shown in Table 3.6, the prototype design occupies just 15% of available slices (10% of slice LUTs and 1% of slice registers), 23% of available I/O blocks and just 1% of available RAM. HAoS is divided into two clock domains : the REG BANK, which is connected to the CPU INTERFACE (see Figure 3.9), runs on a higher clock rate (100 MHz) in order to provide faster read/write operations to the CPU, while the rest of the design is clocked in a (8 times) slower rate. The performance of the design of this initial stage is increased later by various enhancements and optimizations (detailed in the next chapter).

Table 3.6. HAoS Prototype (64 systems) Implementation Statistics on Virtex-6 LX240T FPGA. Excludes the CPU interface and the optional on-chip CPU

	Used	Available	%
Slices	5759	37680	15
Slice LUTs	15487	150720	10
Slice Registers	6019	301440	1
I/O Blocks	143	600	23
RAMs	5	416	1
DSP Blocks	1	768	1

3.9 Summary

In this chapter, the first Hardware Architecture of Systemic computation (HAoS) was introduced. An investigation was presented on how a hardware design can practically encompass the architectural properties of SC, addressing research challenge *Chg2*, while the support for several of the natural properties of Table 1.1 are also discussed, addressing *Chg1*. A number of FPGA-based potential architectures were initially considered while section 3.2 explained the design decisions that lead to the base HAoS system and instruction set, presented in sections 3.3-3.6.

HAoS is a custom but generic computer architecture implementing the SC paradigm. In contrast to conventional architectures that sequentially execute a set of instructions, it defines a pool of operands and operations, which in SC terms are systems and transformation functions, and detects in a parallel fashion which of them may result in an operation, or SC interaction, based on enabling patterns, or SC schemata, embedded in the operands. HAoS accomplishes this parallel detection by using a Ternary Content Addressable Memory, which matches templates of potential interacting systems to the available systems defined by the SC program.

A basic programming model was presented in section 3.7 while the functional behaviour of the first systemic processor is verified using a set of test programs, covering various scenarios, presented in section 3.8 along with initial implementation estimates. The base HAoS system is optimized and extended to a complete SC programming platform in the next chapter, resulting to an increase in efficiency and user-friendliness, and thus making our solution more practical and viable.

Chapter 4

The HAoS Programming Platform

Having verified the core functionality of the base design by simulation in section 3.8.1, the next steps are to investigate the most suitable, considering current technologies, implementation for the communication interface to the optional CPU, further optimize the performance of HAoS and complete the design by providing the CPU, its interface to HAoS and supporting development software in order to have a complete, viable and practical standalone SC programming solution.

Following those steps, this chapter addresses research challenges *Chg2* (SC specific architecture support) and *Chg3* (targeting a practical and efficient implementation), focusing mainly on *Chg3*, as special attention is given to the efficiency of the communication interface, the HAoS logic attached to it and various other blocks of the base design while the implementation of some of the architectural features of SC (schemata matching and random system selection) are revised. A set of software tools were developed to ease programmability and increase user-friendliness.

Part of the work presented in this chapter has been published in [159] and [192].

4.1 HAoS-CPU Communication Interface Investigation

As mentioned in section 3.4, the use of the CPU after the SC program is loaded is optional for the HAoS prototype and depends on the user processing requirements. Since HAoS on-chip processing capabilities are limited by the basic instruction set in Table 3.4, it is safe to assume that the CPU may be useful for a wide range of practical user applications. Thus, addressing the design practicality and overall system efficiency (*Chg3*), an investigation of the implementation of the communication interface between HAoS and the CPU, given below, is important in order to avoid having a communication overhead as the performance bottleneck.

The main design requirements for the communication link are high throughput, low latency and user-friendliness, meaning that it should be based on a widely used interface in order to minimize user effort. Since the maximum supported clock rate of our prototype is estimated at 100 MHz (based on the implementation statistics given by the Xilinx development tools) on the CPU INTERFACE boundary (see Figure 3.9), if we assume for simplicity that only single-byte data accesses are supported, a minimum data rate requirement of 100 MB/s is posed on the communication link in order to have full utilization. Latency is also crucial, as some off-chip communication interfaces may provide adequate bandwidth but nevertheless pose an unacceptable latency constraint.

We should further consider that the selected communication interface will determine the use of either an external more powerful CPU (using a commonly used but slower communication protocol) or a less powerful embedded (on-chip or on-board) CPU, using a relatively faster local bus. For a more realistic performance estimate, we should not only consider the maximum performance potential of the hardware but we should combine this with the actual response times caused by the software (operating system, drivers and user application programming interface implementation).

Another significant consideration is that the HAoS-CPU communication will comprise quite small packets. Typically these will be less than 10 bytes for control instructions (low-level accesses of HAoS control registers which will be frequently used by the driver and also offered as part of the API to the user to enhance accessibility) and considerably less than 100 bytes for data exchange (input and output arguments of the transform task, see Figure 3.10). The availability of IP cores to support these interfaces and the effort required for drivers development is also important. Finally, the selected interface should be supported by the available FPGA development board (in our case, the Xilinx ML605).

The external CPU option seems more appropriate since modern CPUs run more than one order of magnitude faster than embedded ones (the Intel i7 range runs typically at frequencies of 2-3GHz while the maximum frequency for a modern on-chip CPU, e.g. the Xilinx MicroBlaze, is 100-250 MHz [193]). The most commonly used communication interfaces for modern computers are USB, PCI-Express and Ethernet (see Table 4.1). All of them are mature technologies which are constantly revised to support greater bandwidths. While Hi-Speed USB (or USB 2.0) is currently the most widely adopted interface, it specifies a maximum bandwidth of 480 Mbits/s [194]. Its successor, SuperSpeed USB (or USB 3.0) specifies a maximum theoretical full-duplex communication rate of 5 Gbits/s [194]. PCI-Express, featuring a point-to-point topology with separate full-duplex byte streams (1-32 lanes) connecting the device to a root complex [195], has had four revisions that gradually increased bandwidth (the theoretical maximum per lane is [195]: 250 MB/s for the older Gen1, 500 MB/s for the widely used Gen2, 1GB/s for the more recent Gen3 and 2 GB/s for the recently announced Gen4). Gigabit Ethernet is the last option supporting 1 Gb/s while higher bandwidths (10G recently got more industrial interest, while 40G, 100G and 400G solutions are also currently sampled) are also supported for specialized network devices (usually using optical mediums).

Table 4.1. Commonly used interfaces for off-board communication and their nominal raw bandwidths

	USB		PCI-Express (per lane)				Ethernet
Revision	2.0	3.0	1.0	2.0	3.0	4.0	Gigabit
Nominal Max Raw Bandwidth	60 MB/s	625 MB/s	250 MB/s	500 MB/s	1 GB/s	2 GB/s	125 MB/s

The theoretical maximum bandwidth that the most recent versions of all the aforementioned interfaces provide appears to be sufficient for the HAoS-CPU data rate requirement. However, their sustained performance in a working system can be considerably less due to various software and hardware sources of overhead. An quantitative example is given in [196], where a bus mastering design (implemented on a Virtex-5 FPGA) over PCI Express is measured on a Windows system. Sustained software performance can be nearly 17 times slower than the theoretical maximum for a PCI Express Gen1 x1 link [196], mainly due to the very slow interrupt response rate of the operating system and the fact that transaction requests wait for transaction completions. Although techniques for minimizing those overheads (use of a linked list or a circular buffer of transaction descriptors for interrupts and employing a parallel transaction handling state machine) are suggested in [196] and implemented in [197], [198], [199], there is still an inevitable deviation from the theoretical maximum.

While USB 2.0 would be the most convenient option from the viewpoint of the user, it does not satisfy our bandwidth requirement. USB 3.0 provides adequate bandwidth, but it has not yet been widely adopted, so FPGA development boards with this feature are still rare and, moreover, a USB 3.0 device IP is not offered with standard industrial design tools (while designing such a complex core would require considerable effort). An implementation of the Gigabit Ethernet approach as a PC-FPGA communication interface, sending UDP datagrams over IP, is given in [200] and refined in [201]. The design leaves reliability to be implemented at the user level but combines a Look Up Table (LUT), which stores all the static fields that need to keep being resent during communication, with hardware-aware optimizations which make it more attractive than alternative reliable, but more complex, full TCP/IP implementations which require an embedded CPU [200].

However, even such a light-weight protocol suffers from a big overhead when really small packets are frequently sent. These small packets carry HAoS control-related information and may not be grouped together to form larger contiguous blocks (in order to provide a flexible API to the user). Even sending minimally-sized raw Ethernet packets, considering that their minimum size is 64 bytes (accounting for header and framing bytes - preamble, start of frame, MAC destination and source, ethertype, frame check sequence and interfame gap), results in more than 85% overhead for control packets (typically less than 10 bytes). While they are slightly smaller, similar protocol overheads exist for the other external communications interfaces mentioned above. PCI Express Gen1 and Gen2 specify a 20% overhead due to their 8b/10b symbol encoding scheme (used for clock recovery), consume 20-28 bytes for their header and framing and also suffer from traffic, link and flow control protocol overheads [202]. Due to these overheads, latency is increased (practically 20-30us for a Gen1 x8 4-byte transfer [197]) while the actual throughput is decreased, negating the performance advantage of external interfaces for typically-sized data traffic.

Table 4.2 gives examples of the sustainable bandwidth of the interfaces discussed in this section for various configurations. It is noted that the final real system bandwidth is the result of various factors, including protocol selection and overheads but also implementation choices, system integration, software support and optimizations.

In order to minimize protocol overheads, the alternative is to use a local communication interface, placing the CPU on-board. While FPGA development boards that provide an off-chip hard processor cores are not new, another approach (recently commercially available at the time of writing) attempts to overcome overheads caused by off-chip communications by combining relatively powerful hard (ARM-based) CPUs and programmable logic on the same die [206], [207]. This is a quite promising approach, as it is the first step towards practical low-latency embedded applications. While still in its infancy, the power of the processors used are still limited and the cost of reprogrammable logic comparatively high. Moving to smaller fabrication processes in the future can make revisions of this hybrid technology a very strong candidate for truly-optimized heterogeneous processing.

Table 4.2. Sustainable Bandwidth Results for various practical configurations implementing common communication interfaces on FPGAs

Protocol / Interface	USB 2.0	PCI-Express 1.0 x1			
Reference	[203]	[199]			
Experimental Configuration	Xilinx SpartanII-E through CY7C68001 USB controller on Windows. Write to FPGA File size: 400MB	Altera Stratix IV GX to Intel X58 Chipset on Windows. Full-Duplex File size: 100KB Max Payload: Read 256b, Write 512b	Altera Arria II GX to Intel X58 Chipset on Windows. Full-Duplex File size: 100KB Max Payload: Read 256b, Write 512b	Altera Stratix II GX to Dell 490 5000X on Windows. Full-Duplex File size: 100KB Max Payload: Read 64b, Write 256b	Altera Stratix II GX to Nvidia CK804 on Windows. Full-Duplex File size: 100KB Max Payload: Read 64b, Write 128b
Sustainable Bandwidth	54 (W) MB/s	219(R)/211(W) MB/s	217(R)/204(W) MB/s	162(R)/224(W) MB/s	185(R)/207(W) MB/s

Protocol / Interface	UDP/IP over Gigabit Ethernet	PCI-Express 2.0 x1		PCI-Express 2.0 x4	
Reference	[200], [201]	[199]	[198]	[199]	[198]
Experimental Configuration	Xilinx Virtex-5 SX95-1 on HTG-V5-PCIE board to Dell Latitude e4300 on Linux. Full-Duplex Payload: 1472 bytes Success Rate: ~99%	Altera Stratix IV GX to Intel X58 Chipset on Windows. Full-Duplex File size: 100KB Max Payload: Read 256b, Write 512b	Xilinx Virtex-5 on ML555 board to Dell Power Edge with Intel E5000P Chipset on Windows Full-Duplex. File Size: 32KB Max Payload: Read 64b, Write 128b	Altera Stratix IV GX to Intel X58 Chipset on Windows. Full-Duplex File size: 100KB Max Payload: Read 256b, Write 512b	Xilinx Virtex-5 on ML555 board to Dell Power Edge with Intel E5000P Chipset on Windows Full-Duplex. File Size: 32KB Max Payload: Read 64b, Write 128b
Sustainable Bandwidth	113.11(R) / 111.67(W) MB/s	438(R) / 425(W) MB/s	164(R) / 222(W) MB/s	1691(R) / 1631(W) MB/s	680(R) / 864(W) MB/s

Protocol / Interface	PCI-Express 2.0 x8			AXI4	AXI4-Lite
Reference	[199]	[198]		[204]	[205]
Experimental Configuration	Altera Stratix IV GX to Intel X58 Chipset on Windows. Full-Duplex File size: 100KB Max Payload: Read 256b, Write 512b	Xilinx Virtex-6 on ML605 board to Intel X38 Chipset on Windows Full-Duplex. File Size: 512KB Max Payload: Read 64b, Write 128b	Xilinx Virtex-6 on ML605 board to Intel X58 Chipset on Windows Full-Duplex. File Size: 512KB Max Payload: Read 128b, Write 256b	Xilinx Kintex-7 on KC705 board. From 16 on-chip 1080p video sources 32bits/pixel @ 75Hz To off-chip memory and on-chip video IP Data Width: 512bits running @ 200 MHz	Xilinx Kintex-7 on KC705 board. Interconnect handling 32 slaves during video demonstration Data Width: 32bits running @ 100 MHz
Sustainable Bandwidth	2956(R) / 2955(W) MB/s	1686(R) / 1691(W) MB/s	3297(R) / 3297 (W) MB/s	9492 (R) MB/s	180(R) / 180 (W) MB/s

The other option is to use an embedded soft CPU. While this approach has minimal overheads, since all communications are happening at wire speed, part of the available programmable resources is occupied by the relatively low-performance soft processor. Advantages of this approach are that the design tools provide full support on embedded design, the processor can be customized to include only the features that are required (optimizing speed and area) and that bare-metal applications are also supported, since an operating system is optional, depending on user requirements. An indicative collection of soft processor architectures, appropriate to be embedded on an FPGA is given in Table 4.3. It is noted that some of the metrics below are given for reference as they are highly dependent on the device being used and the revision of the implementation tools. Considering the available area and performance figures below, LEON4 and MicroBlaze are the most dominant choices. While the former is an open-source solution, the inherent compatibility of MicroBlaze with the Xilinx toolchain makes it a more favourable option for the prototype HAoS implementation.

Out of the supported on-chip interconnect interfaces [208], the Processor Local Bus (PLB) mainly targets PowerPC processors and is now outdated while the Fast Simplex Link (FSL) is a point-to-point FIFO-like interface; thus they are both inappropriate for the MicroBlaze memory-mapped control register interface for HAoS. The other options are the three variations of the Advanced eXtensible Interface (AXI) of the ARM AMBA v4.0 interconnect protocol specification (in short AXI4).

The three types of AXI4 are [225]: (a) AXI4 for burst-enabled memory-mapped communication, (b) AXI4-Lite for simple memory-mapped communication ideally to and from control and status registers and (c) AXI-Stream for high-speed streaming data. Considering the mainly controlling nature of the HAoS interface, the small size of the data to be communicated to and from the CPU and the substantially greater area footprint of the AXI4 interface compared to AXI4-Lite while providing adequate bandwidth (supporting a 32-bit interface running up to 200MHz on Virtex-6) and minimizing latency, it was decided that the latter was the optimal option. It is noted that, as AXI protocols are the industry standard for FPGA interconnect interfaces, choosing this option makes the design more future-proof (the hybrid approach in [207] also employs an AXI interface to connect its hard dual-core ARM CPUs with the programmable fabric).

Table 4.3. Indicative Collection of Available Soft Processors. CPI: Cycles Per Instruction, MMU: Memory Management Unit, MUL: Hardware Multiplier, FPU: Floating Point Unit, Area: given for specific family and corresponding metric, DMIPS/MHz: Dhrystone Millions of Instructions per Second Per MHz, (MHz): Max Frequency for family stated in Area column

Soft Core [Reference]	Architecture	Bits	License	Pipeline Depth	CPI	MMU	MUL	FPU	Area Family Metric	DMIPS /MHz (MHz)	Comments
Sirocco S1 [209]	SPARC-v9	64	Open-source (GPL)	6	1	+	+	+	60K(EE) 40K(SE) 37K(ME) Virtex5 LUTs	-	Single-core version of UltraSPARC T1
LEON3 &[210] LEON4	SPARC-v8	32	Open-source (GPL)	7	1	+	+	+	3.5K & 4K Virtex5 LUTs	1.4(125) 1.7(150)	
OpenRISC 1200 [211]	OpenRISC 1000	32	Open-source (LGPL)	5	1	+	+	-	2.4K to 4K Virtex5 Slices	(60-125)	
MicroBlaze [193]	MicroBlaze	32	Proprietary	3, 5	1	opt	opt	opt	546 to 1201 Virtex6 LUTs	1.03-1.38 (100-250)	Limited to Xilinx devices, Zero-cost for limited version
aeMB [212]	MicroBlaze	32	Open-source (LGPL)	3	1	-	opt	-	1268 Virtex4 Slices	(88-136)	Open-source clones of MicroBlaze
OpenFire [213]	MicroBlaze	32	Open-source (MIT)	3	1	-	opt	-	641 VirtexII-Pro Slices	0.58 (100)	
Nios II/f [214]	Nios II	32	Proprietary	6	1	+	+	opt	1020 StratixIII ALUTs	1.183 (290)	Limited to Altera devices
Nios II/s [214]	Nios II	32	Proprietary	5	1	-	+	opt	1030 StratixIII ALUTs	0.611 (230)	Limited to Altera devices
Nios II/e [214]	Nios II	32	Proprietary	no	6	-	-	opt	500 StratixIII ALUTs	0.138 (340)	Limited to Altera devices
MP32 [215]	MIPS 2.0	32	Open-source OpenCore+	6	1	+	+	+	5444 StratixIII ALUTs	1.21 (252)	Limited to Altera devices
Lattice Mico32 [216]	Lattice Mico32	32	Open-source	6	1	-	opt	-	2370 Lattice LUTs	(115)	Not limited to Lattice devices
Cortex-M1 [217]	ARMv6	32	Proprietary	3	1	-	+	-	2600 CycloneIII LEs	0.8 (100)	
Diamond 106Micro [218]	Tensilica Xtensa	32	Proprietary, ReadyIP	1	5	-	+	-	-	1.22 (180)	Zero-cost for Synplicity Synplify Users
Freescale V1 Coldfire [219]	Coldfire	16 32 48	Proprietary, zero-cost	2	1	-	-	-	5000 CycloneIII LEs (32bits)	(80)	Zero-cost to Altera devices only
DSPuva16 [220]	DSPuva16	16	Open-source	no	4	-	+	-	635 SpartanII Logic Cells	(40)	DSP-oriented
hyperARM [221]	ARMv4	32	Open-source (AL/GPL)	3	1	-	-	-	2953 VirtexII-Pro Slices	(63)	
PicoBlaze [222]	PicoBlaze	8	Proprietary, zero-cost	no	2	-	-	-	26 Virtex6-3 Slices	(238)	Limited to Xilinx devices
PacoBlaze [223]	PicoBlaze	8	Open-source (BSD)	no	2	-	-	-	200 SpartanII Slices	(46)	An open-source clone of PicoBlaze
Lattice Mico8 [224]	LatticeMico8	8	Open-source	no	2	-	-	-	181 Cyclone LFE2-5 Slices	(99.2)	Not limited to Lattice devices

Following the analysis above, the embedded soft CPU interface appears to be one of the most dominant candidates for the implementation of the HAoS-CPU communication link. It is noted that this is a recommendation, rather than a definite conclusion (considering the requirements stated in the beginning of this section and currently available technologies) and it depends on the processing requirements of the user application and the flexibility of the provided API (control packets could be potentially eliminated if all the software driver logic was mapped on hardware, effectively eliminating the API since the user would be provided with just one function (`transform()`) to interface to hardware).

For applications that utilize heavy-weight functions, the function processing time may overrule the communication overhead, thus making an external CPU interface preferable. This can either be the latest revision of PCI Express (due to the lower overhead and higher bandwidth), if compatible hardware (motherboard, development board) is available or a custom Ethernet-based interface implementing a custom light-weight protocol and a Network Interface Card capable of supporting such a protocol or USB 3.0 (subject to availability) or a future FPGA development board featuring a high-end processor.

The two options may further be combined in a "smart" system that offloads computation to the appropriate CPU depending on the required processing workload. Implementing such a configuration would involve a manual, or ideally automated, computation dispatching mechanism that would assign low-level processing, supported by the hardware-accelerated part of the HAoS instruction set (upper part of Table 3.4), to the built-in on-chip FU, high-level functions of low complexity to the low-end on-chip embedded processor and computation-intensive tasks to the off-board high-end CPU. High-level tasks would be assigned to the appropriate CPU depending on the comparison between the actual computation latency and the communication overhead according to:

If	*Instruction Supported on Hardware*	Select	*HAoS FU*
If	$L_{on} + O_{on} < L_{off} + O_{off}$	Select	*On-Chip CPU*
If	$L_{on} + O_{on} > L_{off} + O_{off}$	Select	*Off-Chip CPU*

where L and O would be the computation latency and communication overhead, respectively, for on-chip (embedded) and off-chip (and probably off-board) processor, accounting for the trade-off between the computational performance and the communication latency of the two solutions.

In summary, the systemic computer is designed for highly parallel software, that resembles natural systems. For such a computer to be practical it must also support sequential operations (e.g. longer mathematical expressions) and thus needs the support of a conventional CPU. The analysis here shows that current communication protocols are largely unsuitable for the task of linking a SC hardware architecture to a CPU. There is a clear need for a more integrated solution for development purposes. An FPGA board with a high-end on-board processor may be one such suitable option in the future (extending the processing capabilities in [207]). For now an embedded soft CPU provides the ability to prototype the HAoS-CPU interface. An optimal solution would be an ASIC, combining HAoS and a hi-speed CPU on-chip, which will minimise the bottlenecks caused by existing technologies.

4.2 CPU Subsystem Integration

Building on the discussion of the previous section, the soft Xilinx MicroBlaze processor was connected, using Xilinx development tools (Embedded Development Kit - EDK and Xilinx Platform Studio - XPS), through one of its available communication interfaces to the base HAoS architecture to result in the first practical hardware Systemic Computation platform.

The available tools enable great flexibility as virtually all the features of the MicroBlaze soft processor are user-defined, letting the user tailor a balanced embedded CPU design in terms of frequency, area and performance. The configuration of the processor embedded in the HAoS platform maximizes the performance of the soft CPU with the inclusion of dedicated hardware blocks (a barrel shifter, a floating-point unit (FPU) also supporting type conversions and square root, 64-bit integer multiplier and divider and a pattern comparator), instruction and data caches (64 KB each) with stream buffers (for instruction prefetching), saved cache victims (faster fetching of recently evicted cache lines) and write-back storage policy (data are not written back to memory immediately but only when needed), and math (FPU and integer divide) exceptions. A hardware debug module was also included, enabling breakpoints and memory address watchpoints, to ease debugging. A dedicated Memory Management Unit (MMU) was not added in the system as it would increase significantly its size and because its provided features, as virtual memory and memory protection, are more useful when an operating system is used. As discussed later in section 4.3.5, an operating system will not be used in the HAoS programming platform as it would run inefficiently on an embedded processor and negatively impact the latency of SC applications.

After the architecture of the MicroBlaze CPU was configured to suit the requirements of the HAoS platform, the custom HAoS logic was connected to the CPU through its AXI4-Lite interface as a peripheral. The connection was implemented using standard Xilinx-provided IP blocks. The MicroBlaze is connected to the AXI4 Interconnect core (IC), which implements the required AXI4-Lite protocol and uses a crossbar topology to route traffic between the various masters and slaves of the bus. Further details on the CPU subsystem integration are given Appendix B.

4.3 Optimizations and Enhancements

Focusing on research challenges *Chg2* and *Chg3* regarding supporting the implied SC architecture and addressing the efficiency and practicality of HAoS respectively, various optimizations and enhancements were made, both into the hardware and the software domain, to the initial design (presented in the previous chapter) in order to increase its performance and also make the prototype more user-friendly and flexible, towards a more practical and viable design.

4.3.1 Refining the Random Selection Logic

The most obvious performance optimization for HAoS, as for any clock-based circuit, was to increase its operating frequency. After analysis of the critical path of the design, the longest combinatorial path was, as expected, in the Random Selection Logic (see Figure 3.11 and section 3.6.2). The RSL was redesigned to incorporate resource sharing along with pipelining.

As mentioned in section 3.6, the BITPOSSEL module of the RSL, combines a parallel bit count with a branchless selection method. The parallel bit count is used to provide partial sums which are then appropriately masked and passed through a barrel-shifter to provide the position of a bit with a given rank in the input bus, resulting in a divide-and-conquer technique. The COUTNONES and BITPOSSEL modules of the RSL are now merged, as the parallel sum-of-bits counter in COUNTONES is reused for the generation of the partial sums during the identification of the position of the selected bit. The length of the barrel shifter is equal to the size of the longest input bus to the RSL which is in turn equal to the number of maximum supported systems. Thus, when this number is increased, the number of logic levels required for the barrel shifter implementation have a considerable impact to the delay along the critical path. For this reason, the conventional barrel shifter is replaced with a parallelized and pipelined version which instead uses an array of multiplexers with registered pre-shifted (by the required pre-calculated number of bits) versions of only the possible subset of shifting combinations of the input buses. While this results in a slightly higher resource utilization as the number of maximum supported systems increases, it provides the ability to minimize its latency and moreover make it independent of the maximum number of systems, making its performance deterministic.

Moreover, since registering the inputs of the RSL or the output of its input selection multiplexer would not further noticeably decrease the critical path, as the combinatorial logic from the TCAM to the input of the RSL adds minimal timing overhead, these registers were not included in the design saving *Number of RSL Input Buses* x *Input Bus Length* bits (for 512 maximum supported systems: up to 5x512=2560 registers).

The critical path delay of the RSL was also greatly affected by the combinatorial divider. Thus, the divider was pipelined (one-level deep), dividing its 16-stage structure (see Figure 3.15), in two groups of 8 stages each with registered inputs and outputs.

After the changes mentioned above were implemented, a static timing analysis revealed that other parts of the design (the TCAM and the Function Unit) also had latencies in the range of 15-20 ns. Thus, since the level of pipelining throughout the RSL achieved to match the critical path outside the RSL, it was decided that a latency of 20ns (which translates to 50MHz of operating frequency) was adequate for the prototype, as deeper pipelining, although possible, would require considerable changes in the control logic and would probably affect resource utilization in order to achieve timing closure. Further details on the optimisations of the RSL are given in Appendix C.

4.3.2 Minimizing the Schemata-Matching Overhead

Standard FPGA CAM design techniques include registered-based, RAM-based and Look-Up- Table-based approaches [173], [226]. Moreover, Xilinx provides a reference design which combines the LUT technique with the optimized shift-register blocks (SRL16E) found in its FPGAs [173]. Although RAM-based CAMs are the most efficient in terms of resource utilization [173], they do not support the ternary mode required for partial schemata matching in SC.

The base HAoS design used the suggested (by Xilinx) SRL16E-based approach which, according to [173], provides efficiency in terms of the trade-off between required area and operating frequency. It was noticed, that as the number of entries for the TCAM increased, depending on the number of maximum supported systems, for deep TCAM implementations (>128 entries) the area footprint of the LUT-based approach was not substantially smaller from the one of the simple register-based design (15%-25% area overhead depending on size) while the two implementations has similar operating frequency (up to ~100MHz). Thus, since the size difference was not prohibitive, the much simpler register-based TCAM structure was preferred. Further details on the TCAM design revision are provided in Appendix D.

The main advantage of this optimization[23] was the reduction of the overall latency of the matching mechanism resulting in increasing the efficiency of valid triplet generation. Since the TCAM is written every time a system is altered during an interaction, replacing the SRL16E-based TCAM with an array of registers and comparators, provided single-clock read and write operations, saving 15 clock cycles for every interaction which changed one system and 30 clock cycles when both systems are changed.

4.3.3 Further Addressing I/O Efficiency

The investigation of the capabilities and limitations of various communication interfaces between HAoS and a CPU, discussed in section 4.1, makes evident the crucial role of the performance of the design on the I/O boundary. Various optimizations have been made in order to obtain faster CPU accesses and minimize the overhead of extracting real-time (during the execution of a SC program) logging information.

As shown in Figure B., the registers in the REG BANK are accessed by the CPU through the AXI4-Lite communication link to, among other functionality, read the active triplet and write back any system which is changed by the current interaction (see Figure 3.10). As the parts of an active triplet that will be used during an interaction depend on the transformation function, HAoS makes available to the user its full contents (shown in Figure 3.8) along with some more useful information (addresses of systems, active scope and active context). In the initial design all this user data are read from and written back to the REG BANK, and then the CU handles updating the local memories and the TCAM with the changed systems.

Looking for a more efficient way, the mechanism that is used when the program is loaded to the local memories was slightly changed in order to enable the CPU to directly write changed systems to HAoS memories. However, since writing a triplet to the memories is performed in one clock cycle, to reduce latency, the whole user data would

[23] Practically here we traded area for performance, choosing the bigger but faster registered-based TCAM.

have to be updated when a change was made. Enabling the option of independently writing parts of the triplet would greatly increase the control logic complexity and the required area footprint.

Further addressing this communication challenge, a write-detection mechanism was devised, inspired by the "dirty-bit" scheme commonly used in page replacement and data caches [227]. As mentioned above, since all user data are available in the beginning of an interaction, the user may read only the parts of the triplet that are going to be used in his custom transformation function. The great enhancement comes when writing the transformed triplet back to the memories.

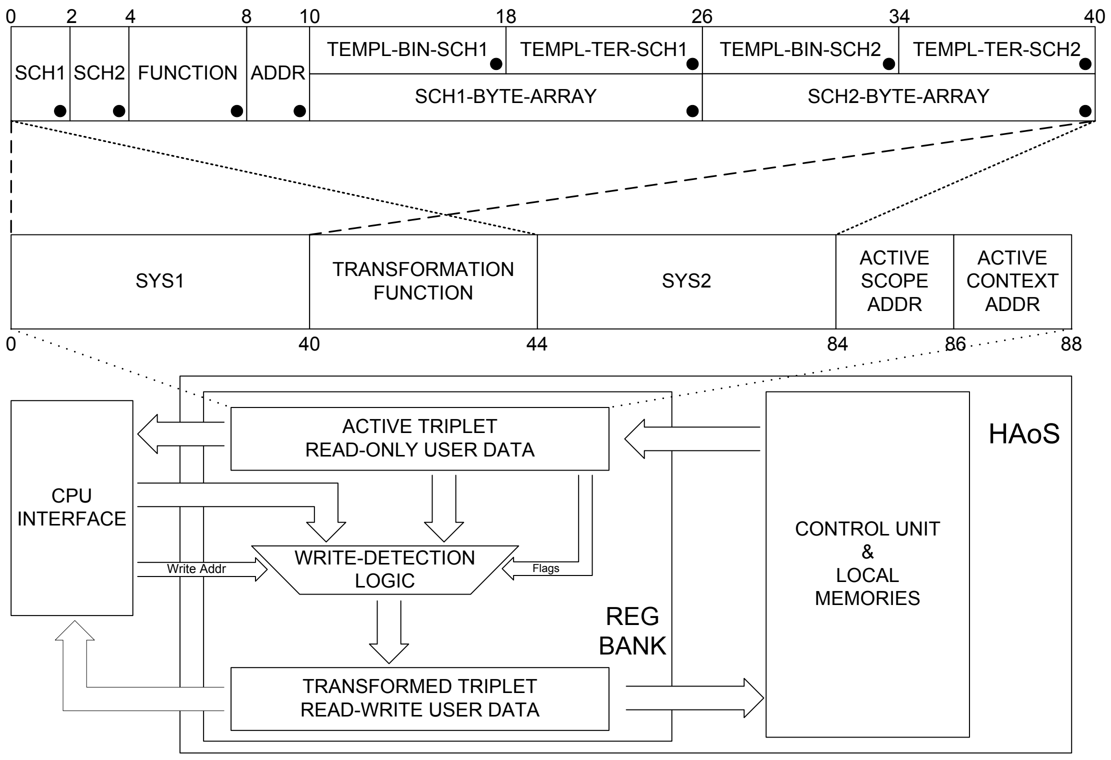


Figure 4.1. Revised Triplet Memory Map and Write-Detection Mechanism. In the upper part, the revised registers organization for each system in a triplet shown along with the sizes (in bytes) for each field. Fields from left to right: schemata 1 & 2 (for data system), transformation function, system address, binary and ternary parts for each schemata of a context system share the same address space with a byte-array formatted version of the respective schemata of a data system. All fields have an associated write-detection flag (shown here with a dot) which is set when a field is modified. In the middle, the two systems along with the active interaction function, scope and context addresses form the user data. In the bottom, when writing-back the transformed triplet after an interaction, the write address from the CPU is used to update only those fields that have actually been changed while the rest are copied over from the local copy (active triplet), minimizing the required CPU I/O operations

Each field of the transformed triplet is now associated with a write-detection flag. This flag array is reset when an interaction is assigned to the CPU. The address of the registers that hold each individual field of the transformed triplet (see Figure 4.1) is already given in the predetermined memory map of the CPU (the memory management subsystem of the CPU accesses the REG BANK as any other memory location). While in "Transform" state (see Figure 3.10), when each such field is altered by the CPU, the decoded write address from the memory subsystem is matched against each field address and sets its respective flag. At the end of the "Transform" state, the active triplet (user data before interaction) is copied to the transformed triplet (user data after interaction) address space, updating at the same time only the fields that were actually changed by the CPU. Using this relatively simple write-detection approach, the need of accessing individual fields when writing the triplet is avoided, preserving the low area footprint of the HAoS memories writing logic, but also minimizing the required user accesses to enable the write-back of the interaction result.

In order to further minimize the user effort while manipulating the HAoS user data, taking into account that each SC schema (16-bit in this implementation) may be used as a whole (e.g. a 16-bit number) or as a bit-array (e.g. a 16-element chromosome), each schema can be accessed (read/written) in both modes (2-byte value and 16-byte array with one effective bit each). This provides the user with the flexibility of being able to avoid time-consuming bit-manipulation through bit-masking while processing the data by operating on an array and also saving bit-to-byte software conversions as this is handled by hardware.

Furthermore, the parts of a compressed template of a context system (see Figure 3.8) were carefully re-arranged from SCH1-FUNCTION-SCH2 to SCH1-SCH2-FUNCTION to get more compact memory utilization and faster accesses as the respective registers in the REG BANK were also re-arranged in order to overcome any compiler byte-alignment restrictions. This way, the whole template can be accessed by 2 consecutive 4-byte memory read operations rather than three separate ones (one for each of the three fields).

4.3.4 Further Addressing User-Friendliness with a Functional Model

While the developed simulation environment provides extended debugging capabilities, it requires access and expertise on electronic design tools which should not be a requirement for developing SC models to run on HAoS. Furthermore, such low-level system simulations can be extremely time-consuming. Thus, in order to expedite natural

SC models development, a software equivalent functional model of the suggested HAoS architecture (a high-level simulator of the HAoS circuitry) was built, making possible to quickly verify the functional behaviour of a SC program even without the need of having the hardware platform.

The HAoS functional model is based on the original implementation in [24], but provides a software interface to develop abstract high-level interaction behaviours, similar to the (C/C++) plug-in approach in SCoPE. Once the functionality of the required contextual behaviour is verified, the user can easily reuse the plug-in (with minimal implementation-specific changes) in the compatible HAoS development flow.

4.3.5 Further Addressing Programmability

In order to also enhance the user experience and further address the programmability of HAoS, the challenge of loading the program to the platform, being able to extract debug information during runtime and storing this log information for post-processing were carefully examined taking into consideration that it would be preferable if HAoS was a stand-alone self-contained solution.

As discussed in section 3.7, SC models are first developed using the SC language (see Listing 3.1). The compiler has been updated to incorporate abstract transformation functions in order to enhance flexibility by supporting high-level processing through the CPU, resulting in human-readable assembly code (see Figure 3.17). However, this format is not optimal for the program to be loaded to HAoS. Thus, a post-compiler tool was developed to transform the human-readable assembly code to binary format with minimum size in order to minimize the amount of data to be transferred to the HAoS local memories and the processing time during program loading.

This SC binary generator tool effectively assigns one bit for each element of the scopetable, while cleverly separates data from context systems as the former can be further compressed while the latter may not, since system templates of contexts carry already compressed information. Therefore, each line of the scopetable (see Figure 3.17) requires the number of systems contained in the program to be divided by 8 and rounded up to the closest integer amount of bytes, the transformation function is always 4 bytes as it can never have a ternary part, each schemata of data systems is compressed to 2 bytes while each template of a context system requires 16 bytes, 8 for its binary part and 8 for its ternary part. The transformation function information is not included in the binary SC format, as all interactions supported natively by HAoS have a predetermined opcode while all CPU-supported interactions can be checked for validity during runtime through the software backend.

After the SC binary file, containing all required information, is prepared, the program should be loaded to HAoS memories. The most straight-forward solution would be to store the SC program on either the on-chip Block-RAM or the on-board DDR3 RAM memory resources of the soft CPU. While the first option would enable the fastest loading time, this would reduce the already limited available on-chip memory resources. While this size limitation could be resolved by using the bigger on-board RAM memory, a connection of the platform with a conventional computer would still be implied and required in order to transfer the developed program to HAoS.

Thus, in order to enhance flexibility and make HAoS a standalone platform, it was decided that the program should be stored on and loaded from a form of non-volatile memory. Since the selected FPGA development board featured a Compact Flash card controller (a common feature for development boards), this was chosen to be used as the main storage of the prototype, since when it is FAT-formatted, it can also support a basic file system.

Using the Compact Flash card and its file system, also addresses another very important programmability aspect. This is the ability to efficiently log runtime information in a console-like manner. Although access to a real-time console is possible during live hardware debugging (using Xilinx tools), this results in excessive run times as all text is communicated to a separate computer through a high-latency UART channel. For SC applications which require that results are logged throughout the execution of the program, just the data-logging overhead can account for the majority of the run time. Storing any output data on the onboard CF card drastically reduces the required runtime due to logging and again results in a standalone platform.

It is noted that in order to ease development, SC programs with low size requirements can be hardcoded in software and loaded on HAoS on-chip memories along with the accompanying low-level driver. A tool converting the SC binary file to ASCII text (in order to be embedded in the user code) was developed to enable this functionality which can be very useful during the first stages of development of a SC model, as initially a lightweight version of the model can be more easily and quickly verified through multiple revisions of the code until the desired behaviour is achieved. An example of the resulting translated SC binary to ASCII is given in Appendix H for the example SC program discussed in section 3.7 (see Figure 3.17).

As explained in section 4.1, having decided that a soft processor will be used in the HAoS prototype platform, using the Xilinx MicroBlaze processor was the most straightforward decision as this offers a complete solution which is supported by the available Xilinx design tools out of the box, in contrast with other open source and proprietary alternatives. As the overhead of a complete operating system would have a high impact on the performance of the platform, especially on such a low-performance processor, it was decided that SC high-level interaction processing should be run as a bare-metal application (referred to as "standalone" operating system option by Xilinx), which is a set of low-layer software modules used to access processor specific functions. Therefore, a low-level driver had to be developed in order to achieve communication between the MicroBlaze and HAoS.

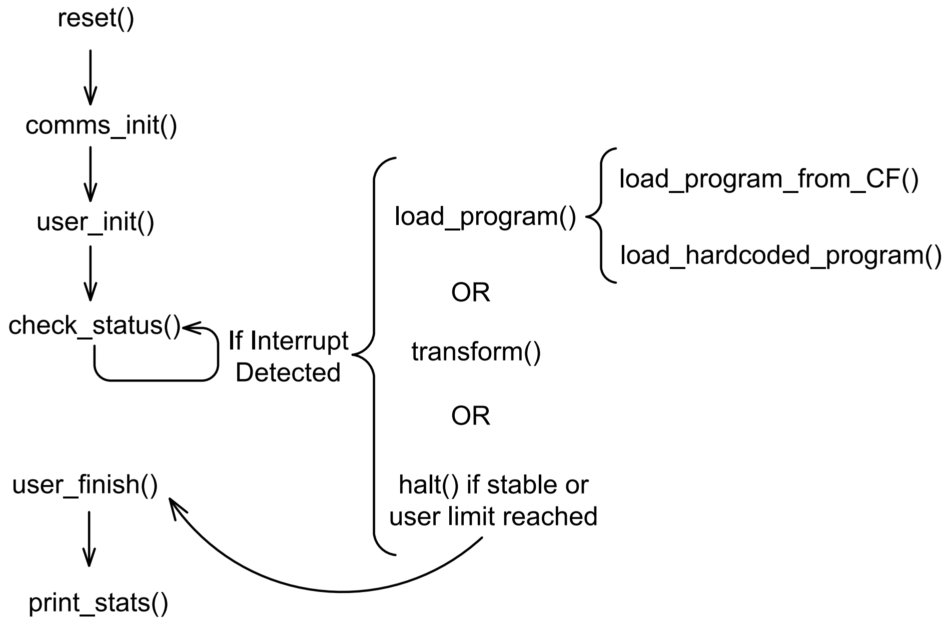


Figure 4.2. HAoS driver flow diagram

The HAoS driver handles all required background functionality. Its flow diagram is given in Figure 4.2. It resets HAoS at the beginning, initialises any used communication interfaces and loads the program either externally, from the CF card, or internally, from the embedded user code, and then the loaded SC program starts executing. Then the driver waits for an interrupt from HAoS, by constantly reading the predetermined HAoS status register, to either pass control to the user code to perform some high-level interaction or halt the system in case all systems have become stable or the user-defined maximum number of interactions has been reached or a user-defined condition has been satisfied. At the end of the execution, it also optionally gives some useful statistics (execution time, percentages of aborted iterations due to either schemata mismatches or unavailable matching contexts in the selected scope, executed transformation functions, number of interactions and average execution times). Effectively, all these background processes are transparent to the user, which only has to define the transformation functions that are defined in the SC source code and are supposed to be executed on the CPU.

The HAoS software framework is completed by a basic but comprehensive API, in order to enhance the flexibility of the platform and the accessibility of the user to the internal state of HAoS. The API among others, provides the user with read and write access to any HAoS memory-mapped control register, and also offers optimized low-level access routines to the schemata byte-arrays, scopetable manipulation, direct access to the full contents of the HAoS local memories (TCAM and system RAMs) and the high-precision (10ns resolution) HAoS real-time counter while it gives to the user the option of executing initialization and termination code, respectively, before and after the execution of the main SC program. A summary of the functions provided by the API is given in Appendix E while more detailed information can be found on the official HAoS webpage [191].

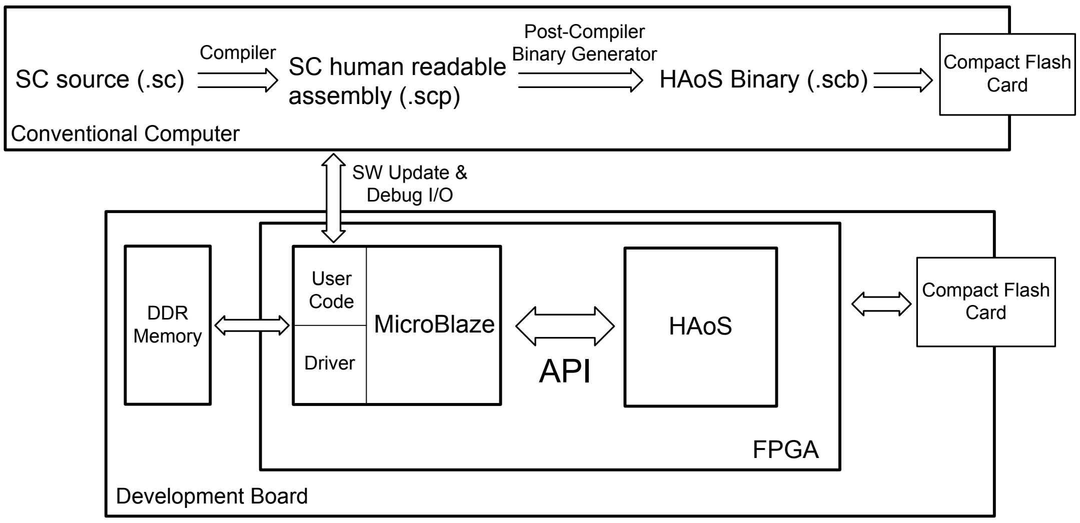


Figure 4.3. HAoS programming toolchain and software framework illustrating the complete suggested programming platform

The discussion above is summarized in Figure 4.3. The provided toolchain to convert the SC source to the final HAoS binary is shown in the upper part, while the lower part gives an overview of the software framework (and its association to the partitions of the hardware platform), where the program is loaded from either the CF or part of the user code, the driver handles background processes while the user just focuses on writing the high-level implementations of the required interaction transformation functions, communicating, when needed, with HAoS through the provided API.

4.3.6 Refinements Results

Various optimizations made to the prototype HAoS architecture are discussed in the previous sections. These enhancements are made to address mainly research challenge *Chg3* (see section 1.4), in order to increase performance, in terms of latency and operating frequency, and I/O efficiency but also improve qualitative aspects as user-friendliness and programmability.

Table 4.4. Benchmark timing improvements reflecting various architectural optimizations, using the 64-system base configuration with MicroBlaze running at 200MHz. Results given are averaged over 10 runs. Reported timing for each row is obtained using optimizations stated in all rows preceding it. On average, the CPU consumes ~40ms for the transformation functions and ~15ms for the low-level driver functionality.

Optimization Description	**Benchmark Timing (ms)**
No Optimization - CPU Writes Back the Triplet to HAoS Memories in consecutive writes Writing logging information (20 ASCII characters) to off-board terminal through debug UART and USB	768.213
CPU Writes Back the Triplet to HAoS Memories in consecutive writes - Writing logging information (20 ASCII characters) to on-board Compact Flash card	186.315
CPU Writes the Triplet to HAoS Registers - HAoS then writes it back to memories	176.613
CPU reads/writes only what is needed since Hardware Write-Detection is enabled	135.928
HAoS offers byte-aligned schematas in software-aware formatted registers for optimized CPU access	121.428
Enable hardware random numbers from the LFSR instead of using standard PRNG software functions	109.431
Optimised read/write functions using full data width for CPU schemata access	105.877
Minimized schemata-matching overhead using a register-based TCAM (single clock write latency)	101.704
Replaced barrel-shifter in BITPOSSEL with parallel pipelined shifter and multiplexers fed with pre-calculated constants accounting for every possible shifting combination	98.704
Increased HAoS operating frequency from 12.5MHz to 50MHz. Merged COUNTONES with BITPOSSEL to form RSL and optimized its critical path	82.934

In order to quantify the performance improvements, a classic computational problem (the binary knapsack problem solved using a genetic algorithm optimization - see section 5.1) is used here as a benchmark and the performance of the 64-systems HAoS is measured in terms of the duration of the execution of the program until 10000 interactions have been reached. In this configuration the MicroBlaze runs at 200MHz while accurate (±10ns) timing measurements are obtained by the dedicated real-time counter on HAoS. The optimizations results are given in Table 4.4.

As shown above, while all optimizations have a positive contribution to overall system performance, the ones that provide the most major improvements are the Hardware Write-Detection mechanism (see section 4.3.3) and the optimizations on the critical path of the RSL (see section 4.3.1) that allowed a higher operating frequency. Furthermore, printing logging information on an off-board terminal (e.g. a laptop connected to the board through USB), would heavily impact the performance of the system due to the increased latency of the UART while disabling logging would negatively impact the user experience. The solution of storing real-time information locally on the SD card enables logging with a minimal impact to performance (compared to the terminal approach).

Further addressing *Chg3*, ease-of-use is improved by the functional model which enables users to start developing and verify the basic functionality of SC programs without the need of the hardware platform, while a complete software framework is also provided to improve programmability and forms the base for the formal HAoS model development methodology, introduced in the next section, to enhance user-friendliness and support the utility and viability of the HAoS prototype platform.

4.4 Addressing Scalability for Single-Chip Implementations

It is important to note that depending on the number of systems required for a SC model, the HAoS architecture can be easily scaled to accommodate any number of systems as long as the design area can fit on the selected FPGA device (assuming a single-FPGA implementation). HAoS has been written in highly-parameterized VHDL code. Thus, scaling the design is a matter of changing a single parameter, the length of the address bus (which is equal to the base-2 logarithm of the number of maximum supported systems). In this way, the size of the SC model, in terms of systems, is limited solely by the size of the available FPGA device.

As mentioned in section 3.8, the available Xilinx ML605 development board, features the Virtex-6 LX240T FPGA which is a mid-range 40-nm based device, with high-end devices built on 28-nm processes offering even 10 times more reprogrammable fabric real estate and significant performance potential [228]. Table 4.5 shows the implementation statistics of the available variations of the HAoS platform of Figure B.2, scaling the number of systems, including also the case of the area footprint of the platform without HAoS (number of systems equals zero, just the MicroBlaze subsystem - 18% of available slices). These figures are in agreement with the initial estimates of Table 3.6 and give the utilization of slices, LUTs, registers, Input/Output ports and DSP blocks for designs ranging from 32 to 1024 systems.

Table 4.5. HAoS platform implementations statistics as the number of maximum systems increases. Figures based on Virtex-6 LX240T utilization. The MicroBlaze subsystem including all peripherals except HAoS requires approximately 18% of available area. Numbers of slices, LUTs, registers, I/Os, RAMs and DSP blocks with respective percentages used are given for designs supporting 32-1024 systems (1024-systems configuration does not include the latest design changes)

Maximum Systems		0		32		64		128		256		512		1024*	
	Total	**Used**	**%**	**Used**	**%**	**Used**	**%**	**Used**	**%**	**Used**	**%**	**Used**	**%**	**Used**	**%**
Slices	37680	6841	18	12235	32	13492	35	15525	41	18269	48	24882	66	34522	91
Slice LUTs	150720	14283	9	27636	18	29972	19	34338	22	43146	28	61481	40	98511	65
Slice Registers	301440	15061	4	22733	7	25400	8	30818	10	41727	13	63768	21	108361	35
I/O Blocks	600	193	32	193	32	193	32	193	32	193	32	193	32	193	32
RAMs	416	56	13	58	13	61	14	64	15	70	16	106	25	148	35
DSP Blocks	768	6	1	7	1	7	1	7	1	7	1	7	1	7	1

It is interesting to note that the size of the design appears to scale linearly (considering the limits imposed by a single-chip implementation) as the number of systems increases, as illustrated in Figure 4.4.

This implies that, assuming availability (and affordability) of the largest modern FPGA device (Virtex-7 2000T with 305400 slices), SC models with up to 8196 systems may be efficiently modelled with the single-FPGA HAoS platform (based on a projection of the number of slices required according to the linear regression equation for the used slices of Figure 4.4).

While the performance of HAoS will be identical for designs supporting different maximum number of systems, fine-tuning the size of the design for a particular application may permit more functions to be hardware-accelerated, increasing overall performance. If such an addition is not required, the design featuring the greatest number of systems may always be used.

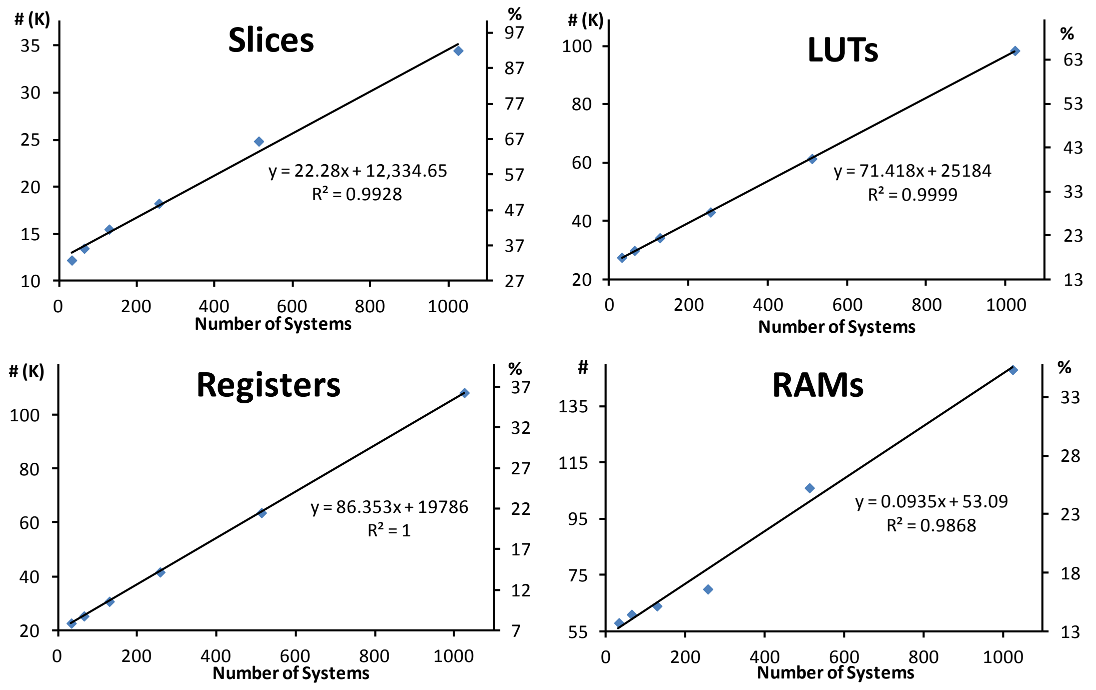


Figure 4.4. Linearity on area utilization as the number of maximum supported systems is increased. Linear regression lines and determination coefficients given for slices, LUTs, registers and RAMs

It is also noted that while the MicroBlaze processor of the hardware platform is clocked at 100MHz, higher operating frequencies are achievable and have in fact been tested to be fully operational, running at 200MHz, near the lower bound of the design sizes. This is expected as, when the area utilization is low, the implementation tools have more flexibility and achieving timing closure is more feasible. However, the lower 100MHz CPU frequency has been selected for the evaluation purposes of this section in order to have a uniform performance along all the size variations of the platform.

The discussion above makes evident that the specification (number of maximum systems, performance of the soft processor, operating frequency of the HAoS subsystem) of the HAoS platform strongly depends on the characteristics of the FPGA device it is implemented on and that the prototype is merely an example of what a mid-range device can accomplish. It is expected that as new FPGA technologies emerge, the custom HAoS logic, having been written in completely vendor-agnostic fully-synthesizable code, can be adopted with minimal effort to achieve greater performance.

4.5 HAoS Model Development Methodology

Building upon the discussion of previous sections and further focusing on the practical aspect of using the platform, a methodology for developing natural models targeting

HAoS is suggested in this section and illustrated in Figure 4.5 with a layered format to separate the distinct development phases.

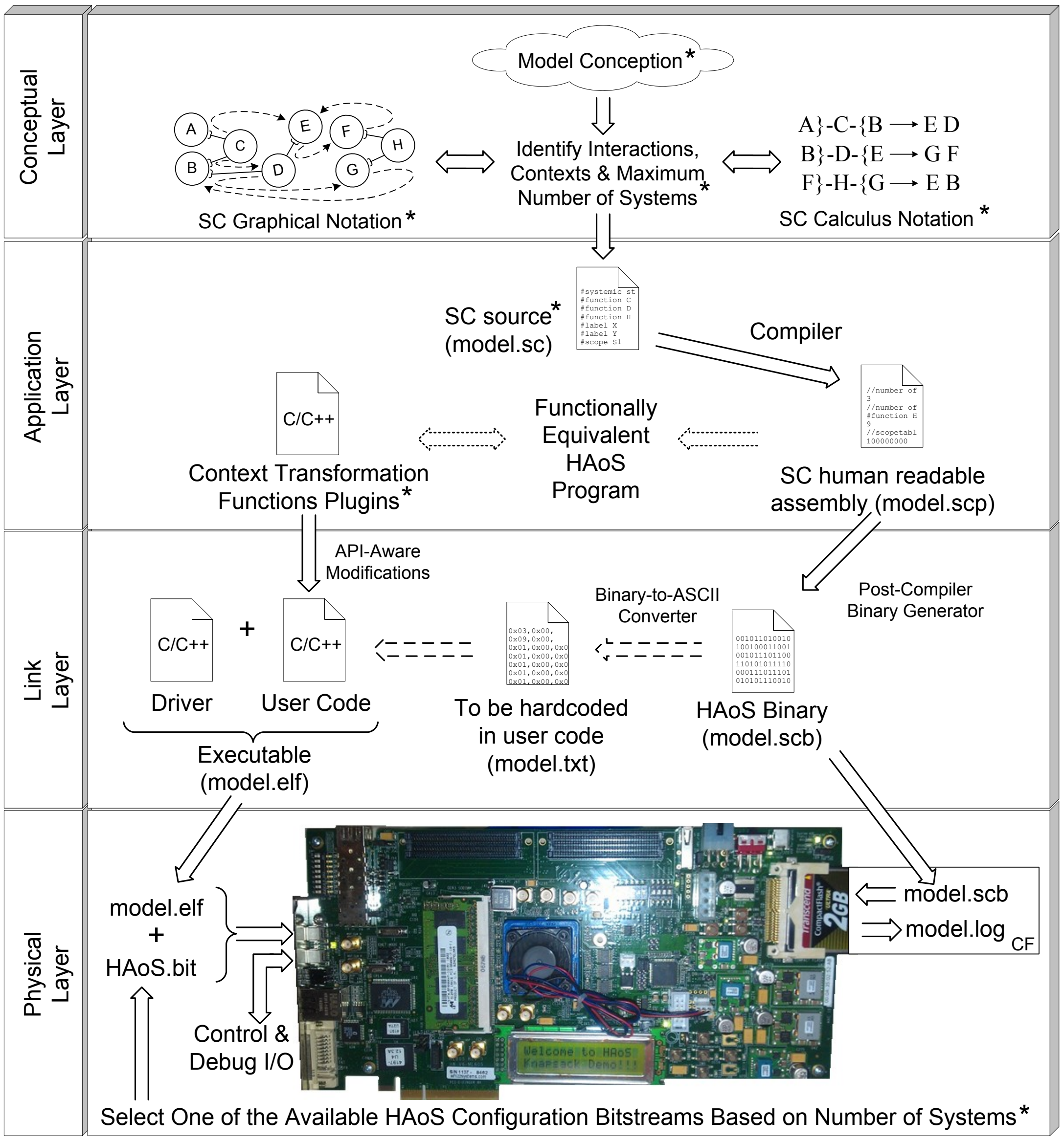


Figure 4.5. HAoS model development methodology (* implies user input)

Assuming that an existing natural system or process needs to be simulated, it is important to first understand its behavioural dynamics and identify its quantitative characteristics in order to conceptualize it (Conceptual Layer). At this stage, a systemic analysis is necessary to identify the interacting systems, the interactions among them (any contextual behaviour defining their transformation functions) and their organisation (using scopes). The SC calculus notation can be used to describe the interactions, while the SC model may be visualized using the SC graphical notation (see Figure 2.9). Having a proper SC graphical notation of the model can make writing of the SC source code

(*.sc - see Listing 3.1) quite trivial as each element in the SC graph corresponds directly to a specific part of the code. This fact implies that source code extraction from SC graphs can be automated in the future, enabling building SC models by using a high-level SC graph tool. This direct mapping also extends in the SC calculus notation making the transition from the Conceptual Layer to the next layer, the Application Layer, fully automated once these SC high-level tools are developed.

The Application and Link Layers form the software framework discussed in sections 4.3.4 and 4.3.5. In the Application Layer, the SC source code is translated to human-readable assembly code (*.scp), which is then used as input to the HAoS functional model along with the high-level processing plugins, implied by transformation functions not supported natively by HAoS (not included in the HAoS instruction set - see Table 3.4). The source code and plugins are then revised until the desired behaviour is accomplished. The Link Layer is the back-end phase were the SC binary (*.scb) is generated by the post-compiler and, depending on how the program is going to be loaded, it is either transferred to the Compact Flash Card or converted to ASCII text (*.txt) to be embedded to the user code. Slight modifications may be needed at this point to the interaction plugins prepared in the Application Layer to account for low-level communication to HAoS through the provided API. Finally, the user code is linked with the HAoS driver (using the Xilinx Software Development Kit) to generate the bare-metal executable (*.elf) which will run on the MicroBlaze processor.

At the Physical Layer, the HAoS platform is implemented on the target FPGA board. Based on the number of systems of the SC model, the appropriate configuration bitstream (*.bit) is selected and combined with the output executable of the Link Layer to form the final bitstream to program the FPGA device. The SC model simulation starts by asserting the on-board hardware reset. The CF card acts as the storage unit of the platform, storing the HAoS binary program (*.scb) and runtime log information (*.log). A summary of the various file types used along the suggested HAoS model development framework is given in Appendix F.

Before the final deployment of the SC model, live hardware debugging is also supported through the Xilinx Software Development Kit (SDK) [225] following a typical debugging flow in an Eclipse-based environment. The choice of the MicroBlaze processor (section 4.1) ensured that software development is seamlessly integrated in the HAoS embedded system design flow, as the complete architecture can be exported from the hardware environment directly to the software environment. The SDK tools take advantage of this compatibility and configure the compiler and the debugger according to the underlying hardware design in an automated way while the memory mapping of the various peripherals is configured by auto-generated linker scripts. The maturity of the tools and the inclusion of the specialized hardware Xilinx MicroBlaze Debug Module in the system enable full source-level debugging capabilities as all typical debugging features (like setting breakpoints and watchpoints, examining program variables and the contents of system memory, stepping through program execution and viewing the call stack) are supported. During debugging, access to the internal state of the HAoS custom logic is obtained through reading the appropriate registers of the REG_BANK using the provided API functions (see Appendix E) while the host computer running the SDK communicates with the FPGA development board through the UART of the embedded system using a standard USB cable.

4.6 Summary

In this chapter, the base HAoS system is extended to a practical hardware platform accompanied by a software framework to provide a complete SC programming platform. A thorough investigation of potential communication interfaces is provided in section 4.1. The analysis suggests that current technologies and protocols are widely inappropriate for the real low-latency high-bandwidth solution required for linking the SC architecture to a hi-end CPU. Thus, the suggested design makes a compromise based on the latency-bandwidth trade-off that current technologies support, and it is concluded that the ideal configuration would involve a high-performance CPU and the reprogrammable logic on the same die, communicating at wire speed (acknowledging the fact that current industrial trends have started adopting this approach).

The rest of the chapter addresses *Chg2* (SC architecture support), by revising the random-selection and schemata-matching hardware blocks, and *Chg3* detailing optimizations and enhancements that increase the efficiency of the design in terms of latency and area, quantifying the results in sections 4.3.6 and 4.4. Combining the updated hardware design with a complete software framework, developed mainly to enhance user-friendliness and programmability, a HAoS model development methodology is then formulated in 4.5 and demonstrated in the next chapter, giving examples of simulating a natural process from conception to obtaining the final results. The time complexity of the schemata matching mechanism is evaluated in section 5.1.5 and reveals that the optimized architecture achieves the task in near constant time.

Chapter 5

Verification and Evaluation

Low-level simulations of the hardware design have previously (section 3.8.1) verified the functional behaviour of the base HAoS design testing various simple scenarios. However, system-level verification is also important to stress the architecture and ensure that the complete platform (including the embedded CPU subsystem and communication interface) is behaving as expected. Since the hardware platform is available, live testing (SC programs executed on hardware) can be much faster than the extremely time-consuming RTL simulations, enabling the testing of more advanced functionality and more complex SC models. System-level hardware verification addresses research challenge *Chg2* as it validates the support of the underlying SC architecture from the suggested design.

Furthermore, executing more complex SC models on the final hardware platform can also be used to evaluate HAoS against alternative simulators, in terms of efficiency - addressing this way research challenge *Chg3*. Thus, after successfully executing the simple test programs of Table 3.5 on hardware, three practical bio-inspired models, presented in this chapter, are simulated with our prototype platform, and the results are compared with the outcome of alternative simulation environments confirming that HAoS can be used as a practical simulation solution (addressing the second requirement of *Chg3*).

The selected models attempt to cover a wide range of possible SC applications. First, a genetic algorithm optimization of the binary knapsack problem gives an example of how evolutionary methods can be implemented with SC to solve a classic synthetic computational problem. Then, moving to a more practical application, we model a well-studied biochemical process, the MAPK signalling cascade. Finally, increasing significantly the complexity, a SC application modelling the effect of chromosome missegregation during cellular division and typical treatment approaches on cancer growth is presented. All the models presented below, have been previously introduced targeting different platforms and are reused for a thorough verification and evaluation of

HAoS. The first two models are based on previously developed SC applications, retargeted here to the HAoS platform, while the cancer SC application has been developed from scratch.

Part of the work presented in this chapter has been published in [37], [160] and submitted for publication in [230].

5.1 A Genetic Algorithm Optimization of the Binary Knapsack Problem

The most complex test case among the initial verification scenarios of Table 3.5 is the genetic algorithm (GA) optimization of the binary knapsack problem [231]. The knapsack (or rucksack) problem is a classical example of combinatorial optimization [232] which involves finding an optimal object in a finite set of objects, essentially exploring a search space for the best solution to a given problem. Other typical examples in this category of problems are the Travelling Salesman Problem, Minimum Spanning Tree Problem and Job Assignment Problem [232] having in common that an optimum instance is required, but examining all the possible permutations to identify it is not usually desirable or feasible.

For this reason, alternative approaches and numerous algorithms can be found in the literature [232] addressing the various types of combinatorial problems. Among them, a Genetic Algorithm (described earlier in the context of Evolvable systems, section 2.2.2) is a well-suited method for solving the binary knapsack problem, as it uses evolutionary search techniques to identify a sufficiently good solution. The SC model presented below follows the approach introduced in [34], running on the GPU-based SC implementation, in order to directly compare the performance results obtained by HAoS to prior SC implementations.

5.1.1 The Binary Knapsack Problem

In the general knapsack problem, there are *n* types of items. Each type *i*, has an associated non-negative value v_i and weight w_i. The maximum combined weight of items that can fit in the knapsack is W. The binary (or 0-1) knapsack problem also poses a restriction on the number x_i of copies of each type of object to zero or one. The problem is mathematically formulated as:

$$\text{Maximize } \sum_{i=1}^{n} v_i x_i \text{ where } \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}$$

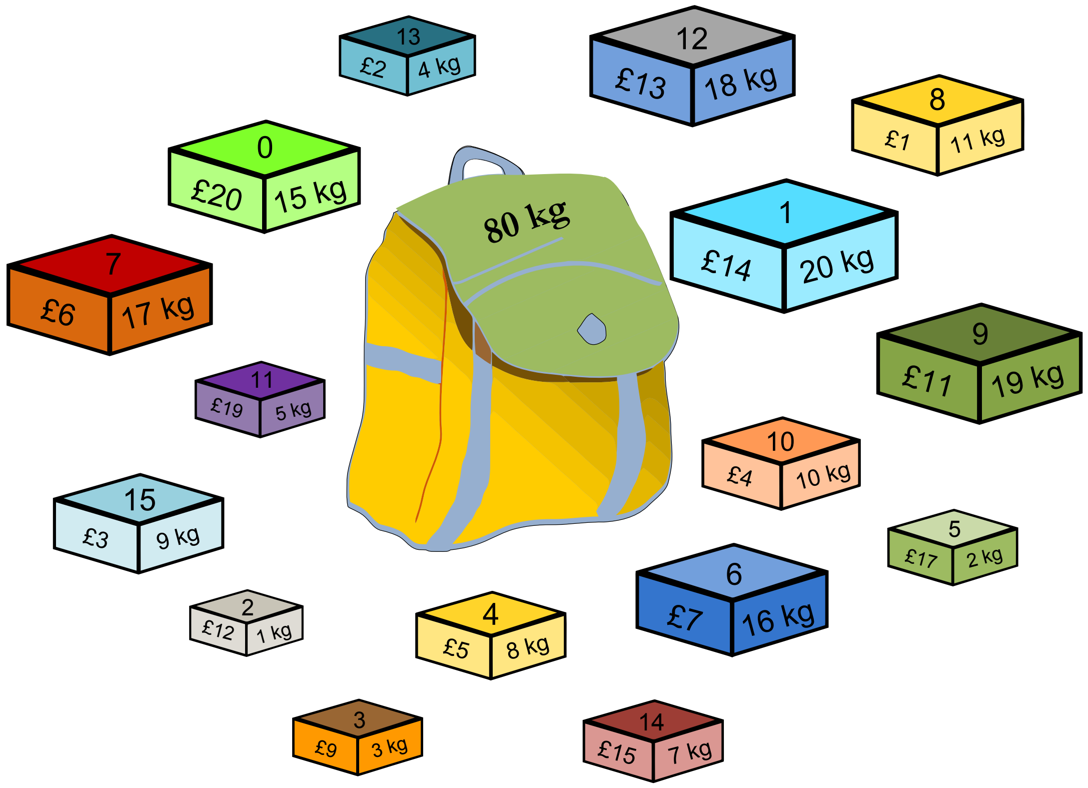


Figure 5.1. 16-Element Binary Knapsack Problem where W = 80kg

The configuration of the specific test case for the binary knapsack problem is shown in Figure 5.1 with $W = 80\ kg,\ n = 16\ (i = 0 \rightarrow 15)$ and various randomly selected combinations of weight and value for the available items.

5.1.2 Applying a Genetic Algorithm to the Binary Knapsack Problem

In order to solve a problem with a GA-based approach, a population of candidate solutions is evolved by altering a set of properties for each candidate. For the binary knapsack problem, each solution may or may not include one copy of each available item. Each solution is represented by an n-bit binary string, where n is the number of available items and each bit represents if a specific item is (if the bit is set) or is not (if the bit is cleared) selected to be part of the solution. Thus, the string, or chromosome, holds the binary decisions making up each distinct solution for the given problem.

This representation is illustrated in Figure 5.2, giving as an example the optimal solution of the 16-element binary knapsack problem of Figure 5.1. The position of each bit in the chromosome corresponds to the distinct type of each item (shown at its top facet in Figure 5.1). The weight and value for each solution, according to its chromosome, is calculated by summing the weights and values of the items at the corresponding positions with set bits. For the configuration of Figure 5.1, the optimal chromosome gives a maximum value of 124 for a total weight of 79.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Position - Type
0	1	0	1	1	0	0	0	0	0	1	1	1	1	1	1	Optimal Chromosome
9	7	4	18	5	10	19	11	17	16	2	8	3	1	20	15	Weight (79)
3	15	2	13	19	4	11	1	6	7	17	5	9	12	14	20	Value (124)

Figure 5.2. Representation of the optimal solution for the Binary Knapsack Problem

Following a classical GA methodology, a set (or population) of solutions is initialized with random values for each bit in their chromosomes. Then, a set of genetic operators is used to alter the genetic material of each solution. For simplicity, only three standard genetic operators, illustrated in Figure 5.3, are applied to the candidate solutions of the Binary Knapsack Problem[24]: Binary (or Single Point) Mutation which performs a random bit flip, One-Point Crossover that swaps the genetic content of the two parents around a randomly selected point and Uniform Crossover where each bit of the resulting solution may come from each parent with a 50% probability.

The selection of solutions to propagate to the next generation is straightforward as the fitness function in this case simply gives the weight of the chromosome, so valid solutions with greater weight are fitter. However, it is noted that as the genetic alterations are random, the resulting offspring may become invalid if its total weight exceeds the predetermined threshold *W*. For this reason, each genetic operation also includes a guarding functionality to prevent invalid solutions by selectively decreasing the weight of an unacceptable chromosome until its weight is below *W*.

[24] It is noted that other types of mutation (as boundary, uniform and Gaussian) and crossover (as two-point, cut-and-slice and half-uniform) are also commonly used. Moreover, other genetic operators (as regrouping, colonization-extinction and migration) are also suggested in the literature [250]

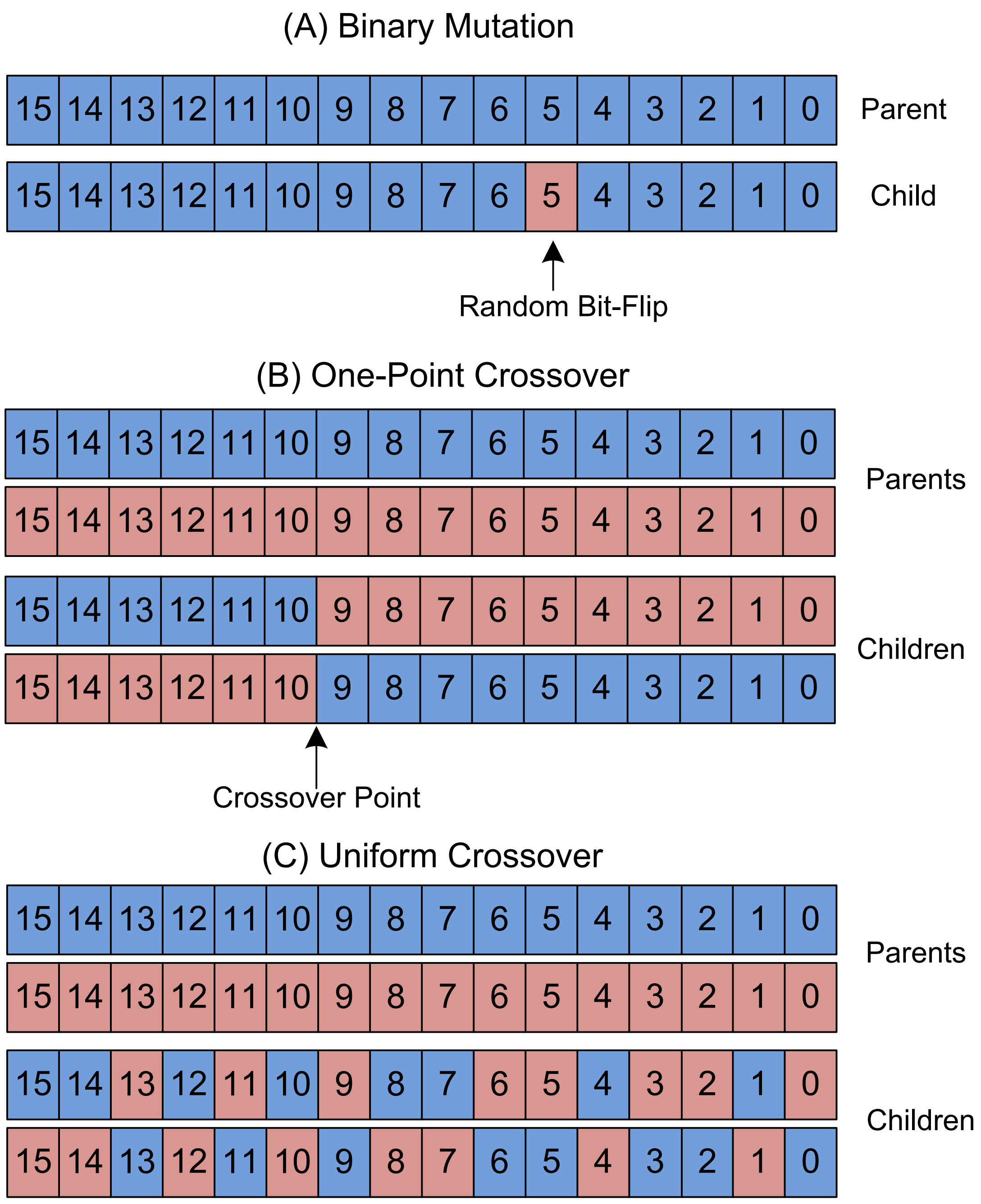


Figure 5.3. Standard genetic operators: Binary Mutation, One-Point and Uniform Crossover

5.1.3 Systemic Analysis

Having described the genetic optimization approach of the binary knapsack problem in the previous sections, a systemic analysis is required in order to identify the systems, the interactions among them and the scopes they belong to before building the corresponding SC model (implementing the conceptual layer of the suggested model development methodology, see Figure 4.5).

In this case, associating the data systems with the candidate solutions is a straightforward choice, as the similarity of the binary representation of a chromosome (see Figure 5.2) with the representation of a HAoS data system (see Figure 3.8a) is evident. Thus, the genetic content of each chromosome is stored in one of the schemata of a SC system. This implies that as the size of each schema is set to 16 bits in this HAoS implementation, a restriction of a maximum of 16 items is posed to this knapsack problem.

As systems are initialized to random values after the beginning of the program, the SC model should initially include only non-initialized solutions. Moreover, the fittest solution should be uniquely stored in a different type of system, which will be updated only when required (only when a fitter solution than the previously fittest solution has been found). Thus, three distinct types of data systems, stored in the second schema of each data system, are required to represent all possible solution: non-initialized, initialized and fittest.

The obvious context systems needed, defining interactions between the parent solution systems, correspond to the three genetic operators used in our problem (see Figure 5.3). Additionally, following the discussion above, a context handling the initialization of solutions (an "initializer" context) is also required, setting the bits of the chromosome-representing schema randomly to 1 or 0 (with 50% probability each) and changing their type-representing schema from non-initialized to initialized. Moreover, another "output" context should be responsible for updating the fittest solution, by comparing its weight with the weight of a randomly chosen initialized system and updating it when a new maximum weight is found.

Considering the required scopes of interaction, since the main transformation activities of this SC model are performed by the genetic operator contexts on initialized solutions, a dedicated "computation" scope is defined to separate them from the secondary tasks of initialization and output. This implies that non-initialized solutions can be part of the root (or "main") scope but they need to be moved in the computation scope when initialized by the initializer context. The output context (along with the fittest solution) can also be contained in the root scope. However, since interacting systems must be in the same scope, and since the fittest solution belongs to the root scope, initialized solutions should also be part of the main scope (in order to be able to interact with the final solution during updating). This denotes that initialized solutions are part of both the computation and the main scopes. The genetic optimization binary knapsack SC model, according to the systemic analysis above is summarized in Figure 5.4.

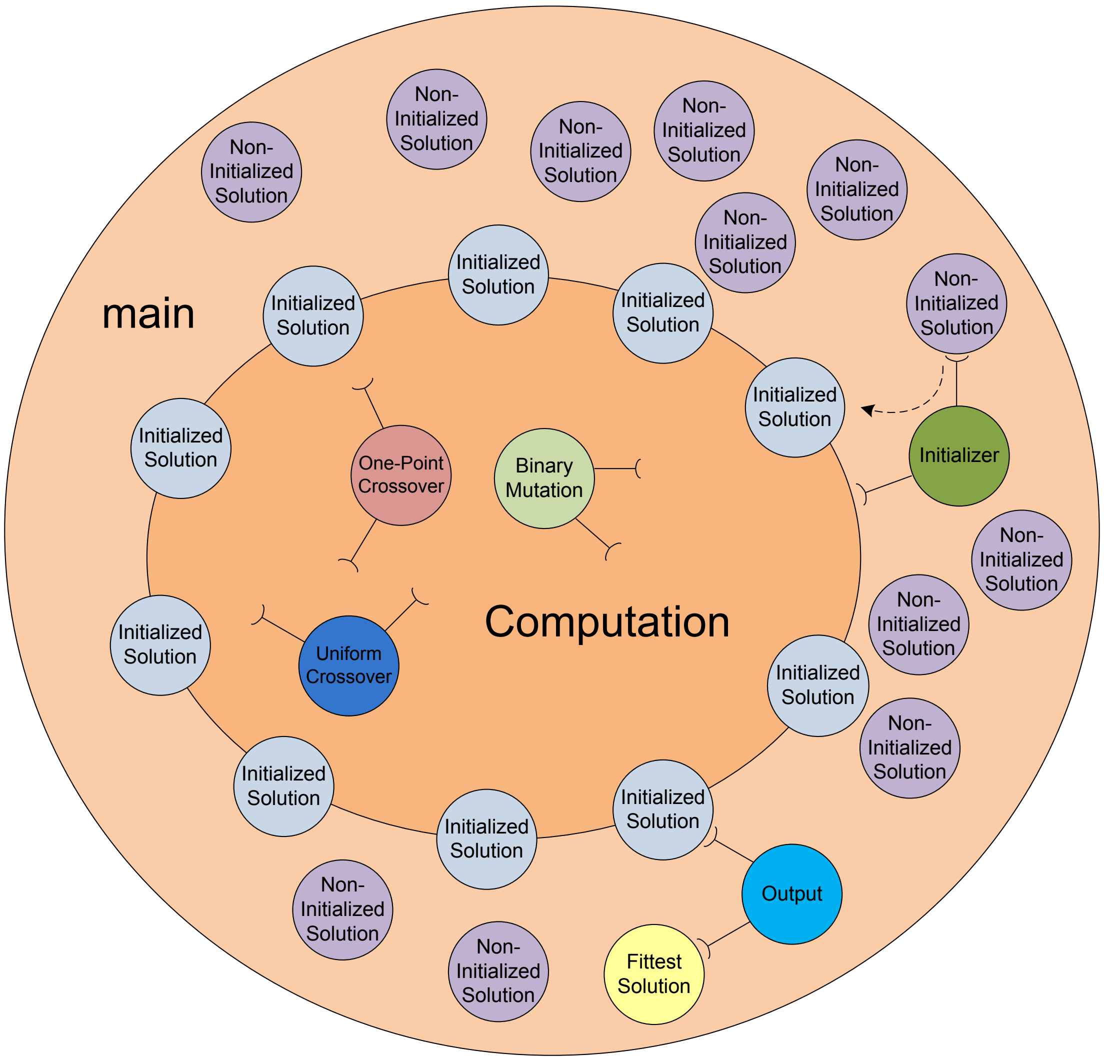


Figure 5.4. The binary knapsack SC model. Non-initialized solutions are initialized by the initializer context and added into the computation scope where they are transformed through genetic operations. The output context updates, if necessary, the fittest solution.

5.1.4 SC Binary Knapsack Model Implementation

The HAoS binary knapsack model (and the two other bio-inspired models presented later in this chapter) has been implemented applying the suggested development methodology[25] of section 4.5. The systemic analysis performed at the Conceptual Layer, resulting in laying out the model on SC graphical notation, makes the development of the

[25] The low level details of the implementation of the Application, Link and Physical Layers will be omitted here. All source code and configuration bitstrings can be found in the official HAoS webpage [191].

SC source code straightforward. In order to show this, the source code of the binary knapsack model is given in Listing 5.1. Its direct mapping to Figure 5.4 is evident as, after the functions and some useful labels are defined, the data systems, contexts and scopes are defined exactly as represented graphically. A description of the SC contexts functionality is given in Table 5.1.

Listing 5.1. Binary Knapsack SC model source code (50 solutions)

```
#systemic start
// define the functions
#function Output            %b00000001010000000000000000000000
#function Initialize        %b10000001010000000000000000000000
#function UniformCrossover  %b01000001010000000000000000000000
#function OnePointCrossover %b11000001010000000000000000000000
#function BinaryMutation    %b00100001010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label comp       %b1111111100000000
#label Sol        %b1000000000000000 // Initialized Solution System Type
#label nonInitSol %b0100000000000000 // Non-Initialized Solution System Type
#label FittestSol %b1100000000000000 // Fittest Solution System Type
#label zero2      %b00000000000000000000000000000000

// declare the scopes
main (%d0 %d0 %d0)
computation ( comp %d0 comp )

// data systems
OutSolution ( zero %d0 FittestSol )    // Fittest Solution
[1:50]solution ( zero %d0 nonInitSol ) // Non-Initialized Solutions

// context systems

// The initializer context defines an interaction between a non-initialized
// solution and the computation scope
initializer ([zero zero2 nonInitSol] Initialize(0,0) [comp zero2 comp])

// The output context defines an interaction between an initialized solution
// and the fittest solution
output  ([dontcare zero2 Sol] Output(0,0) [dontcare zero2 FittestSol])

// The genetic operator contexts define interactions between initialized solutions
uniformCross ([dontcare zero2 Sol] UniformCrossover(0,0) [dontcare zero2 Sol])
onePointCross ([dontcare zero2 Sol] OnePointCrossover(0,0) [dontcare zero2 Sol])
binMutation ([dontcare zero2 Sol] BinaryMutation(0,0) [dontcare zero2 Sol])

// set up the scopes
#scope main
{
       OutSolution
       [1:50]solution

       initializer
       output

       computation
}
#scope computation
{
       uniformCross
       onePointCross
       binMutation
}
#systemic end
```

Table 5.1. Summary of the Knapsack SC model functions. All functions run on the CPU.

Function Name	**Description**
Initialize	Initializes an non-initialized solution with random bit values, transforms it to initialized and inserts it also in the computation scope
UniformCrossover	Performs Uniform Crossover (each child bit can come from any of the parents with 50% probability - see Figure 5.3) using two initialized solutions as parents
OnePointCrossover	Performs One-Point Crossover (the child is produced by two consecutive parts, one from each parent, while the point that defines the length of the parts is chosen randomly - see Figure 5.3) using two initialized solutions as parents
BinaryMutation	Performs Uniform Crossover (a random bit-flip is performed to the parent to result in the child - see Figure 5.3) using one initialized solution as parent
Output	Compares a random solution with the fittest and updates the fittest if needed

Experiment Setup

The setup of the binary knapsack experiment replicates the configuration presented in [34], and shown earlier in Figure 5.1, in order to enable a direct comparison and evaluation of the performance of HAoS against prior SC implementations. In particular, identical copies[26] of the SC source code are used to run the experiment on HAoS, the original sequential (section 2.4.1) and the GPU-based (section 2.4.3) implementations. SCoPE (section 2.4.2) is excluded here, as it uses a different SC language and compiler. It is noted that the CPU is heavily used in this case, as the available low-level hardware-supported HAoS instructions are not suitable for the required high-level GA tasks.

The experiment involves running the 16-item SC binary knapsack model using 50, 100, 200, 400, 800 and 1000 solutions. As the time of convergence to the optimal chromosome can vary greatly for different runs and since this metric is mainly affected by the sequence of the selected genetic operations applied to the candidate solutions, it was decided that the three platforms would be evaluated based on reaching a certain amount of interactions, set in this experiment at 10000 (following the setup in [34]). An Intel® Core™ i7 950 CPU at 3.06GHz with 4 GB of RAM running on 32-bit Windows 7 and an NVIDIA GeForce GTX 260 GPU with 192 CUDA cores where used for the sequential and GPU versions. HAoS, as mentioned in sections 4.2 and 4.3.6, uses a

[26] The source code was not optimized for HAoS in order to enable a more fair comparison.

MicroBlaze processor with 64KB of RAM running at 100 MHz while the custom logic is clocked at 50MHz.

5.1.5 Results

The binary knapsack problem was one of the initial verification tests (see Table 3.5) that validated the functionality of HAoS by simulation. The verification environment lacked[27] precise timing information, as the CPU INTERFACE (see Figure 3.9) was emulated by a generic register-based mechanism implementing a handshake protocol and the timing of functions running on the processor was estimated by averaging the results of intrusive software-based profiling. Thus, timing results presented in this chapter will be solely based on live testing on the hardware platform.

However, in order to show that the verification environment can be used, if required (mostly for debugging purposes), even for such a high-level model, an excerpt from its output near the end of the simulation is given in Listing 5.2 showing that the optimal chromosome, with right weight and value (see Figure 5.2), is correctly identified (also noting the relatively long run time).

Listing 5.2. Verification environment output for Binary Knapsack SC model

```
#Time 52093085ns :: KSBINMUTATE{CPU}: (22591,22591) => (6207,22591)
[((sys1.sch1),(sys2.sch1)) => ((sys1.sch1),(sys2.sch1))] <sc:0,cxt:53,it:9998>
#Time 52097645ns :: KSOUTPUT   {CPU}: IF DIFF THEN BETTER (22587,22591) =>
(22587,22591) (sys1@51,sys2@1) [sys1.sch2:1=>1,
sys2.sch2:3=>3][((sys1.sch1),(sys2.sch1)) => ((sys1.sch1),(sys2.sch1))]
<sc:0,cxt:53,it:9999>
#
#Time 52098245ns :: SC Top Test finished..
#
#Time 52098245ns :: The SC program was loaded at 2613975.
#
#Binary Knapsack Problem Solution is 0101100000111111 (found at iter. 2259
<@13929180ns>)
#It has a weight of 79.000000 and a value of 124.000000
#
#Aborted 11 times due to schemata mismatch out of 10000 iterations.
#Aborted/Successful CAM compare Ratio : 0.11%
#
#Process time 305.01 seconds (simulation real time duration)
```

The comparison results for the different experiment configurations in terms of number of systems are given in Figure 5.5 as a semi-log graph (left) in order include the exponential growth of runtimes for the sequential implementation, while on the right

[27] Applies for behavioural (RTL) simulations. Precise and very precise timing can be obtained in the verification environment by running post-synthesis and post-place-and-route simulations including the processor subsystem but this results in excessive run times.

HAoS performance is compared only with the GPU-based version. All the implementations eventually identify the correct solution.

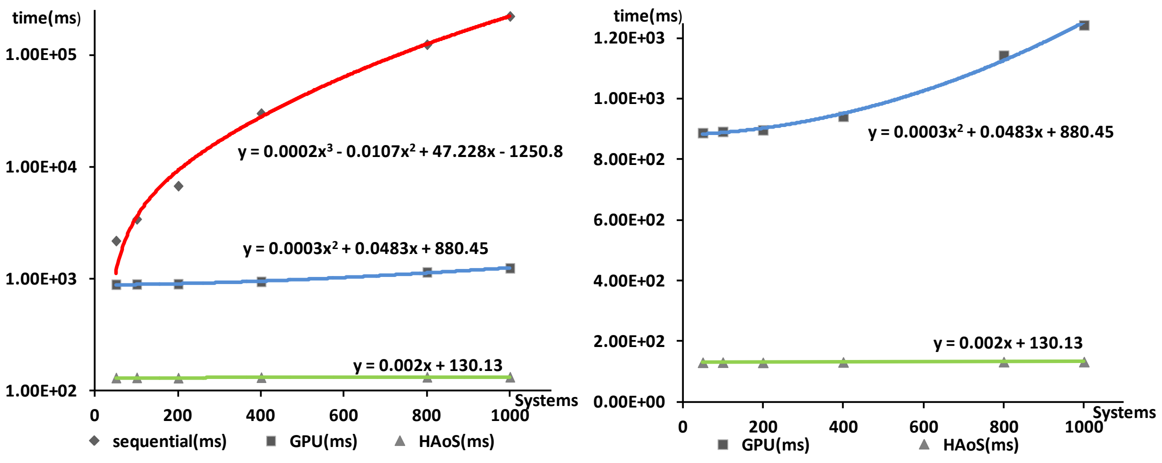


Figure 5.5. Binary knapsack problem experimental results across a range of number of systems comparing the sequential, GPU and HAoS SC implementations (left - semi-log) - zoomed in (right)

5.1.6 Analysis

Since the code implementing the transformation functions of the SC model is intentionally identical to the code running in the other implementations, the comparison results mainly represent the efficiency difference of the three platforms on valid triplet generation, affected mainly by the implementation of the schemata matching mechanism. As the number of system increases, the sequential implementation struggles as it handles schemata matching with an inefficient loop-based approach resulting in time complexity of $O(n^2)$, while the GPU version, by utilizing multiple stream processors, parallelizes part of this loop and achieves to decrease it in $O(n)$ [34] (shown in Figure 5.5 if the minimal highest orders factors are ignored). The truly parallel nature of the TCAM is the differentiating feature for HAoS since schemata matching is executed in constant time (one clock cycle – implying $O(1)$), shown in Figure 5.5.

As more clearly shown in Figure 5.6 (illustrating the normalised performance of HAoS in relation with the sequential and GPU SC versions), the superiority of the HAoS platform against prior implementations is evident as HAoS performs more than 1000x better than the sequential solution in the best case and approximately 8x-9x when compared with the GPU implementation.

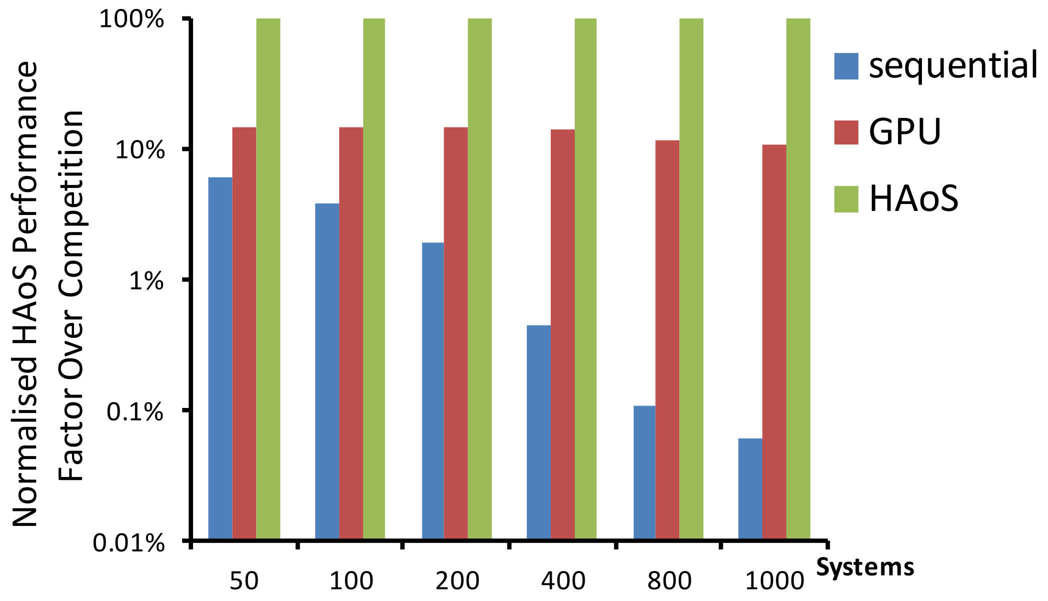


Figure 5.6. Binary knapsack experiments HAoS normalised performance compared to the sequential and GPU implementations

Moreover, it is important to be noted that, these results do not apply only in the problem class which the knapsack experiment belongs in, but they can be generalized to any problem that may be solved with SC given that HAoS provides sufficient computational resources when compared to alternatives. This is attributed to the importance of the schemata matching mechanism. The performance during the simulation of any natural system under the SC paradigm always rely on valid triplet generation (finding the two systems to interact according to the schemata templates of a, third, context system) and the actual computation which changes the systems according to the transformation function of the context system. Being able to perform the computational part, by executing code written in some well-established high-level language, is essential in order to achieve a generic and practical architecture not limited to the complexity of the computation. However, attempting to also simulate the schemata matching step, which is highly-parallel in nature, to the same sequential logic results in the findings of Figure 5.5 when compared to the suggested HAoS architecture for ascending number of systems, due to the inherent parallelism of the employed TCAM.

The schemata matching task can be viewed as a lookup in a 3-dimensional search space for each scope, where each dimension represents the indexes of each system in the valid triplet (s1 for first interacting system axis, s2 for second interacting system axis and c for context axis), or a 4-dimensional search space with the fourth dimension (s) representing the scopes of each system. In the current configuration the coordinates on s and c axes (valid scope and active context) are chosen randomly, as explained in section 3.5, and then HAoS locates the appropriate coordinates on axes s1 and s2 by performing two consecutive lookups using the TCAM to identify the interacting systems, and thus pinpoint the next valid triplet. Using a second TCAM, would enable the identification of both interacting systems at the same time, while using such a double TCAM structure (or effectively a dual-port TCAM) for each context system would enable a one clock cycle latency of finding all active triplet points in the 3-dimensional search space for each scope.

Extending this thought, again multiplying the number of TCAMs with the number of scopes would allow a truly parallel schemata matching mechanism that would give all interacting systems pairs for all contexts in all scopes at once. Such a structure would increase the number of required TCAM storage exponentially. Whereas the TCAM in the current prototype requires (N is the number of maximum supported systems):

$$Size_{TCAM} = (Length_{TCAM}) * (Width_{TCAM}) = (2 * Size_{Schemata}) * (N) = (2 * 16) * 1024 = 4KB$$

in the hypothetical 4-dimensional TCAM scenario it would require :

$$Size_{TCAM4d} = 2 * N * N * Size_{TCAM} = 4 * Size_{Schemata} * N^3 = 8GB$$

which is an enormous size when considering on-chip distributed memory today but may become feasible in the future or using an alternative TCAM implementation approach, e.g. an external TCAM configuration. The HAoS prototype, in this context, is a compromise between the inefficient sequential approach and the truly parallel but currently infeasible 4d approach for schemata matching.

Apart from the encouraging evaluation results, it should be noticed that the knapsack problem here acts as more than an example of a common synthetic computational problem being solved. Implementing a genetic algorithm in such a native way implies that the design encompasses, to some level, a lot of the natural properties of Table 1.1, addressing research challenge *Chg1*. As it has been shown in [22], such a system can present behavioural natural properties as self-adaptation, self-organization, fault-tolerance and self-maintenance while also implementing a stochastic, distributed and approximate computational model. Moreover, the successful execution of this first high-level SC model confirms the support of the suggested design for the underlying architecture of SC, addressing research challenge *Chg2*. The results additionally address research challenge *Chg3* by verifying the efficiency of HAoS against prior SC implementations.

5.2 Simulation of a Biochemical Process with HAoS: the MAPK Signalling Cascade

Enzymes regulate various cellular functions by catalyzing chemical reactions among biological molecules. One of the enzymes known to be responsible for gene expression and cell fate induction is the protein kinase which adds phosphate groups to proteins (a process called phosphorylation). Extracellular stimuli (mitogens) can activate protein kinases (Mitogen Activated Protein Kinases - MAPK) and start a chain reaction known as the MAPK signalling cascade [233], resembling the behaviour of a biological ultrasensitive switch which can bring the cell to discrete states.

The MAPK cascade model presented in this section was introduced by Huang and Ferrell in [233]. The authors give estimated results based on the numerical solution of rate differential equations derived by the involved biochemical reactions which are in accordance with in vitro experimental results presented in the same paper. The same model was used later in [234] as an application for their stochastic π-calculus simulator and in [22] as a case study to demonstrate the visualization framework of the high-level SCoPE implementation (see section 2.4.2 - Figure 2.14 actually represents a MAPK cascade model).

5.2.1 The MAPK Signalling Cascade

During the process involved in the MAPK signalling cascade, mitogens activate a MAPKKK (mitogen-activated protein kinase kinase kinase or MAPKKK or for simplicity here KKK) which in turn phosphorylates a MAPKK (mitogen-activated protein kinase kinase or MAPKK or KK) which itself phosphorylates a MAPK (mitogen-activated protein kinase or MAPK or K). The product of the first step, the activated KKK, is denoted as KKK*. In the next steps, one or two phosphate groups (P) are added to K and KK and result in single (KP and KKP) and double (KPP and KKPP) phosphorylated kinases.

The cascade can return to its initial state with the addition of phosphatase enzymes (KKPase and KPase) which remove a phosphate group from its substrate (this reverse process is called dephosphorylation). The phosphorylation and dephosphorylation processes along with their associated chemical reactions are illustrated in Figure 5.7.

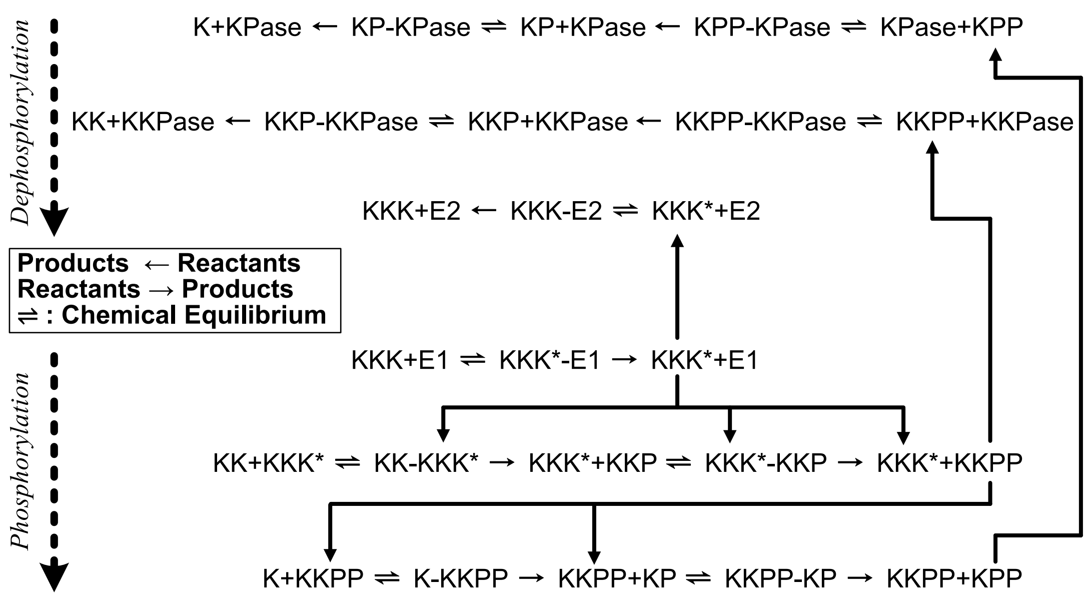

Dephosphorylation
K+KPase ← KP-KPase ⇌ KP+KPase ← KPP-KPase ⇌ KPase+KPP
KK+KKPase ← KKP-KKPase ⇌ KKP+KKPase ← KKPP-KKPase ⇌ KKPP+KKPase
KKK+E2 ← KKK-E2 ⇌ KKK*+E2
Products ← Reactants
Reactants → Products
⇌ : Chemical Equilibrium
KKK+E1 ⇌ KKK*-E1 → KKK*+E1
Phosphorylation
KK+KKK* ⇌ KK-KKK* → KKK*+KKP ⇌ KKK*-KKP → KKK*+KKPP
K+KKPP ⇌ K-KKPP → KKPP+KP ⇌ KKPP-KP → KKPP+KPP

Figure 5.7. Simplified biochemical description of the MAPK signalling cascade ignoring phosphate groups. The cascade is traversed forwards during phosphorylation (lower half, left-to-right) to result in a high concentration of activated kinases KKK*, KKPP and KPP and backwards during dephosphorylation (upper half, right-to-left) to return to its initial state. E1 and E2 represent the mitogens which activate KKKs and deactivate KKK*s, respectively. The intermediate product of each reaction (written here as the reactants connected by a hyphen) may either give the final products or be transformed back to the initial reactants[28].

5.2.2 Systemic Analysis

Taking into consideration the chemical reactions of the cascade (see Figure 5.7), a systemic analysis of the model is performed to identify the systems to interact and the form of the interactions between them. Since this is a biochemical model involving enzyme activity, selecting the level of abstraction of the SC model at the enzyme level is a straightforward decision.

Considering first the phosphorylation process, looking at the bottom half of Figure 5.7, since the addition of the mitogens E1 performs the activation of proteins KKK and transforms them in activated proteins KKK*, it is evident that the mitogens act as context systems, defining the interaction of a data system KKK with an implied free phosphate group (PF) resulting either on the KKK binding the PF and becoming KKK* with a bound phosphate group (PB) or no system being transformed. This binding of a

[28] To be more exact, the intermediate products are in chemical equilibrium state with the reactants, meaning that no real transformation can happen between them as both reactants and products are present at concentrations which have no further tendency to change with time.

PF and its transformation to a PB is implied for every reaction during the phosphorylation process and will not be repeated in the following analysis.

Moving in the next pair of reactions, a KKK* may transform a KK into a KKP and if this reaction is performed, the KKK* may also further transform the product KKP into a KKPP. Thus KKK* acts as a context, defining interactions between KK or KKP data systems and the implied PFs. It should be noted that while KKK is a data system, it is transformed to a context system (KKK*) when activated by E1 in the previous step. This implies that the context adapting functionality (discussed in section 3.6.1 and tested in section 3.8.1) is required by this SC model.

Looking at the last set of reactions, the product of the previous reaction KKPP (which has resulted from data system KKP) acts as a context, defining interactions between K or KP data systems and the implied PFs. The possible products of these interactions are respectively KP or KPP data systems. Thus the data system KKP is transformed to the context system KKPP. As shown in the bottom right part of Figure 5.7, KKPP and KPP are the last products of the phosphorylation process, so monitoring their concentration can give the state of the MAPK signalling cascade biological switch (the two distinct states of the switch are represented by either very high or very low concentration of these products) during an experiment using this model.

Using the same thought process and the chemical reactions of the top half of Figure 5.7, a systemic analysis can also be performed for dephosphorylation. For each reaction during this process, an unbinding of a bound phosphate group PB and its transformation to a free phosphate group PF is implied. Skipping the detailed explanation, the phosphatase enzymes KPase and KKPase and the mitogen E2 act as contexts. KPase may transform data systems KPP and KP to data systems KP and K, respectively. KKPase may transform data systems KKP to data systems KK but may also transform context systems KKPP to data systems KKP. This dual functionality cannot be represented by one system. Thus, a context should be used to model the former behaviour of KKPase, while a context adapter system should be used to model the later. Finally, context adapter E2 may transform KKK* contexts back to data systems KKK.

After identifying the systems involved in the SC MAPK cascade model above, the last part of the systemic analysis is with regards to the required scopes of interaction. An exact representation of the model would require that once a phosphate group P, modelled as a separate system, is bound by a protein kinase during phosphorylation, this P could not be able to be re-bound by another kinase. Furthermore when this specific kinase would interact during dephosphorylation, it would need to unbind the specific P that had bound before. This functionality, representing the physical location in a real biochemical system could be implemented using scopes. However, it would add considerable complexity to the SC model.

Since the main focus of this MAPK model lies on the concentrations of the activated kinases, representing the phosphate groups as a distinct system each is not a firm requirement. The total of Ps can also be modelled as a container system holding in its schemata counters for the number of the free groups PF and of the ones bound to a kinase PB. Furthermore, this model assumes that an adequate number of Ps is available, equal or greater than $|KKK|+2*(|KK|+|K|)$ (as one P can be bound by a KKK and two Ps can be bound by KKs and Ks), which would be enough to phosphorylate all kinases. Thus, the way Ps are modelled becomes irrelevant as it would only have an effect on the outcome of the model only in the case of having a shortage of P, since this would disable some reactions. So, in order to avoid adding unnecessary complexity to the SC model, all phosphate groups can be modelled by a single system taking part in all interactions or even be safely ignored, as long as the contexts appropriately alter the kinase systems. Following this approach, the number of total required systems is drastically decreased and also no additional scopes are needed, apart from the root scope.

In order to make the systemic analysis more clear, the systems and interactions of the HAoS MAPK model are described in SC calculus notation [22] in Table 5.2. The designations (P[F]) and (P[B]) as the second interacting system denotes that a free or bound phosphate group, respectively, would be used if each P was represented by a separate system while all interactions would involve the same P system if all Ps were represented by a container system. The parenthesis reminds us that the P system may even be ignored (in that case, the second template would match with any system). The simplified SC model discussed above is given in SC graphical notation in Figure 5.8.

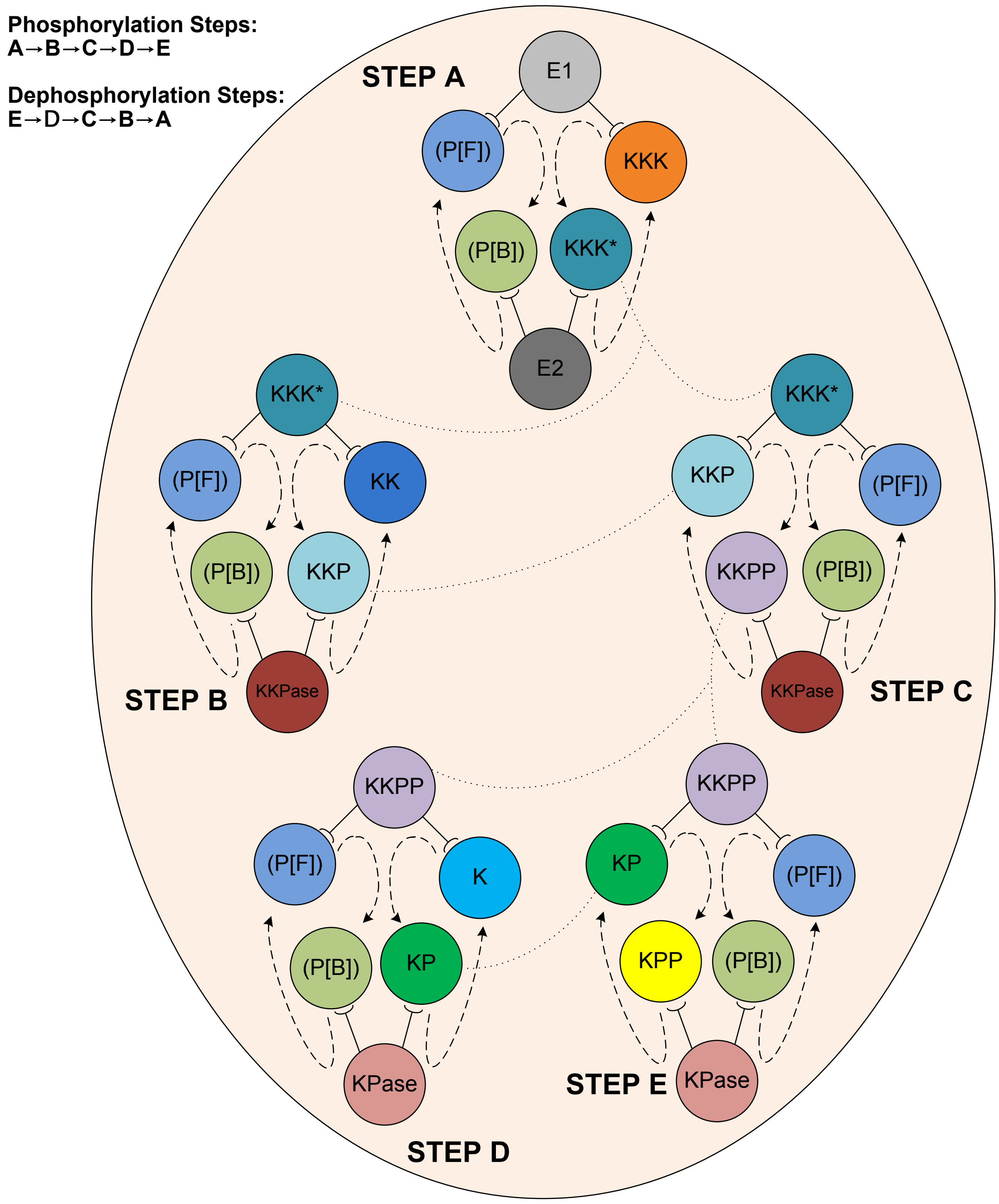


Figure 5.8. The HAoS MAPK model in SC graphical notation. During phosphorylation, E1 mitogens activate KKKs, which become KKK*s and phosphorylate KKs, which, when double phosphorylated, become KKPPs and phosphorylate Ks. This process is reversed during dephosphorylation with KPase and KKPase phosphatases and E2 mitogens bringing the cascade to its initial state. Systems with the same colour or connected with a dotted line may represent the same system (redrawn here for clarity). Phosphate groups may be modelled as separate systems including information about their binding ([F]:free, [B]:bound) or as one counter system representing all of them (P) or may even safely ignored as they do not have an impact on the behaviour of the model.

Table 5.2. The HAoS MAPK model interactions in SC calculus notation. The notation *S_1 }- C -{ S_2* indicates that the systems S_1 and S_2 match the schemata of a context system C and they interact in the scope of C according to its transformation function, while the | symbol separates different outcomes that an interaction may have. During phosphorylation a free phosphate group (PF) becomes bound (PB) to a kinase, while it is released during dephosphorylation when phosphate groups are included in the model. The different types of systems according to the systemic analysis are represented as follows here: Bold for contexts, italic for context adapters and normal for data

	Interacting Systems	→	Result
Phosphorylation	KKK }- **E1** -{ (P[F])	→	**KKK*** (P[B]) | KKK (P[F])
	KK }- **KKK*** -{ (P[F])	→	KKP (P[B]) | KK (P[F])
	KKP }- **KKK*** -{ (P[F])	→	KKPP (P[B]) | KKP (P[F])
	K }- **KKPP** -{ (P[F])	→	KP (P[B]) | K (P[F])
	KP }- **KKPP** -{ (P[F])	→	KPP (P[B]) | KP (P[F])
	Interacting Systems	→	Result
Dephosphorylation	KPP }- **KPase** -{ (P[B])	→	KP (P[F]) | KPP (P[B])
	KP }- **KPase** -{ (P[B])	→	K (P[F]) | KP (P[B])
	KKPP }- ***KKPase*** -{ (P[B])	→	KKP (P[F]) | **KKPP** (P[B])
	KKP }- **KKPase** -{ (P[B])	→	KK (P[F]) | KKP (P[B])
	KKK* }- *E2* -{ (P[B])	→	KKK (P[F]) | **KKK*** (P[B])

5.2.3 SC MAPK Signalling Cascade Model Implementation

Following the systemic analysis of the previous section, the MAPK HAoS model includes all types of supported systems: data, context and context adapters. Observing the systemic interactions of Table 5.2, data systems are differentiated by the number of protein kinases (Ks) and phosphate groups (Ps). So a straightforward approach to represent this information is to assign one bit for each K and each P in each data system. If separate phosphate groups are included in the model, another pair of bits can be used to represent the P data system type and its binding state. Furthermore, noting that contexts KKK*, KKPP and KPase can define interactions performing the same transformation on two different types of data systems (KKK* can select KKK or KK, KKPP can select K or KP and KPase can select KPP or KP), these interactions can be grouped together. Ternary bits (denoted with a question mark: ?) can be used in the data template defined in the schemata of these contexts, so that both possible data system types can be selected during schemata matching. This is shown in the labels section of the SC source code of the model, given in Listing 5.3.

Listing 5.3. MAPK Signalling Cascade SC model source code

```
#systemic start

// define the functions needed according to the systemic interactions
#function PHOSPH_E1       %b00000010010000000000000000000000 // Phosphorylation E1
#function PHOSPH_KKK      %b10000010010000000000000000000000 // Phosphorylation KKK*
#function PHOSPH_KKPP     %b01000010010000000000000000000000 // Phosphorylation KKPP
#function DEPHOSP_E2      %b11000010010000000000000000000000 // Dephosphorylation E2
#function DEPHOSPH_KKPASE %b00100010010000000000000000000000 // Dephosphoryl. KKPase
#function DEPHOSPH_KPASE  %b10100010010000000000000000000000 // Dephosphoryl. KPase

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label zero2      %b00000000000000000000000000000000

//// #label phosfree   %b0000001000000000 // uncomment to include phosphate groups
//// #label phosbound  %b0000001100000000 // uncomment to include phosphate groups
#label kkk        %b1110000000000000
#label kk         %b1100000000000000
#label k          %b1000000000000000
#label kkp        %b1100100000000000
#label kp         %b1000100000000000
#label kpp        %b1000110000000000
#label kkORkkp    %b1100?00000000000 // using ternary bit to match both kk and kkp
#label kkpORkkpp  %b11001?0000000000 // using ternary bit to match both kkp and kkpp
#label kpORkpp    %b10001?0000000000 // using ternary bit to match both kp and kpp
#label kORkp      %b1000?00000000000 // using ternary bit to match both k and kp

main (%d0 %d0 %d0) // declare the main scope

// data systems
[0:9]kkk ( zero %d0 kkk )
[0:99]kk ( zero %d0 kk )
[0:99]k  ( zero %d0 k )
//// [0:409]phosphate ( zero %d0 phosfree )
// uncomment commented contexts and comment the line above them to include Ps

// context systems

// Phosphorylation
e1 ([dontcare zero2 kkk] PHOSPH_E1(0,0) [dontcare zero2 dontcare])
//// e1 ([dontcare zero2 kkk] PHOSPH_E1(0,0) [dontcare zero2 phosfree])
kkkst ([dontcare zero2 kkORkkp] PHOSPH_KKK(0,0) [dontcare zero2 dontcare])
//// kkkst ([dontcare zero2 kkORkkp] PHOSPH_KKK(0,0) [dontcare zero2 phosfree])
kkpp ([dontcare zero2 kORkp] PHOSPH_KKPP(0,0) [dontcare zero2 dontcare])
//// kkpp ([dontcare zero2 kORkp] PHOSPH_KKPP(0,0) [dontcare zero2 phosfree])

// Dephosphorylation
e2 ([kkkst] DEPHOSP_E2(0,0) [dontcare zero2 dontcare])
//// e2  ([kkkst] DEPHOSP_E2(0,0) [dontcare zero2 phosbound])
kpase ([dontcare zero2 kpORkpp]  DEPHOSPH_KPASE(0,0) [dontcare zero2 dontcare])
//// kpase ([dontcare zero2 kpORkpp] DEPHOSPH_KPASE(0,0) [dontcare zero2 phosbound])
kkpasekkp ([dontcare zero2 kkp]  DEPHOSPH_KKPASE(0,0) [dontcare zero2 dontcare])
//// kkpasekkp ([dontcare zero2 kkp] DEPHOSPH_KKPASE(0,0) [dontcare zero2 phosbound])
kkpasekkpp ([kkpp] DEPHOSPH_KKPASE(0,0) [dontcare zero2 dontcare])
//// kkpasekkpp ([kkpp] DEPHOSPH_KKPASE(0,0) [dontcare zero2 phosbound])

#scope main // set up the main scope
{
        e1
        e2
        kpase
        kkpasekkp
        kkpasekkpp
        [0:9]kkk
        [0:99]kk
        [0:99]k
        //// [0:409]phosphate
}

#systemic end
```

Table 5.3. Summary of the MAPK SC model functions. All functions run on the CPU.

Function Name	**Description**
PHOSPH_E1	Represents the functionality of enzyme E1, transforming KKK to KKK* or leaving them unchanged
PHOSPH_KKK	Represents the functionality of activated kinases KKK*, transforming KK and KKP to KKP and KKPP respectively, or leaving them unchanged
PHOSPH_KKPP	Represents the functionality of activated kinases KKPP, transforming K and KP to KP and KPP respectively, or leaving them unchanged
DEPHOSP_E2	Represents the functionality of enzyme E2, transforming KKK* to KKK or leaving them unchanged
DEPHOSPH_KPASE	Represents the functionality of phosphatases KPase, transforming KPP and KP to KP and K respectively, or leaving them unchanged
DEPHOSPH_KKPASE	Represents the functionality of phosphatases KKPase, transforming KKPP and KKP to KKP and KK respectively, or leaving them unchanged

Context adapter systems e2 and kkpasekkpp, which respectively transform kkkst (KKK*) and kkpp contexts back to data systems, match the context systems according only to their transformation function, so the name of the systems to be matched are used in the SC source code instead of separate templates. Finally, not all defined systems need to be part of the main scope at the beginning of the SC program, as contexts KKK* and KKPP are products of interactions which occur along the execution of the model.

As discussed in the previous section, the MAPK model used in this experiment is a simplified[29] version, in terms of the representation of the phosphate groups. While this version ignores Ps, the SC source code given in Listing 5.3 includes for completeness the changes (in comments) that would be needed to include Ps in the model[30].

[29] If the simplification of the model (resulting in decreasing the overall number of required systems) was not possible, the size of the model would be restrictive for execution on the implemented prototype HAoS platform, which after the last revisions of the design officially supports models involving up to 511 systems.

[30] The inclusion of a data system representing all phosphate groups would also be quite simple as this system would need to be defined and included in the main scope. Moreover, a single matching label (e.g. "phos") corresponding to the type of this phosphate system would replace

On the high-level software development side of the model, context transformation functions plugins (Application Layer, Figure 4.5) were developed to implement the transformation activity of each interaction. These functions, running on the embedded processor, mainly handle the required systems' type alterations (see Table 5.2 and Table 5.3) and the logging of the output from the model (concentration, represented by the number of systems, of the final products of the chemical reactions) on the SD card. It is noted that in the case of context adapting the software should notify the hardware (through appropriately setting a configuration register) that a system has changed its type (from data to context or vice versa).

Experiment Setup

The MAPK signalling cascade model has been previously simulated with the Stochastic Pi Machine (SPiM) simulator in [234] and with the high-level SC implementation (SCoPE) in [22]. Thus, modelling the MAPK cascade with HAoS using the same configuration and initial conditions previously used in the experiments presented in [234] and [22] enables the direct comparison and evaluation of our prototype platform against these alternative simulators in terms of quality of results and performance (based on execution speed).

The common configuration used in all three modelling environments, in accordance with the experiment in [234], involves 10 KKKs, 100 KKs and 100 Ks kinases, 1 E1 and 1 E2 enzymes, and 1 KKPase and 1 KPase phosphatases. All protein kinases are initialized to a non-phosphorylated state. All chemical reaction rates (which in this model are translated to interaction probabilities) are set to a nominal value of 1. This implies that since every interaction may either change the interacting systems or leave them unchanged, each outcome has a probability of 0.5.

5.2.4 Results

Since the results of the SPiM simulator, modelling the cascade, have been shown in [234] to be in agreement with the actual response of this signalling network, observed in a wet lab, we can use them as a reference to validate the functionality of HAoS. As illustrated in Figure 5.9, the simulated behaviour of the cascade is shown to be in

all "phosfree" and "phosbound" labels in all context definitions, as all interactions would involve this single P data system.

agreement for all three used simulation environments ((a) SPiM, (b) SCoPE and (c) HAoS).

However, while all three simulators capture the functional behaviour of the cascade correctly, their simulation running times vary significantly. Simulating biological processes was the key consideration for their design but their performance depends heavily on their implementation. While, as previously discussed, SCoPE is a high-level software (C++) implementation of SC, the SPiM simulator is a functional programming [235] (F#) software implementation of the SPiM language, developed by Microsoft Research, which is based on stochastic π-calculus [49] and standard kinetic theory of physical chemistry [234]. HAoS being a hardware-based implementation, it benefits from the inherent parallelism of the TCAM and low-level hardware latency-aware optimizations. As seen in Table 5.4, HAoS outperforms the alternative software-based simulators: SPiM by a factor of 17.3 and SCoPE by a factor of 11323.6 in the case of the MAPK cascade.

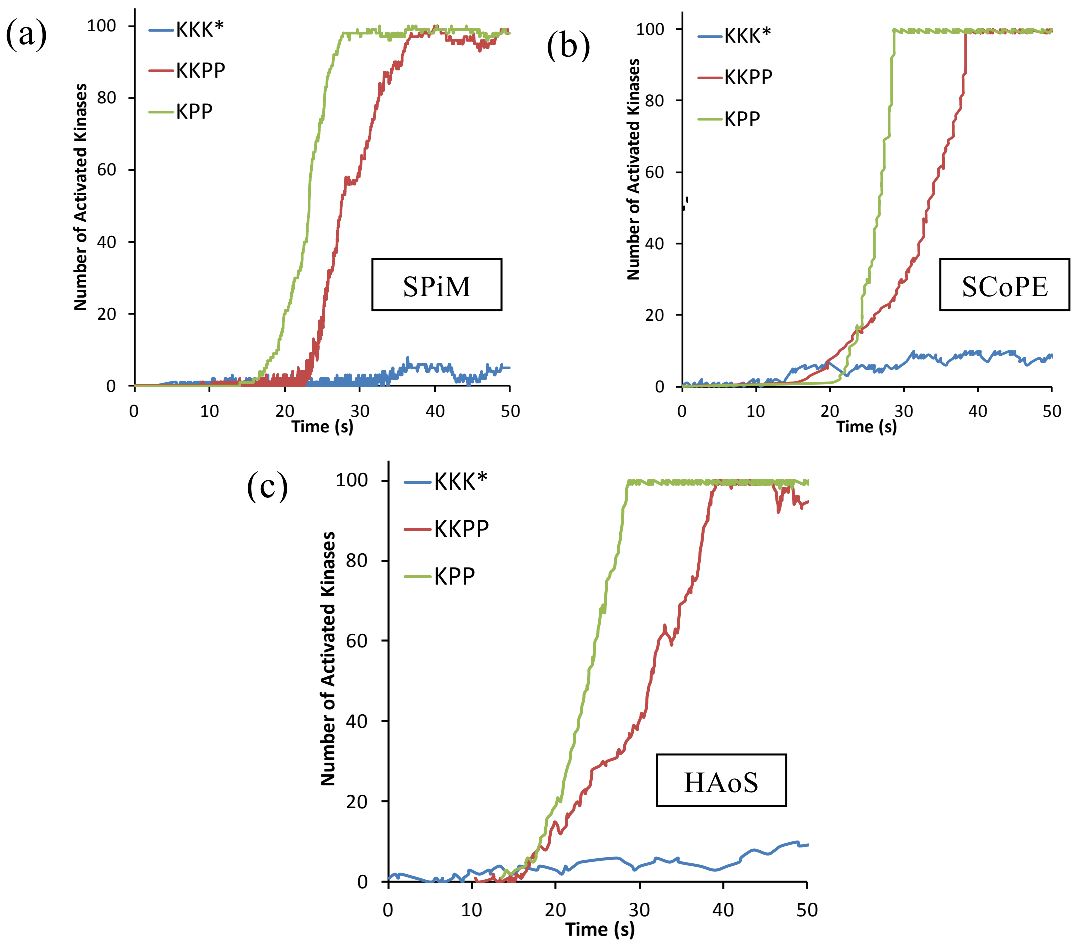


Figure 5.9. Traversing the MAPK signalling cascade with (a) SPiM, (b) SCoPE and (c) HAoS with initial state: 10 KKKs, 100 KKs, 100 Ks, 1 E1, 1 E2, 1 KKPase and 1 KPase

Table 5.4. Performance of the HAoS, SPiM (v1.13) and SCoPE simulators based on simulation duration, simulating 50 seconds of the MAPK cascade evolution, with initial conditions as stated in Figure 5.9. Values shown are the average over 20 runs acquired using PowerShell on Windows 7 64-bit, Core i7 Q840 CPU with 8 GB RAM for the software-based simulators, while HAoS is implemented on a Xilinx Virtex-6 FPGA utilizing a MicroBlaze soft processor running at 100MHz with 64KB of dedicated memory.

	HAoS	SPiM	SCoPE
msec	58.1	1004.9	657902.5
factor	1	17.3	11323.6

5.2.5 Analysis

As shown in Figure 5.9, the behaviour of the MAPK signalling cascade as a biological switch that can bring the cell in discrete states is captured by all three used modelling environments, as all instances of KK and K kinases are double phosphorylated and result in KKPP and KPP respectively. However, minor differences can be observed in the output of the three simulators regarding the signal response sensitivity of the model in terms of the rate of KPP production as a result of KKPP activations near both states and especially as the concentrations reach the total number of the available kinases plateau. It is noticed that the SPiM simulator reflects more accurately the decrease in available kinases, resulting in reaching the final state (after a decrease in the activation rate) in a more gradual manner presenting a more rounded sigmoidal finish. Both SC simulators present a more abrupt finish as the chemical rates are translated to interaction probabilities in the software level, resulting in a less sensitive behaviour as fine-tuning their values is also affected by the interaction order mechanism employed in the implementation.

The timing results of Table 5.4, also illustrated as the performance factor provided by HAoS against SPiM and SCoPE (after normalisation) in Figure 5.10, reveal the low efficiency of the high-level SCoPE implementation, as a result of its increased provided flexibility and the ineffective brute-force schemata matching mechanism which iterates through random triplets of systems until one that can define an interaction is identified. The SPiM simulator, while running on a conventional CPU, achieves a considerably better performance due to its optimized implementation of stochastic π-calculus. It models the various interactions with separate processes communicating through predetermined channels with dynamically adjusted interaction rates, affected by the number of possible combinations of inputs and outputs on each channel [234]. It is important to notice that while the SC models set the level of abstraction at the enzyme level, modelling each kinase as an individual entity, the SPiM model simulates all instances of the same type of kinases in a separate unified group, resulting in a reduced simulation complexity and increased performance.

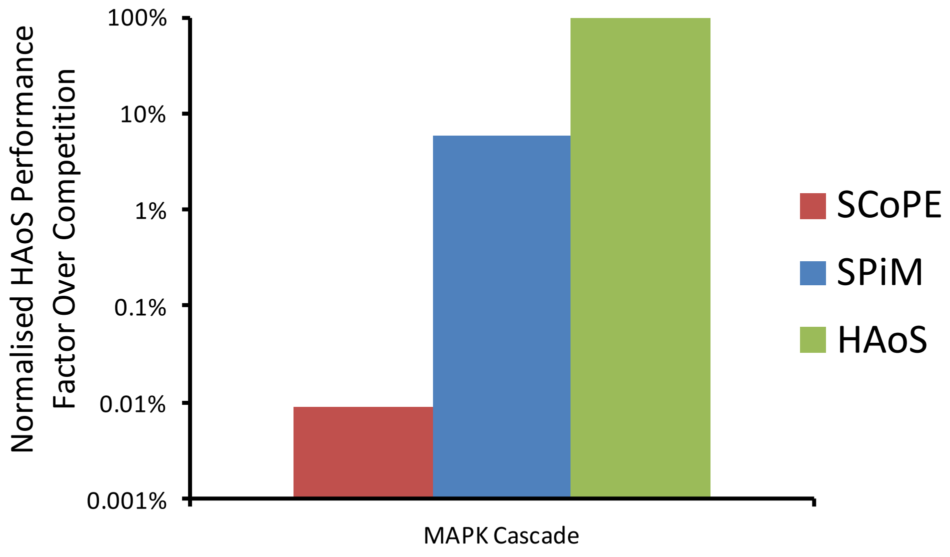


Figure 5.10. MAPK cascade experiment HAoS normalised performance compared to the SPiM and SCoPE simulators

Yet, in spite of this relative difference in the implementation of the MAPK model, HAoS still achieves to outperform SPiM due to its highly parallel nature and low-level optimizations which implement the implied SC architecture efficiently. Thus, these results confirm that research challenges *Chg2* and *Chg3* have been adequately addressed since the suggested platform provides support for the architectural features of SC while achieving this with efficiency. This is shown by the capacity of HAoS outperforming not only prior SC implementations, as SCoPE, but also rival simulators, as SPiM.

5.3 Modelling the Effect of Chromosome Missegregation and Typical Cancer Therapy Approaches in Tumour Evolution with HAoS

Medical research is given a high priority amongst all research activity, mainly because it usually addresses issues that may have a profound role in the course of human life. A cure for cancer may be called the “holy grail” of medical research on life-threatening diseases, due to the increasing levels of cancer-related mortality being observed during the last decades. Cancer is a group of diseases, having in common irregular cell growth, which are commonly associated with multiple external factors but without a registered common cause [236]. The ultimate goal of cancer research is to provide an effective way of prevention, diagnosis and therapy for the large number of individual cancer diseases, but in order to accomplish that, researchers should first gain understanding of the complex underlying tumour development pathways. The explosive technological advancements of the past century have been enabling this by means of wet lab experiments but also by more efficient computational models (in silico) to assist and sometimes guide in vivo (in living biological organisms) and in vitro (in a test tube) experimental research.

While there is a wide range of types of cancers, usually classified according to the organ developing unregulated cell growth, most of them have been linked to a variety of genetic irregularities along the development of the tumour [237]. Whether this abnormalities play a causal and initiatory role or if they are just consequences of cancer is still an open question [237]. An example of such a genetic anomaly is aneuploidy - defined as a cellular state of having an abnormal number of chromosomes [238]. One of the mechanisms associated with this lack or excess of chromosomes in cells is chromosome missegregation - the erroneous duplication of chromosomal genetic material during cell division [238], resulting in a change in the number of chromosomes in daughter cells, also known as aneuploid cells.

This section presents a reimplementation of a model encapsulating the role of chromosome missegregation in the development of a tumour. In order to further show the modelling capabilities of SC and HAoS, we are not limiting the biological model in just simulating the interactions between the cells in a tissue, but also demonstrate that external stimuli can also be integrated in it, by means of human-induced changes in the internal state of the tissue - caused by typical cancer treatment approaches, chemotherapy and surgery. The chromosome missegregation reference model is presented in [238] and is implemented optimally in a high-level software programming language (C++). This test case attempts to show how we may approach the implementation of such a high-level model using SC and the HAoS development tools presented earlier and evaluates the functionality and performance of HAoS and its high-level functional model (essentially the HAoS functional simulator) against an optimal high-level software implementation. The selected reference biological model is not demonstrated just as a real-world application but it was specifically chosen as a worst-case scenario, as explained later, in terms of performance comparison with a rival software implementation approach in order to stress the HAoS programming platform to its computational limits.

5.3.1 The Cancer Model

The reference cancer model, drawn from [238], is an agent-based model. An agent, in a computer-based simulation context, is defined as a self-contained entity with a set of pre-defined initial characteristics, according to a number of base behavioural rules, and with the potential of being self-adaptable - adjusting its behaviour by learning from experience and altering its base rules [238]. The agents are chosen according to the selected level of abstraction of the model, and in this case the agents are the cells of an organ or biological tissue.

Description of the Reference Model

The behaviour of the tissue is regulated by the intrinsic characteristics of the population of cells. Each cell initially contains two sets of identical chromosomes with a set of regulatory genes, each responsible to control a specific cellular process. A pair of each type of gene is distributed among the pair of chromosomes to reflect the nature of the simulated diploid genome. The key cellular processes modelled and regulated by the genes are [238]:

- Cellular Division: The biological process where a cell duplicates its DNA and then separates the two copies giving birth, to two genetically identical daughter cells, replacing the parent cell.
- Cellular Apoptosis: The process of regulated cellular death to prevent excess growth and maintain a homeostatic state - preserving a stable cell number and tissue structure.
- Chromosome Segregation: The process of redistributing genetic material (DNA) between daughter cells during the mitotic step of division. Errors during this process may result in an asymmetrical distribution of chromosomes - commonly known as missegregation. Genes regulating this process are known to increase fidelity when present.

Cellular division genes are an abstraction of proliferation controlling genes, known as proto-oncogenes, apoptosis genes are an abstraction of tumour suppressor genes while chromosome segregation regulatory genes represent genes that control reliable segregation [238]. Following this brief introduction, the reference cancer model as described above is illustrated in Figure 5.11.

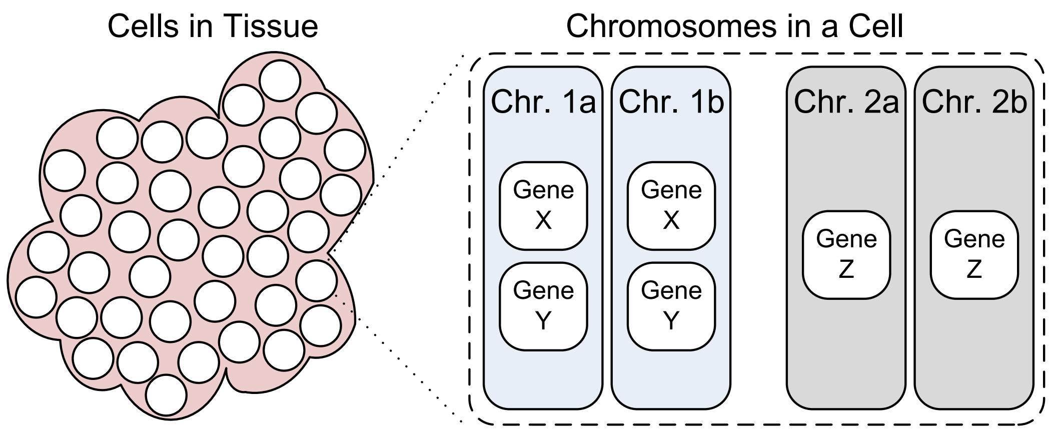


Figure 5.11. The reference cancer model. (Left) Abstract cells in a biological tissue are used as agents. (Right) Each cell includes a pair of chromosomes; each of them initially has the same genetic content - number of genes. Each gene controls a specific cellular process: Division, Apoptosis and Chromosome Segregation

In addition, in order to explore the effects of the initial genetic configuration and gene linkage (genes being part of the same chromosome) in cell growth and genetic diversity, three different chromosomal gene distributions are used. Initially, each chromosome pair has two copies of the gene or genes it contains. The segregation regulatory genes are always part of the second chromosome pair. Division genes are genetically linked with apoptosis genes (both being part of the first chromosome pair) in Chromosome Distribution A. In Distribution B, apoptosis genes are contained in the first chromosome pair while division genes are part of the second chromosome pair. In distribution C these positions are reversed (division genes in the first pair and apoptosis genes in the second). The three chromosomal distributions are illustrated in Figure 5.12.

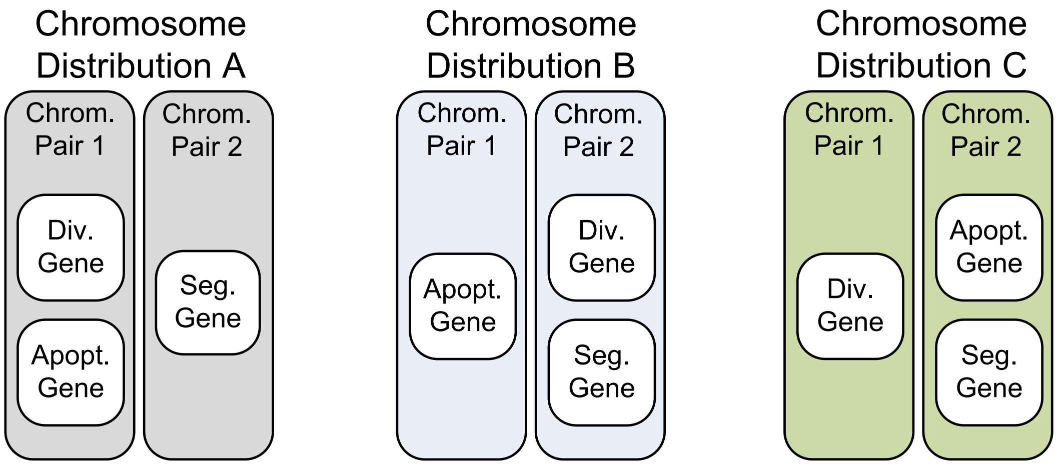


Figure 5.12. The three genetic configurations, employing different gene chromosomal linkage, are used to explore the effect of the initial genetic distribution in the overall tissue growth and behaviour

The reference model also investigates the response of the simulated tissue in typical cancer treatments. Thus, four therapy-related scenarios are examined for each chromosome distribution simulated:

- Therapy Scenario A: No therapies
- Therapy Scenario B: Surgery only - localized tumour removal
- Therapy Scenario C: Chemotherapy only - drug or radiation based attack on excessively dividing cells, usually using bio-markers
- Therapy Scenario D: Both therapies - a combination of surgery followed by chemotherapy

Implementation of the Reference Model

The specific mechanics of the reference model, as it was implemented in [238], are described in Algorithm 5.1.

For each iteration of the simulation (which implements a separate experiment), and according to the simulated chromosome distribution, the regulatory genes control the fate of each cell in the tissue. The corresponding process for each gene is executed according to a probability *p* proportional to the number of copies *N* of the specific regulatory gene in the chromosome, and in extent in the specific cell, and a fixed parameter *r* associated with the empirical rate of the process, derived from relative literature and experiments performed in a lab environment. The probabilities of apoptosis p_{ap}, division p_{div} and missegregation p_{msg} (p_{msg} is adjusted to the number of different chromosomes to be selected, in this case 4: chr1a, chr1b, chr2a, chr2b) are given according to Equations 5.1.

$$\begin{aligned} p_{ap} &= r_{ap} N_{ap} \\ p_{div} &= r_{div} N_{div} \\ p_{msg} &= r_{msg} (4-N_{msg}) \end{aligned} \tag{5.1}$$

Thus, the selected reference model, with behaviour of considerable complexity, will serve as a realistic demonstration application modelling a biological system taking into consideration a multitude of factors, constraints and abstractions. Evidently, agent-based models are suitable candidates for a SC implementation, as the notion of an agent aligns well with the notion of the fundamental SC element, the system. While interactions between cells are not modelled, the fate of the tissue is determined by the behaviour of each cell, controlled by genetic and external stimuli.

Algorithm 5.1. The reference cancer model algorithm [238]**. Different experiments are executed in sequence. Each experiment runs until the tissue has reached a threshold size in number of cells (*TH_{END}*) or a maximum number of generations. During each generation (or timeslot), each living cell in the tissue may die, divide (and missegregate or not) or remain unchanged according to the corresponding probabilities.**

```
Initialize the model with random seed
Set the carrying capacity of the tissue to a fixed number
for all experiments do
 Create tissue with an initial population of cells, each with two diploid chromosomes.
 Each chromosome in each cell is given one or two genes based on chromosome distribution
 repeat
  for all cells in tissue do
   if during surgery then
    Kill current cell if tissue size (total cells) exceeds its initial size
   else if no chromosomes in the cell (mitotic checkpoint) then
    kill current cell
   else if total cells > tissue capacity and apoptosis probability $p_{ap}$ satisfied then
    kill current cell
   else if division probability $p_{div}$ satisfied then
    if during chemotherapy (lasts fixed number of timeslots after cancer detection) then
     kill current cell
    else
     Add mitotic cell (birth of new daughter cell, identical to current parent cell)
     if missegregation probability $p_{msg}$ satisfied then
      randomly select r : one of the four chromosomes in the cell
      perform asymmetrical division instead (increment daughter r, decrement parent r)
     end if
    end if
   else
    current cell remains unchanged
   end if
   go to next cell
  end for
  Update number of cells
  if number of cells > cancer detection threshold ($TH_{DET}$) and no previous therapy then
   initiate therapy (surgery and/or chemotherapy)
  end if
  Increment timeslot t (generation counter - abstract time)
 until reached maximum number of generations or cells (End Threshold - $TH_{END}$)
 print output results
end for
```

The reference model is constructed in [238] as a linked list with each of its elements representing a cell, making traversing through all cells during each generation, or timeslot of simulation, trivial and optimal. In contrast with previous test cases, the reference model does not require that a search space is explored for potential interactions, but only that each cell function, during each abstract time step, may have an outcome according to the nested-if statement of Algorithm 5.1. However, in order to implement the model in SC, the interactions among cells and the tissue should be explicitly defined, thus a systemic analysis of the model is required, given in the next section. It is noted that, since the agents (cells) are selected in a sequential manner (just by visiting the next node of the linked list), their selection process is a best-case scenario, making the model a worst-case scenario in terms of comparison for HAoS.

5.3.2 Systemic Analysis

Following the suggested model development methodology in section 4.5, the previous paragraph describing the reference cancer model corresponds to the initial conceptual layer, since a thorough understanding of an existing model to be implemented in SC is crucial prior to any development effort. As with the previous models presented in this chapter, the next step is a comprehensive systemic analysis to identify the level of abstraction, systems and the contextual interactions among them.

As mentioned before, working on an agent-based model simplifies this task as the level of abstraction and most interacting systems are given as the agents. However, it is important to analyze the dynamics among them to define an optimal way to represent their interactions, which can commonly be an iterative process. Due to the increased complexity of the cancer model, a detailed description of the thought process and decisions leading to a number of possible suitable SC model variations is presented below.

This systemic analysis will result in four SC cancer model alternatives, implemented and compared with the reference model later in this chapter. In order to reach these four final SC models, the analysis will begin with a set of intermediate steps which will add the specific features of the reference model gradually.

SC Cancer Model Development Step 1: The Base SC Cancer Model

Starting with Algorithm 5.1, and retaining the level of abstraction at the cellular level, it is quickly noticed that a pool of data systems is required to represent the living cells. The obvious cellular functions that can act as transformation functions in contextual systems are cell death and cell division. Considering initially therapy scenario N (no therapies), a cell may die at the mitotic checkpoint (when no chromosomes are left in it) or when the tissue has grown over its carrying capacity and the apoptosis probability is satisfied.

The death context mainly tries to identify a living cell and kills it under the conditions mentioned above, where initially the second interacting system may be any other cell (since this is unaffected - death examines a cell at a time). SC rules explicitly state that systems can be transformed but never destroyed so a living cell is transformed to a dead one (or waste system) by the death context if killed.

The divider context, in contrast to death, may affect both interacting systems. The notion of division implies that a new cell will be created, which will ideally be reproduced as an identical copy of the parent cell. As this new cell cannot be created from nothing, the divider will define an interaction between a living cell representing the parent and a waste system that may be transformed to a new living cell through division, representing the daughter cell. The notion of waste systems (possibly previously killed cells) being transformed to living cells is biologically plausible as during the division process the parent cell consumes energy acquired by nutrients in its environment. This initial SC cancer model is illustrated in Figure 5.13.

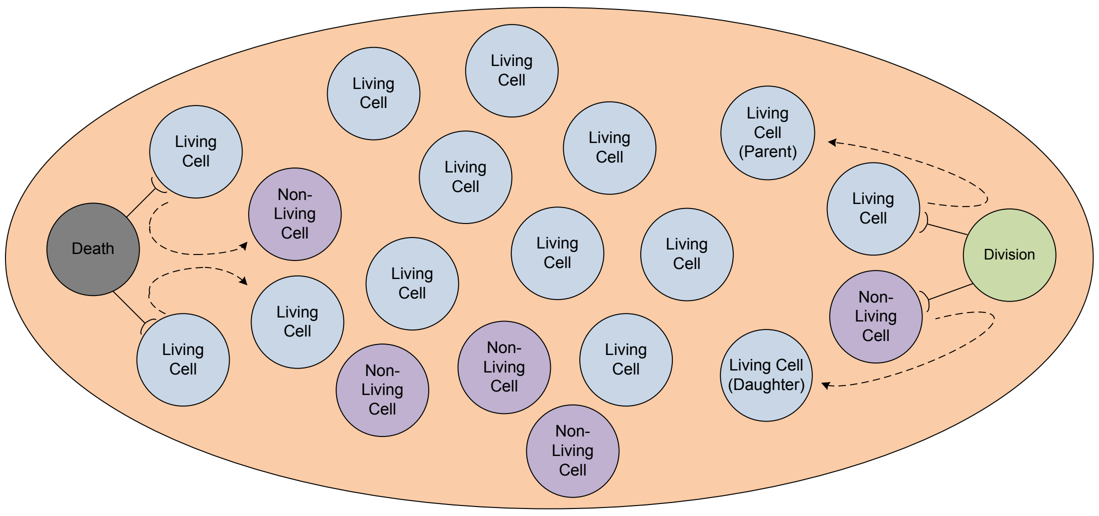


Figure 5.13. Initial SC cancer model. A death context transforms living cells to non-living ones while non-living cells act as daughter cells in division interactions, transformed in living cells by a division context.

SC Cancer Model Development Step 2: Integrating the Tissue

The next step is to integrate the notion of the tissue to our SC model. As the total of the cells makes up the tissue, another data system is needed to represent the tissue which has in its scope all the cell systems. The size of the tissue, in terms of cells, is an important metric for the model. While the total number of cells could be stored in a global variable in the user code portion of the HAoS program, in order to comply with the SC feature of systems having local knowledge, it is instead stored locally in one of the schemata of the tissue data system. Thus, a mechanism for updating the size of the tissue should be added to the SC model - incrementing the total number of living cells with every division and decrementing it with every cell death. Furthermore, while each living cell belongs to (meaning it is in) the scope of the tissue, every non-living cell (either dead or nutrient cell) should be out of its scope. Thus, the processes of cell death and division are broken in two steps: (a) perform the process and (b) update the total number of cells in the tissue and the scope memberships of the interacting systems.

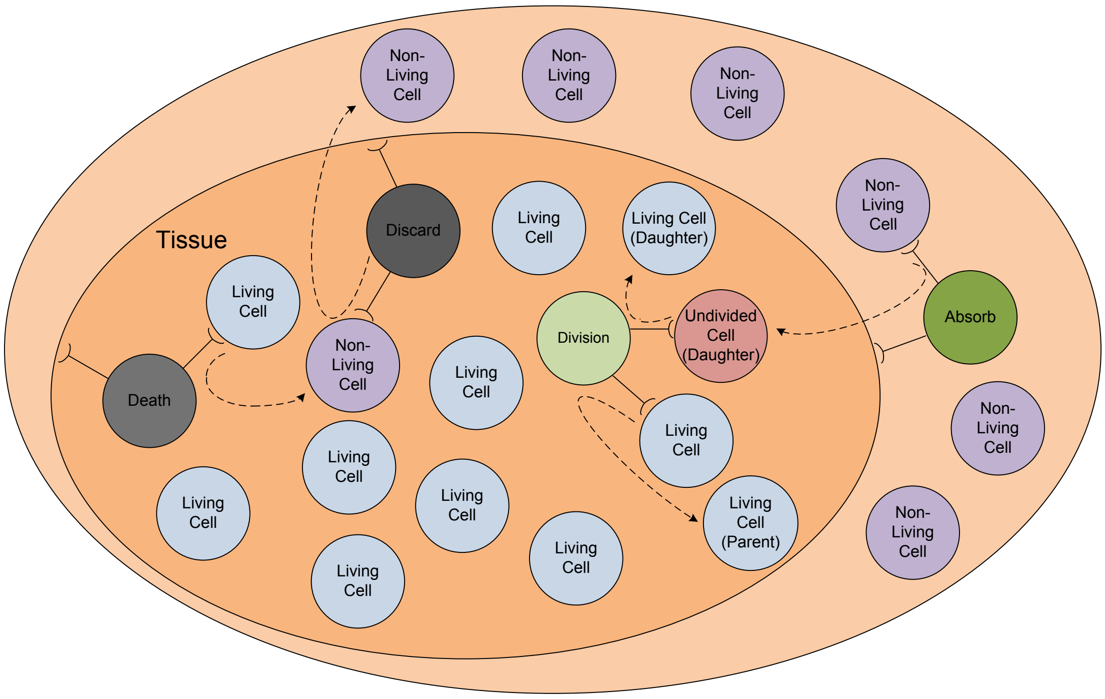


Figure 5.14. Revised SC cancer model with tissue and two-step cell death and division processes. During death a living cell is transformed to a non-living one which is then discarded from the tissue. During division, a non-living cell is absorbed in the tissue and then it is transformed to a living one, becoming the daughter cell

For this reason, two contextual systems are added to the model to handle the tissue size and scopes updating. For each possible cell division, a non-living cell interacts with the tissue through an "absorb" context and if the probability of division is satisfied, it is transferred from the tissue external environment inside its scope changing its type to being (yet) undivided resulting to the size of the tissue being incremented. This undivided cell then interacts with a living cell through the division context in the scope of the tissue and acts as the division daughter cell - becoming a copy of the parent living cell. Cell death is performed in a similar two-step process. A living cell interacts with the tissue through the death context and if the death conditions (from Algorithm 5.1) are satisfied, it is transformed to a dead (non-living) cell. This non-living cell then interacts with the tissue through a "discard" context and it is transferred back to the external environment of the tissue (outside the tissue scope), becoming a nutrient non-living cell (and a valid candidate for division). This discard context also decrements the size of the tissue. Following these thoughts, the updated SC cancer model is shown in Figure 5.14.

As noticed in Figure 5.14, missegregation is not explicitly controlled by a context. This was decided as segregation is part of the division process (happening when the genetic content of the parent cell is copied to the undivided daughter cell to become a new living cell), and as such this functionality is integrated in the division context. Also, while it seems visually suitable to have only living cells as part of the tissue and non-living ones making up its environment, it is reminded that non-living cells are used here as an abstraction for energy consumed or released by the tissue during cell division and death. Taking into consideration that the implied scope manipulation (cells constantly changing scopes as they are discarded from or absorbed into the tissue) may have a considerable computational impact, degrading the (timing) performance of the model, and since this membership does not have an active biological role for the model, it was decided that all systems may be part of the tissue. This way, meaningless scope alterations are avoided as all systems belong to the scope of the tissue.

SC Cancer Model Development Step 3: Integrating the Cancer Therapies

Returning to Algorithm 5.1 and as mentioned in the previous section, the reference model also includes human-induced interference in terms of common cancer treatments, surgery and chemotherapy. Therapies are applied to the tissue following cancer detection - when the number of living cells reaches a predetermined detection threshold (TH_{DET} in Algorithm 5.1). Surgery is performed during one timeslot, immediately after detection and removes a number of living cells, bringing the tissue back to its initial size. Chemotherapy is performed during a fixed duration of timeslots (9 in our experiments, in accordance to [238]), either after detection (therapy scenario C) or after surgery (therapy scenario D). During chemotherapy, cells that are meant to divide, instead die. Thus, cell death during chemotherapy is included in the division process while surgery requires effectively the same functionality by the death context but is executed only during the surgery timeslot. This also implies that the surgery context has priority over all other contexts during surgery if the number of living tissue cells exceeds its initial size.

In order to control the therapy processes, the therapy state of the tissue is embedded, as corresponding flags, in one of its schemata. This is sufficient for controlling chemotherapy (being integrated in division) but not for controlling surgery. The reason is that the division context already defines two interacting cell systems, the parent cell and another one to become the daughter cell, so the state of the tissue cannot be used to block divisions while it is in surgery state, as required by the reference model. Thus an additional intermediate (preparatory) step is needed, defining an interaction between the parent cell and the tissue, in order to be decided if this cell can proceed to division depending on the tissue state. This is accomplished with another "fertilizer" context, which transforms a living cell to a parent cell if the therapy state of the tissue is not in-surgery. The parent cell then interacts with a nutrient cell (the non-living cell to be used during division as a daughter cell) and both cells are transformed by the division context: the parent cell to a living cell and the daughter cell to a divided cell. The divided cell, being the product of the division process, then interacts with the tissue and is finally transformed to a living cell by the absorb context which also updates the size of the tissue. Thus, division is now a three-step process. The therapy-enabled cancer SC model, described above, is shown in Figure 5.15.

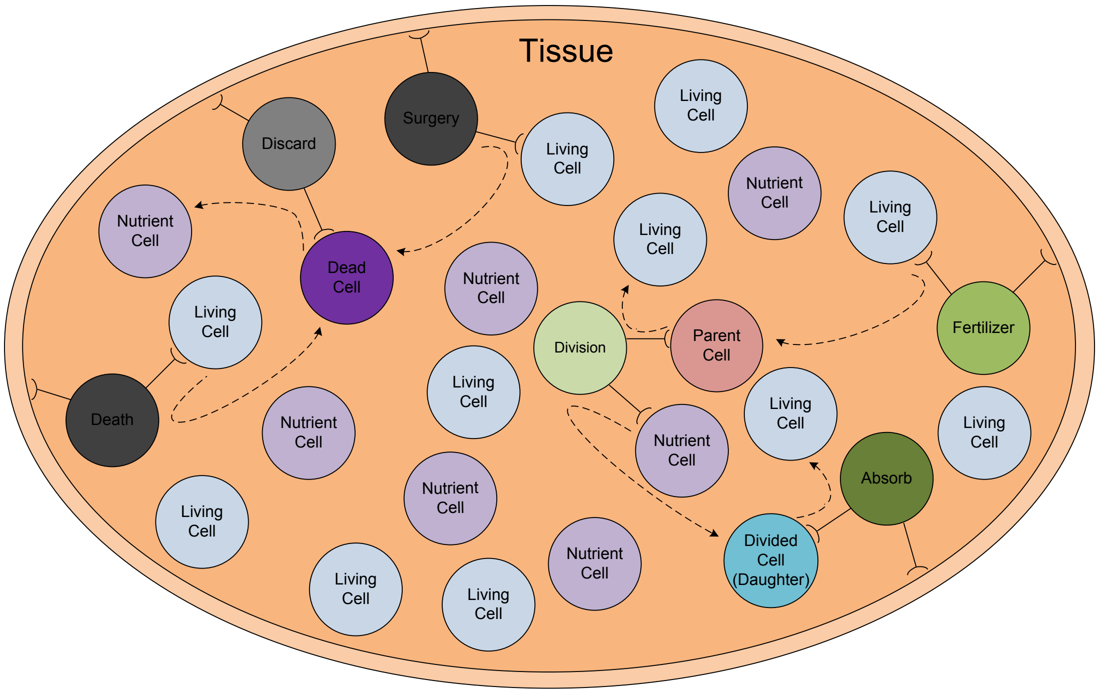


Figure 5.15. Therapy-Enabled SC cancer model. The therapy state of the tissue is locally stored in its data system to enable controlling the surgery and chemotherapy processes. Division is executed in three steps: (i) fertilize a living cell to become a parent, (ii) perform division and segregation of this parent cell to produce a divided cell (iii) the tissue absorbs the divided cell which becomes living and the tissue size is updated

SC Cancer Model Development Step 4: Integrating the Notion of Time

The last consideration before we have a complete SC cancer model reflecting the reference one is integrating the notion of simulated time. The reference model uses abstract time, counted in abstract time units called generations or, during this analysis, timesteps or timeslots. A generation has finished when the main loop in Algorithm 5.1 has visited all living cells and decided their individual fate. It is importance to notice that the model does not use any feedback from this time variable affecting its behaviour. The abstract time is mainly used for convenience as an index when logging simulation output data.

The way the list of living cells is traversed in the reference model is completely different from the approach used in HAoS. As the living cells are stored in a linked list in the software implementation, the list is traversed in the same relative order. This traversal order changes slightly with the local addition of new nodes (daughter cells are placed immediately after parent cells during division) and the removal of nodes (during cell death). In HAoS, the order of the selected cells to be evaluated against the available genetic processes is random, as all matching systems to the template defined by the schemata of the active context system have the same probability of being selected to interact. Evaluation of a cell here stands for the evaluation of the probability of this cell interacting through one of the given context systems and according to this interaction, it might get transformed during one of the genetic processes or remain unaltered. The inherent randomness of HAoS is desirable as the goal of the experiment is to model a stochastic biological system.

However, this means that there is not a convenient way to ensure that all cells are selected before any of them is re-selected. Thus, this functionality needs to be implemented on the SC model level. In order to accomplish this, a dual-phase approach was devised, called here a tic-toc approach. During each phase, the execution follows the flow shown in Figure 5.15 with a main difference. The products of each genetic process are marked (the current phase state of each cell is stored locally in its schemata) and can only interact in the next phase. In essence, all the context systems of the SC model of Figure 5.15 are duplicated, with each of their two copies being able to define an interaction only during one of the two distinct phases.

In this way, the two phases of the model create two “virtual” scopes which are implemented by using the phase state of each system during valid triplet generation.

Ensuring that all living cells are selected during each phase and then the phase finishes is accomplished by storing the current phase state in the tissue system along with a counter which is set equal to the number of total living cells of the tissue in the beginning of each timeslot and is decremented after each cell is evaluated. Essentially, this counter monitors the number of remaining living cells to be evaluated in the current timeslot, ensuring that the timeslot can finish and the phase can change only when it reaches zero.

SC Cancer Model 1: Time-Enabled Model

According to the discussion above the functionality of the time-enabled SC cancer model is summarized as follows: All living systems and the tissue are initialized in the tic phase. The tic-marked contexts can only define interactions when the tissue is on the tic phase. Thus, all tic-marked contexts are enabled and all toc-marked contexts are disabled. Division is executed in three steps as in the therapy-enabled model of Figure 5.15 (intermediate products are also tic-marked) but its final products, the new living cell (daughter cell in division) created by the absorb context and the living cell (parent cell in division) transformed by the division context are marked as toc, disabling any further interactions during this tic phase. In the case of cell death, the dying cell also gets toc-marked in order to be disabled for the current phase. However, since the final product of cell death is a nutrient cell, which can be used in both phases, it does not need to get phase-marked. It is noticed, that interacting cells are always phase-marked in the first step of any genetic process, even if the associated probability with this process is not satisfied

This means that any cell that is evaluated is changing phase, even if it remains unchanged, to avoid interacting twice during the same phase and ensure the remaining living cells counter is correctly updated. After the last tic-marked cell has been evaluated, all living cells should be toc-marked and the remaining living cells counter should be zero. At this point, the counter is updated to the number of total cells and the tissue changes its phase state to toc. All toc-marked contexts are now enabled while all tic-marked contexts get disabled as they can no longer define interactions as the tissue is now toc-marked. A mirror process to the one described above (toc instead of tic) is executed until the toc phase ends, and tic phase begins again. The time-enabled (tic-toc) model is illustrated using SC graphical notations in Figure 5.16.

.

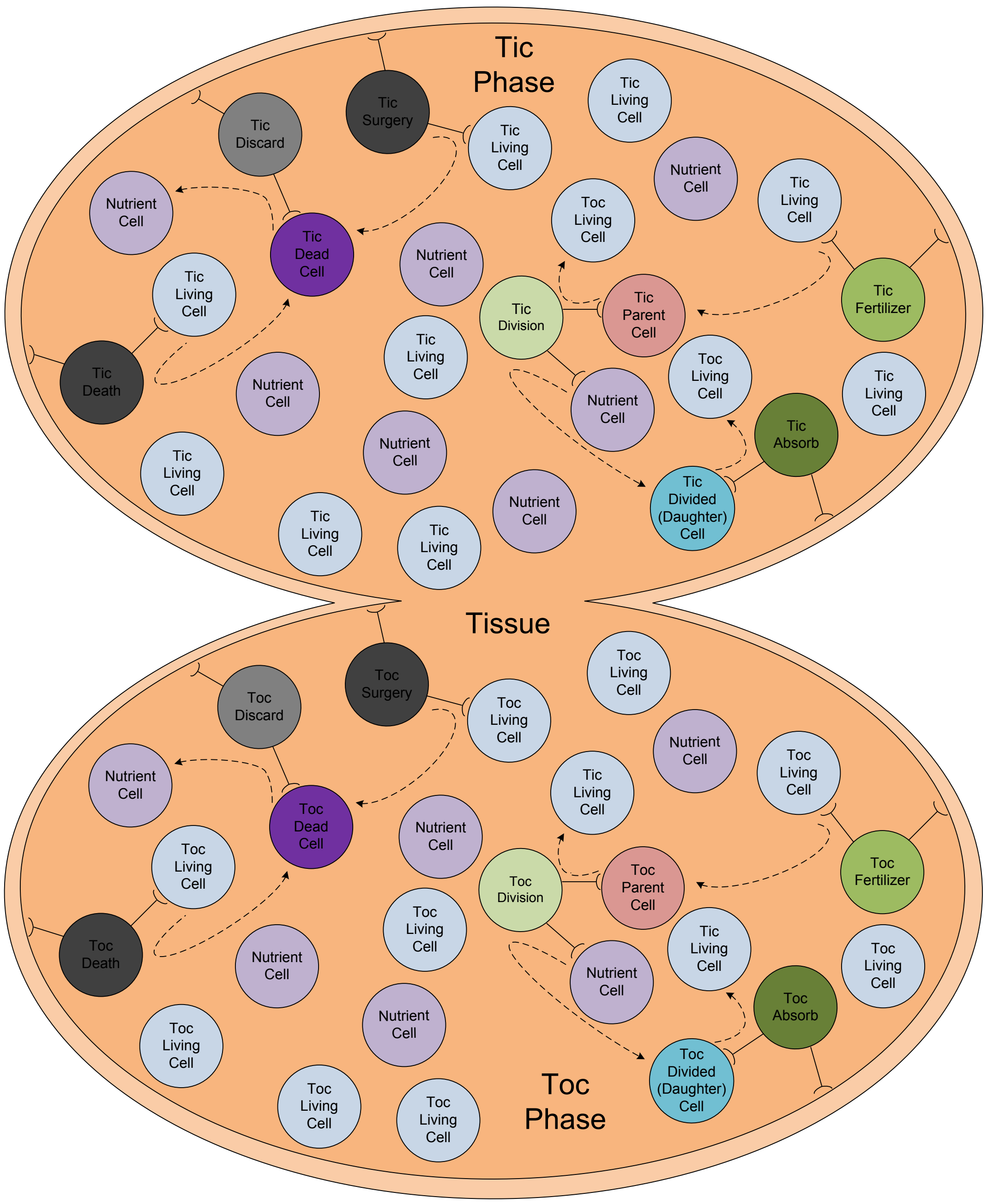


Figure 5.16. Time-Enabled (Tic-Toc) SC cancer model. Two mutually-exclusive phases are used to ensure all cells are evaluated exactly once before advancing to the next timeslot. Both phases are part of the tissue scope, yet the phase state of the tissue determines which one is enabled at each timeslot

When an interaction results in intermediate products of a genetic operation, these products remain in the same phase until they are consumed. A cell changes phase when the resulting system is the final result of a genetic operation or if the initiating context of cell death or division (death or fertilizer contexts respectively) leaves the evaluated cell unchanged (meaning that the respective death or division probability is not satisfied). All contexts involving the tissue except the division context may change the phase of the tissue after the last cell in a timeslot is evaluated. This implies that the fertilizer and death contexts can change the tissue phase when the evaluated (last) living cell remains unaltered while absorb and discard contexts may change it when they consume the last intermediate result (divided and dead cell respectively).

Table 5.5. Time-Enabled (Tic-Toc) cancer SC model interactions

	Interacting Systems		*Results*
Tic Phase	Tic Living Cell }- Tic Fertilizer -{ Tic Tissue	→	(Toc Living Cell | Tic Parent Cell | Tic Dead Cell) (Toc Tissue | Tic Tissue)
	Tic Parent Cell }- Tic Division -{ Nutrient Cell	→	(Toc Living Cell) (Tic Divided Cell)
	Tic Divided Cell }- Tic Absorb -{ Tic Tissue	→	(Toc Living Cell) (Tic Tissue | Toc Tissue)
	Tic Living Cell }- Tic Death -{ Tic Tissue	→	(Toc Living Cell | Tic Dead Cell) (Tic Tissue | Toc Tissue)
	Tic Dead Cell }- Tic Discard -{ Tic Tissue	→	(Nutrient Cell) (Tic Tissue | Toc Tissue)
	Tic Living Cell }- Tic Surgery -{ Tic Tissue	→	(Toc Living Cell | Tic Dead Cell) (Tic Tissue | Toc Tissue)
Toc Phase	Toc Living Cell }- Toc Fertilizer -{ Toc Tissue	→	(Tic Living Cell | Toc Parent Cell | Toc Dead Cell) (Tic Tissue | Toc Tissue)
	Toc Parent Cell }- Toc Division -{ Nutrient Cell	→	(Tic Living Cell) (Toc Divided Cell)
	Toc Divided Cell }- Toc Absorb -{ Toc Tissue	→	(Tic Living Cell) (Toc Tissue | Tic Tissue)
	Toc Living Cell }- Toc Death -{ Toc Tissue	→	(Tic Living Cell | Toc Dead Cell) (Toc Tissue | Tic Tissue)
	Toc Dead Cell }- Toc Discard -{ Toc Tissue	→	(Nutrient Cell) (Toc Tissue | Tic Tissue)
	Toc Living Cell }- Toc Surgery -{ Toc Tissue	→	(Tic Living Cell | Toc Dead Cell) (Tic Tissue | Toc Tissue)

In more detail, a fertilizer context may leave the type of the interacting living cell unaltered (changing only its phase) or change it to a parent cell with the same phase in case of the first step of normal division or kill it (change it to a dead cell) also with same phase, in case of division during chemotherapy. As mentioned above, it may also change the phase of the tissue at a timeslot transition. The division context always transforms the parent interacting cell to a living cell changing its phase and also changes the interacting nutrient cell to a divided cell (setting its phase to the current one). The absorb context, in the last step of division, changes the divided cell to a living cell with different phase while it may also change the phase of the tissue. Death and surgery contexts change the phase of the interacting living cell when they do not kill it, while they leave its phase unchanged when they do (changing its type to dead). They may also alter the tissue phase when changing timeslot. Finally, the discard context always transforms the dead interacting cell to a nutrient cell and may also change the tissue phase. The calculus notation of the time-enabled SC cancer model representing its behaviour in terms of SC interactions, as it is explained above is given in Table 5.5.

SC Cancer Model 2: Timeless Model

The time-enabled SC cancer model of Figure 5.16 satisfies all the requirements and features all the characteristics of the original model. However, in order to accurately model its abstract time, the complexity of the model was considerably increased. Taking into consideration that time, although convenient for logging output information, should not otherwise greatly affect the model behaviour, a timeless variation of the cancer model was also implemented to determine the level of this effect. The timeless model is equivalent in terms of SC graphical notation with the therapy-enabled model, illustrated in Figure 5.15, and overcomes the problem of sampling the internal state of the tissue (total number of cells and number of chromosomes) by continuously monitoring and storing all changes in the relevant internal variables immediately after each alteration is caused by any genetic process (instead of only sampling it at the end of the timeslot).

In the case of the timeless SC cancer model, the systemic interactions are similar to the ones of the time-enabled model excluding any phase-related features. The calculus notation of the timeless model is given in Table 5.6, making this similarity obvious when compared to Table 5.5.

SC Cancer Model 3: Approximate Time Model

In order to further explore the effect of abstract time and reduce the complexity of the time-enabled model of Figure 5.16, a hybrid model in terms of time was also implemented. This “approximate time” model waives the restriction of all cells being evaluated at each time-step but keeps the remaining living cells counter functionality to keep track of approximate time. Essentially, it ensures that a number of cells equal to the size of the tissue are evaluated before advancing to the next time timeslot. As living interacting cells are selected on a purely random manner with the same probability, it is expected that the behaviour of the model will remain concise - while not all cells may be evaluated during a single time-step, all of them will interact with the same probability and approximately the same frequency over the duration of the experiment. Thus, the main difference of this model compared to the timeless one is that it keeps the notion of the timeslot (implying that an adequate number of cells are evaluated before advancing time), but it uses a less strict mechanism to achieve this (since not every cell is evaluate exactly once in each timeslot). In terms of the SC graphical notation, this hybrid model, attempting to address the trade-off between complexity and functionality, is also represented by Figure 5.15. Additionally, since no change has been made to the approximate time model in the way systems interact with each other, when compared to the timeless model, the calculus notation is the same for both models.

Table 5.6. Timeless and Approximate Time cancer SC models interactions

Interacting Systems						*Results*
Living Cell	}-	Fertilizer	-{	Tissue	→	(Living Cell | Parent Cell | Dead Cell) (Tissue)
Parent Cell	}-	Division	-{	Nutrient Cell	→	(Living Cell) (Divided Cell)
Divided Cell	}-	Absorb	-{	Tissue	→	(Living Cell) (Tissue)
Living Cell	}-	Death	-{	Tissue	→	(Living Cell | Dead Cell) (Tissue)
Dead Cell	}-	Discard	-{	Tissue	→	(Nutrient Cell) (Tissue)
Living Cell	}-	Surgery	-{	Tissue	→	(Living Cell | Dead Cell) (Tissue)

SC Cancer Model 4: Optimized Approximate Time Model

While developing a SC model, ensuring that it behaves correctly (in this case, in a similar way to the reference model) is crucial. However, the SC model developer should take into consideration the performance of the model as well. In an attempt to demonstrate example optimizations that can be made on the SC model side, an optimized version of the approximate time model was also developed.

The most evident way to optimize a SC model is to ensure that valid triplet generation is performed in an optimal way. This means that HAoS should be able to identify triplets of interacting systems without (or, more realistically, without numerous) mismatches (see section 3.5). The hardware ensures that even in cases that only one pair of systems can match the templates defined by the schemata of the context, this pair will be efficiently identified, if such a pair exists. If a pair does not exist, this results in a schemata mismatch and another context is selected. Effectively, while the hardware attempts to find triplets, from an interaction (or processing) point of view, mismatches result in idle time as no interactions are performed. The SC model should take this fact into consideration and try to minimize mismatches, in order to increase efficiency and consequently overall performance.

Taking a closer look at Figure 5.15 and according to the discussion integrating therapies in the cancer model, we notice that the surgery context performs the exact same task with the death context in a different occasion. However, while the death context can define interactions along the course of the simulation, the surgery context is only functional during the surgery timeslot. This implies that, as the surgery context is randomly chosen with the same probability as every other context during the execution of the HAoS cancer program, it produces mismatches on the majority of the times it is selected. Thus, embedding the surgery task in the death context increases the efficiency of the SC program.

In addition, an extra optimization can be made with regards to the transition from a timeslot to the next one. After all the cells have been evaluated in any timeslot, there may be outstanding interactions to be made in order for intermediate products of the two main genetic operations (cell division and death) to be consumed (dead cells in the case of death or parent and divided cells in the case of division - see Figure 5.15). This ensures that each genetic operation finishes in the timeslot it was initiated and the total number of cells is reported correctly for each timeslot. Effectively, once the remaining living cells counter reaches zero, the fertilizer and death contexts do not define any further interactions until the next timeslot commences, waiting for the division, absorb and discard contexts to consume the cells produced in the intermediate cells. Thus, in order to reduce mismatches in this case, an extra flag, stored in the tissue system, is used to disable the selection of the fertilizer and death contexts when no more interactions initiating division or death may be performed in the current timeslot. This way, we achieve optimal timeslot transitions in terms of performance.

Another observation regarding to the optimal flow of the SC program can be made when we take a higher-level view on the execution of tasks in terms of interactions. A high-level task is broken down in more than one interaction in two cases. Either the task requires that more than two systems must interact under a single context to accomplish the expected functionality or a succession of steps should be performed, implying a number of contexts being selected sequentially, to implement a *chain* of events. Supporting a context type that could define interactions among multiple systems would be impractical form an implementation point of view as it would increase the size of such context systems (an extra schema would be added to such contexts, to define a template for matching systems, for every extra supported system). However, the two cases could be merged, as multiple interacting systems could be selected by subsequent contexts.

Nevertheless, as the selection of contexts happens in a random manner to reflect the stochastic behaviour of natural systems, breaking down a task in elementary steps will usually result in suboptimal performance. While the subsequent contexts will eventually be selected, the implied sequence of interactions defines a pipeline which stalls until the next context (the next step in the correct order along the desired chain of events) is randomly chosen. Thus, having a way to control the selection of contexts in the case of chained interactions, effectively implementing micro-interactions (analogous to micro-instructions in micro-programmable control units), would greatly increase efficiency.

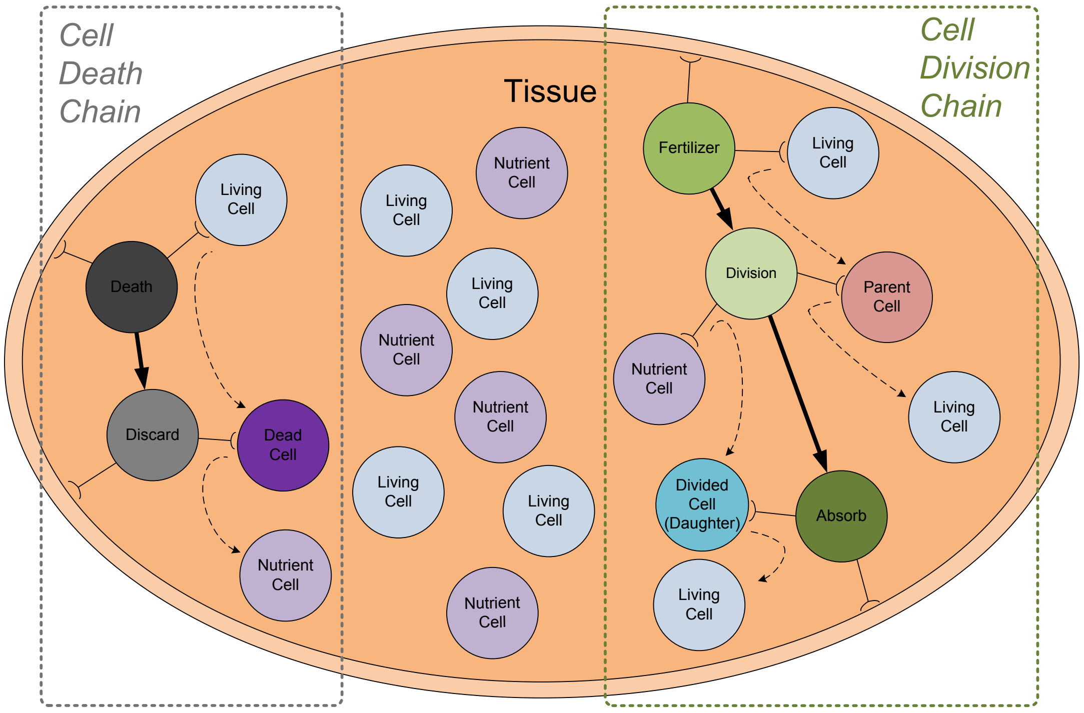


Figure 5.17. Optimized SC cancer model. The surgery functionality is now embedded in the death context while the contexts implementing the two main genetic operations, death and division, are now chained

Table 5.7. Optimized Approximate Time cancer SC model interactions

		Interacting Systems				*Results*
Living Cell	}-	Fertilizer	-{	Tissue	→	(Living Cell | Parent Cell | Dead Cell) (Tissue)
		↓				
Parent Cell	}-	Division	-{	Nutrient Cell	→	(Living Cell) (Divided Cell)
		↓				
Divided Cell	}-	Absorb	-{	Tissue	→	(Living Cell) (Tissue)
Living Cell	}-	Death	-{	Tissue	→	(Living Cell | Dead Cell) (Tissue)
		↓				
Dead Cell	}-	Discard	-{	Tissue	→	(Nutrient Cell) (Tissue)

Returning to the cancer model, the two main genetic operations involved, cell division and death, are broken down to three and two steps respectively. If *context chaining* was supported, cell division would be defined as a fertilizer-divider-absorb context chain while cell death would be implemented as a death-discard context chain. As described in the next paragraph, the context-chaining feature was added in HAoS, realized mainly in the user software domain, to showcase this functionality. The optimized approximate-time cancer SC model (integrating the surgery functionality in the death context, implementing optimal interaction transitions and context chaining) is shown in Figure 5.17. Its calculus notation is given in Table 5.7. In terms of SC interactions, its main differences with the previous models are the omission of the surgery context and the inclusion of context chains (implied by the vertical arrows).

Systemic Analysis Summary

To sum up, as in traditional software programming, there is more than one way to build a SC model representing a natural system. This section has demonstrated this, taking a complex biological model, a tissue developing cancer caused by genetic defects - and provided a thorough explanation of the thought process while building such a SC model. In this case, four candidate cancer SC models are presented: a time-enabled model, a timeless model, an approximate time model and an optimized approximate time model. The building elements in these resulting SC models have distinct biological meanings representing, in the form of SC systems, biological structures or processes, shown in Table 5.8. The next section focuses on the implementation of those models while the comparison experimental results are presented later in this chapter.

5.3.3 SC Cancer Model Implementation

Following the discussion of the previous paragraph, the four resulting cancer SC models attempt to explore the trade-off between functionality, performance and convenience. Before presenting the results evaluating these metrics in the next section, some implementation-specific topics need to be addressed. These include some considerations to be made before developing the final SC source code and performing the setup of the cancer experiments.

Developing the SC source code

While using SC graphical notations to present the four suggested cancer models is visually appealing and straightforward, using their SC calculus notations, to describe their systemic interactions, can greatly expedite the development of the SC source code.

Table 5.8. Biological representation of the systems of the SC cancer models

SC System	**Biological Representation (Analogous to)**
Living Cell	A cell of the biological tissue.
Death Context	Death is the context in which a living cell interacts with the tissue; it combines pressure for space, apoptosis and therapeutical interventions in one abstract form and may result in the living cell becoming a dead cell, representing the Programmed Cell Death (PCD) [239] biological process.
Discard Context	Discard is a context in which a dead cell interacts with the tissue; representing the biological mechanism (termed as efferocytosis [240]) which is responsible for the removal of apoptotic bodies (dead cells), by special cells, called phagocytes, that engulf and consume the dead ones. Phagocytes, using special receptors in their surface, identify dead cells by recognising special molecules which are placed to their cell surface in the last stages of cell death [240]. Matching the phagocytes receptors with these compatible special molecules is similar to the schemata matching mechanism of SC, with systems matching their schemata to the templates of context systems. This process releases energy to the environment, represented here in the form of a nutrient cell.
Dead Cell	The result of cell death; a cell showing organized degradation of cellular organelles which is finally broken into (several) apoptotic bodies [240]
Nutrient Cell	Nutrient cells represent the energy and nutrients in the tissue environment that may be released by cell death and may also be used to make new cells during division
Fertilizer Context	Fertilizer is the context in which a living cell interacts with the tissue; representing the preparatory step for division, known as the interphase[31] [241], making the parent able to initiate the mitosis process. Errors during this phase may kill the cell.
Division Context	Division is the context in which a living (parent) cell interacts with a nutrient cell; representing the mitotic phase of division [241], resulting in two daughter[32] cells with identical genetic information if no errors occur or different genetic information in the case of missegregation (resulting in aneuploid cells).

The SC calculus notations of the cancer models, shown in Tables 5.5 (time-enabled model), 5.6 (timeless and approximate time model) and 5.7 (optimized approximate time model), define their respective interactions and in addition include information about the phase of systems (in the case of the tic-toc model) and the type of interacting cells. However, in order to write the final SC source code, some additional information must be included in order to correctly implement the selection of appropriate cells in the therapy cases (surgery and chemotherapy). As explained in the previous section this is accomplished with the inclusion of flags regarding the therapy state of the tissue in its

[31] The first part of this phase, called G_1, is regulated by the MAPK cascade [241], presented in section 5.2.

[32] However, only one of the resulting cells is tagged as daughter in this SC model.

schemata. The format of the bit-fields stored in the data systems of the SC cancer models is illustrated in Figure 5.18.

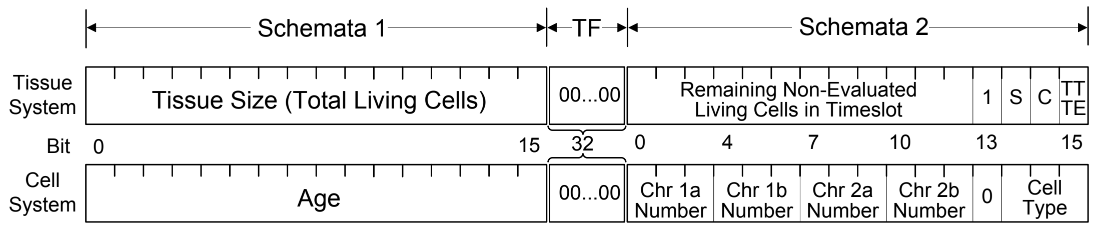


Figure 5.18. SC cancer model data systems and their contents. Bit 13 of schemata 2 defines the system as tissue or cell. The tissue size and cell age are stored in the respective schemata 1. Their 32-bit transformation function (TF) is zero. Schemata 2 holds the remaining non-evaluated cells counter, the surgery state (S), the chemotherapy state (C) and the phase state (tic-toc, TT) in case of the time-enabled model or the optimal timeslot ending flag (TE) in the case of the optimized approximate-time model. For cell systems, it stores the number of the different chromosome types of the cell and also the cell type (living, parent, dead, nutrient or divided along with its phase in the case of the time enabled model).

The proper system selection for the model is accomplished by appropriately using “don't care” bits in the respective therapy state bits in the templates defined by the schemata of each context. According to this, since the fertilizer context may initiate division during either a non-therapy timeslot or a chemotherapy timeslot, the tissue will be selected (and thus a fertilizing interaction can be defined) if its surgery bit (S) is set to 0 (since it is not in surgery) and for any value of its chemotherapy state bit (as it may or may not be in chemotherapy).

In a similar fashion, the absorb context may not consume divided cells only during surgery, so in this case: S is set to “don't care” (or “X”) and C is set to 0. The same principle is applied to the rest of the contexts. The death context will not kill living cells during surgery if the surgery functionality is not embedded to it. In this case, the surgery may kill living cells only during the surgery state while their total number exceeds the initial tissue size (see Algorithm 5.1). If the surgery and death functionalities have been merged, then the death context, similarly to the discard context (as cells may die at any time point), can interact with the tissue in any therapy state (both C and S are set to “X”). The complete SC code of the cancer models, according to the discussion above, is given for reference in Appendix G.

SC Models' Parameters Setup

In the reference cancer model experiments, as they are presented in [238], two parameters of the model (introduced in section 5.3.1, Algorithm 5.1) representing number of cells thresholds, TH_{END} (the end of the simulation threshold) and TH_{DET} (the cancer detection threshold), are set higher than the systems capacity of our HAoS prototype. For this reason these two parameters have been scaled down during our experiments. This, however, does not have an impact in the model behaviour as these parameters just define checkpoints in time for those two specific events. Thus, the main difference observed with the original simulations is with regards to the timeslot which cancer is detected (the tissue reaches a cancerous state as the total number of cells exceeds TH_{DET}) and the duration of the simulation (which runs until the tissue size exceeds TH_{END}). Since the first change implies just a shift of the cancer detection time and as the model does not change its behaviour towards the end of the simulation, these modifications are assumed to be acceptable.

While the parameters mentioned above have a minimal impact on the model behaviour, the parameters setting the intrinsic rates of apoptosis r_{ap}, division r_{div} and chromosome missegregation r_{msg} (see Equations 5.1) have a major effect on the model, as they affect the respective genetic operations probabilities and in extent the cell population of the tissue. The nested-if structure in Algorithm 5.1 implies that each cell may result in the outcome of any of the possible cases but importantly also implies a selection priority. Obviously this does not apply in systemic interactions, which happen in a stochastic way. Due to the complex nature of the algorithm and mainly because of the conditional feedback mechanisms (the probabilities are affected by the number of chromosomes which are affected by the probabilities), deriving the intrinsic rate constants r with an analytical way was avoided and a brute-force approach was followed instead.

Thus, in order to derive the adjusted values of the intrinsic rate constants r to be used in the four SC cancer models, the original algorithm was altered to reflect the different nature of interactions when these are implemented in a systemic way (the resulting "systemic style" algorithm is given in

Algorithm 5.2). The main changes to the original algorithm involved the removal of any priority in the selection of the genetic operation that may be performed on the cell and also taking into consideration the number of context systems involved. Since the rate constant for apoptosis should always be equal to the rate constant for division, to preserve the homeostatic behaviour of the tissue, the goal was to find a pair of constants $r_{ap/div}$ - r_{msg} (apoptosis/division and missegregation rate constants) to result in a similar behaviour to the one given by the reference model.

Algorithm 5.2. The reference cancer model algorithm written in a "systemic way" in order to derive adjusted values for the parameters setting the intrinsic rates of the genetic operations due to the difference on probability mechanics between all four SC models and the reference one. The priority selection of the original model is broken. Interactions that may initiate cell death / division have the same probability.

```
Initialize the model with random seed
Set the carrying capacity of the tissue to a fixed number
for all experiments do
 Create tissue with an initial population of cells, each with two diploid chromosomes.
 Each chromosome in each cell is given one or two genes based on chromosome distribution
 repeat
  for all cells in tissue do
   if during surgery then
    Kill current cell if tissue size (total cells) exceeds its initial size
   else
    if context that may initiate cell death is selected then
     if total cells > tissue capacity and apoptosis probability $p_{ap}$ satisfied then
      kill current cell
     else
      current cell remains unchanged
     end if
    else if context that may initiate cell division is selected then
     if division probability $p_{div}$ satisfied then
      if during chemotherapy then
       kill current cell
      else
       Add mitotic cell (birth of new daughter cell, identical to current parent cell)
       if missegregation probability $p_{msg}$ satisfied then
        randomly select r : one of the four chromosomes in the cell
        perform asymmetrical division instead(increment daughter r,decrement parent r)
        if no chromosomes left in parent cell (mitotic checkpoint) then
         kill parent cell
        end if
       end if
      end if
     else
      current cell remains unchanged
     end if
     go to next cell
    else
     re-evaluate current cell (until a genetic operation is attempted)
    end if
  end for
  Update number of cells
  if number of cells > cancer detection threshold ($TH_{DET}$) and no previous therapy then
   initiate therapy (surgery and/or chemotherapy)
  end if
  Increment timeslot t (generation counter - abstract time)
 until reached maximum number of generations or cells (End Threshold - $TH_{END}$)
 print output results
end for
```

A series of simulations using the altered algorithm were executed for a range of different $r_{ap/div}$ - r_{msg} pairs, comparing the similarity to the reference algorithm based on metrics relevant with the constants: mean number of generations run until simulation finish, mean number of missegregations per generation and mean cancer detection (or diagnosis) generation. The qualifying pair was the one with the minimum value for the root mean square of the differences of the corresponding values to these metrics between the altered and the original algorithm.

The "systemic" variation of the reference model thus enabled the discovery of the parameters to be used by all models. For the final setup of all the experiments, the end of the simulation threshold (TH_{END}) was set at 200 generations, the initial size of the tissue was set at 100 cells, the cancer diagnosis threshold was set at 200 cells and the tissue carrying capacity was set at 150 cells. For the original cancer model the genetic operation constants for apoptosis, division and chromosome missegregation were set at [238]:

$$r_{ap} = r_{div} = 0.045,\ r_{msg} = 0.02$$

while for the SC cancer models, using the methodology mentioned above, they were set respectively at:

$$r_{ap} = r_{div} = 0.09,\ r_{msg} = 0.02$$

Experiments Setup

In order to get a fair comparison in terms of both functionality and performance, the same methodology presented in [238] was followed to obtain the results of each batch of simulations.

Due to the complexity of the experiment, it was decided that this test case is also ideal for the evaluation of the functionality of the developed HAoS functional model (functional model of the hardware circuitry, not to be mistaken with the simulated biological models) which is essentially a high-level HAoS simulator. All SC cancer models were tested both live on hardware with HAoS and using its simulator.

As discussed in the cancer model presentation section, three gene distributions (see Figure 5.12) with different gene chromosomal linkage and four therapy scenarios (with/without surgery and/or chemotherapy) are examined. In total, ten cancer models are involved in the experiments: the four SC cancer models described in the previous section (time-enabled, timeless, approximate-time and its optimized variation) running both on hardware and on the HAoS simulator[33], the reference model from [238] and, for completeness, its altered variation (coded in a systemic-aware way). Each of these ten models was executed in batches of 20 experiments[34] for each possible gene distribution and therapy scenario combination and the mean of those simulations is used to represent the final results.

Due to the high number of possible comparison metrics, a set of indicative selections were made to compare the cancer models for the multitude of simulated configurations. The behavioural features compared along the duration of the simulations were the tissue size (in cells) and the number of regulatory genes for each genetic operation (division, apoptosis and segregation) in the case of the non-therapy scenario. For the therapy scenarios, the models were compared based on the average apoptosis-to-division gene ratio, which is characteristic of the model behaviour according to [238].

Following the methodology in [238], for each batch of experiments, the output from the models (tissue size and number of genes) was stored in separate text files for each simulation. These log files were then used for post-processing (using Mathematica), transforming the results in a form more suitable for statistical analysis (analyzed then with Excel).

5.3.4 Results

The comparison results are given below for all the cancer models: the original one from [238], the original recoded in a systemic style (OriginalSystemicStyle) and the time-enabled (TicToc), timeless (noTime) and approximate time (ApproxTime) with and without optimizations (opt and nopt) for both the HAoS simulator (simHAoS) and the hardware platform itself (HAoS).

In the case of the models simulating a therapy scenario, the results are re-aligned taking the diagnosis time as a common reference time point to make the comparison more comprehensive. Also in the case of the timeless SC cancer models, since the notion of time is missing, a direct comparison of the genetic evolution of the tissue with the other

[33] While a model runs on the HAoS hardware platform and its software simulator unaltered, it is expected to give slightly different results in terms of behaviour and quite different results in terms of performance. Thus, it is accounted as two separate models in the context of this analysis.

[34] Each experiment here is a simulation of one of the resulting cancer models for a specific pair of gene configuration - therapy scenario.

models is not possible. For this reason, the number of the main genetic operations (in the form of tissue interactions) is used to monitor their growth instead of time. In order to enable their visual comparison, the results from the timeless models are plotted along with the ones acquired by the other models - appropriately scaling the respective axes.

Point-to-Point Model Behaviour Comparison Results

The point-to-point comparison results, giving simulation results for all ten cancer models in the same graph for each experiment configuration, are presented in Figures 5.19 - 5.23. Figures 5.19 - 5.21 give the output of the models when no therapy is used for all chromosome distributions comparing the resulting average number of cells and all three types of genes. Figures 5.22 - 5.23 compare the apoptosis-to-division gene ratio only for chromosome distributions B and C (as distribution A shows a homeostatic tissue behaviour which does not require treatment) for therapy scenarios B, C and D involving surgery, chemotherapy and both therapies, respectively.

As seen in Figures 5.19 - 5.23, the behaviour of the tissue is correctly captured by the SC cancer models, since the results are quite similar in most cases. Especially in the experiments without therapies involved (Figures 5.19 - 5.21), the similarity regarding the tissue size and numbers of regulatory genes is evident, as all models converge on the same results. A slight difference is observed only in the case of the number of cells for gene distribution A. Since all simulations carry on until the maximum number of generations, the tissue shows the expected homeostatic behaviour but the inherent randomness in SC causes a wider oscillation in the evolution of the number of total cells, resulting in the tissue converging in a slightly higher number of cells.

Same observations can be made for the therapy-enabled results (see 5.22 - 5.23) comparing the apoptosis-to-division gene ratio since there is a high degree of correlation between the SC cancer series and the reference one.

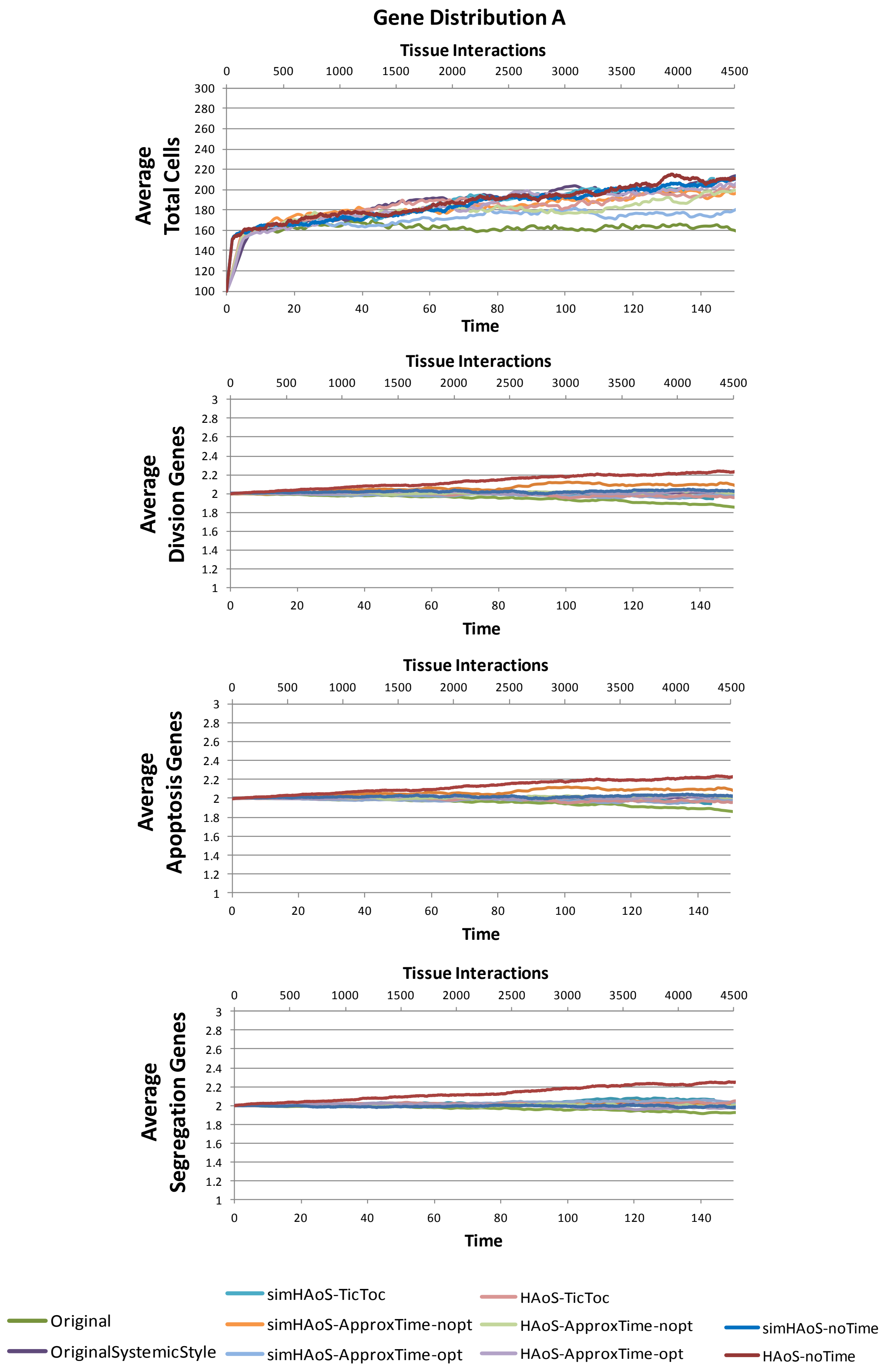


Figure 5.19. Non-therapy cancer models comparison for gene distribution A. The results shown give the average from 20 runs taken by each of the ten models for the tissue size in cells (first row), division genes (second row), apoptosis genes (third row) and segregation genes (fourth row) for gene distribution A.

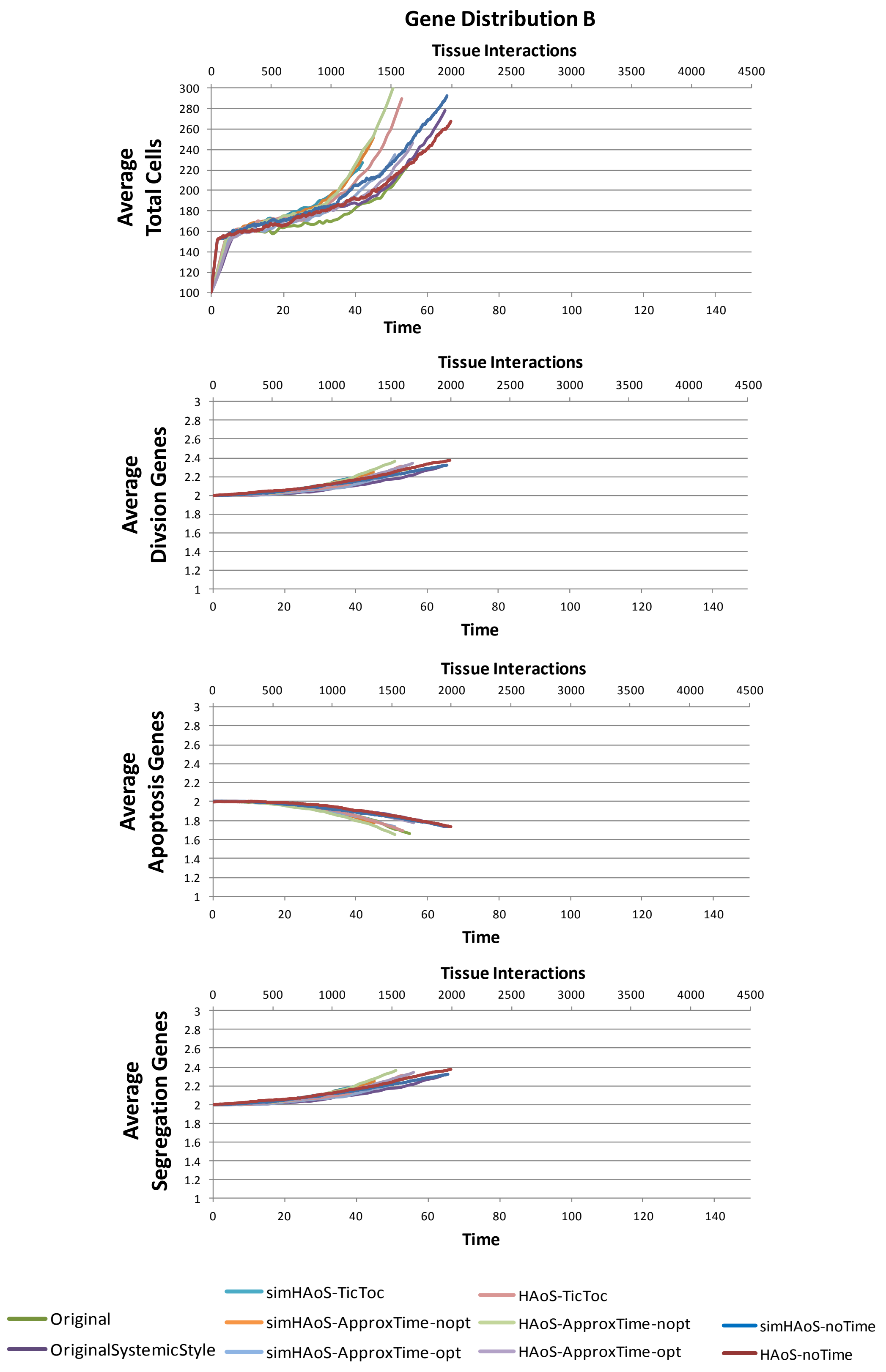


Figure 5.20. Non-therapy cancer models comparison for gene distribution B. The results shown give the average from 20 runs taken by each of the ten models for the tissue size in cells (first row), division genes (second row), apoptosis genes (third row) and segregation genes (fourth row) for gene distribution B.

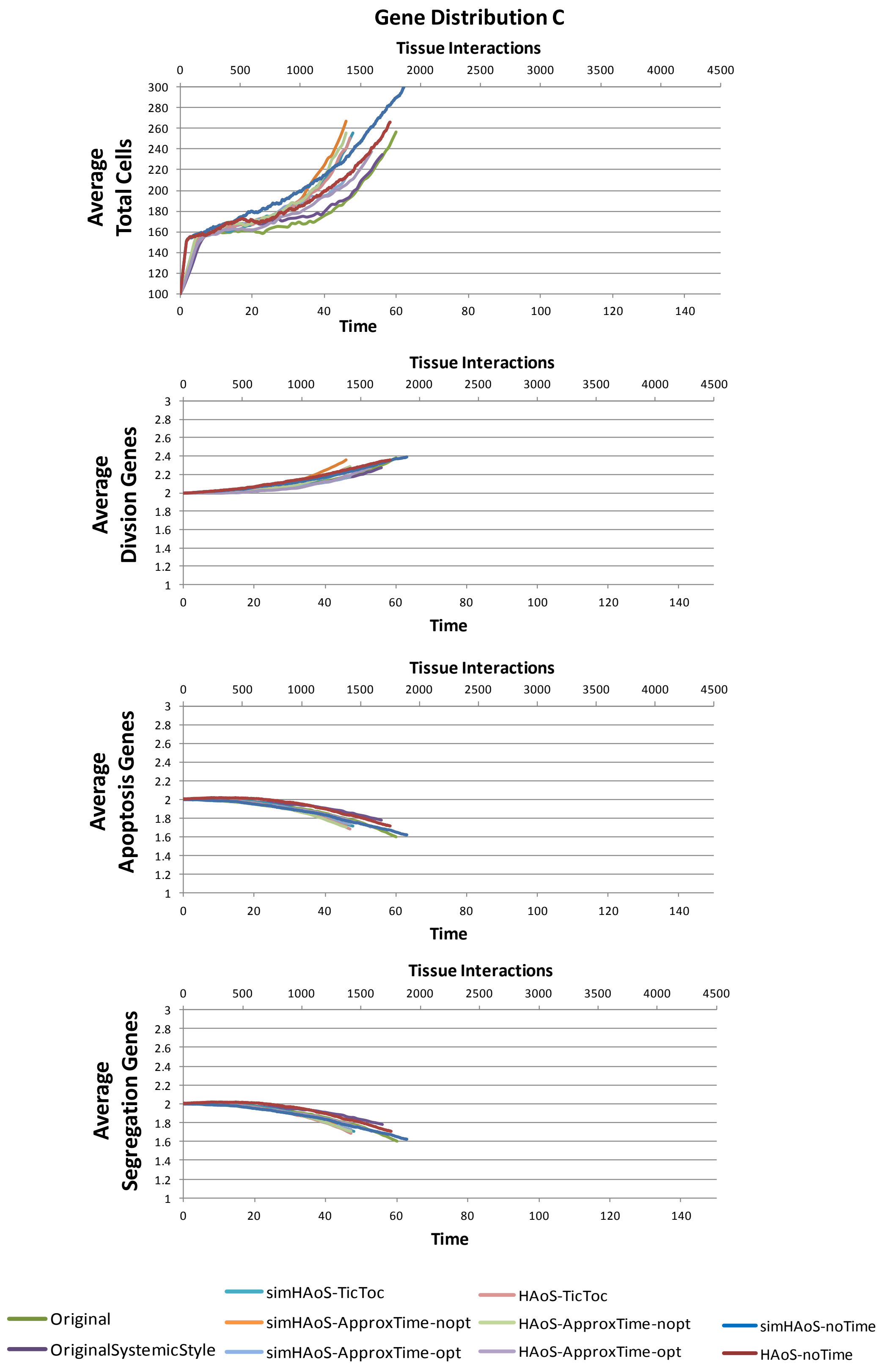


Figure 5.21. Non-therapy cancer models comparison for gene distribution C. The results shown give the average from 20 runs taken by each of the ten models for the tissue size in cells (first row), division genes (second row), apoptosis genes (third row) and segregation genes (fourth row) for gene distribution C.

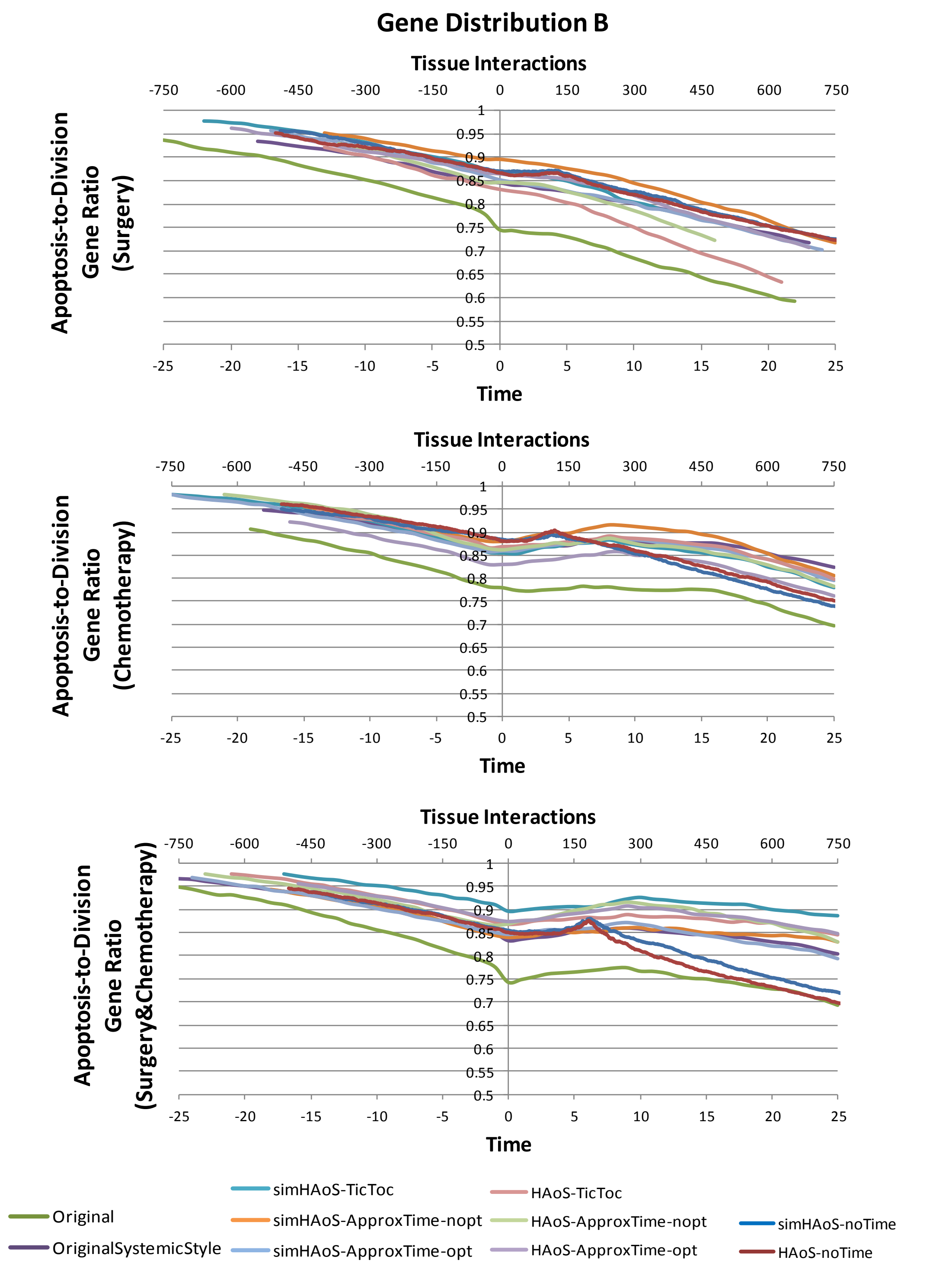


Figure 5.22. Therapy-enabled cancer models comparison near cancer diagnosis for gene distribution B. The results shown give the average from 20 runs taken by each of the ten models for the ratio of the number of apoptosis regulatory genes to the number of division regulatory genes for therapy scenario B (only surgery - first row), C (only chemotherapy - second row) and D (both therapies - third row) for gene distribution B. The results are plotted from 25 timeslots before until 25 timeslot after cancer detection for models supporting time and from 750 tissue interactions before until 750 interactions after detection for the timeless models.

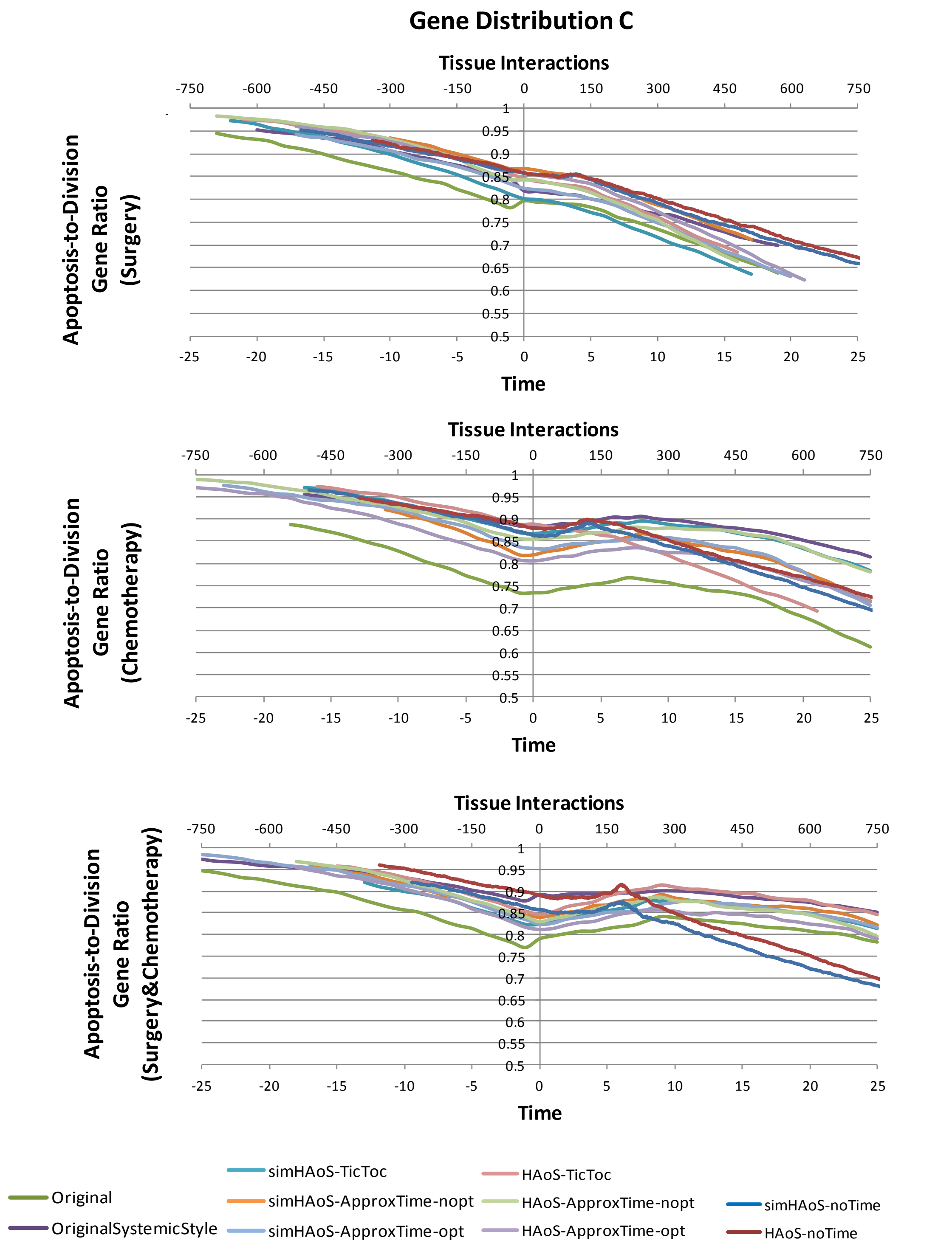


Figure 5.23. Therapy-enabled cancer models comparison near cancer diagnosis for gene distribution C. The results shown give the average from 20 runs taken by each of the ten models for the ratio of the number of apoptosis regulatory genes to the number of division regulatory genes for therapy scenario B (only surgery - first row), C (only chemotherapy - second row) and D (both therapies - third row) for gene distribution C. The results are plotted from 25 timeslots before until 25 timeslot after cancer detection for models supporting time and from 750 tissue interactions before until 750 interactions after detection for the timeless models.

Performance Comparison Results

While all suggested cancer models have been able to give results similar to the expected ones, their performance is the differentiating factor that will enable us to select the most optimal implementation. The average absolute and normalized (in terms of each generation) execution times for all cancer models, gene distributions and applied therapies are given in Table 5.9.

Table 5.9. Absolute and normalized average execution times for all simulated cancer scenarios

	Absolute Execution Time (ms)								
Gene Distribution	Distr. A	Distribution B				Distribution C			
Therapies	None	None	Surgery	Chemo	Both	None	Surgery	Chemo	Both
Original	1192.1	336.8	460.2	478.2	362.0	328.4	457.4	463.4	345.6
OriginalSystemicStyle	1648.7	353.2	393.0	405.0	398.0	366.1	363.4	359.6	378.7
simHAoS-TicToc	81862.0	19536.0	40969.0	20000.2	18013.0	16646.0	18194.0	39619.0	41590.0
simHAoS-ApproxTime-nopt	25880.0	4387.0	4965.0	5243.0	5622.0	4106.0	4890.0	4765.0	5195.0
simHAoS-ApproxTime-opt	7362.0	2182.0	2234.0	2338.0	2407.0	3239.0	2006.0	2262.0	2599.0
HAoS-TicToc	422.6	146.6	146.0	167.3	155.3	136.7	139.8	139.9	156.0
HAoS-ApproxTime-nopt	384.5	126.7	136.1	148.3	149.7	124.4	125.6	145.4	136.9
HAoS-ApproxTime-opt	328.5	112.3	111.6	139.9	135.3	105.3	104.9	122.0	137.2
simHAoS-noTime	29218.0	19084.0	18438.0	20625.0	19362.0	11272.0	11878.0	13281.0	15077.0
HAoS-noTime	310.7	172.0	177.2	187.0	378.0	160.2	167.8	182.5	170.0
	Normalized Execution Time (Per Generation)								
Gene Distribution	Distr. A	Distribution B				Distribution C			
Therapies	None	None	Surgery	Chemo	Both	None	Surgery	Chemo	Both
Original	3.97	4.33	5.14	4.49	3.28	4.30	5.19	4.73	3.15
OriginalSystemicStyle	5.50	4.33	4.25	3.75	3.48	4.58	4.17	3.60	3.50
simHAoS-TicToc	284.39	302.65	523.23	225.74	190.31	261.52	259.54	481.98	439.87
simHAoS-ApproxTime-nopt	86.30	70.93	70.58	61.21	60.22	69.36	68.01	60.58	59.88
simHAoS-ApproxTime-opt	24.46	29.37	27.50	24.07	24.45	44.22	26.55	25.70	26.07
HAoS-TicToc	2.10	2.23	2.08	1.88	1.77	2.18	2.06	2.08	1.74
HAoS-ApproxTime-nopt	1.91	2.11	1.95	1.76	1.65	2.03	1.92	1.75	1.61
HAoS-ApproxTime-opt	1.63	1.49	1.29	1.46	1.38	1.44	1.43	1.35	1.47
	Normalized Execution Time (Per Tissue Interaction)								
simHAoS-noTime	5.84	7.24	6.83	7.36	6.84	4.80	4.42	5.12	5.60
HAoS-noTime	0.06	0.06	0.07	0.07	0.13	0.06	0.07	0.07	0.07

5.3.5 Analysis

Model Behaviour

As observed in all charts, the apoptosis genes gradually become less than the division genes (the ratio is always below 1). The reason is that the cancer model implies a positive feedback on division genes growth. This means that as the number of genes grows, the probability of division grows as well (as this probability is proportional to the number of genes) resulting in cells with at least the same number of division genes (since genetic material gets written from the parent to the daughter cell). However, in the case of death, when the number of apoptosis genes grows, the probability of the cell dying grows as well, meaning its genes are lost and not carried in the next generation, as in the case of division. As seen in the therapy related results, the SC models tend to overestimate the gene ratio compared to the original one - implying that the positive division gene feedback in the SC models is weaker, mainly due to the lack of priority in selecting the genetic operation to each cell.

Another, less noticeable difference, is that the reference series tend to have a greater change in the gene ratio at the time of therapy from the SC model ones, especially for therapies involving surgery. This is an artefact of the structural design of the original cancer model as daughter cells (new nodes in the linked list) are positioned next to parent cells creating locally elevated concentrations of division genes, due to the positive feedback mechanism. While cells are evaluated as the linked list is traversed, surgery removes a range of cells adjacent to each other in the reference model, making it more probable that all the cells of such clusters of higher division genes will remain or be removed from the tissue after surgery. In the SC models however this does not happen as the selection of cells created during division and killed during surgery happens in a random manner.

Finally, a notable difference between the models supporting the notion of time and the timeless ones can be found on the way the various metrics are monitored and illustrated in the figures above. As mentioned earlier, the values for each metric are sampled in the end of each timeslot for the models supporting timeslots while they are continuously sampled in the timeless models. This is most evident in the surgery timeslot where a big number of cells are killed, changing considerably the genetic state of the tissue. While for the time-supporting models the surgery happens in one timeslot and is plotted as a sudden change of the tissue, for the timeless models the surgery operation is unravelled as a big number of subsequent cell deaths that are shown in the resulting figures.

Since a stochastic biological model is simulated, some variation in the results is naturally expected. To compare the level of behavioural similarity of the SC cancer models to the reference one in a more clear way, the differences of the averages of all time-supporting models to the values of the original one are plotted in Figures 5.24 - 5.28 while their respective mean error, standard deviation and correlation are given in Table 5.10. The timeless models are excluded from these comparisons as a point-to-point comparison between timeslot and tissue interactions would not be beneficial. To further support the fact that some level of variation is natural and acceptable, a second batch of experiments using the reference model (Original2) was conducted and their difference to the reference simulations is also included in the analysis below.

As seen in Figures 5.24 - 5.28, a clear pattern cannot be identified in the behaviour of each model when compared to the original one due to the stochastic nature of this cancer model and the variability of the intermediate states due to its complexity. In general, the second batch of experiments using the reference model gave, as expected, results that are more similar to the original ones. However, notably this was not always the case (when taking into consideration the results from all the different scenarios). This is also confirmed by the statistical comparison results of Table 5.10.

This test case has stressed the simulating abilities of both the HAoS functional simulator and the hardware platform itself. As shown in Figures 5.19 - 5.28 and Table 5.10 the developed high-level HAoS simulator succeeds on modelling the behaviour of the platform, capturing within an acceptable statistical error the results given simulating a complex biological model.

Following the discussion above, we can conclude that HAoS (and its accompanying simulator) can adequately model a fairly complex biological system. As in traditional programming, more than one ways can be used to describe such a system. While capturing the functionality of such a model is essential, its performance in terms of execution speed is also usually critical.

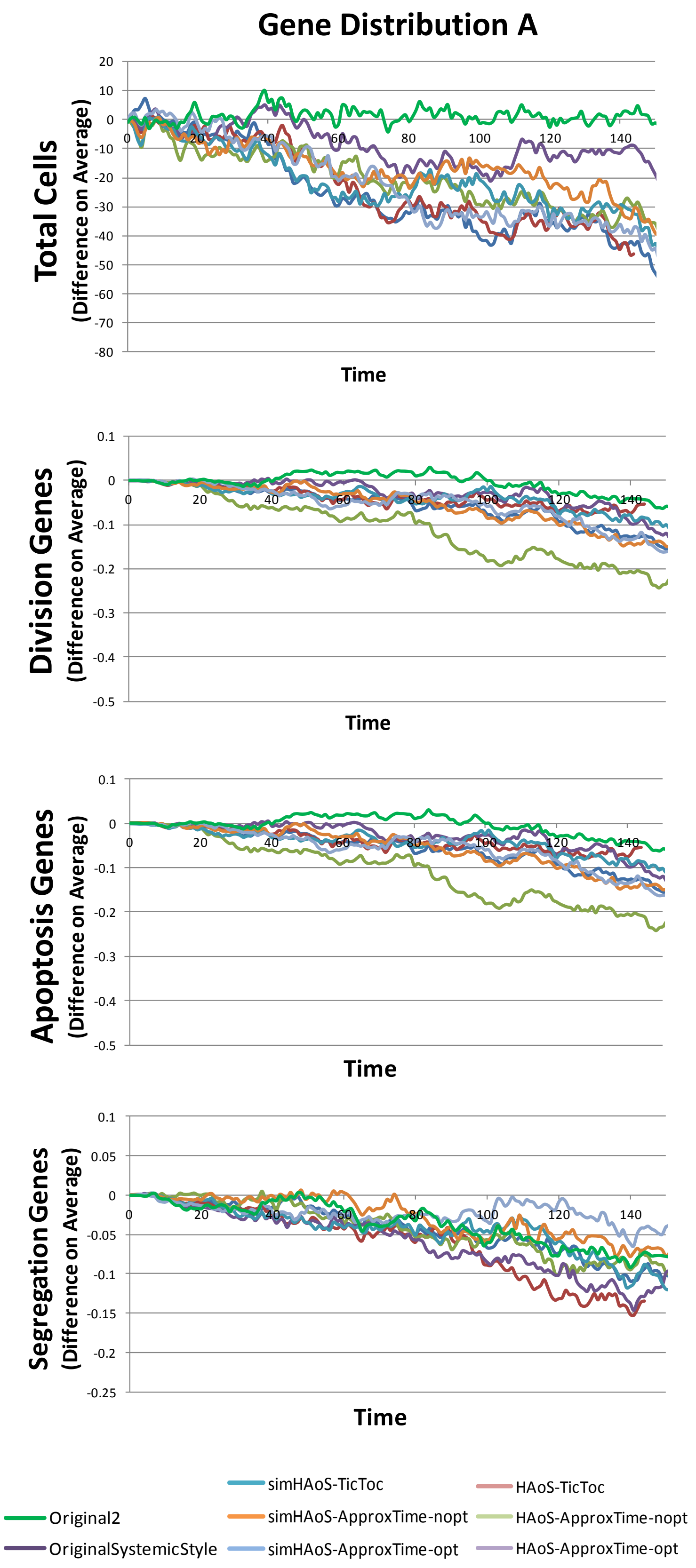


Figure 5.24. Non-therapy cancer time-supporting models results differences for gene distribution A against the reference model on tissue size and regulatory genes

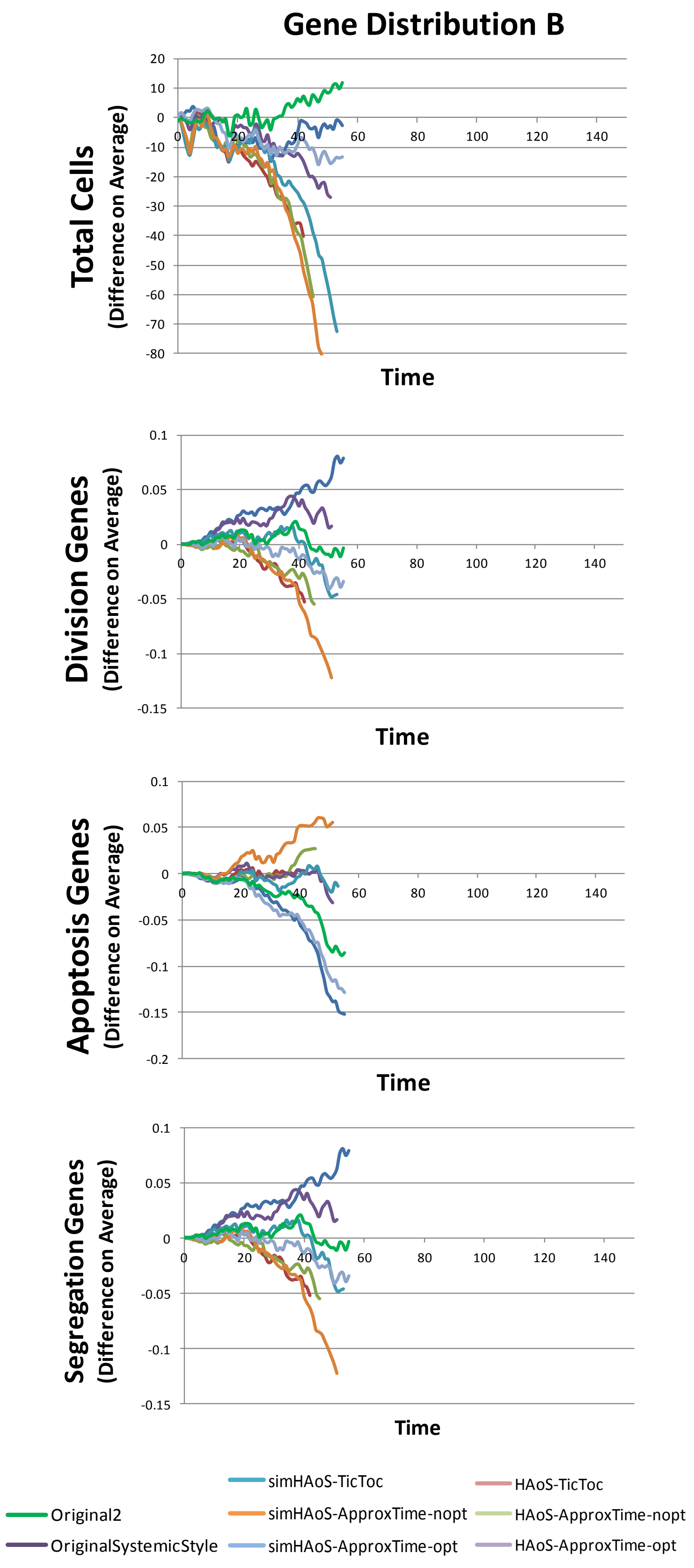


Figure 5.25. Non-therapy cancer time-supporting models results differences for gene distribution B against the reference model on tissue size and regulatory genes

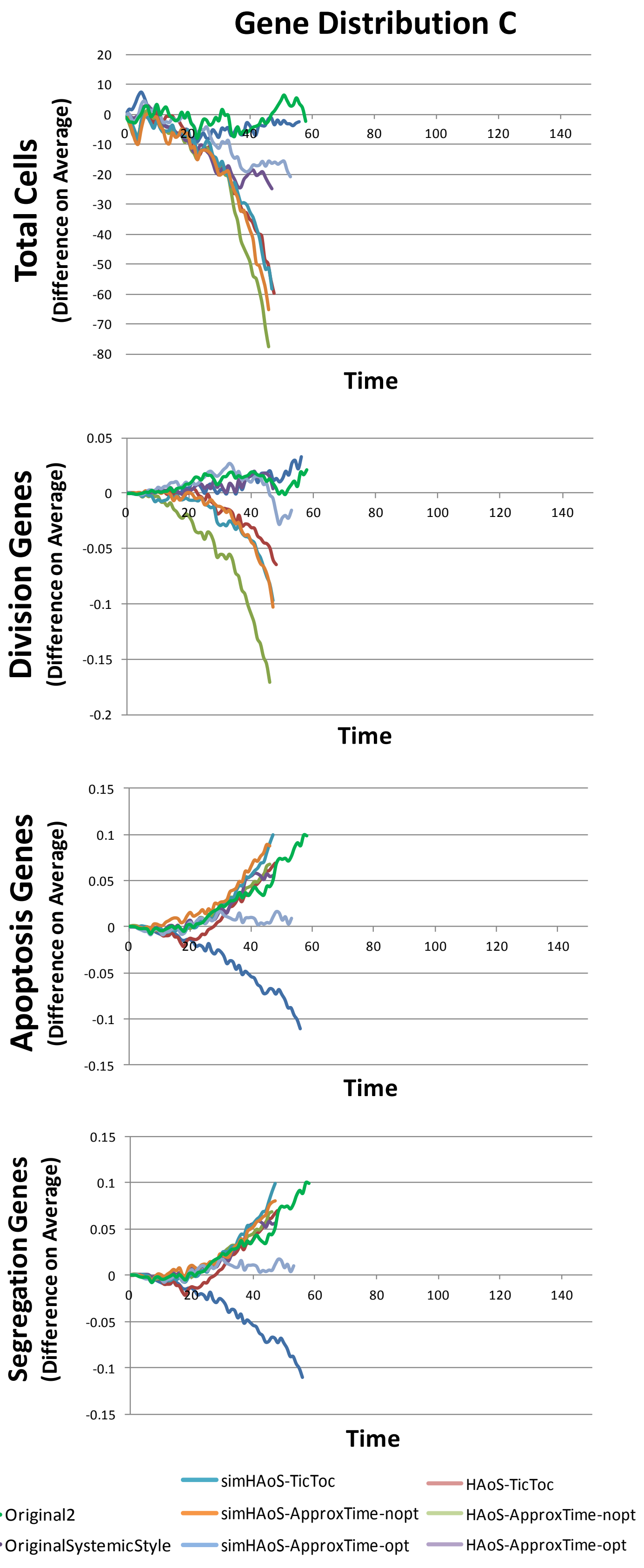


Figure 5.26. Non-therapy cancer time-supporting models results differences for gene distribution C against the reference model on tissue size and regulatory genes

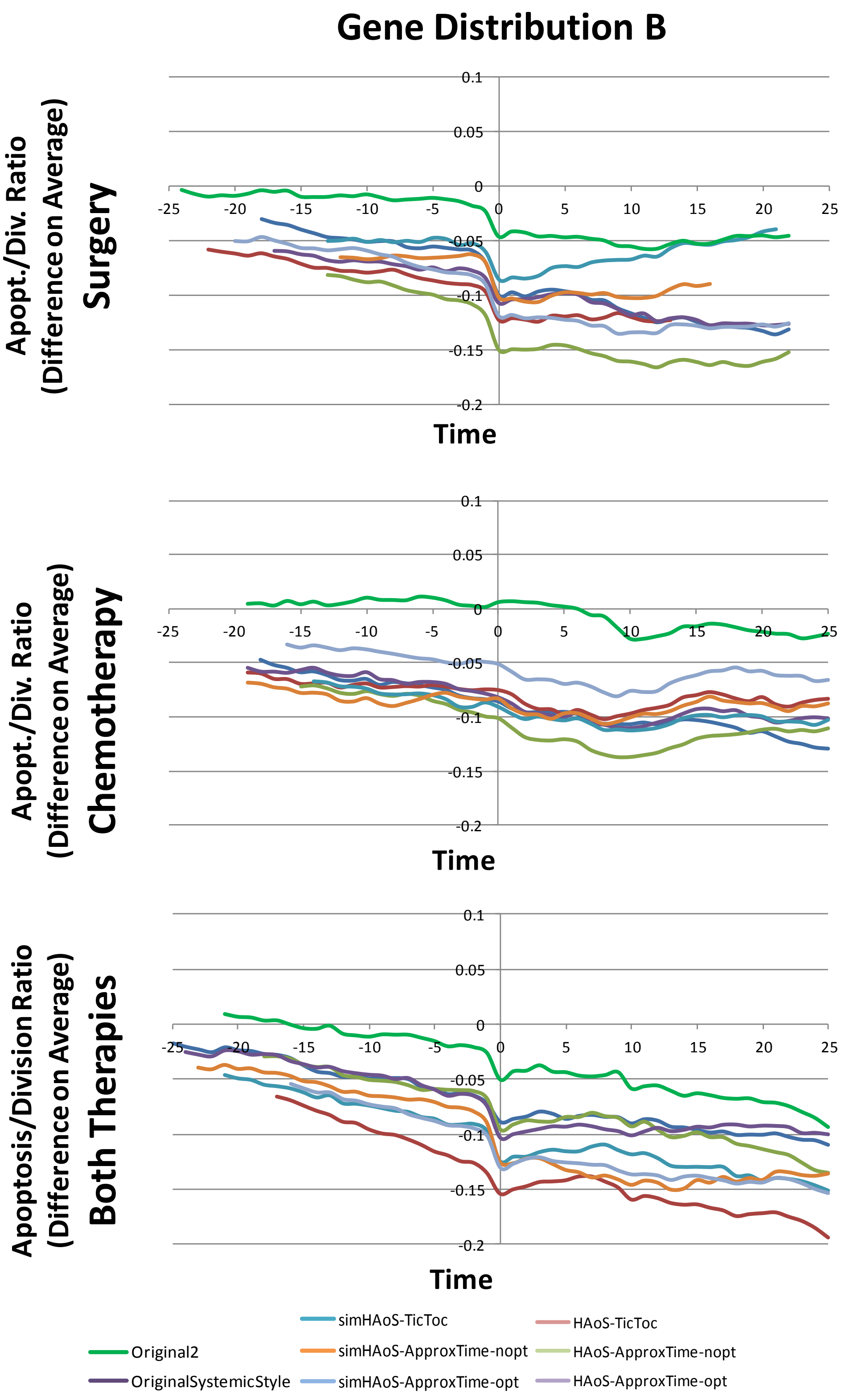


Figure 5.27. Differences on apoptosis-to-division ratio between the therapy-enabled time-supporting cancer models for gene distribution B against the reference one around diagnosis timeslot

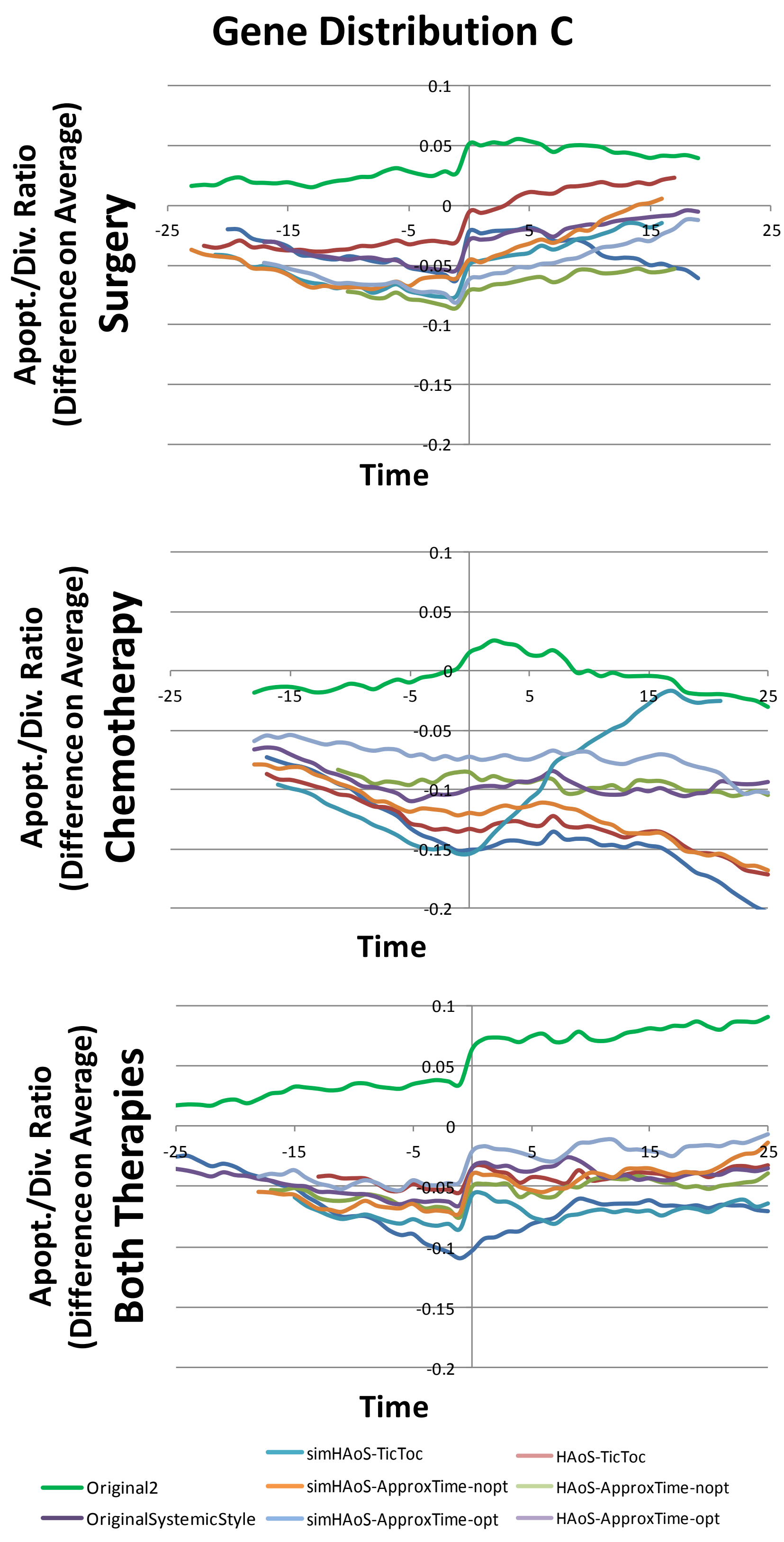


Figure 5.28. Differences on apoptosis-to-division ratio between the therapy-enabled time-supporting cancer models for gene distribution C against the reference one around diagnosis timeslot

Table 5.10. Statistical comparison of the time-enabled cancer models to the reference one in terms of mean error (ME), standard deviation (STD) and correlation (COR)

	Total Cells								
	Distr. A			Distr. B			Distr. C		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-41.097	19.962	0.407	-5.413	4.913	0.981	-3.426	3.663	0.991
simHAoS-TicToc	-22.118	14.377	0.639	-15.150	11.645	0.948	-16.714	16.597	0.931
simHAoS-ApproxTime-nopt	-32.314	15.975	0.443	-17.253	15.534	0.939	-20.041	21.199	0.892
simHAoS-ApproxTime-opt	-15.489	9.433	0.548	-8.120	7.975	0.985	-11.090	9.118	0.964
HAoS-TicToc	-27.836	15.299	0.560	-19.570	18.229	0.964	-16.166	15.406	0.937
HAoS-ApproxTime-nopt	-21.457	11.770	0.623	-25.431	26.779	0.935	-17.555	17.025	0.916
HAoS-ApproxTime-opt	-27.597	16.510	0.537	-7.084	5.607	0.992	-7.827	7.664	0.982
Original2	0.143	2.738	0.900	2.220	4.362	0.992	-1.036	3.311	0.994

	Average Division Genes								
	Distr. A			Distr. B			Distr. C		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-0.175	0.135	-0.743	0.032	0.022	0.997	0.007	0.008	0.999
simHAoS-TicToc	-0.038	0.022	0.830	-0.011	0.017	0.997	-0.014	0.018	0.998
simHAoS-ApproxTime-nopt	-0.190	0.105	0.341	-0.013	0.014	0.998	-0.047	0.047	0.998
simHAoS-ApproxTime-opt	-0.132	0.117	0.410	0.020	0.012	0.991	0.006	0.006	0.998
HAoS-TicToc	-0.066	0.053	0.732	0.000	0.016	0.992	-0.021	0.024	0.997
HAoS-ApproxTime-nopt	-0.094	0.078	-0.791	-0.027	0.037	0.995	-0.018	0.026	0.995
HAoS-ApproxTime-opt	-0.081	0.061	0.433	-0.009	0.013	0.999	0.006	0.012	0.990
Original2	-0.028	0.031	0.977	0.004	0.007	0.997	0.009	0.007	0.998

	Average Apoptosis Genes								
	Distr. A			Distr. B			Distr. C		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-0.175	0.135	-0.743	-0.041	0.047	0.993	-0.036	0.030	0.998
simHAoS-TicToc	-0.038	0.022	0.830	-0.001	0.004	0.998	0.011	0.025	0.995
simHAoS-ApproxTime-nopt	-0.190	0.105	0.341	0.002	0.010	0.996	0.016	0.022	0.998
simHAoS-ApproxTime-opt	-0.132	0.117	0.410	-0.002	0.008	0.997	0.017	0.022	0.997
HAoS-TicToc	-0.066	0.053	0.732	-0.005	0.007	0.998	0.020	0.030	0.997
HAoS-ApproxTime-nopt	-0.094	0.078	-0.791	0.021	0.022	0.997	0.025	0.028	0.999
HAoS-ApproxTime-opt	-0.081	0.061	0.433	-0.039	0.039	0.998	0.004	0.007	0.998
Original2	-0.028	0.031	0.977	-0.024	0.026	0.997	0.026	0.032	0.999

	Average Segregation Genes								
	Distr. A			Distr. B			Distr. C		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-0.033	0.038	-0.241	0.032	0.022	0.997	-0.036	0.030	0.998
simHAoS-TicToc	-0.060	0.046	-0.862	-0.011	0.017	0.997	0.011	0.025	0.995
simHAoS-ApproxTime-nopt	-0.037	0.034	-0.350	-0.013	0.014	0.998	0.016	0.022	0.998
simHAoS-ApproxTime-opt	-0.085	0.043	-0.340	0.020	0.012	0.991	0.017	0.022	0.997
HAoS-TicToc	-0.050	0.029	-0.382	0.000	0.016	0.992	0.020	0.030	0.997
HAoS-ApproxTime-nopt	-0.026	0.025	0.158	-0.027	0.037	0.995	0.020	0.025	0.999
HAoS-ApproxTime-opt	-0.025	0.013	0.826	-0.009	0.013	0.999	0.004	0.007	0.998
Original2	-0.037	0.022	0.150	0.004	0.007	0.997	0.026	0.032	0.999

	Average Apoptosis to Division Ratio (Gene Distribution B)								
	Surgery			Chemotherapy			Both Therapies		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-0.086	0.035	0.996	-0.090	0.022	0.982	-0.068	0.029	0.996
simHAoS-TicToc	-0.093	0.024	0.995	-0.080	0.011	0.987	-0.136	0.035	0.972
simHAoS-ApproxTime-nopt	-0.134	0.031	0.980	-0.106	0.021	0.907	-0.079	0.030	0.974
simHAoS-ApproxTime-opt	-0.096	0.024	0.998	-0.084	0.019	0.973	-0.070	0.029	0.987
HAoS-TicToc	-0.059	0.013	0.988	-0.094	0.014	0.975	-0.102	0.031	0.989
HAoS-ApproxTime-nopt	-0.085	0.017	0.984	-0.087	0.009	0.990	-0.100	0.041	0.911
HAoS-ApproxTime-opt	-0.097	0.033	0.987	-0.056	0.013	0.975	-0.114	0.031	0.954
Original2	-0.028	0.020	0.996	-0.005	0.014	0.978	-0.036	0.029	0.975

	Average Apoptosis to Division Ratio (Gene Distribution C)								
	Surgery			Chemotherapy			Both Therapies		
	ME	STD	COR	ME	STD	COR	ME	STD	COR
OriginalSystemicStyle	-0.039	0.013	0.988	-0.138	0.033	0.994	-0.067	0.021	0.948
simHAoS-TicToc	-0.013	0.024	0.994	-0.129	0.021	0.998	-0.043	0.006	0.973
simHAoS-ApproxTime-nopt	-0.066	0.010	0.995	-0.095	0.006	0.996	-0.053	0.008	0.971
simHAoS-ApproxTime-opt	-0.029	0.016	0.994	-0.095	0.011	0.990	-0.044	0.011	0.978
HAoS-TicToc	-0.049	0.019	0.985	-0.093	0.046	0.863	-0.071	0.007	0.973
HAoS-ApproxTime-nopt	-0.043	0.023	0.986	-0.120	0.024	0.991	-0.049	0.015	0.944
HAoS-ApproxTime-opt	-0.051	0.018	0.992	-0.073	0.012	0.998	-0.030	0.015	0.951
Original2	0.034	0.014	0.996	-0.006	0.014	0.979	0.053	0.026	0.967

Model Performance

As seen on the top portion of Table 5.9, simulations are lengthier for gene distribution A as the models reach the maximum number of generations (since the homeostatic behaviour of the tissue keeps its size well under the maximum cells threshold). As expected, the execution times for the software implementations are similar (10-15% variation). The HAoS simulator needs considerable more time to identify interacting systems triplets. Comparing the (normalized) results from the cancer models executed on HAoS, the time-enabled model requires additional time to implement the tic-toc time phase mechanism. The timeless model cannot be directly compared (its timing is provided in Table 5.9 for reference) as it is normalized against tissue interactions rather than timeslots and also logs more output information as it monitors more events. The approximate time models are faster than the time-enabled, while as expected the few optimizations of its optimized variation (HAoS-ApproxTime-opt) result in it having the best execution times.

While HAoS still outperforms the reference cancer simulation program, the outperforming factor (relative difference in the performance) is smaller than the ones achieved in the previous sections (knapsack problem and MAPK cascade). This is however expected as the cancer model was a corner case, comparing the simulation capabilities of HAoS with a dedicated software implementation running on a high-end desktop computer and identifying interacting systems in a trivial way. Yet, as shown in Figure 5.29, HAoS achieved on average more than 60% performance incresase.

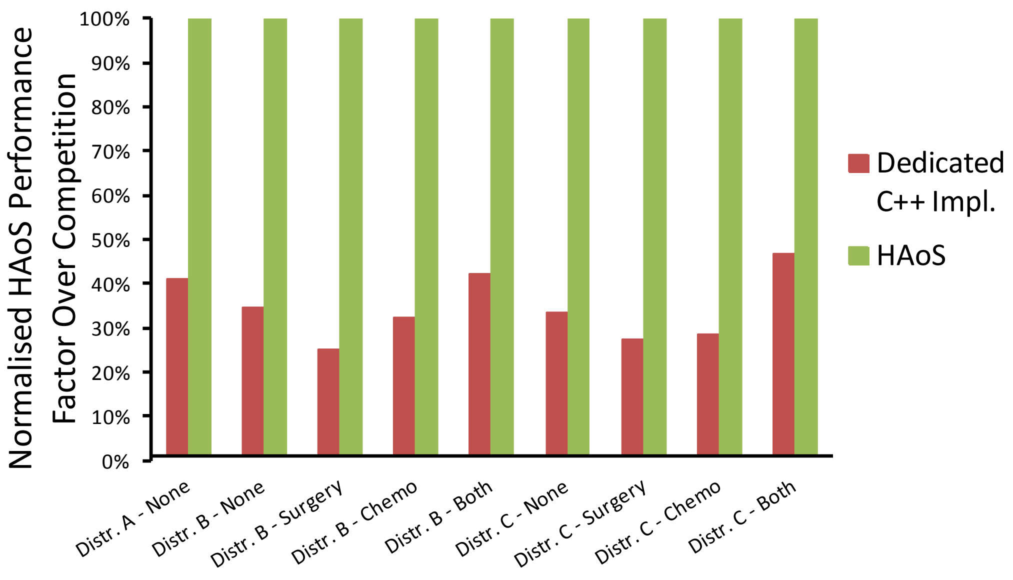


Figure 5.29. Cancer growth experiment HAoS normalised performance against the dedicated c++ implementation

Moreover, further profiling the optimized SC cancer model running on hardware, showed that an additional performance increase can be achieved with a purely-hardware implementation of context-chaining. Since the current context-chaining approach relies on the user to explicitly define the next context to be executed in the chain by software, this results in many time-consuming and execution-blocking memory accesses which may be avoided by a smart implementation preloading the possible (or permitted) chains in the HAoS memories along with the SC program loading. However, this implies that several changes would be required in the hardware, driver and compiler code and, thus the more simplistic current approach was preferred for our prototype.

The complexity of this high-level cancer model and its successful execution on the HAoS platform confirms that the implied SC architecture (systems, scopes, contexts, schemata matching) is effectively supported, meeting this way research challenge *Chg2*. The cancer experiments proved that HAoS can compete with dedicated solutions on modelling real-world biological models, confirming that research challenge *Chg3* has also been adequately met, in terms of both practicality and efficiency.

5.4 Summary

This chapter presented the implementation of three bio-inspired models, using the developed prototype HAoS programming platform, in order to verify its functionality and evaluate its performance against alternative solutions. The three models were carefully selected to represent verification test cases of increasing complexity, testing all aspects of the suggested architecture, both in the hardware and the software domain. These demonstration HAoS applications were developed using the suggested development methodology of section 4.5. Each of the preceding sections provides an introduction to the bio-inspired model, followed by a systemic analysis, details on the model implementation, the experimental setup and the obtained results.

First, a SC application implementing a genetic algorithm optimization of the binary knapsack problem showed the suitability and compatibility of the SC hardware architecture with standard evolutionary methods. This test case was used to evaluate the performance of HAoS against the original sequential and the GPU-based implementations of SC. The results showed the superiority of HAoS, mainly based on the fine-grained parallelism of the TCAM, even when compared with a powerful GPU.

The next application modelled a well-studied biochemical process, the MAPK signalling cascade. Being more complex in nature, it tested more advanced functionality like context adapting and enabled the evaluation of HAoS against the flexible high-level SCoPE SC implementation and a stochastic π-calculus simulator written using functional programming. HAoS again matched the behaviour of the alternative simulators providing a considerable performance gain.

The last HAoS application modelled the effect of genetic abnormalities and therapeutic approaches on cancer growth. In this test case, a comprehensive analysis on the thought process required to build a considerably complex SC model was provided, along with examples of optimizations that can be made at the SC source code and transformation function plugin level to take full advantage of the underlying hardware architecture. The notion of context chaining was introduced as a means of controlling SC interactions that define a chain of events. Additionally, this model was used to validate the functionality of the developed high-level HAoS simulator (a program with functional behaviour similar to the circuitry). The performance of HAoS was evaluated against an optimized dedicated software implementation and showed a competitive advantage, considering that this case represented a worst-case scenario in terms of comparison, due to the straightforward selection of the agents.

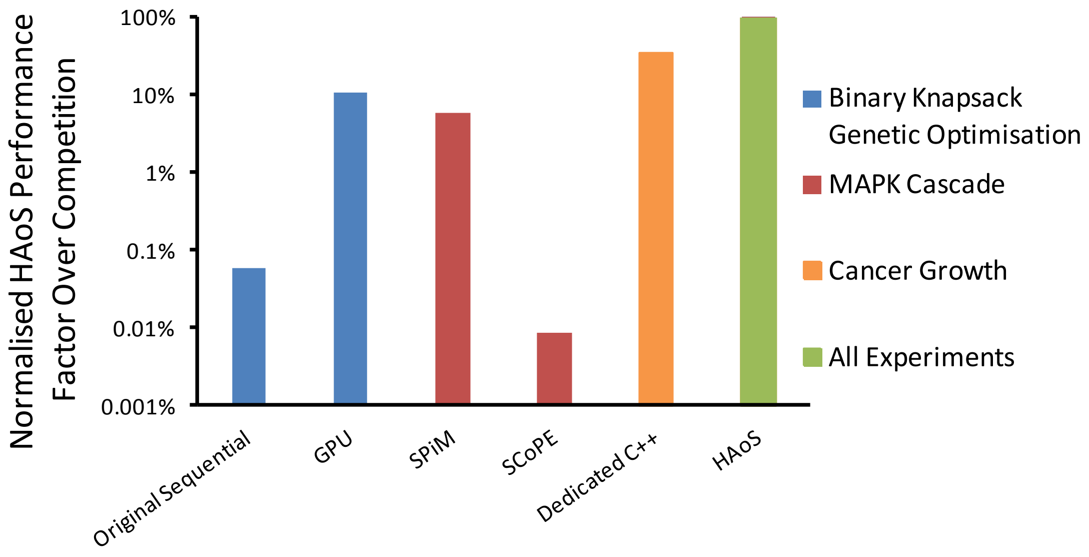


Figure 5.30. Performance Evaluation Results Summary

Analysis for all three HAoS applications showed that research challenges *Chg2* and *Chg3* have been met, as the suggested programming platform successfully simulated a range of SC models of increasing complexity, confirming the support for the implied SC architecture. The evaluation results, as collectively shown in Figure 5.30, show that HAoS consistently outperformed the rival simulators in all cases, confirming it shows the capacity to be used as an efficient and practical simulation solution alternative.

Chapter 6

Conclusion

This final chapter summarises and concludes the thesis. At first, a summary of the work presented in this thesis is provided while revisiting its objectives. Then the contributions of the thesis are listed, followed by a critical evaluation of the research outcomes. Future work is suggested for further investigation and development of the HAoS programming platform. The thesis finishes by describing how the contributions address the three main research challenges and provide evidence to support its hypothesis.

6.1 Summary of Work Revisiting the Objectives

This thesis focuses on the practical hardware implementation of the Systemic Computation paradigm. The objectives of this work, identified and listed in section 1.5, are reviewed below summarising the work presented in this study.

1. Review the work done on Natural Computation with a focus on hardware-based approaches.

An introduction to Natural Computation was given in section 1.1. The computational and behavioural properties of Natural Computation were listed, against the opposing properties of conventional computation, in Table 1.1 (page 19), and they were outlined at the end of the same section. Understanding the concept of those properties is very useful because, in essence, they define natural computation.

Chapter 2 provided a thorough literature review on several approaches to Natural Computation. Various, software and hardware, approaches and computational paradigms on Natural Computation were listed in Table 2.1 (page 30). The software approaches and the computational paradigms were briefly discussed in section 2.1, while conventional (Chip Multiprocessors, supercomputers, pure peer-to-peer networks and GPUs) and unconventional (ubiquitous computing, wireless sensor networks, FPGAs, computing with unconventional materials) hardware-based approaches to natural computation were critically described in section 2.2. An overview of a set of indicative projects (POEtic, PERPLEXUS, SpiNNaker, Molen, DodOrg) was also given to show how various approaches are applied to accomplish, model or mimic natural computation.

2. Review and assess the work done on Systemic Computation (theory and implementations) to date.

A brief introduction on the SC theory was provided in section 1.2 giving its roots. In addition, the conventions followed by SC in order to model biological processes effectively, were listed. In essence, those conventions define the SC paradigm. SC was further described in section 2.3, as it was introduced in the original paper by Bentley. The SC conventions were discussed and the SC graph notation and systems representation was illustrated. A simple demonstration of computation and the progression of a simple SC program were also given to illustrate how SC can be used.

The three prior SC implementations were discussed in section 2.4. The original SC implementation was a low-level simulation of a systemic computer and provided a proof-of-concept for the SC theory. It provided a basic instruction set, an assembly language and corresponding compiler. The second SC implementation was a high-level simulation of a systemic computer and provided flexibility with a high-level SC programming language, a compiler and a virtual machine, a complete runtime environment and visualization tools. The third implementation used the power and parallelism of a GPU, to accelerate SC programs execution with great success, compared to the two previous attempts, since the acceleration factor was in the order of one hundred.

3. Investigate the suitability of available hardware implementation platforms for SC by evaluating them in terms of their ability to support the natural properties of SC (Chg1), the implied SC architecture (Chg2), and practicality/efficiency (Chg3) and select the most appropriate.

In order to evaluate and investigate the suitability of the available implementation platforms, the features that should be incorporated by a practical SC hardware implementation platform, taking into consideration the research challenges, were determined in section 2.5. These included the compatibility of the platform with the SC natural properties (section 1.1) and the SC architecture features (systems, scopes, contexts and interactions), and also I/O efficiency, programmability, design friendliness, technology maturity and scalability.

Signifying and understanding the advantages and disadvantages of each available hardware implementation approach to natural computation in Chapter 2 was crucial to identify which of them could be used as a suitable implementation platform for SC. For this reason, a critical discussion concluded the description of each approach, with regard to their compatibility with the SC paradigm, and those that could define a SC hardware implementation platform were evaluated against the identified implementation requirements. The summary of the evaluation was given in Table 2.3 which was used to discard the less suitable platforms for a SC implementation. FPGAs were finally selected among the two other candidates, GPUs and wireless sensor networks.

4. Analyse the SC architectural features and create a prototype hardware implementation designed to support the SC architecture.

The first Hardware Architecture of Systemic computation (HAoS) was introduced in Chapter 3. The main SC architectural features, focusing more on the computational rather than the behavioural aspects of Natural Computation (see Table 1.1), were depicted and discussed in section 3.2, while potential architectures had been listed earlier in section 3.1. HAoS is a novel custom digital design, which addresses the SC architecture parallelism requirement by exploiting the inbuilt parallelism of an FPGA and by using the highly efficient matching capability of a Ternary Content Addressable Memory (TCAM). Basic processing capabilities were embedded in HAoS, in order to minimize time-demanding data transfers, while the optional use of a CPU provides high-level processing support. The suggested architecture was detailed and its underlying building blocks were discussed in sections 3.4 - 3.6. The CPU interface (see Figure 3.9) was only simulated at this point.

The functional simulation-based verification methodology along with a set of test programs was given in section 3.8.1. Since the target development board had been identified (based on the supported functionality and maturity of the FPGA device family it includes) to be the Xilinx ML605 board, accurate implementation estimates of HAoS for the on-board Virtex-6 LX240T FPGA device were acquired through Xilinx developments tools and summarized in Table 3.6 (section 3.8.2). This first HAoS implementation supported a maximum number of 64 systems.

5. Create a complete and standalone practical SC programming platform with the ability to meet the three challenges.

After a thorough investigation of the most suitable implementation approach for the HAoS-CPU communication interface in section 4.1, it was decided that a soft embedded processor, implemented on the reconfigurable logic, minimized the communication overhead and provided the ability to prototype the communication link. The CPU subsystem, along with various peripherals, was integrated to the HAoS initial design (in section 4.2) through an AXI4-Lite based interface, which was well-suited to handle communications involving control and status registers, as the ones in the register bank of HAoS.

After the custom design was combined with the on-chip soft processor, the Compact Flash Card interface was added to the platform to enable SC programs loading and runtime information logging, making the prototype a standalone solution for simulating natural processes. The initial design was expanded to support the maximum number of systems, being limited only by the size of the target FPGA device while scaling became a matter of changing a single parameter as fully-parameterizable code was used throughout the system. The usability and viability of the platform was also greatly enhanced by the accompanying software framework and the suggested model development methodology.

The prototype has the potential of adding great educational value in the academic community as it combines practical aspects of hardware and software engineering with an unconventional computational paradigm focusing on natural systems modelling.

6. Analyse and address the limitations of the hardware prototype by means of optimizations and enhancements taking into consideration the research challenges.
Various optimizations applied in the initial HAoS architecture in terms of speed and area were discussed in section 4.3. The optimizations included refining the Random Selection Logic by pipelining and careful resource sharing, minimizing the schemata matching overhead by using a register-based TCAM which features a single-clock cycle read and write latency and further addressing I/O efficiency by devising a write-detection mechanism. These optimizations enabled the increase of the operating frequency and throughput and the decrease of the overall latency compared to the initial design. Timing-based verification was conducted to validate the optimizations before the design was implemented on hardware (downloaded on the FPGA). The enhancements included addressing user-friendliness, by providing a HAoS functional simulator to expedite and ease SC models development, and programmability by introducing a complete HAoS programming toolchain and an accompanying software framework, which were then used to formulate a HAoS model development methodology (in section 4.5).

7. Evaluate the ability of the prototype SC platform to meet the three challenges by simulating natural models against alternative solutions.
An initial evaluation against prior SC implementations was provided in section 5.1, where the same SC source code solving a typical genetic algorithm optimization

problem, the binary knapsack problem, was executed by the original SC implementation, the GPU SC implementation and HAoS. SCoPE was excluded from this evaluation since it uses a different version of the SC language but it can be safely assumed that its performance would be similar to the original version (both being purely software implementations relying solely on conventional hardware architectures). Experimental results showed that HAoS provides an effective solution in terms of efficiency versus flexibility trade-off and can potentially outperform prior implementations.

A well-studied biochemical process, the MAPK signalling cascade, was the second SC model, developed using the methodology of section 4.5 and presented in section 5.2, simulated with the HAoS prototype platform. Although this experiment was used as a means of verification and evaluation of the platform against alternative high-level simulators, SCoPE and SPiM, it also provided an example of simulating a highly stochastic and approximated model.

The third and most complex SC model, out of the models presented in this thesis, examined the role of chromosome missegregation, a cellular anomaly of genetic origin, in the development of a tumour. External stimuli were also modelled in the form of typical cancer therapies, chemotherapy and surgery. In contrast with the other models, presented in the first two sections of Chapter 5, which retargeted previously introduced SC models to the HAoS programming platform, the cancer SC model was developed from scratch in the context of this thesis. For this reason and due to the increased complexity of this model, a thorough explanation and a detailed systemic analysis were provided before reaching the alternative ways of approaching and implementing such a model. Thus four variations of this SC model were presented, all representing the functionality of the reference model, trying to identify a balanced choice in terms of accuracy and efficiency. This test case was also used to evaluate the correctness of the HAoS functional simulator. The results showed that HAoS can outperform the optimized C++ reference model while correctly modelling its complex behaviour.

6.2 Contributions

This work contributes to the fields of systemic computation, natural computation and, in general, computer science by providing:

- A critical review of hardware-based approaches to systemic and natural computation and identification of the requirements of an implementation platform, in order to support a practical SC hardware implementation.

- Critical analysis of natural computation implementation platforms with respect to SC and the derived requirements.
- Determination of the most appropriate hardware implementation platform for a practical SC implementation.
- Design of the first hardware SC architecture taking into consideration the flexibility and performance trade-off.
- Introduction of a complete and practical standalone platform to simulate natural systems, accompanied by
  - a programming toolchain,
  - a software framework and
  - a model development methodology
- A custom hardware write-detection mechanism used to decrease CPU accesses to a local resister file.
- A custom random selection circuit that selects a set bit from a given bus and returns its position.
- Introduction of the concept of context chaining in SC applications.
- SC programming examples executed on hardware showcasing efficient natural systems modelling.
- Introduction of a SC cancer model focusing on chromosome missegregation and including genetic and external stimuli.

6.3 Critical Evaluation

The various design decisions and choices involved in the development of the resulting prototype, from the selection of hardware implementation platform to the layout of the hardware architecture and then the development of its accompanying software, have been explained throughout chapters 2 to 4. These decisions have been primarily based on finding a balance between efficiency and flexibility and were driven by currently available technologies and design methodologies. Yet, it has been evident from the analysis given before any decision was made, that there was usually no single correct answer to each design challenge faced along this work.

Implementation Platform Selection

According to the analysis of section 2.5, FPGAs were selected as the most suitable platform for a hardware implementation of SC. While this decision was made due to maturity of the FPGA technology and its great potential for fine-grained parallelism, advancements in emerging technologies as Quantum and DNA computing may enable the implementation of SC in a more natural substrate in the future.

Regarding the other strong candidates for realizing SC, from the discussion of section 2.5, both Wireless Sensor Nodes and GPUs provide advantages that would make them suitable candidates. The comparison between the FPGA-based HAoS prototype and the GPU SC implementation using the binary knapsack test case (section 5.1) reveals the superiority of a dedicated hardware architecture over the power of a GPU. However, while FPGAs are widely used having a plethora of commercial applications, GPU design advancements are mainly driven by the power-hungry and ever-demanding gaming industry. Thus, the two fields should continue evolving with the same pace for the FPGA platform to continue being the most favourable option. A quantitative comparison between the FPGA-based and a WSN-based approach was not possible as such an implementation is not available yet. Thus, since the features of a WSN network are well-aligned with the natural properties of SC, a WSN SC implementation may still prove useful to realize.

Hardware Architecture

As stated in section 1.2, the two main tasks implied by SC are the identification of the interacting systems and then the transformation of those systems. The competitive advantage of the suggested hardware architecture lies on the efficiency of the TCAM performing schemata matching in a parallel manner, regarding the first implied task, and the low-latency communication with the embedded CPU, regarding the second. However, it is clear that there is still great performance gain potential regarding the parallelization of the transformation task which may be addressed by using on-chip available resources or additional off-chip processing elements.

The rationale behind choosing the soft processor in section 4.1 is the extremely low communication overhead which can however be negligible when the runtime of a task increases. While the embedded processor approach was proven sufficient to prototype the suggested hardware architecture (according to its evaluation against alternative simulators in Chapter 5), more computationally intensive natural models may require more raw processing power which may be addressed by the computation-offloading hybrid approach suggested in section 4.1 and further low-level hardware optimizations and enhancements.

While the suggested hardware architecture was designed to implement the SC paradigm in an efficient and practical way, it is acknowledged that it does not fully support all the natural properties that are implied by SC. Evidently, the focus of this work was more towards the computational rather than the behavioural properties supported by the SC concept in theory. The FPGA platform was selected for the increased level of support it can provide for these implied properties and some of them like parallelism and stochastic interactions have been implemented on the hardware level. The behavioural properties are left to be simulated by the SC applications running natively on the HAoS platform, e.g. self-adaptation and fault-tolerance can be sufficiently demonstrated using a genetic algorithm [22] which being greatly compatible with SC, it can be easily simulated and efficiently mapped to the underlying architecture of HAoS as shown in section 5.1.

Prototype Implementation

The suggested hardware design has been written in highly-parameterized VHDL code, enabling its effortless migration to any FPGA device. The selected FPGA development board featured a midrange FPGA device in terms of size and included a rich set of features[35]. As FPGA technology evolves and modern devices provide more efficient reprogrammable solutions, HAoS is not constrained to a specific vendor, it can be easily scaled just by changing a single parameter in the source code (as long as the design fits to the target device) and is fairly future-proof as it uses an industry-standard communication interface to its embedded CPU. Using this flexibility, the number of maximum supported systems using a single device may be adequate to simulate fairly complex natural systems. However, scaling the architecture further than a single FPGA device, realizing a distributed architecture of HAoS nodes, may be beneficial for real-world modelling scenarios.

SC Model Development and HAoS Programming

The three SC models presented in Chapter 5 attempted to cover a wide[36] range of SC applications with varying levels of behaviour complexity. The suggested model

[35] As various communication interfaces (PCI Express Gen2, USB 2.0, Gigabit Ethernet and DDR3 memory interface) and on-board peripherals (FMC expansion connectors, SD card controller and an LED screen).

[36] Wide in the context of a research thesis.

development methodology can facilitate further exploring new models while the provided programming toolchain and software framework can assist in making HAoS programming straightforward for potential SC programmers, familiar with conventional programming methodologies and techniques. However, an understanding of the main concepts of SC, the basic mechanics of the underlying hardware and software development for an embedded processor is encouraged for efficient HAoS model development.

6.4 Future Work

Although the current prototype has been proven to be fully functional, there are several further improvements that could be made to increase its efficiency. The following suggestions address the deficiencies of the HAoS platform, identified in the critical evaluation section above.

Hardware Architecture

In order to maximize the utilization of on-chip resources, the vast number of available DSP building blocks can be used as discrete parallel processing elements and any remaining FPGA fabric can be used as a dynamically reconfigurable area for predetermined hardware-supported functions. As FPGAs provide the flexibility to partially reconfigure the device to implement a different circuit every time on a predetermined area of the available fabric, a different circuit could be downloaded on the FPGA, according to the requirements of a systemic program, which then would highly optimize the performance of the system. The supported reconfigurable function set could either include only predetermined hardware functions or any supported function by a high-level synthesis tool or C-to-HDL compiler [164], [242], [243]. Essentially this feature would imply that instead of having a fixed instruction set supported by the FU (see Figure 3.9), the FU would be itself reconfigurable and tailored to the specific SC application, maximizing this way resources utilization.

Another enhancement to the suggested hardware architecture would be the addition of an extra communication interface on the platform to provide the possibility of HAoS offloading computationally-intensive tasks to a conventional CPU through a PCI-Express link. The motivation behind this is the acknowledgement that a low-performance soft processor can be a poor choice if a heavy computational task is required. A smart solution would execute hardware supported tasks within the HAoS FU, low-demanding general-purpose processing tasks to the low-latency low-throughput on-chip processor and computationally expensive processing tasks to the high-latency high-throughput off-chip conventional CPU. Thus, adding this option to the platform, would give the user the choice between having a standalone solution and fine-tuning the system performance according to the processing requirements of the given application.

More low-level enhancements that would increase the efficiency and flexibility of the architecture are also possible. Some examples would be supporting variable parts next to the transformation function section of a system holding auxiliary program information, an efficient implementation of context chaining supported inherently by hardware and also further increasing the size of the schemata of a number of systems in order to be able to hold more local information.

Prototype Implementation

Naturally, the raw performance of the platform may also be increased by retargeting the HAoS architecture to the latest FPGA family, using a bigger and faster FPGA device to enable more systems to be modelled (using a single device) and the operating frequency of the custom logic to be increased without further architectural changes to the design. Also the operating frequency of the processor may also be increased by an order of magnitude, if the selected FPGA device makes use of a powerful hard CPU, instead of the low-end soft CPU of the HAoS prototype, implemented next to the reconfigurable logic (as it is the case for the recently commercialized Xilinx Zynq[37] Extensible Processing Platform [202] and the recently announced Altera Stratix 10 FPGA platform which includes a quad-core A53 64-bit ARM processor fabricated on an Interl 14-nm 3D Trigate Transistor process, as discussed in section 4.1). This approach would maintain the low-latency communication advantages of the suggested design. A migration to such a platform would not impose altering the suggested communication interface, as the selected AXI4 based communication protocol (being an industry standard) used by HAoS, is readily supported by such modern solutions. However, minor changes would be required on the software side.

[37] The configurable logic provided by these devices is still limited since a full dual-core ARM CPU is also implemented on the same chip. The largest currently available Zynq device (Z-7100) provides 444K reconfigurable logic cells while the midrange Virtex-6 FPGA used for implementing HAoS includes 241K cells. However, with FPGA manufacturers moving to smaller technology nodes (currently announced down to 14-16nm), these limitations will be more efficiently addressed as the technology matures. The highest operating frequency for the ARM Cortex-9 CPU in the Zynq family is currently 1GHz (Z-7045-3 device) [202].

An investigation on the scaling of the HAoS platform beyond the single-FPGA implementation is also suggested. This could be accomplished with using an external CAM configuration in order to address the increasing number of systems requirement. This approach would have a performance penalty but it would enable a broader range of SC applications. As the on-chip CAM is the most area-consuming block of HAoS, its absence will provide a high number of maximum supported systems, even when only one FPGA device is used. Moreover, external TCAMs can easily be incorporated to an FPGA-based design through dedicated or generic communication interfaces and also be scaled by cascading multiple devices [244][245]. A second approach that could address the scalability of the HAoS architecture is an FPGA cluster [93], with each FPGA defining a separate scope or part of a scope and system transfers/exchanges happening between the discrete FPGA nodes. Taking into consideration budget limitations of a hardware realization of this approach, the functionality may be simulated and tested on a configuration initially using a small number of FPGAs. A network interface will probably have to be designed at the bounds of each chip. A shared bus topology or a wireless link may help addressing the communication-related scalability issues. Address-Event Representation [111] may also be considered to be adopted by the design in order to compensate for the limitation of the I/O pins of the FPGAs.

SC Natural Models Development and HAoS Programming

A natural extension of this work would involve exploring more natural models and developing SC applications which would fully exploit the efficiency of the suggested programming platform. Specifically, SC applications showing the level of support for the behavioural properties of Natural Computation would be especially interesting, using the work presented in [22] (exploiting self-adaptation, robustness, fault-tolerance, homoeostasis and self-organisation) as a starting point.

The HAoS programming framework could also be greatly enhanced by further automating parts of the Conceptual and Application Layer (see Figure 4.5) with the addition of a high-level SC graph tool which would translate the graphical notation of a SC model to calculus notation and the corresponding SC source code.

Implementation Platform

As noted in the previous section, it would be interesting to also explore additional suitable implementation platforms and evaluate their performance and compatibility with the SC paradigm. An obvious candidate would be a WSN-based [79] approach while alternatively a SpiNNaker-based [132] implementation would also be appealing once the final platform is available.

6.5 Closing Words

The hypothesis of this thesis was to prove the viability and utility of a practical SC hardware implementation. In order to accomplish this, an overview was first given on the fields of Natural and Systemic Computation to introduce their base concepts and non-conventional nature. Then, in order to provide evidence to support the hypothesis, three research challenges had to be addressed:

Chg1: How can a hardware platform support the central SC natural properties?

Acknowledging the fact that the implementation of a hardware platform that fully supports all the SC inherent natural properties is not yet realistic, this thesis attempted to identify a compromise based on the various trade-offs provided by current technologies and design techniques. A critical analysis of hardware-based approaches to natural computation was presented, followed by the identification of the key requirements for an implementation platform which would provide sufficient support for as many as possible of the implied natural properties of SC, focusing more on the computational part. The investigation of the compatibility of the various available implementation platforms with the desired properties led to the selection of FPGAs as the most suitable choice to implement the first Hardware Architecture of Systemic computation (HAoS). The natural properties were also taken into consideration along the design of the custom hardware and its accompanying software. After analysis of the available options, constraints and trade-offs, a few of the properties (as stochastic execution and parallelism) were incorporated to the suggested hardware platform, while the rest remained to be supported on a software level (e.g. by using a genetic algorithm).

Chg2: How can a hardware platform support the underlying architecture of SC?

In order to support the underlying SC architecture, a hardware platform should be implemented on a substrate which is compatible with the specific features of SC: systems, scopes, contexts and interactions[38]. For this reason, the compatibility with these features was also considered in the implementation platform investigation. Since the FPGA platform was selected, a design analysis concluded that systems and scopes would be stored on system RAM to optimize area utilization, while register-based constructs

[38] Including schemata matching and random selection

would provide parallel access to performance-critical status information. A new, more hardware-friendly, systems representation and coding method was devised in order to optimally map the architecture to the hardware resources. A Ternary Content Addressable Memory was selected to handle the demanding task of schemata matching to implement valid triplet generation in a purely parallel manner. An optimized and dedicated hardware state-machine was implemented to control the interaction flow. In addition, a custom circuit was designed to handle the random selection task, which was used to randomly identify a valid scope among all scopes in the SC program, a context among all contexts in a scope and a pair of interacting systems among all matching systems. The support for the SC architecture was further revised with low-level optimizations and it was also evaluated and verified using high-level bio-inspired SC models running live on the suggested hardware platform.

Chg3: How can a hardware platform meet the first two challenges while also being practical and efficient?

Practicality and efficiency were also considered during the investigation of the most appropriate technology/substrate, which would be used to implement HAoS. The requirements regarding this challenge were identified to be I/O efficiency, programmability, design-friendliness, technology maturity and scalability. After the selection of the implementation platform and the introduction of the base HAoS architecture, the design effort was focused on optimizations and enhancements targeting a more practical and efficient simulation platform. A critical analysis regarding the HAoS-CPU communication interface led to the selection of an embedded CPU due to the minimal communication latency. Low-level optimizations in the RSL, the TCAM and the I/O boundary increased significantly the efficiency of the platform. Additionally, the user experience and the level of practicality where substantially enhanced by the functional model of the design (HAoS simulator), the programming toolchain, the software framework and the programming methodology, which greatly expedited SC models development targeting HAoS. The efficiency of the platform was evaluated by simulating natural models and it was validated by outperforming prior SC implementations and alternative simulation environments.

To sum up, this thesis met all three research challenges since the resulting prototype was realized on the most compatible to the desired natural properties implementation platform (*Chg1*) and implemented the SC paradigm and its implied architecture (*Chg2*) in a practical way, employing widely-used programming techniques and methodologies (using C/C++ for transformation functions implementation) on a mature technology (FPGAs combined with embedded processing). Additionally, the efficiency of the platform (*Chg3*) was shown through evaluation, as HAoS has been shown to have the capacity of outperforming competing solutions proving the viability and utility of the suggested design (illustrated in Figure 6.1). Thus, by meeting the research challenges, this thesis provides compelling evidence to support the hypothesis that it is possible to implement a practical Systemic Computation hardware architecture that is viable and useful.

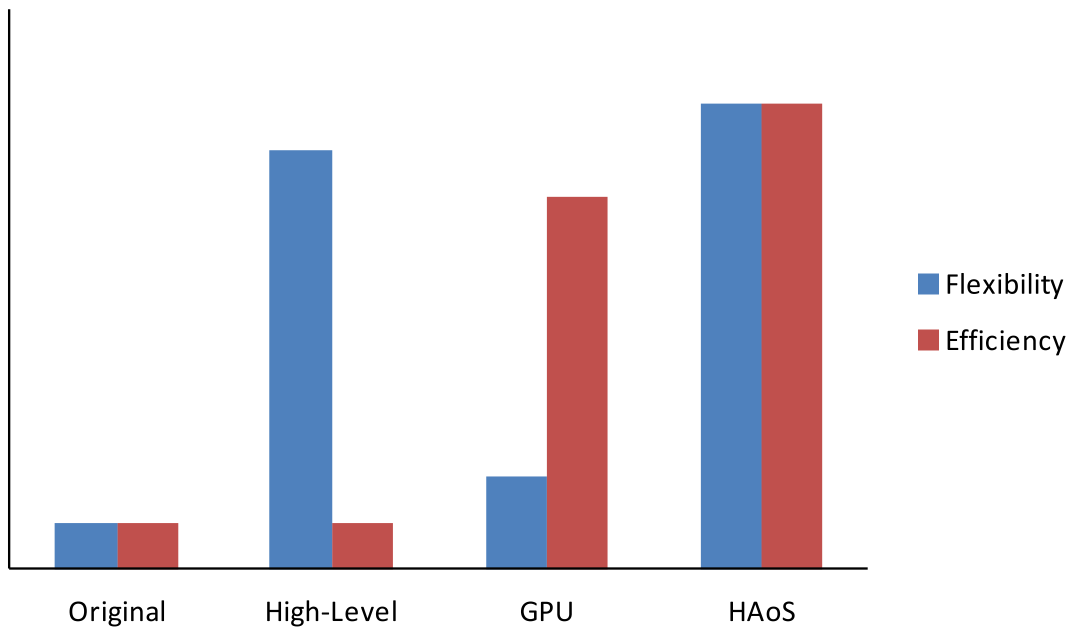


Figure 6.1. Comparison in flexibility and efficiency provided by the HAoS programming platform, contributed by the work presented in this thesis, against to prior SC implementations. The suggested practical hardware-based implementation provide a balanced SC programming solution

Throughout this study, it has been highlighted that nature seems to work in a massively parallel fashion. The creation of new computer architectures better suited to model natively natural systems is the dream of many hardware engineers. This thesis is a stepping stone towards that goal.

References

[1] P. J. Bentley, "Everything Computes," in *Paper accompanying Keynote Seminar in Proceedings of Third Iteration, the Third International Conference on Generative Arts*, 2003, pp. 15–24.

[2] G. Păun, "From Cells to (Silicon) Computers, and Back," in *New computational paradigms: changing conceptions of what is computable*, B. Cooper, B. Lowe, and A. Sorbi, Eds. Springer Verlag, ISBN: 978-0387360331, 2008, pp. 343–371.

[3] S. Stepney, S. Braunstein, J. Clark, A. Tyrrell, A. Adamatzky, R. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, and D. Partridge, "Journeys in non-classical computation I: A grand challenge for computing research," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 20, no. 1, pp. 5–19, Mar. 2005.

[4] S. Stepney, S. L. Braunstein, J. A. Clark, A. Tyrrell, A. Adamatzky, R. E. Smith, T. Addis, C. Johnson, J. Timmis, P. Welch, R. Milner, and D. Partridge, "Journeys in non-classical computation II: initial journeys and waypoints," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 21, no. 2, pp. 97–125, Apr. 2006.

[5] H. Markram, "The blue brain project," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, 2006.

[6] R. Carlson, "The pace and proliferation of biological technologies," *Biosecurity and Bioterrorism: Biodefense Strategy, Practice, and Science*, vol. 1, no. 3, pp. 203–214, 2003.

[7] M. Schulz, "The end of the road for silicon," *Nature*, vol. 399, no. 6738, pp. 729–730, 1999.

[8] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[9] R. E. Blankenship, *Molecular mechanisms of photosynthesis*. Blackwell Science Oxford, ISBN: 0632043210, 2002, p. 321.

[10] E. R. Kandel, J. H. Schwartz, and T. M. Jessell, *Principles of neural science*. McGraw-Hill, ISBN: 0838580688, 2000.

[11] L. N. de Castro, "Fundamentals of natural computing: an overview," *Physics of Life Reviews*, vol. 4, no. 1, pp. 1–36, 2007.

[12] J. H. Holland, *Adaptation in natural and artificial systems*. MIT press Cambridge, MA, ISBN: 978-0-262-08213-6, 1992.

[13] M. H. Hassoun, *Fundamentals of artificial neural networks*. The MIT Press, ISBN: 978-0262082396, 1995, p. 511.

[14] L. N. de Castro and J. Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*. London: Springer-Verlag, ISBN: 978-1-85233-594-6, 2002, p. 380.

[15] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm intelligence: from natural to artificial systems*. New York, NY, USA: Oxford University Press, ISBN: 0195131592, 1999, p. 307.

[16] J. Kennedy, R. C. Eberhart, and others, "Particle swarm optimization," in *Proceedings of IEEE international conference on neural networks*, 1995, vol. 4, pp. 1942–1948.

[17] S. Wolfram, *A New Kind of Science*. Wolfram Media, Inc., ISBN: 1-57955-008-8, 2002.

[18] A. Lindenmayer, "Mathematical models for cellular interaction in development, Part 1 and 2," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 280–315, 1968.

[19] C. G. Langton, *Artificial life: An overview*. Cambridge Ma.: The MIT Press, ISBN: 0-262-62112-6, 1997, p. 352.

[20] L. M. Adleman, "Computing with DNA," *Scientific American*, vol. 279, no. 8, pp. 34–41, 1998.

[21] C. Bennett and D. DiVincenzo, "Quantum information and computation," *Nature*, vol. 404, no. 6775, pp. 247–255, Mar. 2000.

[22] E. Le Martelot, "Investigating and Analysing Natural Properties Enabled by Systemic Computation within Nature-inspired Computer Models." EngD Thesis, Department of Electrical & Electronic Engineering, UCL, London, p. 363, 2010.

[23] L. Kari and G. Rozenberg, "The many facets of natural computing," *Communications of the ACM*, vol. 51, no. 10, pp. 72–83, 2008.

[24] P. J. Bentley, "Systemic computation: A model of interacting systems with natural characteristics," *International journal of parallel, emergent and distributed systems*, vol. 22, no. 2, pp. 103–121, 2007.

[25] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 1, p. 230, 1937.

[26] S. Wolfram, *Theory and applications of cellular automata*. World Scientific, ISBN: 978-9971501235, 1986.

[27] M. Cook, "Universality in Elementary Cellular Automata," *Complex Systems*, vol. 15, pp. 1–40, 2004.

[28] D. Eriksson, "A principal exposition of Jean-Louis Le Moigne's systemic theory," *Cybernetics and Human Knowing*, vol. 4, no. 2–3, pp. 33–77, 1997.

[29] L. Von Bertalanffy, *General system theory: Foundations, development, applications*. G. Braziller New York, ISBN: 978-0807604533, 1968.

[30] R. Descartes, "Discourse on the method, trans. J. Cottingham, R. Stoothoff, and D. Murdoch, The Philosophical Writings of Descartes, vol. 2." Cambridge University Press.(Original published in 1637.), 1984.

[31] R. Fuenmayor, "The roots of reductionism: A counter-ontoepistemology for a systems approach," *Systemic Practice and Action Research*, vol. 4, no. 5, pp. 419–448, 1991.

[32] P. W. Anderson, "More is different," *Science*, vol. 177, no. 4047, pp. 393–396, 1972.

[33] B. Boehm and C. Abts, "COTS integration: plug and pray?," *Computer*, vol. 32, no. 1, pp. 135–138, Jan. 1999.

[34] M. Rouhipour, P. J. Bentley, and H. Shayani, "Systemic Computation using Graphics Processors," in *Proc. of 9th International Conference on Evolvable Systems - From Biology to Hardware*, 2010.

[35] W. A. Richards, *Natural computation*. Cambridge, MA, USA: Mit Press, ISBN: 0262680556, 1988, p. 480.

[36] Y. Gurevich and S. Shelah, "Expected computation time for Hamiltonian path problem," *SIAM Journal on Computing*, vol. 16, no. 3, pp. 486–502, 1987.

[37] C. Sakellariou and P. J. Bentley, "Computing Nature at the Intersection with Chemistry: Innovative Architectures," to appear in *Genesis Engines: Computation and Chemistry in the Quest for Life's Origins*, 1st ed., B. Damer, R. Gordon, and J. Seckbach, Eds. Springer, 2013.

[38] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, Dec. 1975.

[39] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," in *Operating Systems, Proceedings of an International Symposium*, 1974, vol. 16, pp. 171–187.

[40] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE transactions on software engineering*, vol. SE-11, pp. 1491–1501, 1985.

[41] J. C. Giarratano and G. D. Riley, *Expert Systems: Principles and Programming*, 4th ed. Course Technology, ISBN: 978-0534950538, 2004, p. 856.

[42] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, ISBN: 978-0521899437, 2008, p. 504.

[43] "Biotica." Online : http://www.mimetics.com/vur/biotica.html, Accessed: August, 2013.

[44] S. Ulam, “On Some Mathematical Problems Connected with Patterns of Growth and Figures,” in *American Mathematical Society Proceedings of Symposia in Applied Mathematics*, 1962, vol. 14, pp. 215–224.

[45] J. von Neumann, *Theory of Self-Reproducing Automata*. Champain, IL, USA: University of Illinois Press, ISBN: 9780252727337, 1966.

[46] J. H. Holland, *Emergence: From Chaos to Order*. Oxford University Press, ISBN: 978-0192862112, 2000.

[47] R. Milner, “The Polyadic pi-Calculus: A Tutorial.” Technical Report, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1991.

[48] G. Boudol, “Asynchrony and the Pi-calculus.” Technical Report: Rapports de Recherche, No 1702, Unite de Recherche, Inria-Sophia Antipolis, p. 15, 1992.

[49] C. Priami, “Stochastic π-Calculus,” *The Computer Journal*, vol. 38, no. 7, pp. 578–589, Jul. 1995.

[50] L. Cardelli and A. D. Gordon, “Mobile Ambients,” in *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, 1998, pp. 140–155.

[51] C. A. Petri, “Communication with Automata.” Applied Data Research Inc, Princeton NJ, 1966.

[52] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[53] R. Milner, “Bigraphical Reactive Systems,” in *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, 2001, pp. 16–35.

[54] B. Goertzel, “Multiboundary Algebra as Pregeometry,” *Electronic Journal of Theoretical Physics*, vol. 4, no. 16(II), pp. 173–186, 2007.

[55] A. Regev, “BioAmbients: an abstraction for biological compartments,” *Theoretical Computer Science*, vol. 325, no. 1, pp. 141–167, Sep. 2004.

[56] G. Păun, “Introduction to membrane computing,” in *Applications of membrane computing*, G. Ciobanu, M. J. . Pérez-Jiménez, and G. Paun, Eds. (Eds), Heidelberg: Springer, ISBN: 978-3-540-25017-3, 2006, pp. 1–42.

[57] L. Cardelli, “Brane Calculi. Interactions of biological membranes,” in *Proceedings of Computational Methods in System Biology 2004 (CMSB 2004)*, 2005, pp. 257–280.

[58] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and A. Troina, “The calculus of looping sequences for modeling biological membranes,” in *Proceedings of the 8th international conference on Membrane computing*, 2007, pp. 54–76.

[59] L. Cardelli, “Bioware languages,” in *Computer Systems: Theory, Technology and Applications*, A. Herbert and K. Sparck Jones, Eds. (Eds), Springer-Verlag, ISBN: 978-0387201702, 2004, pp. 59–65.

[60] C. Hewitt and H. Lieberman, “Design Issues in Parallel Architecture for Artificial Intelligence,” *Artficial Intelligence Memo - No. 750*, 1983.

[61] A. N. Afuah and J. M. Utterback, “The emergence of a new supercomputer architecture,” *Technological Forecasting and Social Change*, vol. 40, no. 4, pp. 315–328, 1991.

[62] L. Hammond, B. A. Nayfeh, and K. Olukotun, “A Single-Chip Multiprocessor,” *Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[63] S. L. Graham, M. Snir, and C. A. Patterson, *Getting up to speed: The future of supercomputing*. National Academy Press, ISBN: 978-0-309-09502-0, 2005, p. 308.

[64] M. J. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[65] M. Baker and R. Buyya, “Cluster computing: the commodity supercomputer,” *Software: Practice and Experience*, vol. 29, no. 6, pp. 551–576, 1999.

[66] “Computer Cluster Links.” Online: http://www.tu-chemnitz.de/informatik/RA/cchp/, Accessed: August, 2013.

[67] E. Marcus and H. Stern, *Blueprints for High Availability: Designing Resilient Distributed Systems*. John Wiley & Sons, ISBN: 978-0471356011, 2000, p. 368.

[68] M. Yevmenkin, D. Fullagar, C. Newton, and J. G. Koller, “Load-Balancing Cluster.” US Patent App. 12,390,560, Google Patents, United States, p. 8, 2009.

[69] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, “Peer-to-peer computing,” Tech. Rep. HPL-2002-57, Citeseer, HP Labs, Palo Alto, 2002.

[70] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, “A survey of general-purpose computation on graphics hardware,” in *Computer Graphics Forum*, 2007, vol. 26, no. 1, pp. 80–113.

[71] M. Macedonia, “The GPU enters computing’s mainstream,” *Computer*, vol. 36, no. 10, pp. 106–108, 2003.

[72] M. Weiser, “The Computer for the Twenty-First Century,” *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[73] D. Saha and A. Mukherjee, “Pervasive computing: a paradigm for the 21st century,” *Computer*, vol. 36, no. 3, pp. 25–31, 2003.

[74] A. C. W. Wong, D. McDonagh, O. Omeni, C. Nunn, M. Hernandez-Silveira, and A. J. Burdett, “Sensium: an ultra-low-power wireless body sensor network

platform: Design & application challenges," in *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, 2009, pp. 6576–6579.

[75] N. Shadbolt, "Ambient intelligence," *IEEE Intelligent Systems*, vol. 18, pp. 2–3, 2003.

[76] D. Wright, S. Gutwirth, M. Friedewald, E. Vildjiounaite, and Y. Punie, *Safeguards in a world of ambient intelligence*. Springer-Verlag New York Inc, ISBN: 978-1-4020-6661-0, 2008.

[77] D. K. Arvind and K. J. Wong, "Speckled computing: Disruptive technology for networked information appliances," in *IEEE International Symposium on Consumer Electronics*, 2004, pp. 219–223.

[78] N. Gershenfeld, R. Krikorian, and D. Cohen, "The Internet of Things.," *Scientific American*, vol. 291, no. 4, pp. 76–81, 2004.

[79] F. Zhao and L. Guibas, "Wireless sensor networks," in *Communications Engineering Desk Reference*, 1st Editio., Academic Press, ISBN: 978-0123746481, 2009, p. 247.

[80] K. Romer and F. Mattern, "The design space of wireless sensor networks," *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 54–61, 2004.

[81] F. L. Lewis, "Wireless sensor networks," in *Smart environments: technologies, protocols, and applications*, D. J. Cook and S. K. Das, Eds. Wiley-Interscience, 2004, pp. 1–18.

[82] Moteiv Corporation, "Tmote Sky Datasheet v1.0.4." Online: http://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/tmote_sky_datasheet.pdf, Accessed: August, 2013.

[83] E. Y. Song and K. Lee, "Understanding IEEE 1451-Networked smart transducer interface standard - What is a smart transducer?," *Instrumentation Measurement Magazine, IEEE*, vol. 11, no. 2, pp. 11–17, 2008.

[84] P. J. Bentley, "Designing Biological Computers: Systemic Computation and Sensor Networks," in *Bio-Inspired Computing and Communication*, 2008, pp. 352–363.

[85] M. Bolton, "Introduction and PGA bibliography," *Microprocessors and Microsystems*, vol. 13, no. 5, pp. 299–303, 1989.

[86] R. H. Freeman, "Configurable electrical circuit having configurable logic elements and configurable interconnects." US Patent 4,870,302, Google Patents, USA, 1989.

[87] S. Hauck, "The roles of FPGA's in reprogrammable systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[88] H. Shayani, P. J. Bentley, and A. M. Tyrrell, “An FPGA-based Model suitable for Evolution and Development of Spiking Neural Networks,” in *Symposium A Quarterly Journal In Modern Foreign Literatures*, 2008, pp. 23–25.

[89] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, “NetFPGA--An Open Platform for Gigabit-Rate Network Switching and Routing,” in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, 2007, pp. 160–161.

[90] K. Bondalapati and V. K. Prasanna, “Reconfigurable computing systems,” *Proceedings of the IEEE*, vol. 90, no. 7, pp. 1201–1217, 2002.

[91] A. H. Jackson and A. M. Tyrrell, “Implementing asynchronous embryonic circuits using AARDVArc,” *Proceedings 2002 NASA/DoD Conference on Evolvable Hardware*, pp. 231–240, 2002.

[92] G. Tempesti, D. Mange, P.-A. Mudry, J. Rossier, and A. Stauffer, “Self-Replicating Hardware for Reliability: The Embryonics Project,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 3, no. 2, p. 9, 2007.

[93] O. Mencer, K. H. Tsoi, S. Craimer, T. Todman, W. Luk, M. Y. Wong, and P. H. W. Leong, “Cube: A 512-fpga cluster,” in *Programmable Logic, 2009. SPL. 5th Southern Conference on*, 2009, pp. 51 – 57.

[94] T. Higuchi, Y. Liu, and X. Yao, *Evolvable hardware*. ISBN: 978-0387243863, Springer, 2006, p. 224.

[95] P. C. Haddow and A. M. Tyrrell, “Challenges of evolvable hardware: past, present and the path to a promising future,” *Genetic Programming and Evolvable Machines*, vol. 12, no. 3, pp. 183–215, Sep. 2011.

[96] H. Shayani, “A Practical Investigation into Achieving Bio-Plausibility in Evo-Devo Neural Microcircuits Feasible in an FPGA.” EngD Thesis, Department of Computer Science, UCL, London, p. 343, 2013.

[97] H. Restrepo and D. Mange, “An Embryonics implementation of a self-replicating universal Turing machine,” *Evolvable Systems: From Biology to Hardware*, pp. 74–87, 2001.

[98] A. Stauffer, D. Mange, G. Tempesti, and C. Teuscher, “BioWatch: A giant electronic bio-inspired watch,” in *2001. Proceedings. The Third NASA/DoD Workshop on Evolvable Hardware.*, 2001, p. 185.

[99] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, “Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware,” in *Lecture Notes In Computer Science; Vol. 1259. Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware*, 1996, pp. 33–54.

[100] G. W. Greenwood and A. M. Tyrrell, *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems*. Wiley-IEEE Press, ISBN: 0471719773, 2006.

[101] A. M. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J. M. Moreno, J. Rosenberg, and A. E. P. Villa, "Poetic tissue: An integrated architecture for bio-inspired hardware," in *Proceedings of the 5th international conference on Evolvable systems: from biology to hardware*, 2003, pp. 129–140.

[102] J. M. Moreno, Y. Thoma, E. Sanchez, J. Eriksson, J. Iglesias, and A. Villa, "The poetic electronic tissue and its role in the emulation of large-scale biologically inspired spiking neural networks models," *Complexus*, vol. 3, no. 1–3, pp. 32–47, 2006.

[103] A. Upegui, Y. Thoma, and J. M. Moreno, "Bio-Inspired Features Of The Ubichip." PERPLEXUS Project Technical Report, Online: http://www.perplexus.org/project/ results/assets/Deliverable_2.1.pdf, Accessed: August, 2010.

[104] Y. Thoma, G. Tempesti, E. Sanchez, and J.-M. M. Arostegui, "POEtic: an electronic tissue for bio-inspired cellular applications.," *Bio Systems*, vol. 76, no. 1–3, pp. 191–200, 2004.

[105] W. Barker, D. M. Halliday, Y. Thoma, E. Sanchez, G. Tempesti, and A. M. Tyrrell, "Fault Tolerance Using Dynamic Reconfiguration on the POEtic Tissue," *Evolutionary Computation, IEEE Transactions on*, vol. 11, no. 5, pp. 666–684, 2007.

[106] A. Koopman, *Hardware-Friendly Genetic Regulatory Networks in POEtic tissue*. M. Sc. thesis. Institute for Information and Computing Sciences, Utrecht University, 2004.

[107] R. Paricio and J. M. Moreno, "POEtic-cubes: acquisition of new qualia through apperception using a bio-inspired electronic tissue," in *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, 2005, pp. 783–789.

[108] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. M. Moreno, J. Madrenas, and G. Sassatelli, "The perplexus bio-inspired hardware platform: A flexible and modular approach," *International Journal of Knowledge-based and Intelligent Engineering Systems*, vol. 12, no. 3, pp. 201–212, 2008.

[109] E. Sanchez, A. Perez-Uribe, A. Upegui, Y. Thoma, J. M. Moreno, A. Napieralski, A. Villa, G. Sassatelli, H. Volken, and E. Lavarec, "PERPLEXUS: Pervasive Computing Framework for Modeling Complex Virtually-Unbounded Systems," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, 2007, pp. 587–591.

[110] J. M. Moreno and J. Madrenas, "A reconfigurable architecture for emulating large-scale bio-inspired systems," in *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, 2009, pp. 126–133.

[111] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 2000.

[112] Y. Thoma, A. Upegui, A. Perez-Uribe, and E. Sanchez, “Self-replication mechanism by means of self-reconfiguration,” in *Workshop proceedings at the 20th International Conference on Architecture of Computing Systems (ARCS 07)*, 2007, p. 8.

[113] C. Müller-Schloer, “Organic computing: on the feasibility of controlled emergence,” in *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2004, pp. 2–5.

[114] “DFG SPP 1183: Organic Computing.” Online : http://www.organic-computing.de/spp, Accessed: August, 2013.

[115] IBM, “Autonomic Computing: IBM’s Perspective on the State of Information Technology, 2001.” Online : http://researchweb.watson.ibm.com/autonomic/manifesto/autonomic_computing.pdf, Accessed: August, 2010.

[116] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, p. 158.

[117] “D-Wave Two Vesuvius Quantum Computer.” Online: http://www.dwavesys.com/, Accessed: August, 2013.

[118] P. Dittrich, “Chemical Computing,” in *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*, vol. 3566, J.-P. Banatre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Springer Berlin / Heidelberg, ISBN: 978-3-540-27884-9, 2005, pp. 19–32.

[119] L. Kuhnert, K. I. Agladze, and V. I. Krinsky, “Image processing using light-sensitive chemical waves,” *Nature*, vol. 337, pp. 244–247, 1989.

[120] D. Margulies, G. Melman, and A. Shanzer, “Fluorescein as a model molecular calculator with reset capability,” *Nature materials*, vol. 4, no. 10, pp. 768–771, 2005.

[121] D. Bray, “Protein molecules as computational elements in living cells,” *Nature*, vol. 376, no. 6538, pp. 307–312, 1995.

[122] A. Adamatzky, B. D. L. Costello, and T. Asai, *Reaction-Diffusion Computers*. New York, NY, USA: Elsevier Science Inc, ISBN: 978-0-444-52042-5, 2005, p. 348.

[123] V. Balzani, A. Credi, and M. Venturi, “Molecular logic circuits,” *ChemPhysChem*, vol. 4, no. 1, pp. 49–59, 2003.

[124] N. Yachie, K. Sekiyama, J. Sugahara, Y. Ohashi, and M. Tomita, “Alignment-based approach for durable data storage into living organisms,” *Biotechnology progress*, vol. 23, no. 2, pp. 501–505, 2007.

[125] J. Baumgardner, K. Acker, O. Adefuye, S. T. Crowley, W. DeLoache, J. O. Dickson, L. Heard, A. T. Martens, N. Morton, M. Ritter, and others, “Solving a

Hamiltonian Path Problem with a bacterial computer," *Journal of biological engineering*, vol. 3, no. 1, p. 11, 2009.

[126] A. Adamatzky, *From utopian to genuine unconventional computers*. Adamatzky A. and Teuscher C. (Eds.), Luniver Press, ISBN: 978-0955117091, 2006.

[127] L. B. Kish, "Noise-based logic: Binary, multi-valued, or fuzzy, with optional superposition of logic states," *Physics Letters A*, vol. 373, no. 10, pp. 911–918, 2009.

[128] S. L. Harding and J. F. Miller, "Evolution in Materio: Evolving Logic Gates in Liquid Crystal," *Journal of Unconventional Computing*, vol. 3, no. 4, pp. 243–257, 2007.

[129] J. C. Rainwater, D. G. Friend, H. J. M. Hanley, A. H. Harvey, C. D. Holcomb, A. Laesecke, J. W. Magee, and C. Muzny, "Report on Forum 2000: Fluid Properties for New Technologies-Connecting Virtual Design with Physical Reality," *Journal of Chemical & Engineering Data*, vol. 46, no. 5, pp. 1002–1006, 2001.

[130] R. Hiremane, "From Moore's law to Intel innovation—prediction to reality," *Technology@Intel Magazine*, p. 9, 2005.

[131] A. K. Geim and K. S. Novoselov, "The rise of graphene," *Nature materials*, vol. 6, no. 3, pp. 183–191, 2007.

[132] A. D. Rast, X. Jin, F. Galluppi, L. A. Plana, C. Patterson, and S. Furber, "Scalable event-driven native parallel processing: the SpiNNaker neuromimetic system," in *Proceedings of the 7th ACM international conference on Computing frontiers*, 2010, pp. 21–30.

[133] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation," *Solid-State Circuits, IEEE Journal of*, vol. 48, no. 8, pp. 1–11, 2013.

[134] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor," in *Neural Networks, 2008. IEEE International Joint Conference on*, 2008, pp. 2849–2856.

[135] K. J. Dugan, J. S. Reeve, and A. D. Brown, "SpinLink: An interconnection system for the SpiNNaker biologically inspired multi-computer," in *UK Electronics Forum*, 2012, pp. 52–58.

[136] E. Le Martelot, P. J. Bentley, and R. B. Lotto, "A Systemic Computation Platform for the Modelling and Analysis of Processes with Natural Characteristics," in *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2007)*, 2007, pp. 2809–2816.

[137] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The molen media processor: Design and evaluation," in *Proceedings of the International Workshop on Application Specific Processors, WASP 2005*, 2005, pp. 26–33.

[138] J. Becker, K. Brändle, U. Brinkschulte, J. Henkel, W. Karl, T. Köster, M. Wenz, and H. Wörn, "Digital on-demand computing organism for real-time systems," in *Workshop Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS'06)*, 2006, vol. 81, pp. 230–245.

[139] M. Gschwind, "The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor," *International Journal of Parallel Programming*, vol. 35, no. 3, pp. 233–262, 2007.

[140] "The Cell project at IBM Research." Online : http://researchweb.watson.ibm.com/cell/, Accessed: August, 2013.

[141] K. Skaugen, "Petascale to exascale: extending Intel's HPC commitment," in *Keynote in Int. Supercomputing Conference (ISC'10)*, 2010.

[142] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, and others, "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH*, 2008, p. 18.

[143] G. Chrysos, "Knights Corner, Intel's first Many Integrated Core (MIC) Architecture Product," in *Hot Chips: A Symposium on High Performance Chips*, 2012.

[144] "SARC : The Scalable computer ARChitecture Project." Online : http://www.sarc-ip.org, Accessed: August, 2013.

[145] D. Ludovici and G. N. Gaydadjiev, "SARC Power Estimation Methodology," in *Proceedings of the 18th Annual Workshop on Circuits, Systems and Signal Processing. ProRisc 2007*, 2007.

[146] "Systems of Neuromorphic Adaptive Plastic Scalable Electronics." Broad Agency Announcement, Defense Advanced Research Projects Agency, USA. Online: https://www.fbo.gov/utils/view?id=0b62b2149395d4bd8a28dff1b9046944, Accessed: August, 2013.

[147] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[148] AMD, *AMD Fusion^{TM} Family of APUs*. Online : http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, Accessed: August, 2013.

[149] J. von Neumann, *The Computer and the Brain*. New Haven, CT, USA: Yale University Press, ISBN: 9780300007930, 1958, p. 96.

[150] E. Fermi, *Thermodynamics*. New York: Dover Publications, ISBN: 978-0486603612, 1956, p. 160.

[151] E. Le Martelot and P. J. Bentley, "Modelling Biological Processes Naturally using Systemic Computation: Genetic Algorithms, Neural Networks, and Artificial Immune Systems," in *Nature-Inspired Informatics for Intelligent Applications and Knowledge Discovery: Implications in Business, Science and Engineering*, R. Choing, Ed. IGI Global, 2008, pp. 204–241.

[152] E. Le Martelot and P. J. Bentley, “On-Line Systemic Computation Visualisation of Dynamic Complex Systems,” in *Proceedings of the 2009 International Conference on Modeling, Simulation and Visualization Methods (MSV’09)*, 2009, pp. 10–16.

[153] E. Le Martelot and P. J. Bentley, “Metabolic Systemic Computing: Exploiting Innate Immunity within an Artificial Organism for On-line Self-Organisation and Anomaly Detection,” *Journal of Mathematical Modelling and Algorithms*, vol. 8, no. 2, pp. 203–225, Mar. 2009.

[154] E. Le Martelot, P. J. Bentley, and R. B. Lotto, “Crash-Proof Systemic Computing: A Demonstration of Native Fault-Tolerance and Self-Maintenance,” in *Proceedings of the Fourth IASTED International Conference on Advances in Computer Science and Technology (ACST 2008)*, 2008, pp. 49–55.

[155] E. Le Martelot, P. J. Bentley, and R. B. Lotto, “Eating Data is Good for Your Immune System: An Artificial Metabolism for Data Clustering using Systemic Computation,” in *Proceedings of the Seventh International Conference on Artificial Immune Systems (ICARIS 2008)*, 2008, pp. 412–423.

[156] E. Le Martelot, P. J. Bentley, and R. B. Lotto, “Exploiting Natural Asynchrony and Local Knowledge within Systemic Computation to Enable Generic Neural Structures,” in *Natural Computing. Proceedings of 2nd International Workshop on Natural Computing (IWNC 2007)*, 2007, vol. 1, pp. 122–133.

[157] Xilinx, “Spartan-3 FPGA Family Data Sheet DS099 (v3.1).” p. 272, 2013.

[158] C. Sakellariou and P. J. Bentley, “Introducing the FPGA-Based Hardware Architecture of Systemic Computation (HAoS),” in *Mathematical and Engineering Methods in Computer Science,* Lecture Notes in Computer Science (LNCS), vol. 7119, Z. Kotásek, J. Bouda, I. Cerná, L. Sekanina, T. Vojnar, and D. Antoš, Eds. Springer Berlin Heidelberg, 2012, pp. 179–190.

[159] C. Sakellariou and P. J. Bentley, “Describing The FPGA-Based Hardware Architecture of Systemic Computation (HAoS),” *Journal of Computing and Informatics*, vol. 31, no. 3, pp. 485–505, 2012.

[160] C. Sakellariou and P. J. Bentley, “Building a Bio-Inspired Computer: The Hardware Architecture of Systemic Computation (HAoS),” in *Frontiers of Natural Computing Workshop*, York, 2012.

[161] Xilinx, “Virtex-6 Product Brief.” Online: http://www.xilinx.com/publications/prod_mktg/Virtex6_Product_Brief.pdf, Accessed: September, 2013.

[162] Xilinx, “Partial Reconfiguration User Guide,” vol. UG702. 2013.

[163] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. D. George, “Survey of C-based application mapping tools for reconfigurable computing,” in *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD’05)*, 2005.

[164] P. P. Berdychowski and W. M. Zabolotny, "C to VHDL compiler," in *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2010, 77451F*, 2010, vol. 7745.

[165] M. Rouhipour, P. Bentley, and H. Shayani, "Fast Bio-Inspired Computation using a GPU-based Systemic Computer," *Special Issue on Parallel Computing Systems & Applications, in Journal of Parallel Computing, Parallel Architectures and Bioinspired Algorithms*, vol. 36, no. 10–11, pp. 591–617, 2010.

[166] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu, "QP: A heterogeneous multi-accelerator cluster," in *Int. Conf. on High-Performance Cluster Computing*, 2009.

[167] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 115–124.

[168] R. Brennan, M. Manzke, K. O'Conor, J. Dingliana, and C. O'Sullivan, "A scalable and reconfigurable shared memory graphics cluster architecture," in *Proceedings of the 2007 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2007)*, 2007, pp. 284–290.

[169] M. Dutton and D. Keezer, "An architecture for graphics processing in an FPGA (abstract only)," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, p. 283.

[170] R. Bittner and E. Ruf, "Direct GPU/FPGA Communication via PCI Express," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, 2012, pp. 135–139.

[171] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, 2009, pp. 35–42.

[172] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on FPGAS," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 531–534.

[173] K. Locke, "Parameterizable Content-Addressable Memory." Xilinx Application Note XAPP1151, 2011.

[174] L. Shu, K. Song, and Y. Wang, "The Implementation and Research on Pseudo Random Number Generators with FPGA," *Journal of Circuits and Systems*, vol. 3, no. 8, pp. 121–124, 2003.

[175] D. B. Thomas, L. Howes, and W. Luk, "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 63–72.

[176] K. Wold and C. H. Tan, “Analysis and enhancement of random number generator in FPGA based on oscillator rings,” *Int. Journal of Reconfigurable Computing*, vol. 2009, pp. 1–8, Jan. 2009.

[177] M. Dichtl and J. Golic, “High-Speed True Random Number Generation with Logic Gates Only,” in *Cryptographic Hardware and Embedded Systems - CHES 2007*, vol. 4727, P. Paillier and I. Verbauwhede, Eds. Springer Berlin / Heidelberg, 2007, pp. 45–62.

[178] S. Konuma and S. Ichikawa, “Design and Evaluation of Hardware Pseudo-Random Number Generator MT19937,” *IEICE Transactions*, pp. 2876–2879, 2005.

[179] S. M. Qasim, S. A. Abbasi, and B. Almashary, “A review of FPGA-based design methodology and optimization techniques for efficient hardware realization of computation intensive algorithms,” in *Multimedia, Signal Processing and Communication Technologies, 2009. IMPACT ’09. International*, 2009, pp. 313–316.

[180] R. Manohar, “Reconfigurable Asynchronous Logic,” in *Custom Integrated Circuits Conference, 2006. CICC ’06. IEEE*, 2006, pp. 13–20.

[181] D. Shang, F. Xia, and A. Yakovlev, “Asynchronous FPGA architecture with distributed control,” in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 1436–1439.

[182] M. Simlastik and V. Stopjakova, “Automated Synchronous-to-Asynchronous Circuits Conversion: A Survey,” in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, vol. 5349, L. Svensson and J. Monteiro, Eds. Springer Berlin / Heidelberg, 2009, pp. 348–358.

[183] M. Krstic, E. Grass, F. K. Gurkaynak, and P. Vivet, “Globally Asynchronous, Locally Synchronous Circuits: Overview and Outlook,” *Design Test of Computers, IEEE*, vol. 24, no. 5, pp. 430–441, 2007.

[184] S. E. Anderson, “Bit Twiddling Hacks.” Online: http://graphics.stanford.edu/~seander/bithacks.html, Accessed: September, 2013.

[185] H. Lipmaa and S. Moriai, “Efficient algorithms for computing differential properties of addition,” in *Fast Software Encryption*, 2002, pp. 336–350.

[186] J.-P. Deschamps, G. D. Sutter, and E. Cantó, *Guide to fpga implementation of arithmetic functions*. Springer Verlag, ISBN: 9789400729865, 2012, p. 508.

[187] M. J. Flynn and S. S. Oberman, *Advanced computer arithmetic design*. John Wiley & Sons, Inc., ISBN: 9780471412090, 2001, p. 344.

[188] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann, ISBN: 9780123838728, 2011, p. 856.

[189] R. W. Ward and T. C. A. Molteno, *Table of Linear Feedback Shift Registers*. Electronics Group, University of Otago, New Zealand, 2012.

[190] E. Le Martelot, P. J. Bentley, and R. B. Lotto, “A systemic computation platform for the modelling and analysis of processes with natural characteristics,” *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation - GECCO ’07*, p. 2809, 2007.

[191] C. Sakellariou, “The FPGA-Based Hardware Architecture of Systemic Computation.” Online: http://www0.cs.ucl.ac.uk/staff/C.Sakellariou/HAoS/sw/, Accessed: September, 2013.

[192] C. Sakellariou and P. J. Bentley, “Extending the Hardware Architecture of Systemic Computation to a Complete Programming Platform,” in *IEEE International Conference on Evolvable Systems (ICES 2013), part of the IEEE Symposium Series on Computational Intelligence (SSCI 2013)*, Singapore, April 2013.

[193] “MicroBlaze Soft Processor Core.” Online: http://www.xilinx.com/tools/feature/csi/microblaze.htm, Accessed: August, 2013.

[194] “USB Implementers Forum Specifications.” Online: http://www.usb.org/developers/docs/, Accessed: August, 2013.

[195] “PCI Special Interest Group.” Online: http://www.pcisig.com/, Accessed: August, 2013.

[196] R. Bittner, “Bus mastering PCI express in an FPGA,” in *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 273–276.

[197] R. Bittner, “Speedy Bus Mastering PCI Express,” in *22nd International Conference on Field Programmable Logic and Applications*, 2012.

[198] J. Wiltgen and J. Ayer, “Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express® Solutions.” Application Note 1052, Xilinx, 2010.

[199] Altera, “PCI Express High Performance Reference Design.” Application Note AN-456-1.5, 2010.

[200] N. Alachiotis, S. A. Berger, and A. Stamatakis, “Efficient PC-FPGA Communication over Gigabit Ethernet,” in *Computer and Information Technology, IEEE 10th International Conference on*, 2010, pp. 1727–1734.

[201] N. Alachiotis, S. A. Berger, and A. Stamatakis, “A Versatile UDP / IP based PC ↔ FPGA Communication Platform,” in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, 2012, pp. 1–6.

[202] A. Goldhammer and J. Ayer, “Understanding Performance of PCI Express Systems,” *Memory*, vol. 350. Xilinx, pp. 1–18, 2008.

[203] F. A. Jolfaei, N. Mohammadizadeh, M. S. Sadri, and F. FaniSani, “High Speed USB 2.0 Interface for FPGA Based Embedded Systems,” in *Embedded and*

Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on, 2009, pp. 1–6.

[204] S. R. Jonnalagada and V. Krishna, “Designing High-Performance Video Systems in 7 Series FPGAs with the AXI Interconnect.” Xilinx Application Note XAPP741 (v1.1), 2012.

[205] Xilinx, “AXI Interface Based KC705 Embedded Kit MicroBlaze Processor Subsystem Data Sheet.” Datasheet SD669 (v1.1), 2012.

[206] Altera, “SoC FPGA Product Overview.” Advance Information Brief, AIB-01017-1.3, 2012.

[207] Xilinx, “Zynq-7000 Extensible Processing Platform Summary.” User Guide 804 (v1.1), 2011.

[208] Xilinx, “MicroBlaze Processor Reference Guide.” User Guide UG081 (v14.1), 2012.

[209] “Simply RISC S1 Core.” Online:http://www.srisc.com, Accessed: August, 2013.

[210] “Leon Processors.” Online: http://gaisler.com/index.php/products/processors, Accessed: August, 2013.

[211] “OpenRISC 1200 Processor.” Online: http://opencores.org/or1k/FPGA_Development_Boards, Accessed: August, 2013.

[212] S. T. Ser Ngiap, “AEMB 32-bit Microprocessor Core.” Online:http://www.aeste.my/aemb, Accessed: August, 2013.

[213] “OpenFire Processor.” Online: http://openfirefpga.sourceforge.net/, Accessed: August, 2013.

[214] “Altera Nios II processor.” Online: http://www.altera.com/devices/processor/nios2/, Accessed: August, 2013.

[215] “MP32 Processor.” Online: http://www.altera.com/devices/processor/mips/mp32/proc-mp32.html, Accessed: August, 2013.

[216] “Lattice Mico32 Processor.” Online: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx, Accessed: August, 2013.

[217] “Cortex-M1 Processor.” Online: http://www.altera.com/devices/processor/arm/cortex-m1/m-arm-cortex-m1.html, Accessed: August, 2013.

[218] “106Micro RISC Controller Core.” Online: http://www.tensilica.com/uploads/pdf/106Micro_fpga.pdf, Accessed: August, 2013.

[219] "Freescale V1 Coldfire Processor." Online: http://www.ip-extreme.com/corestore/, Accessed: August, 2013.

[220] S. de Pablo, J. Cebrián, L. C. Herrero, and A. B. Rey, "A soft fixed-point Digital Signal Processor applied in Power Electronics," in *FPGAworld Conference 2005*, 2005.

[221] "hyperARM." Online: http://code.google.com/p/arm-cpu-core/, Accessed: August, 2013.

[222] K. Chapman, "PicoBlaze for Spartan-6, Virtex-6 and 7-Series (KCPSM6)." Xilinx User Guide, 2012.

[223] "PacoBlaze." Online: http://bleyer.org/pacoblaze/, Accessed: August, 2013.

[224] "LatticeMico8 Open, Free 8-bit Soft Microcontroller." Online: http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx, Accessed: August, 2013.

[225] Xilinx, "EDK Concepts, Tools and Techniques," UG683(v13.4), 2012.

[226] J. L. Brelet, "An overview of multiple cam designs in virtex family devices." Xilinx Application Note XAPP1151, 1999.

[227] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors," *Micro, IEEE*, vol. 17, no. 5, pp. 64–71, 1997.

[228] Xilinx, "7 Series FPGAs," White Paper WP373, 2012.

[229] Xilinx, "Embedded System Tools Reference Manual." User Guide UG111 (v14.3), 2012.

[230] C. Sakellariou and P. J. Bentley, "Modelling Cancer with the Hardware Architecture of Systemic Computation (HAoS) programming platform," submitted to *International Journal of Bio-Inspired Computation*, *Special Issue on Bio-inspired Hardware*, 2013.

[231] S. Khuri, T. Bäck, and J. Heitkötter, "The zero/one multiple knapsack problem and genetic algorithms," in *Proceedings of the 1994 ACM symposium on Applied computing*, 1994, pp. 188–193.

[232] B. B. H. Korte and J. Vygen, *Combinatorial optimization: Theory and Algorithms*, vol. 21. Springer, ISBN: 9784431100218, 2009, p. 739.

[233] C.-Y. F. Huang and J. E. Ferrell, "Ultrasensitivity in the mitogen-activated protein kinase cascade," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 93, no. 19, pp. 10078–10083, 1996.

[234] A. Phillips, L. Cardelli, and G. Castagna, "A Graphical Representation for Biological Processes in the Stochastic pi-calculus," in *Transactions on Computational Systems Biology VII*, 2006, pp. 123–152.

[235] T. Petricek and J. Skeet, *Real World Functional Programming: With Examples in F# and C#*, 1st ed. Greenwich, CT, USA: Manning Publications Co., ISBN: 9781933988924, 2010.

[236] R. A. Weinberg, *The biology of cancer*, vol. 1. Garland Science, New York, 2013, p. 960.

[237] B. A. A. Weaver and D. W. Cleveland, "Aneuploidy: instigator and inhibitor of tumorigenesis," *Cancer research*, vol. 67, no. 21, pp. 10103–10105, 2007.

[238] A. Araujo, "Modelling Chromosome Missegregation in Tumour Evolution." PhD Thesis, Department of Computer Science, UCL, London, p. 251, 2013.

[239] H. Engelberg-Kulka, S. Amitai, I. Kolodkin-Gal, and R. Hazan, "Bacterial Programmed Cell Death and Multicellular Behavior in Bacteria," *PLoS Genet*, vol. 2, no. 10, p. e135, 2006.

[240] R. W. Vandivier, P. M. Henson, and I. S. Douglas, "Burying the dead*: The impact of failed apoptotic cell removal (efferocytosis) on chronic inflammatory lung disease," *CHEST Journal*, vol. 129, no. 6, pp. 1673–1682, 2006.

[241] H. Lodish, A. Berk, and S. Zipursky, "Overview of the Cell Cycle and Its Control," in *Molecular Cell Biology*, 4th ed., New York: Freeman, W H, ISBN: 9780716740827, 2000.

[242] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 18–25, 2009.

[243] A. Virginia, Y. D. Yankova, and K. Bertels, "An empirical comparison of ANSI-C to VHDL compilers: SPARK, ROCCC and DWARV," in *Annual Workshop on Circuits, Systems and Signal Processing, ProRISC*, 2007, pp. 388–394.

[244] Renesas Inc., "20Mbit QUAD-Search Content Addressable Memory." Datasheet Number: R10PF0001EU0100, Online: http://www.renesas.eu/products/memory/TCAM/, Accessed: August, 2010.

[245] Broadcom, "Ayama^{TM} LA-1 Processor." Part Number: 20512A, Online: http://www.broadcom.com/products/Knowledge-Based-Processors/Layers-2---4/Ayama-20512A, Accessed: August, 2013.

[246] G. Smit, P. Havinga, L. Smit, P. Heysters, and M. Rosien, "Dynamic reconfiguration in mobile systems," *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pp. 171–181, 2002.

[247] G. De Michell and R. K. Gupta, "Hardware/software co-design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349–365, Mar. 1997.

[248] S. W. Moore, P. J. Fox, S. J. T. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 133–140.

[249] R. Pradeep, S. Vinay, S. Burman, and V. Kamakoti, “FPGA based Agile Algorithm-On-Demand Co-Processor,” in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 3*, 2005, pp. 82–83.

[250] R. Akbari and K. Ziarati, “A multilevel evolutionary algorithm for optimizing numerical functions,” *International Journal of Industrial Engineering Computations*, vol. 2, no. 2, pp. 419–430, 2011.

Appendix A. SC Example Test Programs Source Code

Listing A.1. Addition in Multiple Scopes (Test1 Example) Source Code

```
#systemic start

// define the functions
#function ADD       %b10000000000000000000000000000000
#function PRINT     %b01000000010000000000000000000000

// define some useful labels
#label num        %b1000000000000000
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label printnum   %b??????????1?????
#label zero2      %b00000000000000000000000000000000

// and the program begins here:
// declare the scopes
scope0 (%d0 %d0 %d0)
scope1 (%d0 %d0 %d0)
scope2 (%d0 %d0 %d0)
scope3 (%d0 %d0 %d0)

// data systems
data01 (num %d0 %d1000)
data02 (num %d0 %d24)

data11 (num %d0 %d1000)
data12 (num %d0 %d130)
data13 (num %d0 %d25)

data21 (num %d0 %d1000)
data22 (num %d0 %d130)
data23 (num %d0 %d25)
data24 (num %d0 %d32)

data31 (num %d0 %d1000)
data32 (num %d0 %d130)
data33 (num %d0 %d25)
data34 (num %d0 %d32)
data35 (num %d0 %d13)

// context systems
sum ([num zero2 dontcare] ADD(0,0) [num zero2 dontcare])
output  ([num zero2 zero] PRINT(0,0) [num zero2 printnum])

// set up the scopes
#scope scope0
{
        data01
        data02
        sum
        output // should print 1024
}

#scope scope1
{
        data11
        data12
        data13
        sum
        output // should print 1155
}

#scope scope2
{
        data21
        data22
        data23
        data24
        sum
        output // should print 1187
}

#scope scope3
{
        data31
        data32
        data33
        data34
        data35
        sum
        output // should print 1200
 }

#systemic end
```

Listing A.2. Subtraction-escape and then multiply and print (Test2 Example) Source Code

```
#systemic start

// define the functions
#function SUBTRACTe %b01000000000000000000001000000000
#function MULT      %b11000000000000000000000000000000
#function PRINT     %b01000000010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label num1       %b1000000000000000
#label num2       %b0100000000000000
#label num3       %b1100000000000000
#label num4       %b0010000000000000
#label scp        %b1111111111111111
#label zero2      %b00000000000000000000000000000000

// and the program begins here:
main (scp %d0 %d0)

// Scope c1
c1 (scp %d0 %d1)
data1 (num1 %d0 %d10)
data2 (num2 %d0 %d3)
datax1 (num3 %d0 %d110) // dummy - does not match
datax2 (num3 %d0 %d120) // dummy - does not match
datax3 (num3 %d0 %d130) // dummy - does not match
datax4 (num4 %d0 %d140) // dummy - does not match
datax5 (num4 %d0 %d150) // dummy - does not match
minus  ([num1 zero2 dontcare] SUBTRACTe(0,0) [num2 zero2 dontcare])

#scope c1
{
        data1
        data2
        minus  // 10-3=7
        datax1
        datax2
        datax3
        datax4
        datax5
}

// Scope c2
C2 (scp %d0 %d2)
data3 (num1 %d0 %d16)
data4 (num2 %d0 %d4)
datay1 (num3 %d0 %d1010) // dummy - does not match
datay2 (num3 %d0 %d1020) // dummy - does not match
datay3 (num3 %d0 %d1030) // dummy - does not match
datay4 (num4 %d0 %d1040) // dummy - does not match
datay5 (num4 %d0 %d1050) // dummy - does not match
minus  ([num1 zero2 dontcare] SUBTRACTe(0,0) [num2 zero2 dontcare])

#scope c1
{
        data3
        data4
        minus  // 16-4=12
        datay1
        datay2
        datay3
        datay4
        datay5
}
```

```
 // Scope main
 times ([num1 zero2 dontcare] MULT(0,0) [num1 zero2 dontcare]) // 12*7=84
 output  ([num1 zero2 dontcare] PRINT(0,0) [num1 zero2 dontcare])

 #scope main
 {
         c1
         c2
         times
         output
 }

#systemic end
```

Listing A.3. Context Adapting (Test3 Example) Source Code

```
 #systemic start

 // define the functions
 #function SUBTRACT  %b01000000000000000000000000000000
 #function ADD       %b10000000000000000000000000000000
 #function COPY      %b01010000000000000000000000000000

 // define some useful labels
 #label zero       %b0000000000000000
 #label dontcare   %b????????????????
 #label num3       %b1100000000000000
 #label scp        %b1111111111111111
 #label zero2      %b00000000000000000000000000000000

 // and the program begins here:
 main (scp %d0 %d0) // main scope

 // data systems
 datax1 (num3 %d0 %d110)
 datax2 (num3 %d0 %d120)
 datax3 (num3 %d0 %d130)
 datay1 (num3 %d0 %d1010)
 datay2 (num3 %d0 %d1020)
 datay3 (num3 %d0 %d1030)

 // context systems
 minusadapt ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt1 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt2 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt3 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt4 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt5 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 minusadapt6 ([num3 zero2 dontcare] SUBTRACT(0,0) [num3 zero2 dontcare])
 addadapt ([num3 zero2 dontcare] ADD(0,0) [num3 zero2 dontcare])

 // context adapter system: transforms subtraction systems to
 // additions ones by copying the contents of add to sub systems
 killminus  ([minusadapt] COPY(0,0) [addadapt])

 #scope main
 {
         minusadapt
         minusadapt1
         minusadapt2
         minusadapt3
         minusadapt4
         minusadapt5
         minusadapt6
         addadapt
         killminus
         datax1
         datax2
         datax3
         datay1
         datay2
         datay3
 }

#systemic end
```

Appendix B. CPU Subsystem Integration Details

The custom HAoS logic connects to the IC through a point-to-point bidirectional slave interface block, the user IP Interface (IPIF) which in addition to the data and address buses provides a set of standardized control signals (like chip select, chip enable, byte enable and acknowledgements). A simplified block diagram of the HAoS CPU communication link (the CPU INTERFACE of Figure 3.9) is shown in Figure B.1.

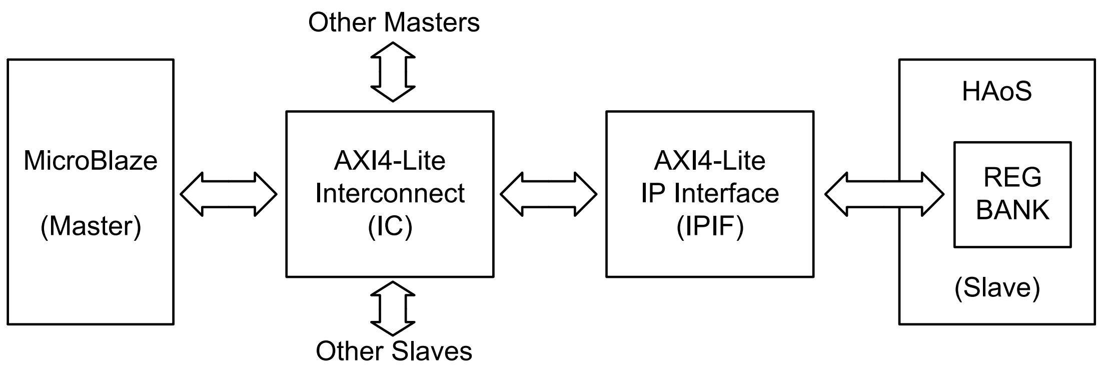


Figure B.1. The AXI4-Lite based HAoS-MicroBlaze communication link

A slight modification was required to the IPIF logic as the Xilinx AXI4-Lite interface natively supports up to only 32 4-byte registers. In order to waive this restriction, the default read/write address decoding logic (the register address would be decoded to give a one-hot 32-bit bus with the set bit at the position of the register to be read/written) and the 32 register-array was replaced by an interface to the HAoS REG BANK providing just the exact access address. This address is then decoded in the REG BANK to give access to any set of the HAoS control registers, depending on the size of the data to be accessed (1-, 2- and 4-byte accesses supported here as the Xilinx AXI4-Lite bus has a width of 4 bytes) and importantly without the requirement of data being aligned on 4-byte words. This results in a slightly increased size of the decoder due to the bigger number of multiplexers needed, but it can enable a more compact usage of the registers since non-word aligned memory accesses of the HAoS register space are now also supported.

Apart from the soft CPU and the communication interface, the hardware platform is completed with some other useful peripherals, shown in Figure B.2. From top to bottom, we have the local Block-RAM based instruction and data memories (64KB), the MicroBlaze processor with its various communication interfaces, the AXI4 interface and its associated bus connecting the processor with the external DDR3 memory (512MB) and the AXI4-Lite interface and its bus to all other peripherals. These are an Ethernet

Controller, on-board switches and push-buttons and LEDs control blocks, the on-board LCD controller, the Flash EEPROM configuration memory controller, the interrupt controller, the Compact Flash card controller, a timer and the UART control block. It is noted that, from the processor point of view, HAoS is just another peripheral in terms of connectivity and accessibility, as it uses a specific address space of the processor memory map.

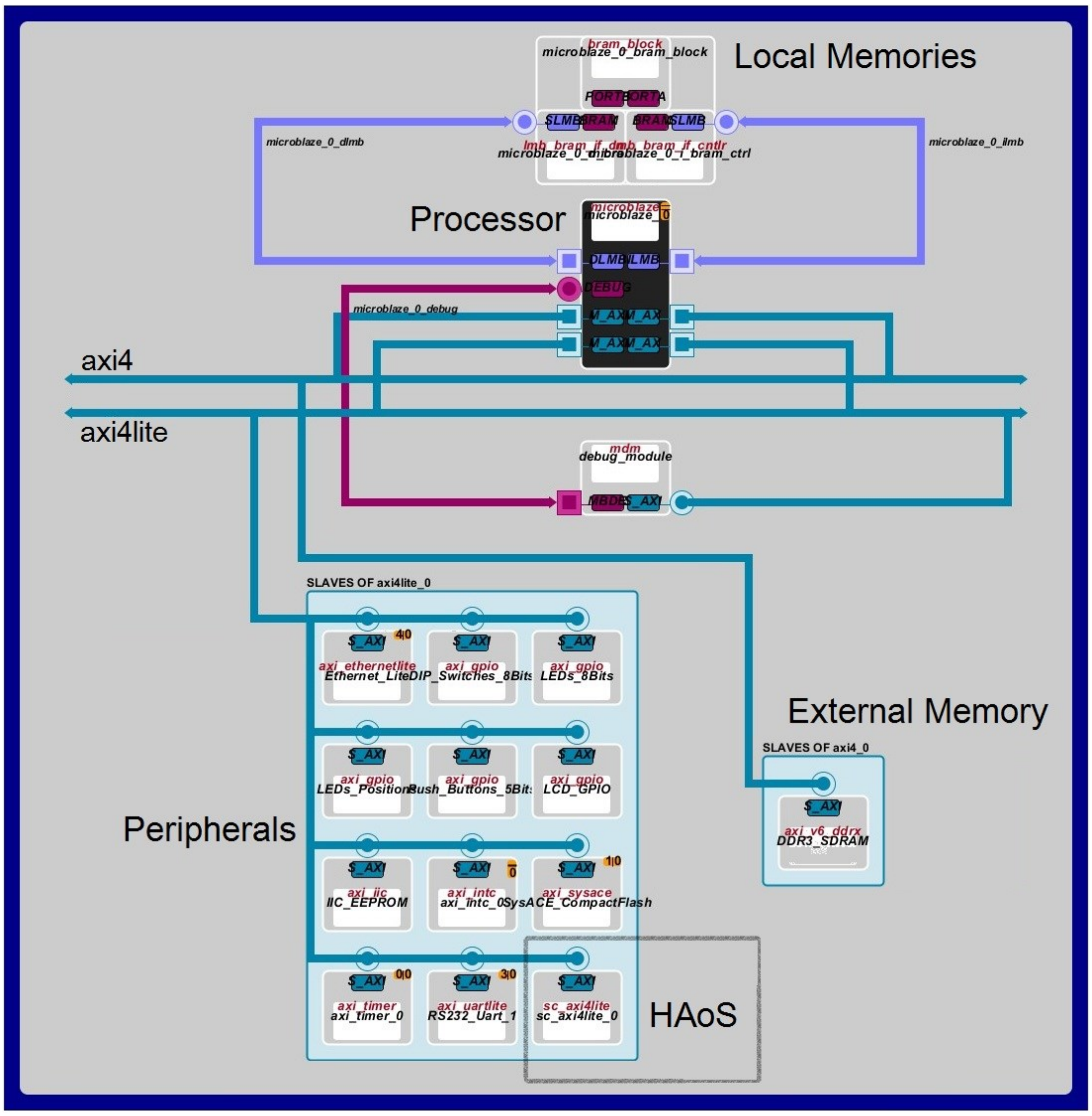


Figure B.2. Top-level On-Chip HAoS Platform Block Diagram

Appendix C. RSL Optimisations Details

The critical path was identified by performing static timing analysis using standard Xilinx tools (PlanAhead). The longest path was implementing valid triplet generation and originated from the TCAM (context template matching result to identify interacting systems), entered the RSL through the input MUX and then it passed from the COUNTONES block to count the number of set bits of the bus. Since the sum of the set bits is used as the divisor of the random number from the LFSR to give the rank of the randomly selected set bit, the critical path then passes from the 16-stage divider to the BITPOSSEL. There the rank is translated to the address of the randomly selected system and the critical path finishes at the memories (binary and ternary RAMs) where this address is used to obtain the interacting system. The critical path, before the optimizations listed below, is illustrated in Figure C.1.

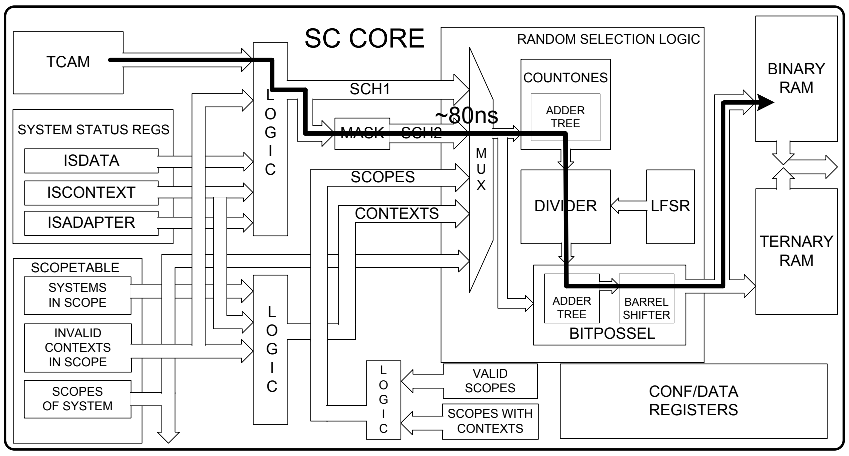


Figure C.1. Critical path of initial HAoS design based on static timing analysis

As mentioned in section 3.6.2, the BITPOSSEL module of the RSL, combines a parallel bit count with a branchless selection method. The parallel bit count is used to provide partial sums which are then appropriately masked and passed through a barrel-shifter to provide the position of a bit with a given rank in the input bus, resulting in a divide-and-conquer technique. As seen in Figure C.2, the COUTNONES and BITPOSSEL modules of the RSL are merged, as the parallel sum-of-bits counter in COUNTONES is reused for the generation of the partial sums during the identification of the position of the selected bit.

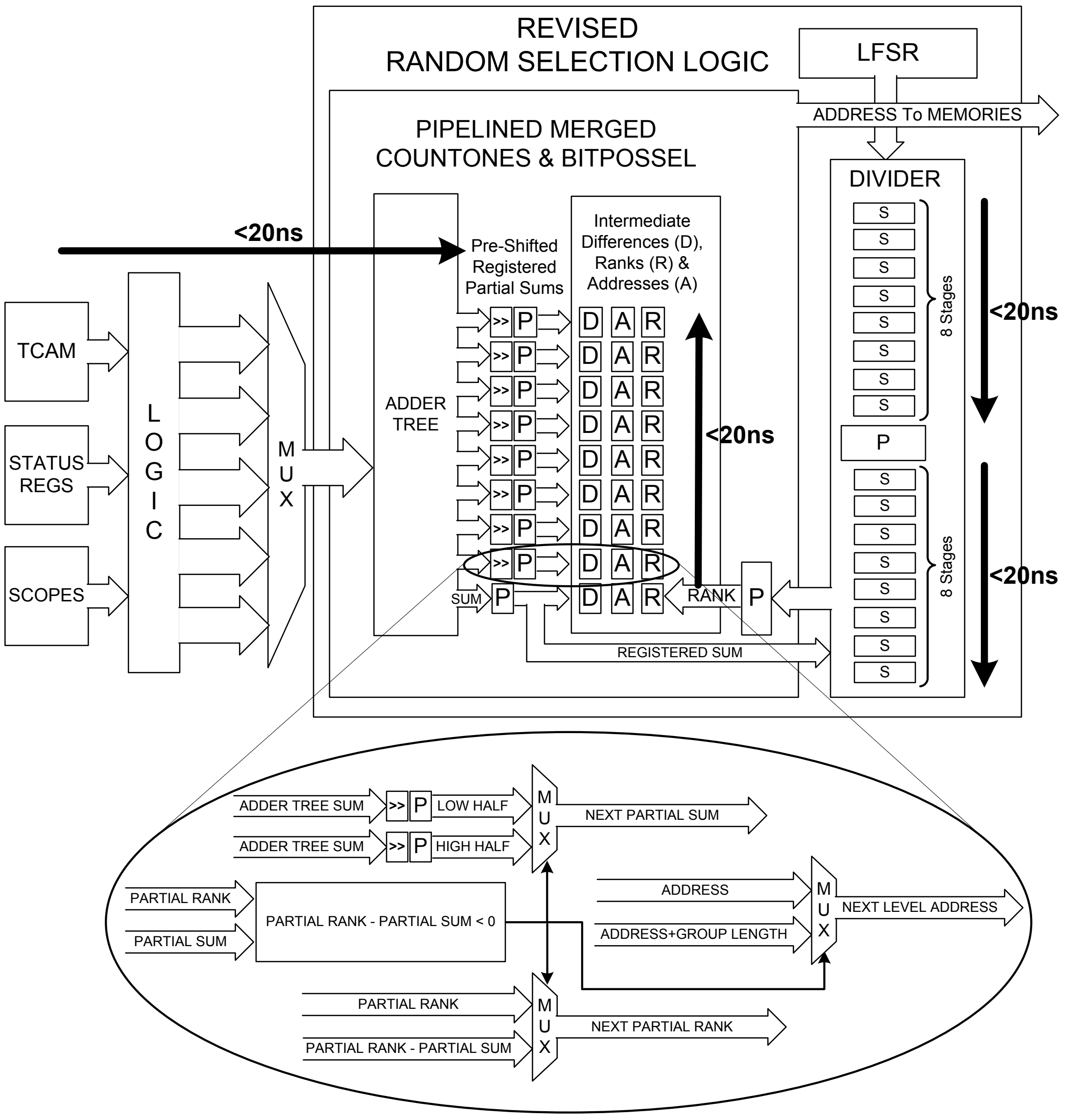


Figure C.2. The Revised RSL module. P stands for pipeline registers. COUNTONES and BITPOSSEL modules have been merged to share the adder tree. The RSL has been carefully pipelined having in mind the trade-off between minimizing latency and excessive resource utilization reducing the critical path from 80ns to 20ns

The length of the barrel shifter is equal to the size of the longest input bus to the RSL which in turn is equal to the number of maximum supported systems. Thus, when this number is increased, the number of logic levels required for the barrel shifter implementation has a considerable impact to the delay along the critical path. For this reason, the conventional barrel shifter is replaced with a parallelized and pipelined version which instead uses an array of multiplexers with registered pre-shifted (by the required pre-calculated number of bits) versions of only the possible subset of shifting combinations of the input buses. Referring back to Figure 3.13 and the discussion of section 3.6.2 regarding finding the position of a set bit given its rank, the BITPOSSEL includes an array of comparators (comparing the intermediate rank to the remaining sum of bits or partial sum of each level, implemented with subtraction) and multiplexers selecting appropriate values for the position (or address) virtual pointer and rank depending on the result of the comparison (middle and bottom part of Figure C.2).

However, for each level, only a subset of shifting combinations of the partial sums is possible (according to the bit-group length of a given level). By replacing the barrel shifter with an array of multiplexers along with using pre-shifted versions of the intermediate sums of the adder tree, we obtain all inputs to the array of units in the BITPOSSEL in parallel (see Figure C.2). Registering those inputs, breaks the critical path after the adder tree and before the comparator tree of the BITPOSSEL, giving the same depth and a latency of less than 20 ns.

Appendix D. Revising the TCAM Design

Each SRL16E primitive is implemented by a LUT and can effectively store one matchable 4-bit data value by driving its output with a set bit only for the corresponding input value (out of the 16 possible permutations). When the SRL16E-based CAM is written, the data input is compared against the output of a counter that cycles through all 16 possible values, and when a match occurs, a '1' is shifted in the SRL16E while zeroes are shifted when the values do not match [173]. Thus, since this design is effectively constructed by a chain of parallel 16-bit shift registers, each write operation, shifting data in, one bit at a time, requires 16 clock cycles. The read operation (which in a CAM is equivalent to a comparison with its input) is performed in a single clock as the data input is partitioned in 4-bit chunks and the chunks are fed as addresses to an array of cascaded SRL16E blocks. Each SRL16E gives a match (a set bit at its output) if its input corresponds to a location that stores a set bit.

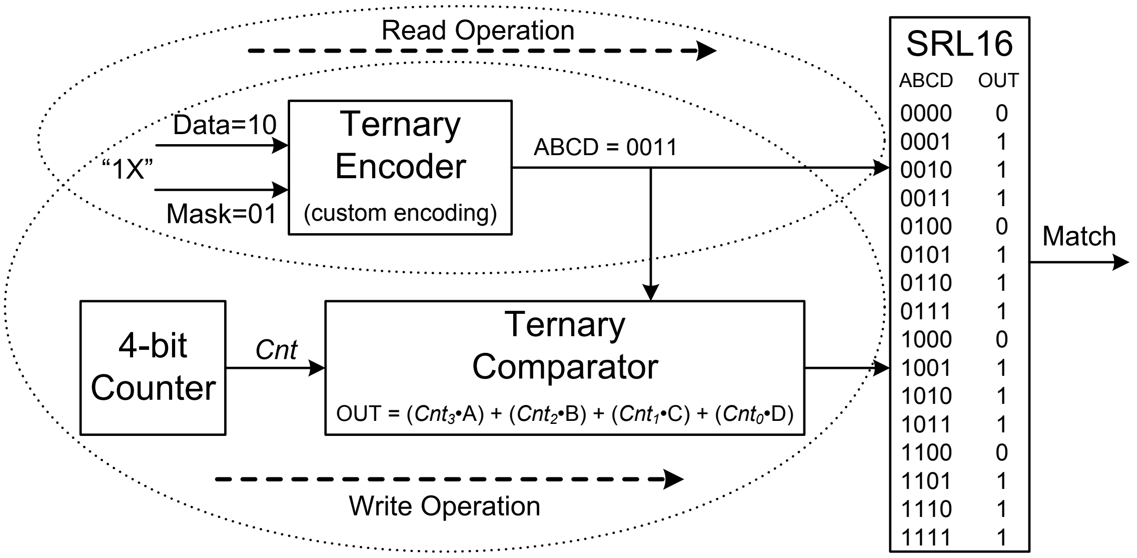


Figure D.1. The SRL16E-based building block of the base HAoS TCAM. The SRL16Es and their associated logic are cascaded using the carry chain between the FPGA slices as a wide AND gate to form wider CAMs. The ternary encoder (using a custom encoding mapping [173]) is used in both read and write operations. Write operation require that all states of a 4-bit counter are "compared" with the encoded value, resulting in a 16 clock cycles write latency

In addition, ternary CAMs support more input combinations as some of the input bits may be "don't care" (X) bits. This implies that the data need to be encoded at the 4-bit input of the SRL16E where data (or binary) bits are combined with mask (or ternary) bits to give a 2-bit ternary-encoded value. Also, the addition of ternary bits also implies that more than one set bits may be stored in an SRL16E as more than one entry may give a match to a ternary value. The SRL16E-based design is summarized in Figure D.1.

The revised design includes a register array which has two partitions, one to hold the data bits and another to store the mask bits. Using registers, write operations are now performed in a single clock cycle. The result of a CAM read operation is available immediately as all comparisons are performed in parallel according to:

Match_bus(i) = and_reduce (data_in xnor binary_regs(i)) or mask_in or ternary_regs(i)

Match_bus is the resulting bus carrying one 1-bit matching flag for each word, *i* is the position of the word, *and_reduce* is a wide AND gate since all the individual bits of the comparison result should be set (a set bit implies a match at that position in the word) in order to have a word match and data_in is the input binary word which is tested for bitwise equality against the binary word stored in position i of the register array (binary_regs(i)). A bit in a word can still be flagged as matched even if the corresponding binary bits do not match if any of the (input - mask_in or locally stored - ternary_regs(i)) ternary bits are set. An extra benefit of using such a parallelized structure for the ternary comparison is that its latency is independent of the depth of the CAM.

Appendix E. Provided HAoS API

Table E.1. Summary of the provided functions (simplified) in the HAoS API

Function Name	Description
scReadX()	Read a HAoS register. X can be 8, 16 or 32 for 1-, 2- or 4-byte read operation
scWriteX()	Write a HAoS register. X can be 8, 16 or 32 for 1-, 2- or 4-byte write operation
readSysXScYArr()	Optimised (uses maximal-length memory access and unrolls the access loop) function to read schema Y of system X where X and Y can be 1 or 2
writeSysXScYArr()	Optimised function to write schema Y of system X where X and Y can be 1 or 2
writeScope()	Write Entry or Entries in HAoS scopetable
loadSumReg()	Write HAoS scopetable sum - Number of systems in a scope
camWrite()	Write Entry in HAoS Ternary Content Addressable Memory
camRead()	Reads System from the RAMs that hold the full contents of the TCAM systems
displayBeats()	Stores Execution Time. May display number of real timer (@100MHz) ticks since it was last reset. Also supports time units
resetRealTimer()	Resets HAoS real time counter
setSwReset()	Asserts HAoS Software Reset
clearSwReset()	Clears HAoS Software Reset
displayStats()	Displays program execution statistics as duration, number of interactions and number of abortions due to context or schemata mismatch
uSleep()	The CPU waits for a user-defined number of us.
lcdPrintString()	Write the lines of the On-board LCD display
printHAoSConstants()	Print all constants defined in the code
encodeSchemata()	Encodes (decompresses) schemata to the appropriate format in order to be written to on-chip memory
readValidTriplet()	Reads the valid triplet, that is the matching systems, transformation function, active scope and context address, and extracts the various user-accessible fields in the triplet driver data structure
writeBackTriplet()	Writes transformed triplet back to HAoS memories
haveTripletWrittenBack()	Transfers transformed triplet back to HAoS registers and lets the hardware do the write-back to the memories.
loadIniSim()	Software Simulation of loading the Initialisation File (for debugging only)
sysAceFReadSim()	Software Simulation of reading the Compact Flash Card
schemataPartToI()	Transforms l bits of binary schemata, starting from bit s, to integer
iToSchemataPart()	Transforms integer into binary with length l and places it in schema starting at bit s

Appendix F. File Types used along HAoS Model Development

Table F.1. Summary of file types used along HAoS model development

File Type	Generated by	Description	Format
.sc	User	HAoS model source code written in the SC language defining transformation functions, systems and scopes	Listing 3.1, Section 3.7
.scp	HAoS Compiler	SC human-readable assembly code corresponding to .sc file	Figure 3.17, Section 3.7
.scb	Post-Compiler Binary Generator	Size optimized binary representation of the .scp file. The exact contents of this file are loaded to HAoS memories	Binary .txt Equivalent, Section 4.3.5
.txt	Binary-to-ASCII Converter	ASCII file, replacing each bit of the .scb file with a '0' or '1' ASCII character. The resulting string can be embedded in the user code, loading the SC program to the HAoS memories directly from MicroBlaze Block RAMs (achieves minimum SC program loading time, convenient for debugging)	Same with .scb but in ASCII, Section 4.3.5, Appendix H
.elf	Xilinx SDK MicroBlaze GNU Compiler & Linker	The HAoS program executable, including the driver and any code if high level functions are used, executed by MicroBlaze	Standardized [229]
.bit	Xilinx ISE Implementation Tools	FPGA configuration bit-string. The reconfigurable logic of the FPGA is programmed according to this file which represents an image of the hardware circuit to be implemented	Proprietary (Xilinx)
.log	HAoS executable	A log storing any text output while the HAoS program runs	N/A

Appendix G. Cancer Models SC Source Code

Listing G.1. Time-Enabled Cancer Model SC Source Code

```
#systemic start

// define the functions
#function KILLCELL      %b10001000010000000000000000000000
#function DIVIDECELL    %b01001000010000000000000000000000
#function ABSORBCELL    %b11001000010000000000000000000000
#function DISCARDCELL   %b00101000010000000000000000000000
#function FERTILIZE     %b10101000010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label zero2      %b00000000000000000000000000000000
#label dontcare2  %b????????????????????????????????

// must embed initial cell population number in tissue
// initial living cells : 100(dec) :
// 001100100(binary) : 001001100(binary-reversed)
#label tissue      %b0010011000001000

#label tissuet     %b????????????1???
#label cellt       %b????????????0???

#label tic_normaltissuet     %b????????????1000
#label tic_surgerytissuet    %b????????????1100
#label tic_maysurgerytissuet %b????????????1?00
#label tic_chemotissuet      %b????????????1010
#label tic_maychemotissuet   %b????????????10?0
#label tic_anytissuet        %b????????????1??0

#label toc_normaltissuet     %b????????????1001
#label toc_surgerytissuet    %b????????????1101
#label toc_maysurgerytissuet %b????????????1?01
#label toc_chemotissuet      %b????????????1011
#label toc_maychemotissuet   %b????????????10?1
#label toc_anytissuet        %b????????????1??1

#label tic_livingcellt %b????????????0100
// initial chromosome values are((1,1),(1,1)):100100100100
#label tic_livingcell  %b1001001001000100
#label tic_parentcellt %b????????????0010
#label tic_parentcell  %b1001001001000010
#label deadcellt       %b????????????0110
#label deadcell        %b1001001001000110
#label nutrientcellt   %b????????????0001
#label nutrientcell    %b1001001001000001
#label toc_livingcellt %b????????????0101
#label toc_livingcell  %b1001001001000101
#label toc_parentcellt %b????????????0011
#label toc_parentcell  %b1001001001000011
#label dividedcellt    %b????????????0111
#label dividedcell     %b1001001001000111

// and the program begins here:
main (%d0 %d0 %d0)
organic_tissue (%d100 %d0 tissue )

[0:99]tissueCells ( zero %d0 tic_livingcell )
[0:299]environmentCells ( zero %d0 nutrientcell )

// account for extra living tissue cells in the end
[0:79]spareEnvironmentCells ( zero %d0 nutrientcell )

// tic phase
[0:2]tic_fertilizer ([dontcare zero2 tic_livingcellt] FERTILIZE(0,0)
[dontcare zero2 tic_maychemotissuet])
tic_divider ([dontcare zero2 tic_parentcellt] DIVIDECELL(0,0)
[dontcare zero2 nutrientcellt])
tic_absorb ([dontcare zero2 dividedcellt] ABSORBCELL(0,0)
[dontcare zero2 tic_maysurgerytissuet])

[0:2]tic_death ([dontcare zero2 tic_livingcellt] KILLCELL(0,0)
[dontcare zero2 tic_maychemotissuet])
tic_discard ([dontcare zero2 deadcellt] DISCARDCELL(0,0) [dontcare zero2 tic_anytissuet])

tic_surgery ([dontcare zero2 tic_livingcellt] KILLCELL(0,0)
[dontcare zero2 tic_surgerytissuet])
```

```
// toc phase
[0:2]toc_fertilizer ([dontcare zero2 toc_livingcellt] FERTILIZE(0,0)
[dontcare zero2 toc_maychemotissuet])
toc_divider ([dontcare zero2 toc_parentcellt] DIVIDECELL(0,0) [dontcare zero2
nutrientcellt])
toc_absorb ([dontcare zero2 dividedcellt] ABSORBCELL(0,0)
[dontcare zero2 toc_maysurgerytissuet])

[0:2]toc_death ([dontcare zero2 toc_livingcellt] KILLCELL(0,0)
[dontcare zero2 toc_maychemotissuet])
toc_discard ([dontcare zero2 deadcellt] DISCARDCELL(0,0) [dontcare zero2 toc_anytissuet])

toc_surgery ([dontcare zero2 toc_livingcellt] KILLCELL(0,0)
[dontcare zero2 toc_surgerytissuet])

// set up the scopes
#scope main
{
        organic_tissue
}

#scope organic_tissue
{
        organic_tissue
        [0:99]tissueCells
        [0:299]environmentCells
        [0:79]spareEnvironmentCells

        [0:2]tic_fertilizer
        tic_divider
        tic_absorb
        [0:2]tic_death
        tic_discard
        tic_surgery

        [0:2]toc_fertilizer
        toc_divider
        toc_absorb
        [0:2]toc_death
        toc_discard
        toc_surgery

}

#systemic end
```

Listing G.2. Timeless Cancer Model SC Source Code

```
#systemic start

#function KILLCELL      %b10001000010000000000000000000000
#function DIVIDECELL    %b01001000010000000000000000000000
#function ABSORBCELL    %b11001000010000000000000000000000
#function DISCARDCELL   %b00101000010000000000000000000000
#function FERTILIZE     %b10101000010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label zero2      %b00000000000000000000000000000000
#label dontcare2  %b????????????????????????????????

// must embed initial cell population number in tissue
// initial living cells : 100(dec) :
// 001100100(binary) : 001001100(binary-reversed)
#label tissue      %b0010011000001000
#label tissuet     %b????????????1???
#label cellt       %b????????????0???

#label normaltissuet      %b????????????100?
#label surgerytissuet     %b????????????110?
#label maysurgerytissuet %b????????????1?0?
#label chemotissuet       %b????????????101?
#label maychemotissuet    %b????????????10??
#label anytissuet         %b????????????1???
```

```
#label livingcellt     %b????????????0100
// initial chromosome values are((1,1),(1,1)):100100100100
#label livingcell      %b1001001001000100
#label parentcellt     %b????????????0010
#label parentcell      %b1001001001000010
#label deadcellt       %b????????????0110
#label deadcell        %b1001001001000110
#label nutrientcellt   %b????????????0001
#label nutrientcell    %b1001001001000001
#label dividedcellt    %b????????????0111
#label dividedcell     %b1001001001000111

// and the program begins here:
main (%d0 %d0 %d0)
organic_tissue (%d100 %d0 tissue )

[0:99]tissueCells ( zero %d0 livingcell )
[0:299]environmentCells ( zero %d0 nutrientcell )

// account for extra living tissue cells in the end
[0:79]spareEnvironmentCells ( zero %d0 nutrientcell )

fertilizer ([dontcare zero2 livingcellt] FERTILIZE(0,0) [dontcare zero2 maychemotissuet])
divider ([dontcare zero2 parentcellt] DIVIDECELL(0,0) [dontcare zero2 nutrientcellt])
absorb ([dontcare zero2 dividedcellt] ABSORBCELL(0,0) [dontcare zero2 maysurgerytissuet])

death ([dontcare zero2 livingcellt] KILLCELL(0,0) [dontcare zero2 maychemotissuet])
[0:1]discard ([dontcare zero2 deadcellt] DISCARDCELL(0,0) [dontcare zero2 anytissuet])

surgery ([dontcare zero2 livingcellt] KILLCELL(0,0) [dontcare zero2 surgerytissuet])

// set up the scopes
#scope main
{
        organic_tissue
}

#scope organic_tissue
{
        organic_tissue
        [0:99]tissueCells
        [0:299]environmentCells
        [0:79]spareEnvironmentCells

        fertilizer
        divider
        absorb
        death
        [0:1]discard
        surgery
}

#systemic end
```

Listing G.3. Approximate-time Cancer Model SC Source Code

```
#systemic start

// define the functions
#function KILLCELL      %b10001000010000000000000000000000
#function DIVIDECELL   %b01001000010000000000000000000000
#function ABSORBCELL    %b11001000010000000000000000000000
#function DISCARDCELL   %b00101000010000000000000000000000
#function FERTILIZE     %b10101000010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label zero2      %b00000000000000000000000000000000
#label dontcare2  %b????????????????????????????????

// must embed initial cell population number in tissue
// initial living cells : 100(dec) : 00110010(binary) : 001001100(binary-reversed)
#label tissue      %b0010011000001000
#label tissuet     %b????????????1???
#label cellt       %b????????????0???

#label normaltissuet     %b????????????100?
```

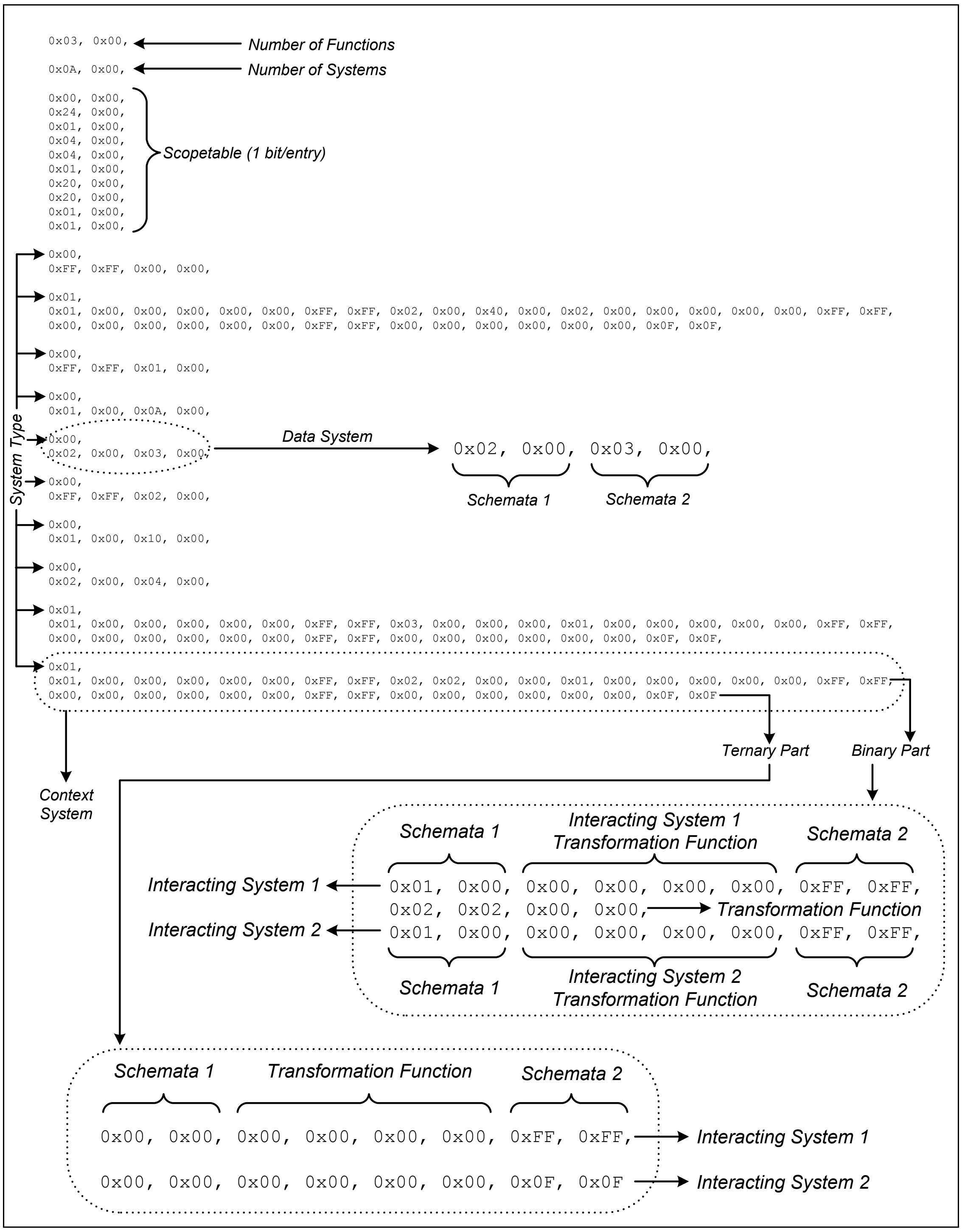
```
#label surgerytissuet    %b????????????110?
#label maysurgerytissuet %b????????????1?0?
#label chemotissuet      %b????????????101?
#label maychemotissuet   %b????????????10??
#label anytissuet        %b????????????1???

#label livingcellt     %b????????????0100
// initial chromosome values are((1,1),(1,1)):100100100100
#label livingcell      %b1001001001000100
#label parentcellt     %b????????????0010
#label parentcell      %b1001001001000010
#label deadcellt       %b????????????0110
#label deadcell        %b1001001001000110
#label nutrientcellt   %b????????????0001
#label nutrientcell    %b1001001001000001
#label dividedcellt    %b????????????0111
#label dividedcell     %b1001001001000111

// and the program begins here:
main (%d0 %d0 %d0)
organic_tissue (%d100 %d0 tissue )

[0:99]tissueCells ( zero %d0 livingcell )
[0:299]environmentCells ( zero %d0 nutrientcell )

// account for extra living tissue cells in the end
[0:79]spareEnvironmentCells ( zero %d0 nutrientcell )

[0:2]fertilizer ([dontcare zero2 livingcellt] FERTILIZE(0,0) [dontcare zero2
maychemotissuet])
divider ([dontcare zero2 parentcellt] DIVIDECELL(0,0) [dontcare zero2 nutrientcellt])
absorb ([dontcare zero2 dividedcellt] ABSORBCELL(0,0) [dontcare zero2
maysurgerytissuet])

[0:2]death ([dontcare zero2 livingcellt] KILLCELL(0,0) [dontcare zero2 maychemotissuet])
discard ([dontcare zero2 deadcellt] DISCARDCELL(0,0) [dontcare zero2 anytissuet])

surgery ([dontcare zero2 livingcellt] KILLCELL(0,0) [dontcare zero2 surgerytissuet])

// set up the scopes
#scope main
{
        organic_tissue
}

#scope organic_tissue
{
        organic_tissue
        [0:99]tissueCells
        [0:299]environmentCells
        [0:79]spareEnvironmentCells

        [0:2]fertilizer
        divider
        absorb
        [0:2]death
        discard
        surgery
}

#systemic end
```

Listing G.4. Optimized Approximate-time Cancer Model SC Source Code

```
#systemic start

// define the functions
#function KILLCELL      %b10001000010000000000000000000000
#function DIVIDECELL    %b01001000010000000000000000000000
#function ABSORBCELL    %b11001000010000000000000000000000
#function DISCARDCELL   %b00101000010000000000000000000000
#function FERTILIZE     %b10101000010000000000000000000000

// define some useful labels
#label zero       %b0000000000000000
#label dontcare   %b????????????????
#label zero2      %b00000000000000000000000000000000
#label dontcare2  %b????????????????????????????????

// must embed initial cell population number in tissue
// initial living cells : 100(dec) :
// 001100100(binary) : 001001100(binary-reversed)
#label tissue      %b0010011000001000
#label tissuet     %b????????????1???
#label cellt       %b????????????0???

#label normaltissuet        %b????????????100?
#label surgerytissuet       %b????????????110?
#label maysurgerytissuet    %b????????????1?0?
#label chemotissuet         %b????????????101?
#label noendmaychemotissuet %b????????????10?0
#label noendanytissuet      %b????????????1??0
#label anytissuet           %b????????????1???

#label livingcellt     %b????????????0100
// initial chromosome values are((1,1),(1,1)):100100100100
#label livingcell      %b1001001001000100
#label parentcellt     %b????????????0010
#label parentcell      %b1001001001000010
#label deadcellt       %b????????????0110
#label deadcell        %b1001001001000110
#label nutrientcellt   %b????????????0001
#label nutrientcell    %b1001001001000001
#label dividedcellt    %b????????????0111
#label dividedcell     %b1001001001000111

// and the program begins here:
main (%d0 %d0 %d0)
organic_tissue (%d100 %d0 tissue )

[0:99]tissueCells ( zero %d0 livingcell )
[0:299]environmentCells ( zero %d0 nutrientcell )

// account for extra living tissue cells in the end
[0:79]spareEnvironmentCells ( zero %d0 nutrientcell )

fertilizer ([dontcare zero2 livingcellt] FERTILIZE(0,0) [dontcare zero2
noendmaychemotissuet])
divider ([dontcare zero2 parentcellt] DIVIDECELL(0,0) [dontcare zero2 nutrientcellt])
absorb ([dontcare zero2 dividedcellt] ABSORBCELL(0,0) [dontcare zero2 maysurgerytissuet])

death ([dontcare zero2 livingcellt] KILLCELL(0,0) [dontcare zero2 noendanytissuet])
discard ([dontcare zero2 deadcellt] DISCARDCELL(0,0) [dontcare zero2 anytissuet])

// set up the scopes
#scope main
{
        organic_tissue
}

#scope organic_tissue
{
        organic_tissue
        [0:99]tissueCells
        [0:299]environmentCells
        [0:79]spareEnvironmentCells

        fertilizer
        death
}

#systemic end
```

Appendix H. HAoS Binary-To-ASCII Conversion Resulting Text File Format

Figure H.1. Annotated HAoS ASCII program example, corresponding to the SC program of section 3.7. The exact representation of the binary file is written in ASCII, separated in bytes (hex form)



```
0x03, 0x00,
0x0A, 0x00,
0x00, 0x00,
0x24, 0x00,
0x01, 0x00,
0x04, 0x00,
0x04, 0x00,
0x01, 0x00,
0x20, 0x00,
0x20, 0x00,
0x01, 0x00,
0x01, 0x00,

0x00,
0xFF, 0xFF, 0x00, 0x00,

0x01,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x02, 0x00, 0x40, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x0F,

0x00,
0xFF, 0xFF, 0x01, 0x00,

0x00,
0x01, 0x00, 0x0A, 0x00,

0x00,
0x02, 0x00, 0x03, 0x00,

0x00,
0xFF, 0xFF, 0x02, 0x00,

0x00,
0x01, 0x00, 0x10, 0x00,

0x00,
0x02, 0x00, 0x04, 0x00,

0x01,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x03, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x0F,

0x01,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x02, 0x02, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x0F
```

```
0x02, 0x00, 0x03, 0x00,
```

```
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x02, 0x02, 0x00, 0x00,
0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
```

```
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0F, 0x0F
```