DISTRIBUTED PROCESSING, RECONFIGURABLE PROCESSES AND ACTIVE NETWORK

YANFENG PENG

Doctor of Philosophy

ASTON UNIVERSITY

September 2003

Aston University

DISTRIBUTED PROCESSING, RECONFIGURABLE PROCESSES AND ACTIVE NETWORK

Yanfeng Peng

DOCTOR OF PHILOSOPHY

2003

SUMMARY

The fast spread of the Internet and the increasing demands of the service are leading to radical changes in the structure and management of underlying telecommunications systems. Active networks (ANs) offer the ability to program the network on a per-router, per-user, or even per-packet basis, thus promise greater flexibility than current networks.

To make this new network paradigm of active network being widely accepted, a lot of issues need to be solved. Management of the active network is one of the challenges. This thesis investigates an adaptive management solution based on genetic algorithm (GA). The solution uses a distributed GA inspired by bacterium on the active nodes within an active network, to provide adaptive management for the network, especially the service provision problems associated with future network. The thesis also reviews the concepts, theories and technologies associated with the management solution.

By exploring the implementation of these active nodes in hardware, this thesis demonstrates the possibility of implementing a GA based adaptive management in the real network that being used today. The concurrent programming language, Handel-C, is used for the description of the design system and a re-configurable computer platform based on a FPGA process element is used for the hardware implementation.

The experiment results demonstrate both the availability of the hardware implementation and the efficiency of the proposed management solution.

Key words: Active network (AN), Genetic algorithm (GA), Handel-C, Re-configurable computing, FPGA.

ACKNOWLEDGEMENTS

I would like to acknowledge the following people for their help to my research work and this Thesis:

Dr.D.J.Holding for supervising my research work and guiding the development of this Thesis

Professor K.J.Blow for supervising my research work, giving the good suggestions and also the discussions.

Professor Ian Marshall (BT Lab) for good suggestion and discussion of the research work

Dr. G.F. Carpenter for making suggestions for improvements in my research work.

Dr. Qiang Gao for making good suggestions for this Thesis.

Dr. XiaoHong Peng for making good suggestions for this Thesis.

Mr. P.R.Trevis for installing and maintaining the machine I used to do my research work and the experiment.

Mr. P.J.Miller for installing and maintaining the machine I used to do my research work and the experiment.

Mr. Milto Mylonas for discussion of the experiment methods

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Introduction to this Thesis

This thesis investigates and implements a system model of active network with active nodes being the network element, which employed a bacterium inspired genetic algorithm (GA) to provide adaptive network management. This “Bacterium Model” was described using a concurrent program language Handel-C and implemented on a FPGA based re-configurable platform. The resulting network performance shows that this bacterial type network management algorithm might be a suitable approach to creating a stable network of autonomous nodes. The hardware implementation of the system model demonstrated the feasibility of employing this adaptive GA based solution for network management in a real network. This Chapter is a basic introduction to the contents of the thesis.

1.2 Background

The telecommunication industry is cautiously navigating its way through a maze with technological crossroads that will ultimately determine what kind of networks the world will use in the 21st century [1]. A most important question is how intelligent the future data networks should be. The requirement for the future network is that it should be smarter and more active, more open and less complex than the present day Internet.

With the rapid Internet proliferation and prevalence, the use of networks and the behaviours of network users are changing and diverse [2]. This brings a great challenge to the traditional network. Several problems with today’s networks have been identified such as the difficulty of integrating new technologies and standards into the shared network infrastructure, poor performance due to redundant operations at several protocol layers, and difficulty of accommodating new services with the existing architectural model [3]. These problems will impede the evolution of network communication technologies and prevent extension of the Internet services without the introducing of new network paradigm [2].

Active networks (ANs) have been shown to address these problems of the traditional networks [4][5][6]. The basic idea is that the network nodes have the ability to perform some computation on the received packets. Thus, applications can inject specific packets into the network that have code (instructions) associated with them and the code is then executed at network elements such as the routers, or switches, as the packet propagates through the network [7]. The aim is to enable users to have access to the services they require (custom services), whilst avoiding any requirement for operators and providers to manage large numbers of services [8].

Active networks can fulfil rapid evolution of network technologies and allow the introduction of service applications without the lengthy process on standardizing the packet format and protocols [2]. New services can be introduced rapidly as active services are based on programs supplied by the users of the services. Thus active network can solve the existing problems in traditional networks.

While suggesting attractive benefits, active networking also poses serious challenges that must be overcome to gain widespread acceptance [9]. One of these challenges is the management of active network. A future network providing active services will be unbounded in both scale and function and the network will be constantly expanded, upgraded, and re-dimensioned [8]. A large range of services will be developed and evolved at an unprecedented rate. It is impossible to predict accurately what the user defined services will look like in the future. However, it is clear that the traditional control and management methods based on obtaining the 'global state' of the system will not be able to cope with the situation due to the massive scale and the changing complexity of the network.

In order to fully realise the intended flexibility of an active network, it will be necessary to combine active services with a highly automated management and control system [8]. Many solutions for active network management have been proposed in recent years [9][10][11][12][13]. A management solution based on the concept of VAN (Virtual Active Network) was discussed in [9][10]. It provides a management framework for active network that allows customers to deploy and manage their own active services in a provider domain. The management solution proposed in [13] is based on the mobile agent technology. Mobile software agents provide a decentralized approach to network management issues because of the distributive nature of the agents themselves. The agents can be defined as autonomous software components operating within a network. Both of these solutions need some architectural support for the management tasks [14].

This thesis applied a novel solution that uses a distributed GA on the active nodes within an active network to provide adaptive management for the network. It was proposed originally in [11] and developed in [12] by Ian Marshall and Chris Roadknight. The solution makes each node within the network responsible for its own behaviour. The whole network is thus modelled as a community of cellular automata. The GA used in this solution is inspired by the mechanisms that bacterium use to transfer and share genetic material. A system model based on this solution was developed and incorporated in a software simulation of a network comprising a number of nodes connected on a rectangular grid. The simulation results showed that this bacterial type network management algorithm might be a suitable approach to creating a stable network of autonomous nodes [11]. Based on this, the work presented in this thesis involves the analysis, design and hardware implementation of an active network of 'bacterium' type active nodes that use GA to implement an adaptive management.

A concurrent programming language – Handel-C [15] was used to describe the prototype system and a re-configurable computing platform comprising a Celoxica RC100 FPGA development board was used to test the hardware design. Two network models were implemented, one for four nodes system and one for nine nodes system. As the design is scalable, the network model can be extended to form a multi-device network with more nodes within it.

1.3 Theories, Concepts and Technologies used in this Thesis

The research work of studying and implementing an active network with adaptive management introduced in this thesis involves some important concepts, theories and technologies such as evolutionary computation, re-configurable computing, cellular computing, concurrent programming and parallel processing. The use of a GA in the management solution relates to evolutionary computation and the use of re-configurable processor such as FPGA for the hardware implementation relates to re-configurable computing. Each node performing cellular computing makes the whole system a community of cellular automata. The concurrent programming language Handel-C is used to describe the system, and the notion of parallel processing is reflected in both of the properties of Handel-C as a software entity and in the concurrent digital electronic FPGA-based design that is synthesised directly from Handel-C.

Evolutionary Computation

The field of evolutionary computation is one of the fastest growing areas of computer science and engineering [16]. It is the study of computational systems that use ideas and get inspiration from natural evolution and adaptation [17]. Evolutionary computation mimics the processes of biological evolution with its ideas of natural selection and survival of the fittest to provide effective solutions for optimisation problems.

Evolution was normally described as a two-step iterative process, consisting of random variation followed by selection. The algorithmic approach begins by selecting an initial set of contending solutions for a particular problem. These "parent" solutions then generate "offspring" by a pre-selected means of random variation. The resultant solutions are evaluated for their effectiveness and undergo selection.

The first approach to evolutionary computation was the GA developed by John H. Holland [18] in the 1960's. It works by creating many random "solutions" to the problem at hand. These solutions can be coded as binary-valued strings. The string is analogous to a 'chromosome', like the real-life, biological equivalents. Each of the solutions will be evaluated using a fitness function. The chromosomes encoding the better solutions are broken apart and recombined through the use of genetic operators such as cross-over, mutation, and recombination to form new solutions (i.e., offspring). These new solutions are generally better than the previous generation (parents). This process will be repeated until an acceptable solution is found within specific time constraints.

The system studied in this thesis uses a bacterium inspired GA which involves mutation and migration processes to implement an adaptive management for the future network and thus performs an evolutionary computation.

Re-Configurable Computing

The re-configurable computing paradigm [19] is a hybrid of two traditional computing technologies, application-specific integrated circuits (ASICs) and general-purpose programmable processors such as microprocessors or DSPs. It arises from the fundamental trade-off between flexibility and performance. ASICs have the advantages of low power dissipation and high clock speeds, thus provide good performance compared with general purpose processor, but have the disadvantage of lacking flexibility because they can only perform the specific tasks for which they were designed. On the other hand, the general-purpose programmable processors have much more flexibility than ASIC because they can be programmed to implement any arbitrary computation but with lower performance levels.

Re-configurable computing aims to overcome this traditional trade-off between flexibility and performance and achieve both the high performance of ASICs and the flexibility of general-purpose processors. Re-configure means the hardware of the re-configurable computing is not static but adapted to each individual application or processing task.

The devices for implementing re-configurable computing are FPGAs which can be easily configured using software to implement a wide variety of processing functions. These devices were introduced in the mid 1980s and were placed on the market at the high-end of programmable logic devices. The new generations of FPGAs have the capacities of up to millions of gates and the technologies are still advancing in a high rate [20].

The system studied in this thesis was implemented in re-configurable FPGA-based hardware. The device that was used in the system is Xilinx Virtex 2000E series FPGA with up to 2 million system gates.

Cellular Computing

Cellular computing [21] is a computational philosophy that is different from conventional computational architecture. It is a natural information processing paradigm, capable of modelling various biological, physical and social phenomena, as well as other kinds of complex adaptive systems [22]. This new computational paradigm provide means for doing computation more efficiently - in terms of speed, cost, power dissipation, information storage, and solution quality.

Moshe Sipper concludes in [21] that cellular computing consists of three principles: simplicity, vast parallelism, and locality. Simplicity means the processor used as the fundamental unit of cellular computing, the cell, is very simple. Vast parallelism means cellular computing involves parallelism on a very large scale and locality means its local connectivity pattern between cells. This has an implication that each cell is only responsible for its own behaviour.

Cellular automata (CA) [21][23][24][25] are quintessential example of cellular computing and also the first to appear on the scene. The system studied in this thesis make each active node within the network only responsible for its own behaviour and thus being modelled as a community of cellular automata.

Concurrent Programming and Parallel Processing

Concurrent programming languages are designed to support concurrent applications in which many parts of a system operate independently and interact [26]. Much of the programming world involves concurrent applications. Examples include operating systems, real-time systems, and simulation [27].

A sequential program specifies sequential execution of a list of statements; its execution is called a process. In contrast, a concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes [28] and the multiple threads of control allows it to perform multiple computations in parallel and to control multiple external activities that occur at the same time [29]. Concurrent programming is then characterized by programming with more than one process.

The strength and weaknesses of a selection of concurrent languages including occam, Ada, Java, and Handel-C etc. are introduced in [30]. The system studied in this thesis use Handel-C as the design language as this language, developed by the Oxford Hardware Compilation Research Group, was designed for compiling programs directly into hardware, which is especially implemented on FPGA. Hardware compilation using Handel-C simplifies the design process, since it provides the convenience of a C-Like high-level programming language to generate the hardware as well [31]. Based on the C language, it is added with the new features, especially that for parallelism which is based on the CSP (Communicating Sequential Processes) [32] model using channels to communicate between processes.

Handel-C was designed primarily for use by a programmer and is designed such that the complex design decisions made at the architectural/hardware level by the hardware engineer do not have to be worried about by the programmer [30]. This is important because by analysing the requirement and determining the major concurrent processes, the designer can make high-level architectural and hardware decisions. The concurrency can be matched to the application or algorithm, and will be translated into structural concurrency in hardware without worrying about any implementation details.

Another major feature of Handel-C is its synchronous semantics. It allows the designer to reason about the execution of parallel programs over time, clock cycle by clock cycle, and thereby support optimisations which are essential when producing efficient hardware implementations.

The Handel-C tools enable a designer to target directly FPGAs in a similar fashion to classical microprocessor cross-compiler development tools, without recourse to a Hardware Description Language (HDL). Thereby allowing even the software engineer to directly realise the raw real-time processing capability of the FPGA.

1.4 Design and Simulation of 'Bacterium-Type' System Model

The software simulation of a system model based on the 'bacterium-type' active network management shows that it might be a suitable approach to create a stable network of autonomous nodes [11]. This thesis presents the work of a hardware implementation of the same model. The first step towards hardware implementation is the design of the system using a description language.

The initial work uses VHDL to describe a bacterium typed system that includes only one node, i.e., the design only implemented one node's function and thus no connection between adjacent nodes were implemented. This is because when the initial work was undertaken the only available device in the lab was a Xilinx XC4005XL FPGA at which the system is aimed to implement. This device was shown to be too small to implement a system with more than one node.

Even though, the simulation result of the one node system described using VHDL still shows the node's basic functions can be implemented. However, as it is a one-node network system, the management algorithm involving the exchanging and interaction of information between nodes cannot be implemented.

Compared with VHDL, the high-level concurrent program language of Handel-C is more flexible and easier to design a complex system at a higher abstract level without the burden of considering the low level structure. Further, the introduction of the Celoxica DK1 Handel-C design environment and the Celoxica RC1000 development system equipped with a Xilinx Virtex 2000 FPGA provides both a modern concurrent software design environment and a solution to the implementation problem. The Handel-C language was then used as the description language instead of VHDL. The Handel-C design implemented a system that includes four nodes connected on a rectangular grid. This model was later extended to form a nine-node model.

The Handel-C design takes advantage of the features of this novel language to obtain the best system performance. The whole system is designed as a hierarchical structure in which the nodes are simply parallel processes that communicate with each other through defined channels, and within each node, the node's separate tasks also comprise a set of communicating parallel processes. This structure maximally takes the advantage of the parallelism feature of Handel-C.

The design was then compiled for simulation. The Handel-C source code was modified and debugged to ensure that the simulation results match the requirements of the system design. The Celoxica DK1 design suite provide a software development environment that includes a GUI (Graphical User Interface) for integrated project management, code editing and source level symbolic debugging [33].

The simulation results show that the bacterium-typed GA (with mutation and migration processes) can be used for the adaptive network management and that the algorithm can be implemented successfully.

1.5 Implementation of "Bacterium Model" in Hardware

Hardware implementation of the studied system model was implemented as an FPGA based system. FPGA was used as it delivers the performance and efficiency of ASICs with the flexibility and short development cycles of PLDs (Programmable Logic Device). Field programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. This user programmability gives the user access to complex integrated designs without the high engineering costs associated with ASICs. In a word, the device combines the performance of the hardware speed and the software flexibility.

While the re-configurable logic lets the user dynamically alter hardware in real time, it blurs the boundary between hardware and software. Further, as an FPGA can be configured to have a parallel architecture, this means the algorithm targeted at an FPGA can exploit parallelism [34].

The whole procedure from source code development to the hardware implementation is illustrated in figure1.1. The source code is the Handel-C description. The Handel-C compiler provided by DK1 Design Suite produce EDIF output from the source code. The EDIF net-list is then mapped into a configuration bit file that can be used to program or configure the FPGA. A re-configurable computer platform from Celoxica -- RC1000 development board, is used to test the design. It is a standand PCI bus card equipped with a Xilinx Virtex family FPGA with up to 2 million system gates. It has 8Mb of SRAM directly connected to the FPGA in four 32-bit wide memory banks.

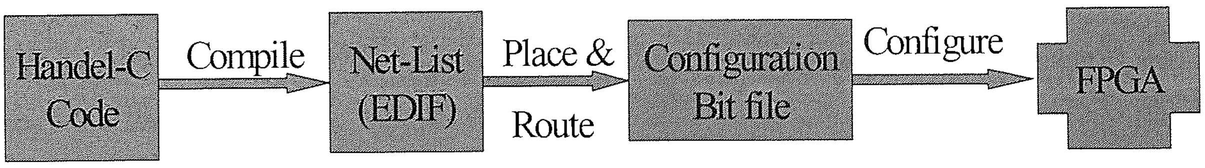


Figure 1.1 From code design to FPGA configuration

Specifying the EDIF output from the compiler produces a net-list that can be used as input to the "Place and Route" tools. As a result of the place and route process, a bit file is created. The bit file can be downloaded to the FPGA using a host program. The host program runs entirely on the host computer, and interacts with the RC1000 board. The four banks of SRAM on the board can be used for communicating between host and the board over the PCI bridge. Data can be sent directly between the off-chip RAM on the board and the hosts' memory without involving the CPU. This is called DMA (Direct Memory Access). FPGA macros are included in the Handel-C program to allow control to pass between the host and the board. There are also macros to access the off-chip RAM on the RC1000 board.

1.6 Experimental Results and Analysis

The experimental results were presented and analysed to demonstrate both the effectiveness and efficiency of the management solution. The QoS measurements that measure the average age of all requests on the network were introduced for both the four nodes and the nine nodes system implementation. The QoS was measured as the load on the network was increased in a series of significant steps. The results demonstrate both the stable behaviour of the system and the efficiency of the management solution of the "Bacterium Model".

The results in terms of the QoS measurements were compared not only between the four nodes and the nine nodes implementation, but also between the previous software simulation and the hardware implementation. This thesis explains why there are differences between them and how these differences are caused.

1.7 Layout of the Thesis

This thesis includes seven chapters. Chapter 2 introduces the background of the work. It explains why the traditional network is now becoming an obstacle for the quick development of the network and cannot meet the various requirements of the users, and a new network paradigm has to be developed [3]. The active network is proved to be the paradigm of future network by many researchers working in this area. Around the management issues for active networks, some solutions were proposed in recent years [8][9][10][13]. GAs were shown to offer a robust approach to evolving effective adaptive control solutions. Therefore a management solution that is based on a distributed GA was studied in this thesis. This solution provides an adaptive control and management for the future network. Chapter 2 introduced the solution in detail and give the software simulation result of the developed system based on this solution. This chapter also introduce a new algorithm and present the simulation result based on this new algorithm. A comparison was made between the two algorithms.

Chapter 3 presents the important concepts, theories and technologies relevant to the developed system. These include Evolutionary Computation, Re-configurable Computing, Cellular Computing, Concurrent Programming and Parallel Processing.

Chapter 4 presents the design and simulation of the developed system. Both VHDL and Handel-C description are introduced. The chapter explains how the system was designed and a comparison between VHDL and Handel-C language is presented. This leads naturally to the decision to develop and implement a scalable network model using Handel-C.

Chapter 5 presents hardware implementation of active network models comprising 4 and 9 nodes that can be implemented within a single FPGA. This chapter briefly introduce the design environment and experiment platform that include the Celoxica DK Design Suite and the re-configurable hardware development platform, Celoxica RC1000. It also contains the introduction of the host program development and the information for hardware implementation.

Chapter 6 presents experiment used to determine the behaviour and performance of the FPGA-based hardware implementation of the experimental active network. The results are analysed to demonstrate the performance of the hardware-implemented GA, and to determine the stability and QoS of the network.

Chapter 7 makes the conclusion, summarises the contributions in this thesis and makes suggestions for future work.

Chapter 2 Background

This chapter introduces the background of the work studied in this thesis. The limitation of existing networks and their management have become an obstacle for the development of future networks that will require the quick introducing of new services and protocols [2][35][36][37]. To accommodate these requirements a new network paradigm known as the active network has been proposed in recent years. Active network is an expanding field of research. It allows the network managers or users to program the network nodes according to their own needs, offering a great amount of flexibility [38]. It becomes therefore an ideal candidate for complementing existing networks towards a truly multi-service environment that can quickly adapt to new service requirements and protocols.

Around this new network paradigm, some solutions for the network management were proposed [10][11][13]. The potential of this new technology has been demonstrated from several ways [39][10]. This chapter reviews and compares several different solutions from the literature, which identify the requirements for 'active nodes' in active systems and highlight the need for scalability and flexibility, even down to the hardware level. The need for flexible adaptive nodes has brought with it a requirement for studies of alternatives to state-based management strategies, particularly those that draw on the notions of massive distribution of resources and distributed control strategies, such as is found in cellular computing. This has led to research on adaptive nodes that provide classes of network services on a local basis and co-exist within a multi-node environment to provide responsive network services. The thesis then concentrates on an adaptive management solution based on a GA. Both the principle of the solution and its software simulation are introduced in this chapter.

2.1 Limitation of Traditional Network and Management

As the Internet and related internetworking architectures and protocols have started to mature in recent years, better understanding has been gained of some of the limits and problems that are still associated with these technologies [40].

Traditional data networks passively transport user data from one end system to another. Normally, the user data is transferred opaquely, i.e., the network is insensitive to the data it carries and they are transferred between end systems without modification [37]. The computation within such networks is very limited, e.g. header processing in packet-switched networks and signalling in connection-oriented networks. The network cannot perform customized computations on the user data and therefore lack the flexibility of implementing user-specific application. This is because the traditional network devices such as routers and switches are closed, vertically integrated systems [41]. In addition, traditional networks were designed and customized for a single network service model. Their functions are rigidly built into the embedded software and are typically limited to simple management, routing, congestion control, and QoS (quality of service). Such an architecture has difficulty in integrating new technologies and standards into the shared network infrastructure and accommodating deployment of new services that dynamically extend the capabilities of the existing architectural model [2]. As the networking and computing is growing so fast, complex network services will have to be introduced at an accelerated rate. Therefore the rapid change in infrastructure technologies calls for a more adaptable service model optimised for flexibility [36].

Currently, the network has more and more users and more and more applications running on the network. The kinds of devices within the network are also increasing. All these changes mean there are more and more demands to the network that has to be handled [42]. Unfortunately, many of the new user demands or applications can not be simply superimposed on the networks because they have to coexist with, or require the adjustment of, the existed services of the networks. In addition, all of the network services are supported by the protocols, which have been standardized long before the new uses and applications were conceived. This makes a lot of constraints for the users or applications and ultimately limits their ability to utilize the current network as flexibly as they want.

Development of new network protocols is generally by a committee approach [43]. Normally it will take a long time to agree on a solution that is satisfactory to all the members. To deploy the newly developed protocols will also take a long time because the manufacturers of network devices (such as routers and switches) can only integrate the new protocols into new equipment or existing equipment that is re-configurable or re-programmable. Unfortunately, much of the current equipment used may not support the new protocols. Unless the old devices are phased out, the users and the applications cannot take the benefits provided by the new protocols.

In addition to the above drawbacks of the existing network, its management is another issue to be addressed. The majority of current network management systems are centralized around some management station [44]. The manager queries the managed objects, tries to build a view of the network and sends alert if a problem is detected. It can also take corrective actions by sending configuration commands to network entities. Such centralised management architecture has many drawbacks, especially when the network grows in size and complexity. As the number of controlled elements grows, the requirements for computational power from the management system and bandwidth from the network that connects it grow too. In a very large network, some of the managed objects may have a long distance from the management station. This will lead to a long delay for the control loops. Further, the traffic for management control wastes bigger portions of the network bandwidth [44].

To summarize the above, existing networks limit the deployment of new network services because they restrict the evolutions of standards and the closed network devices have very limited computation ability. Also it is difficult to integrate systems based on new technology with existing architectures. In addition, as the overall system become more and more complex, the network management is now the biggest cost [45].

2.2 Future Network and Management

As traditional networks and their management have the drawbacks as stated in the former section, a new network paradigm and its management solutions were investigated in recent years [35][42][45][46][47]. The new paradigm for the future network is expected to support very diverse environments (commercial heterogeneous wireline/wireless networks), applications (multimedia, WWW, telnet), and workloads (heterogeneous unicast and multicast streams with different quality of service requirements) [48]. To support such diversity in a single network infrastructure is a big problem because different applications have different requirements from the network. In such a case, it is clear that the network must play a more active role in supporting the needs of the applications and end users [48].

This means the network itself should take an active part in service provision, evolving from a simple transport network toward a large distributed processing environment [49]. The flexibility provided by this ability will allow the network to adapt instantly to changing customer service demands.

In recent years a variety of approaches [37][39][50][51][52][53] have been pursued in order to provide a flexible programmable network infrastructure that could "change its behaviour on drop of a dime" [54]. The development is directed toward open, active, and programmable networks.

Currently, there are several proposals for programmable network infrastructures, such as open signalling [52] and active networks [3]. These proposals share the goal of improving network flexibility and functionality through introduction of an accessible programming abstraction, which may be available on a per-user or even per-packet basis [55]. By opening network equipment to programming (and reprogramming) by the service provider, service providers are freed from the traditional slow process of consensus-based standardization and vendor-dependent implementation [39]. To further open the network to be programmed by the users will make the network even more flexible as the user can program the network for their own application, thus implement customised service. If the network can be programmable, the whole network seems to become a virtual extension of the user's computer. Users will be able to customize their packet forwarding services in terms of reserved resources, QoS, and preferential treatment and even use network resources for distributed processing or storage [39]. For example, the users can make network as a filter to process their packets or storage the computing data distributive on the network.

The concept of active networking emerged from discussions within the broad Defense Advanced Research Projects Agency (DARPA) research community in 1994 and 1995 on the future directions of networking systems [3]. The main idea is to let the messages carry procedures and data, and allow the network to perform customized computations on the user data. This is a natural step beyond traditional circuit and packet switching, and can be used to rapidly adapt the network to changing requirements.

Active network [3][4][5][6] allow the network devices such as routers and switches to be programmed by the users and this gives rise to the notion that data computing can be performed not only at end hosts but also in the network at the intermediate nodes. Thus, an active network may support the deployment and execution of user applications (embedded in the user communications) and offer dynamically customized network services.

Recent research in the area of active networking has demonstrated the potential of this new technology [3][54][44]. From the service point of view, active networking allows customized packet processing inside the network on a per-packet, per-flow, or per-service basis [10]. For example, customized packet processing can be applied to application-aware routing, information caching, multi-party communications, and packet filtering [3]. From the management perspective, active networking technology enables rapid service deployment and flexible service management [56].

Active networking technologies are being made possible by the advances in software and hardware technologies, and in particular in distributed object-oriented engineering [57]. The aim is to provide great flexibility, re-configurability, programmability and management to allow the dynamic and rapid service provision. However, such technologies cannot be deployed successfully without the related management systems being available to provide the support required by telecommunications operators to meet the increasingly sophisticated demands of a customer-oriented market [57].

In recent years, several management solutions [9][11][13] for active network have been proposed and tested. Marcus Brunner and his collaborators have developed a management framework for active networks that allows customers to deploy and manage their own active services in a provider domain [10][58][59]. The key concept in this framework is the Virtual Active Network (VAN). Following them, the VAN captures the functionality and resources that a provider offers to a customer. It can be seen as a generalization of a traditional Virtual Private Network (VPN). From the customers' perspective, the VAN represents the environment in which the customer can install, run, and manage active services without interaction with the VAN provider. From the VAN provider's perspective the VAN provides resource partitioning and customer isolation [9]. In contrast to a traditional VPN, a VAN gives a customer a much higher degree of flexibility and controllability. This is because the framework is not only a service abstraction for customers, it also allows the customers to create service management functionality and manage their services at run-time, without interaction with the provider's management system. This capability is difficult and costly to achieve in today's telecom environments [58]. Furthermore, the VAN concept supports the capability to isolate customers from one another, in order to avoid interference among its services. The framework calls for mechanisms inside the provider's active network nodes, in order to partition resources and police their consumption along VAN boundaries.

Another proposal for the active network management is the use of mobile software agents as an alternative to traditional centralized network management techniques, which was proposed by Rumeel Kazi and Patricia Morreale in [13]. The same work can be observed in [60], proposed by Yasin Kaplankiran et al. Mobile software agents provide a decentralized approach to network management issues because of the distributive nature of the agents themselves. Mobile agents are the active entities that roam across network nodes and perform specialized tasks on behalf of their senders [61]. They work independently or co-operate with other software components. With such distributive nature, mobile agents can resolve any problems that are associated with a centralized manager such as bottlenecking or network congestion [13]. Mobile agent can be used to implement not only distributed management for an active network, but also services inside the network.

Each of the above two solutions provides a way of implementing management for the active network, but they need a complex execution environment (EE) on the nodes and more manual intervention compared with the adaptive control solution proposed by Ian W. Marshall and Chris Roadknight [8][11]. They propose a model for future network management that is based on a bacterium inspired GA. The model makes use of the unique methods that bacteria use to transfer and share genetic material, to create a more robust solution to the service provision problems associated with future data networks.

It is useful to compare these three solutions for active network management. The VAN solution emphasises the exploration of the active node structure, which has to be designed to provide separate execution environments for the different user services. The agent solution uses the active agents to provide a dynamical and distributed management. Also, there is a need for the node structure to provide appropriate execution environments for the different agents. The main difference between these two solutions and the bacterium typed solution is that the latter proposes a general way for adaptive control and management, without considering the detailed structure of the node.

The future active network will be unbounded in both scale and function [8]. The network may be constantly expanded, upgraded and re-dimensioned. On the other hand, as the network is programmable by different users, it is impossible to predict accurately what the user defined services will look or behave like. All these will make the traditional network management, which is based on the knowledge of the global state of the system, become more and more difficult because the global state of the system will be impossible to acquire due to the massive scale and the unpredictability of the user defined services. A more flexible and adaptive solution will be needed to satisfy the requirement of the future active network.

Adaptive control [62] is based on learning and adaptation. The knowledge is leant through performing experiments on the system and storing the results (learning) [8]. The knowledge that the system has learnt is then used to choose the control actions that need to be taken. In particular, adaptive control attempts to automate the behaviour of an expert, monitor the system state and make changes based on the interpretation of the monitoring. This style of control has been proposed for a range of Internet applications including routing [63], security [64][65], and fault ticketing [66].

GAs [67][68][69] are robust search techniques based on the principles of evolution [70]. It is an iterative procedure that consists of a population of individuals (i.e. candidate solutions). Each of these individuals was represented by a finite string of symbols, referred to as the genome, which encode a possible solution in a given problem space. Generally speaking, the GA is applied to spaces that are too large to be exhaustively searched [71]. The standard GA proceeds as follows: at first, an initial population (set of candidate solutions) is generated randomly. The individuals in the initial population are evaluated according to the predefined fitness function. Those individuals with a higher fitness value are chosen to reproduce offspring, i.e., generate a new population partly by recombining and partly by mutating the initial individuals. This new generation again was evaluated according to the fitness function. The above step is repeated until a stop condition is true. Because each time after evaluation, the high quality individuals were chosen to be the elements of the new generation of the population thus make the good individuals have a better chance of 'reproducing', while bad ones are more likely to disappear (being abandoned).

This thesis explored an adaptive solution for active network management that was based on a GA inspired by bacterium and demonstrated its capability to provide robust search techniques based on the principles of evolution.

2.3 A Genetic Algorithm Inspired Solution for Future Network Management

A novel solution for future network management was introduced in [11]. This solution uses a distributed GA on the active nodes within an active network, to provide adaptive management for the network. The particular GA is inspired by the unique methods that bacterium use to transfer and share genetic material, and the model is known as the "Bacterium Model". The solution makes each node within the network responsible for its own behaviour. The whole network is thus modelled as a community of cellular automata. Each member of the community only optimises its own (local) state, handling network requests in the same way as bacteria metabolise energy sources.

2.3.1 Introduction to "Bacterium Model"

The "Bacterium Model" was proposed and developed by Marshall and Roadknight [8][11] to simulate a management solution for future active network. It is a system model that takes the form of cellular automata where each cell (nodes in the simulated network) behaves like a simple organism (bacterium). Suppose each node has a rule set (correspond the genome of the organism) that define the way the node handle the network tasks, The interactions among the cells only take place on a local basis (i.e. each cell only communicates with a few other nearby cells). The organisms were termed bacteria because their reproduction methods were inspired by biological processes and GAs. Rule diversity, which is the essential of the GA, is created in two ways, mutation and plasmid migration. For example, individuals are capable of exchanging elements of their genetic material during their lifetime using the process of 'plasmid' migration (in which sections of 'genome' can be transferred between living organisms). In biological systems, 'plasmid' migration make the bacterium evolve very quickly but with a robustness that has kept many varieties present for several billions of years [72][73]. The "Bacterium model" uses a GA that incorporates a 'plasmid' migration mechanism that allows nodes to exchange their rules with each other. This behaviour allows much quicker reaction to sudden changes in influential environmental factors [8] and can be modelled as a learning mechanism.

The solution was first proposed by Chris Roadknight and Ian Marshall in [11]. Following their description, the model has a large number of nodes connected, say, as a rectangular grid. Each node has a rule set that includes the rules defining the way each node handle their requests. Requests are input to the system by injecting sequences of characters (representing services) at each vertex in the rectangular array. Each node evaluates the items that arrive in its input queue on a FIFO principle. If the request at the front of the queue matches an available rule the request is processed at the node. If there is no match the request is forwarded to one of the adjacent nodes.

Each rule takes the form of {x, y, z} where :

"**x**", is a character representing the type of service requested (suppose there are four types of services, say, "a", "b", "c", "d");
"**y**", is an integer between 0 and pre-defined max queue length (say 200) which is interpreted as the value in a statement of the form "Accept request for service [Val(x)] if queue length<Val(y)";
"**z**", is an integer between 0 and 100 that is interpreted as the value in a statement of the form "Accept request for service [Val(x)] if busyness < Val(z))%".

For example, a rule like {a, 20, 40} means if the incoming request is service "a" and the queue length is less than 20 and the busyness value is less than 40, then the request will be processed. Figure 2.1 shows a diagram of the network with the injection of requests for services on each node. In the diagram some nodes are switched on (solid borders), and some are switched off (dashed borders).

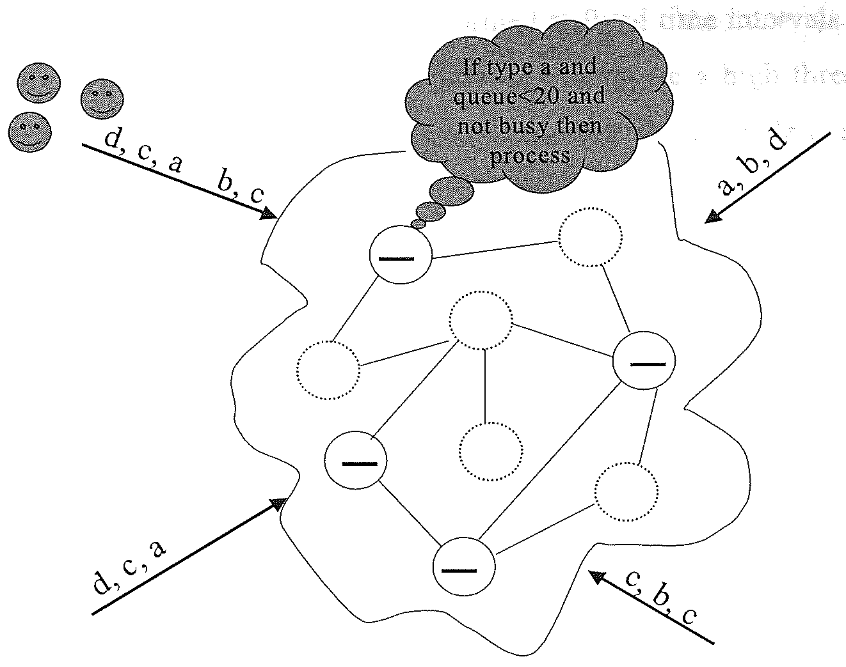


Figure 2.1 Schematic of proposed future network structure

The number of the requests that can be processed per measurement period (expressed in epoch) is pre-defined. The more time a node spends processing requests, the busier it is seen to be. The busyness is calculated by combining the busyness at the previous epoch with the busyness for the current epoch in a 0.8 to 0.2 ratio. For example, if the node has processed three requests this epoch (25 points each) it would have 75 points for this epoch, if its previous cumulative busyness value was 65 then the new cumulative busyness value will be 67. This method dampens any sudden changes in behaviour [11].

Rule diversity is created in two ways, mutation and plasmid migration. Mutation involves the random alteration of only one value in a single rule, for example: from "Accept request for service c if node <90% busy" to "Accept request for service b if node <90% busy" or to "Accept request for service c if node <70% busy". Plasmid migration involves rules from more active nodes being shed or replicated into a 'rule pool' which is accessible to all nodes and subsequently being absorbed into the rule set of less active nodes. Each node has a fixed number of active rules at one time. If a node acquires more than this fixed number of rules through interchange the newest rules are registered as dormant.

Values for queue length and cumulative busyness are used as the basis for migration interchange actions, and an evaluation is performed at fixed time intervals to decide how to change the rule set. If the queue length or busyness is above a high threshold (say 50), a random section of the rule set is copied into the rule pool. If the node continues to exceed the threshold for four evaluation periods, it replicates its entire rules into an adjacent non-active node and activates the node (the node become active). If all adjacent nodes are active, no action is taken at the node. If the busyness is below another low threshold (see 10), a rule randomly selected from the rule pool is injected into the node's rule set. So if a node with the rule set {{a, 30, 40}, {b, 20, 10}} has a busyness of larger than 50 when evaluated, it will copy a random selected rule (suppose {b, 20, 10}) into the rule pool. If another node with the rule set {{c, 25, 30}, {d, 40, 35}} is later judged to be idle it may absorb that rule from the rule pool and the new rule set will become {{c, 25, 30}, {d, 40, 35}, {b, 20, 10}}. In such a way, the nodes perform exchanging of their rules. Figure 2.2 illustrates such a procedure. If a node is 'idle' for three evaluation periods, its active rules are deleted, if dormant rules exist, these are brought into the active domain, if there are no dormant rules the node is switched off (become non-active). The node will not be switched on again until an adjacent busier node activates it.

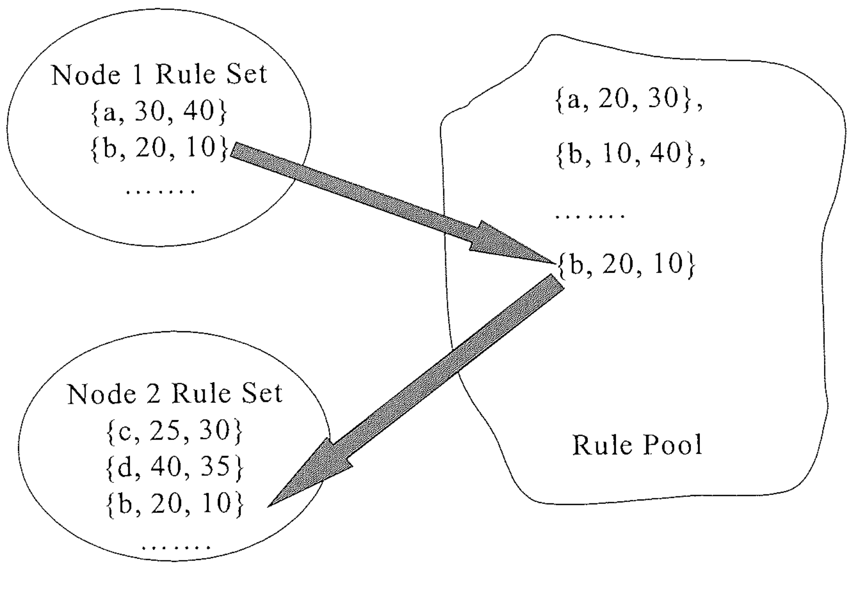


Figure 2.2 Exchange of rules between nodes through the rule pool

2.3.2 Software Simulation of "Bacterium Model"

The original software simulation of "Bacterium Model" was devised by the proposing researchers using Visual Basic Language [11]. The software was executed within a visualisation environment to simulate the behaviour of the model. The simulation was tested with a variety of loading scenarios, its performance when faced with these scenarios showed how suitable the approach is for future network management. Such a test is used as a quality of service (QoS) measurement that measured the average age of all requests on the network.

As part of the research an enhanced version software simulation of "Bacterium Model" was developed using Visual C language by the author. The enhancement lies in both the flexible design and an improved algorithm based on the standard GA. The flexible design means the size of the network can be changed during the simulation process, while the original implementation of VB version in [11] can only simulate a fix-sized network. The improved algorithm involves the changing of the "mutation" algorithm, which is one of the classical GA operators, from the random alteration of a value to a deliberate and reasonable alteration, based on the statistic result of the previous experiment. The QoS measurement was introduced to measure the performance of the network and a comparison between these two types of simulations is given to show the advantage of the new improved algorithm.

2.3.2.1 Original Software Simulation

The original software implementation [11] of "Bacterium Model" (VB version) has 400 vertices connected on a rectangular grid (20x20) and supports four services, represented by A, B, C, and D. The system is initialised through populating a random selection of vertices with active nodes. Each node has a rule set which defines how the node will handle requests for services. The initial nodes have a random selection of rules. The form of the rule has been described in previous sections. A visualisation environment was created for the implementation that allowed the system to be simulated in a flexible manner. The simulation related parameters such as the ratio of requests for the 4 services and the overall number of requests per unit time etc can be varied through the interface provided by the environment. A displaying window that can accommodate up to 400 nodes was used to display the system state.

A quality of service (QoS) measurement was introduced that measured the performance of the network. It measured the average age (expressed in epochs) of all requests on the network. The QoS was measured over time as the load on the network was increased in a series of significant steps, 1,4, 12, 30, 45, 60X the initial load. Figure 2.3 shows how the QoS reacts to these increases. The X (horizontal) axis represents the time (in epoch) and the Y (vertical) axis shows the QoS (average delay of the requests for service in the network, the lower the delay the better the QoS). A short period of worsened service is observed as the network adapts in both size and heterogeneity, followed by a return to more acceptable QoS. At 60X the initial rate, the network is nearing its saturation point, yet performance has degraded very little.

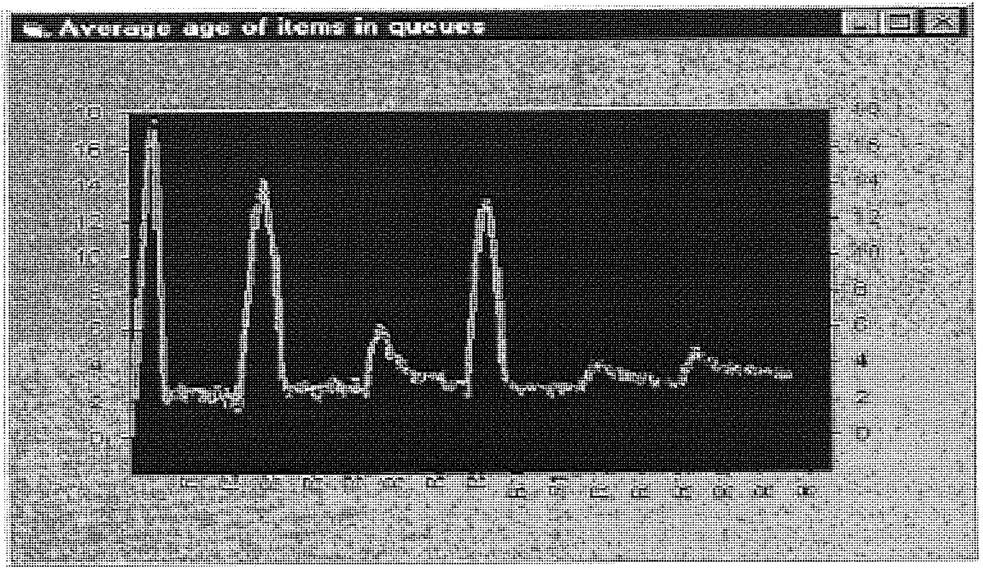


Figure 2.3 QoS level: Average time of requests (epochs) v Time (epochs)

A conclusion was made based on the above result that the bacterial type GA could provide an adaptive management for the network. As each node only responsible for its own behaviour, this provides a simple and efficient scheme without handling most of the high-level network management problems [11].

2.3.2.2 Improved Software Simulation

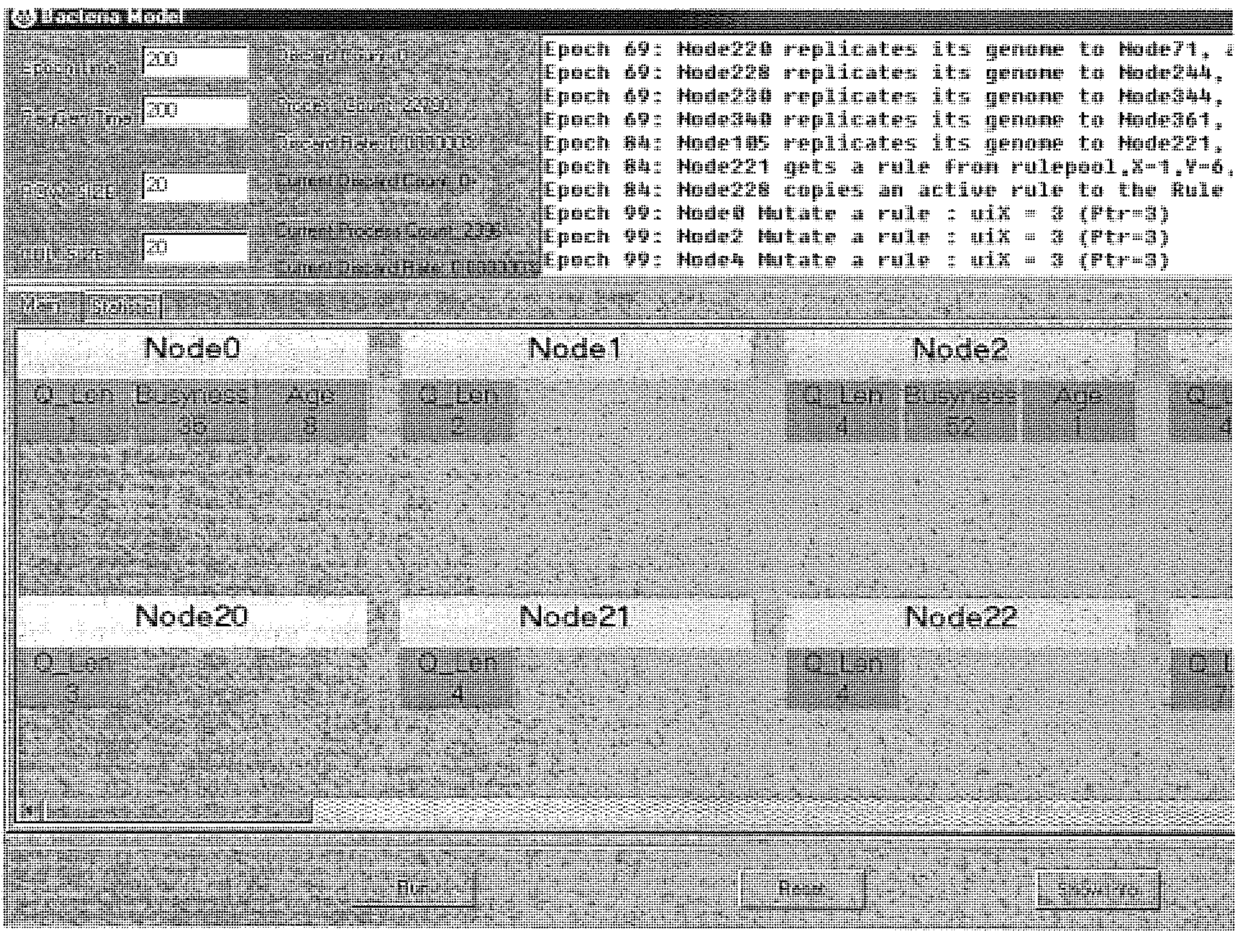
A modified version of software simulation of "Bacterium Model" was devised using Visual C (VC) language as part of the research. Based on the same principle of "Bacterium Model", this newly developed version made some progress to improve the original design and the performance. The original design simulated a fixed sized network with 400 nodes within it, while the new design allows scalability and thus provides more flexibility. The original design introduced a QoS measurement which measured the average age (expressed in epochs) of all requests on the network, while the improved design provides not only the same QoS measurement as the original one but also a new QoS measurement method that measures the discarding rate for requests that have exceeded their 'time to live' (TTL) in the network. The TTL is the pre-defined time constant that decides how long the request can stay in the network. This time constant is defined to prevent the requests stayed in the network too long to cause the congestion and thus decrease the performance.

Improvement of the performance involves the introduction of a new algorithm. This algorithm was developed as a "revised" version of the traditional GA of "mutation". As "mutation" involves random alteration of part of the "genome", the "revised" mutation made a deliberate 'informed' alteration to a rule instead of random alteration based on some statistic result. From this point of view, it is more like a "learning" process rather than a GA.

The standard mutation algorithm in "Bacterium Model" involves a random alteration of only one value in a single rule, as the example in section 2.3.1 showed, any of the three items in a rule can be altered randomly, one at a time. In the new design, the revised algorithm process was developed in which the "mutation" is directed towards a desired solution, rather than being completely random and the algorithm only changed the way the first item in the rule is altered, i.e., the service type, A, B, C or D. The revised "mutation" process can be guided to alter the value of the service type and make the value to be that of the most wanted service. As a result, most of the requests will then match the rules of the nodes and therefore decrease the "discarding rate" and the average age. The other two items, busyness and queue length is handled in the same way as the standard mutation algorithm, i.e., being altered randomly.

Similarly, a modified visual environment was created for the implementation. The environment provides an interface where the parameters can be varied. For example, the ratio of requests for the four services and the size of the network (the number of the nodes within the network) are variable. The parameters of "epoch time" and "request generate time" (representing the ratio of requests) can be changed at the end of a simulation run or during a pause to run. The parameters relate to the network size can only be changed at the end of a simulation run. Unlike the original design, which can only simulate a model of a fix-sized network, such an improvement made the system even more flexible. The size of the network can be made to vary from 2x2 to 20x20.

A displaying window that can accommodate up to 400 nodes was presented to show the system state. Figure 2.4 is a diagram of an example execution, includes both the interface environment and the system simulation state. The left part of the figure is the area for parameter variation and variable displaying. The right part of the figure shows the trace of network state and actions, and the lower part of the figure shows the nodes states. It also includes the node's busyness, queue length and age.



```
Epoch 69: Node220 replicates its genome to Node71, 
Epoch 69: Node228 replicates its genome to Node244,
Epoch 69: Node230 replicates its genome to Node344,
Epoch 69: Node340 replicates its genome to Node361,
Epoch 84: Node105 replicates its genome to Node221,
Epoch 84: Node221 gets a rule from rulepool,X=1,Y=6,
Epoch 84: Node228 copies an active rule to the Rule
Epoch 99: Node0 Mutate a rule : uiX = 3 (Ptr=3)
Epoch 99: Node2 Mutate a rule : uiX = 3 (Ptr=3)
Epoch 99: Node4 Mutate a rule : uiX = 3 (Ptr=3)
```

Figure 2.4 System execution diagram

Two experiments were performed to show how the newly developed algorithm improves the system performance. Based on the above analysis, the improvement to the network performance after using the new 'mutation' algorithm is mainly reflected on the decreasing of the 'discarding rate'. First, a design that employed the traditional mutation algorithm (with discarding request) was simulated and second, a design that employed the new algorithm (also with discarding request) was simulated. The network performances of using these two algorithms were compared with both the QoS (the average age of all requests on the network) and the discarding rate measurement. Figures 2.5 and 2.6 show the QoS measurement under the different algorithms. In both figures, the QoS was measured under the same configurations, i.e., each has 400 nodes connected on a rectangular grid, each has the same initial selection of active nodes, and each initially active node has the same initial rule set. In both figures, the load was increased by 4, 16, 64X the initial load in a same series of significant steps, 500, 1000 and 1500 epoch steps. These figures show how the QoS reacts to the increases. Whenever the load was increased, there was a period of worsened service, but as the network adapts in both size and heterogeneity, the age went down and thus provided a better performance. Table 2.1 and table 2.2 listed the configuration data and the simulation result including the number of active nodes and the discarding rate at the different stage. From the tables, it can be seen that in both cases, after the load was increased to 64X, the discarding rate increased abruptly since from this point all the nodes become active and the network can not adapt in size any more. The QoS also deteriorate from this point in both cases as the age stayed at a high level and does not reduce over time.

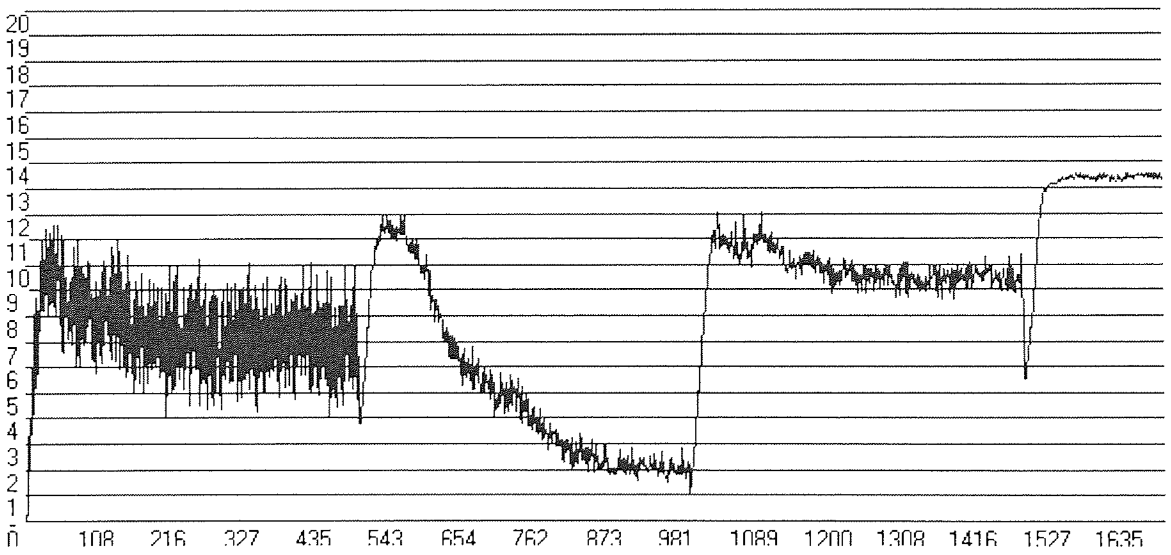


Figure 2.5 QoS (new algorithm): Average time of requests (epochs) v Time (epochs)

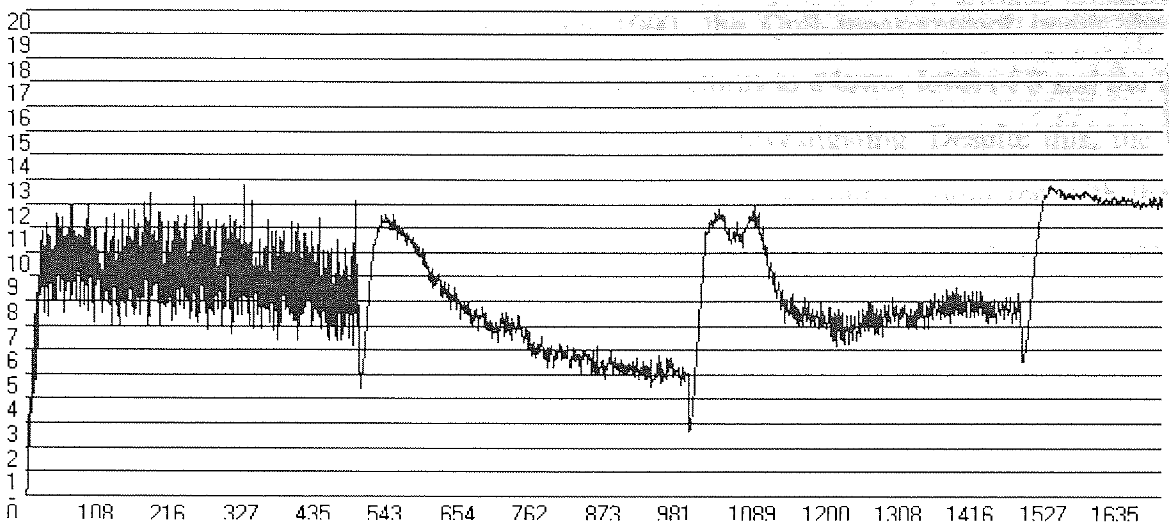


Figure 2.6 QoS (old algorithm): Average time of requests (epochs) v Time (epochs)

Epoch	Network Load	Number of Active Nodes	Discarding Rate
0	initial load	80	0
500	4X initial load	80	6.1%
1000	16X initial load	244	4.7%
1500	64X initial load	400	4.5%
2000	64X initial load	400	9.6%

Table 2.1 Configurations and results under new algorithm

Epoch	Network Load	Number of Active Nodes	Discarding Rate
0	initial load	80	0
500	4X initial load	77	14.9%
1000	16X initial load	217	10.4%
1500	64X initial load	400	10.3%
2000	64X initial load	400	21.2%

Table 2.2 Configurations and results under old algorithm

Compare these two figures and two tables one can see that the network performance under the new algorithm is better than the old one from both the QoS (in terms of average age) measurement and the discarding rate measurement. First, the QoS measurement in figure 2.3 has an average age of 7 during the period between epoch 0 to 500 and figure 2.4 has an average age of 9 during the same period. Similarly, in the second period between epoch 500 and 1000, the former returns to a lower level of 2 and the latter returns to 5. However, after the second increase of the load at epoch 1000, the QoS measurement under the old algorithm is better than the new one as the former returns to a lower level of 8 and the latter only returns to 10. The reason for this is still under investigating. Despite this, the QoS measurement under the new algorithm still possesses an advantage compared with the old one from a long term average view. The most improvement in the performance caused by the new algorithm embodied in the discarding rate measurement. As the tables show, the discarding rate under the new algorithm is much lower than under the old one during every stage of the measurement.

As "random" is the essence of mutation in GA, the revision of guiding the alteration towards a desired direction makes the evolution purposive, not a natural evolvement towards fittest state. From the adaptive control point of view, it is more like a "learning" process rather than a GA as adaptive control is based on learning and adaptation [8]. It gets the knowledge through performing experiments on the system and storing the results (learning). The new algorithm learned from the history record that which direction is the better way to go and then exercises it using the method of a standard GA. It is more like an integrated algorithm combining a learning process with a standard GA. The experiment result has demonstrated its performance for adaptive control.

As the new algorithm is still in its infancy, the later hardware implementation of the developed system was based on the original GA.

2.4 Conclusion

This chapter introduces the background of the research work. As the traditional networks and their management solutions have severe drawbacks, the notion of an active network was proposed as a new paradigm for future network. Many works have been done in this area including both the implementation methods and their management solutions. This thesis focused on a management solution that uses a distributed GA on the active nodes within an active network to provide adaptive management for the network.

After introducing the software simulation and the QoS (in terms of the average age of the requests in network) measurement of the system based on the original proposal, a new algorithm was introduced and simulated. This new algorithm was based on the traditional genetic algorithm of mutation and aims to improve the algorithm performance. It is more like an integrated algorithm combining a learning process with a standard GA.

Two types of software simulations were then implemented and the performance measurement results were given, one with the original GA inspired by bacterium and another with the new proposed algorithm. The QoS performance measurements in both the terms of requests average age and the requests discarding rate were compared between the two simulations. A conclusion was then made that the system that applied the new algorithm is better in performance than the one who applied the original algorithm.

Chapter 3 Theories, Concepts and Technologies

This chapter introduces the theories, concepts and technologies that were used in the development of the research. The content includes evolutionary computation, re-configurable computing, cellular computing, concurrent programming and parallel processing.

3.1 Evolutionary Computation

The work of evolutionary computation was began independently by Ingo Rechenberg (1964) [74] with his work on "Evolutionsstrategie" for function optimisation, Lawrence Fogel (1966) [75] with the evolution of finite state machines through "Evolutionary Programming", and John Holland (1975) [76] with a class of adaptive systems we now call "Genetic Algorithms". It applies the basic principles of evolution to the solution of problems in a variety of domains, especially for addressing complex engineering problems—ones involving chaotic disturbances, randomness, and complex non-linear dynamics—that the traditional algorithms have been unable to conquer [16].

3.1.1 Evolutionary Computation

Evolution is the primary unifying principle of modern biological thought. When this thought was extended beyond the study of life, such as to an optimisation process that can be simulated using a computer or other device and put to good engineering purpose, the concept of evolutionary computation was born. Evolutionary computation is then the computational paradigm that uses evolution principles to provide effective solutions for optimisation problems. When solving engineering problems, evolutionary computation approaches provide considerable advantages such as the adaptability to changing situations and the ability to generate good enough solutions quickly enough for them to be of use [16].

As natural evolution starts from an initial population of creatures, the evolutionary computation begins with an initial population of contending solutions for the problems at hand. The performance is measured to assess the "fitness" of each existing solution. A selection mechanism will then be used to determine which solutions should be maintained as "parents" for the next generation according to their fitness measurement. These "parent" solutions then generate "offspring" by a pre-defined means of random variation. The whole procedure can then be described as a two-step process, consisting of random variation followed by selection [16] and is repeated over successive generations until a satisfied solution was found. Figure 3.1 shows the general process.

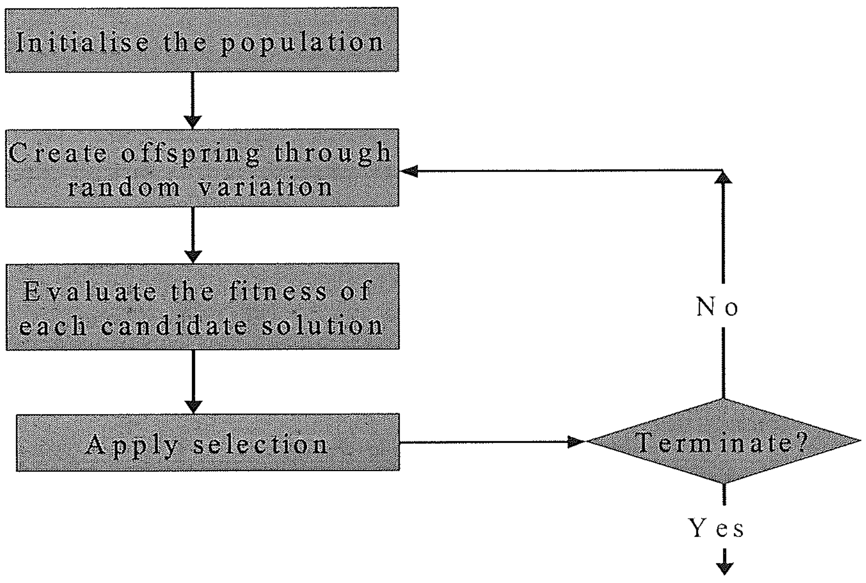

Initialise the population
Create offspring through random variation
Evaluate the fitness of each candidate solution
Apply selection
Terminate?
No
Yes

Figure 3.1 General process of evolutionary computation

In the last several years, the field of evolutionary computation has seen many new ideas, which stem from new biological inspirations. P.J. Bentley and T. Gordon et al [77], researchers from University College London (UCL) outlined the new trends in evolutionary computation and the new branches of research in this area, which aim to extend the capabilities of EC. Following them, the new research tries to apply evolutionary algorithms to dynamic and ill-defined problems, instead of optimising static and simplified functions, embedding knowledge (and constraints) into the search as traditional techniques of EC has done. Such research 'traditionally' took place only in the field of artificial life, but now researchers are investigating how evolution can generate novelty for unconventional applications such as music composition, art, conceptual design and even novel fighter pilot strategies [78].

3.1.2 Application of Evolutionary Computation

Evolutionary computations are already being used to solve a wide variety of real-world problems that pose significant challenges [16]. Application lies in the areas such as engineering design, scheduling, telecommunications, aerospace, environment engineering, signal processing, circuit design, medical engineering, artificial life, network etc. [79]. An important new area for the application is Evolvable Hardware (EVH): the automatic design and synthesis of electronic circuits [77]. The progress in the research of EVH has brought a promising prospect that EC can be used to devise truly intelligent machines.

The GA based management solution introduced in chapter 2 is an example of evolutionary computation application in network. In the recent years, GA has been used more and more as an adaptive control algorithm [70] and this provides a variety of applications. These applications can be contributed to evolutionary computation as GA is one of its avenues.

Previous and more recent work [78] [80] has explored the huge variety of new, creative applications being tackled by evolutionary practitioners today [77]. For example, the work of Ian Parmee concentrates on the development of interactive evolutionary systems that allow users to relax functional constraints for engineering design problems [81].

3.1.3 Genetic Algorithm

The first and most widely used approach to evolutionary computation is the genetic algorithm. The genetic algorithm is a model of machine learning that derives its behaviour from a metaphor of the processes of evolution in nature [67]. It was begun by John Holland, from the University of Michigan in the early 1960s. The first achievement was the publication of "Adaptation in Natural and Artificial System" [76] in 1975. The aim of his work is to improve the understanding of natural adaptation process and to design artificial systems having properties similar to natural systems.

A GA is an iterative procedure that consists of a constant-size population of "individuals", each represented by a finite string of symbols, known as the "genome" (the symbol used is often binary, though other representations have also been used, including character-based encoding, real-valued encoding, and most notably—tree representations) [71]. The genome encodes a possible solution in a given problem space, referred to as the "search space". This space comprises all possible solutions to the problem at hand. The standard GA process is performed as follows: first, an initial population of solutions (individuals) are created randomly. The individuals in the current population will be evaluated according to some pre-defined criterion of performance measurement, referred to as "fitness". After evaluation, the high fitness individuals may be selected to form a new generation of population. The methods that are used to form the new solutions (individuals) are genetically inspired operators such as the most well known "crossover" [82] and "mutation". The former (cross-over) means two selected individuals (called "parents") exchange parts of their genomes (i.e., encoding) to form two new individuals (called "offspring"). The latter (mutation) means the random alteration of bits in the genome (the encoding of the individuals). Such a procedure will be repeated until the termination condition is satisfied, which may be specified as some maximal number of evolutionary step (generation) or as the attainment of an acceptable fitness level. Two examples are presented in figure 3.2 and 3.3 to illustrate these two methods of genetic operator. And the whole procedure of GA can be outlined as the following simple generational algorithm:

1. Randomly generate a population of individuals
2. Evaluate the fitness of each individual
3. Generate a new population by exerting genetic operator (crossover and mutation) on the fittest individuals
4. Repeat steps 1, 2 and 3 until a termination condition is satisfied

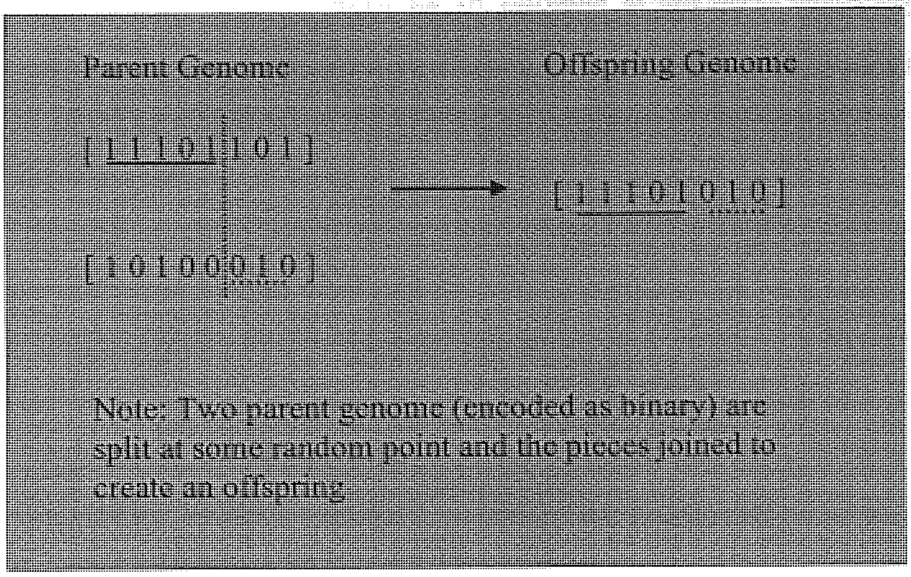

Parent Genome
Offspring Genome
[1 1 1 0 1 1 0 1]
[1 1 1 0 1 0 1 0]
[1 0 1 0 0 0 1 0]
Note: Two parent genome (encoded as binary) are split at some random point and the pieces joined to create an offspring

Figure 3.2 Genetic operator of "crossover"

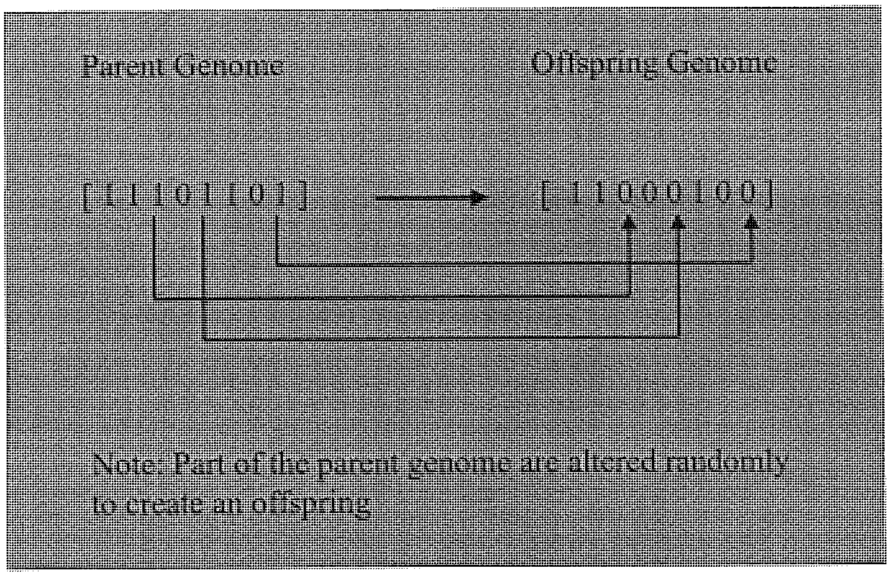

Parent Genome
Offspring Genome
[1 1 1 0 1 1 0 1]
[1 1 0 0 0 1 0 0]
Note: Part of the parent genome are altered randomly to create an offspring

Figure 3.3 Genetic operator of "mutation"

One thing to be noted is that "evolution" (in nature or anywhere else) is not a purposive or directed process [67]. (Incidentally, this conclusion confirmed the algorithm introduced in section 2.3.2.2 cannot be defined as a GA).

GAs are used to solve a large variety of problems that do not have a precisely-defined solving method, or even if they do, when following the exact solving method would take far too much time. Such problems are often characterised by multiple and complex, sometimes even contradictory constraints, that must be all satisfied at the same time. Examples are crew and team planning, delivery itineraries, finding the most beneficial locations for stores or warehouses, building statistical models, etc. One example is the question: "what is the shortest path linking a number of cities? ". The only exact solution for this problem is to try all the paths and compare them. When the number of the cities is large enough, this solution will take a time that is too long to be afforded. In such cases the GA provides excellent and fast answer of approximations to the question [16].

The possible applications of GAs are immense. Any problem that has a large search space could be suitable tackled by GAs. Generally speaking, the GA is applied to spaces that are too large to be exhaustively searched [71]. Actually, it has been successfully applied to numerous problems from different domains, including optimisation, automatic programming, machine learning, population genetics, studies of evolution and learning, and social systems [83].

3.1.4 Evolutionary Computation in "Bacterium Model"

The "Bacterium model" developed by Ian Marshall and Chris Roadknight (the latter has a background in biological research) uses an evolutionary algorithm (specifically a GA with mutation and migration) as the management algorithm for the future network. The model simulates a network made up of interconnected active nodes. Each node in the network has a rule set that comprises several rules, which define how the node will handle the requests from users for services. It is these rules that are encoded as the "genome" in the GA.

Adding rules for reproduction and evolution, and plasmid migration, it becomes possible to envisage each node as a bacterium and each request for a service as food according to Marshall and Roadnight in [11]. This analogy is consistent with the metabolic diversity of bacteria, capable of using various energy sources as food and metabolising these in an aerobic or anaerobic manner [8].

In the model, genetic diversity is created in two ways, mutation and plasmid migration. Roadknight's model defines the mutation process as the random alteration of just one value in a single rule and the plasmid migration process as genes from healthy individuals being shed or replicated into the environment and subsequently being absorbed into the genetic material of less healthy individuals. If plasmid migration doesn't help weak strains increase their fitness they eventually die.

Fitness are calculated every fixed time interval according to the queue length and cumulative busyness. If the queue length or busyness is above a threshold, a random section of the genome is copied into a 'rule pool' accessible to all nodes. If the node continues to exceed the threshold for, say, four evaluation periods, it replicates its entire genome into an adjacent vertex where a node is not present. Roadknight draws an analogy between this and the fact that healthy bacteria with a plentiful food supply reproduce by binary fission. Offspring produced in this way are exact clones of their parent. If the busyness is below a different threshold, a rule randomly selected from the rule pool is injected into the node's genome. If a node is 'idle' for, say, three evaluation periods, its active genes are deleted, if dormant genes exist, these are brought into the active domain, else if there are no dormant genes the node is switched off.

The algorithm described above is an unbounded, distributed GA with "bacterial learning" which uses evolutionary computation theory.

3.2 Re-configurable Computing

Re-configurable computing, which combines the flexibility of general-purpose processors with the efficiency of customized hardware to achieve extreme performance speedup, will change the way computing systems are designed, built, and used [84]. The enabling device for configurable computing is FPGA [85]. FPGAs are large, fast integrated circuits that can be modified, or configured, almost at any point by the end user [86] [87]. For applications characterized by deeply pipelined, highly parallel, integer arithmetic processing, configurable computing machines can outperform alternative solutions by up to an order of magnitude for metrics such as cost and computation speed [85]. Early research in this area of re-configurable computing has shown encouraging results in a number of areas including cryptography, signal processing and searching etc.

3.2.1 Re-configurable Computing

Re-configurable computing is a relatively new computing paradigm that emerged to overcome the traditional trade-off between flexibility and performance. It achieves both the high performance of ASICs and the flexibility of general-purpose application with its new class of architectures. The main characteristic of re-configurable computing (RC) is the presence of hardware that can be reconfigured to implement specific functionality more suitable for specially tailored hardware than on a simple uni-processor [88]. RC system joins microprocessors and programmable hardware in order to take advantage of the combined strengths of hardware and software [89] [90]. In other words, RC retains the high performance (such as the fast execution speed) of dedicated hardware (ASIC based) but also retain a great deal of functional flexibility of a general-purpose processor (microprocessor) [91]. Remarkable performance gains are achieved by placing an algorithm in an FPGA for embedded applications, compared with using a microprocessor or DSP because FPGA take advantage of hardware parallelism while reducing the timing overheads needed for general-purpose microprocessor applications.

The concept of re-configurable computing was proposed in the 1960s, but has become feasible only in recent years following the appearance of re-configurable devices. The key device is FPGA. The concept of re-configurable computing is then defined as the using of re-configurable FPGAs as computing devices, in order to accelerate operations that otherwise would be performed by software. The re-configurable computing systems can be defined as "those machines that use the re-configurable aspects of FPGAs to implement an algorithm". Configurable computing machines are then built around programmable hardware, using fine or coarse grain programmable FPGA devices [92][93]. Typically, configurable computing machines consist of a collection of FPGAs interconnected using a fixed or programmable interconnect [94][95]. The majority of the configurable computing machines [96][97][98][99][100] operate as co-processors [101] and have some local memory. Figure 3.4 shows a model under which most machines operate. The co-processors can be, on demand, configured to perform any desired function.

The logic of an FPGA is determined by a configuration, similar to software, so that an FPGA can be re-configured to perform a variety of computations without redesigning the hardware. The FPGA has to act as an execution engine for a variety of different hardware functions — some executing in parallel, others in serial — much as a CPU acts as an execution engineer for a variety of software threads [91]. Nowadays, FPGAs with capacities of up to millions of gates are available that even allow for partial re-configuration [102]. Partial reconfiguration means that only a part of the device is reprogrammed while the rest remains executing.

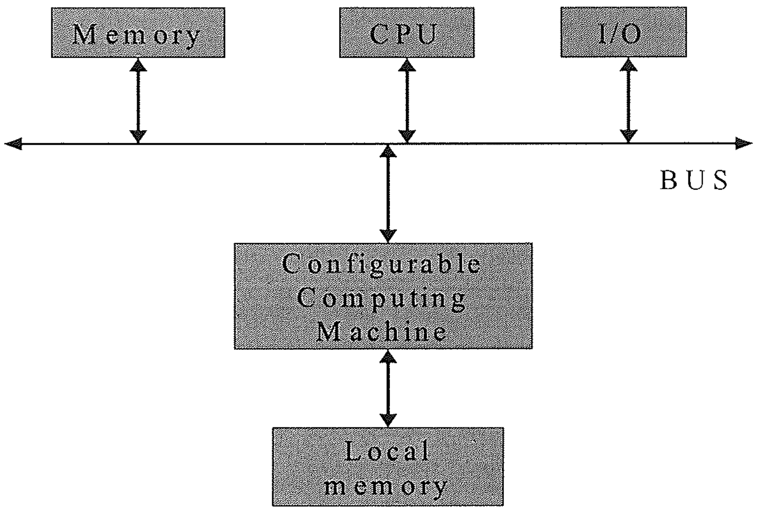


Figure 3.4 Typical model for configurable computing machine

Michael Barr outlined three advantages of re-configurable computing in [91]. The first is its ability to achieve greater functionality with a simpler hardware design. This is because not all of the logic must be present in the FPGA at all times, the cost of supporting additional features is reduced to the cost of the memory required to store the logic design. The second advantage is its lower system cost. Because re-configurable computing systems are up-gradable, this will extend the useful life of the system and thus reduce the lifetime cost. The third advantage is reduced time-to-market. This is obvious because the chip design and prototyping cycles for customised application are saved. Furthermore, the logic design remains flexible right up until (and even after) the product ships. These promising advantages provided by re-configurable computing make it a great candidate for a lot of applications that is introduced in the next section.

3.2.2 Application of Re-configurable Computing

Re-configurable computing technology has acquired a great deal of use in many areas of applications [19]. One example [91] of a theoretical application is a smart cellular phone that supports multiple communication and data protocols, though just one a time. When the phone passes from a geographic region that is served by one protocol into a region that is served by another, the hardware can be automatically reconfigured to meet the different requirement for the needs. Another example of the application is for satellite communications. TSI TelSys [103] has been developing and using the technology for their product of ground-station equipment for satellite communications. This application involves high-rate communications, signal processing, and a variety of network protocols and data formats.

Configurable computing machines have been demonstrated to have a wide variety of applications from logic emulation [104] [105] to algorithm specific hardware execution. Some of the computer intensive tasks that have made effective use of configurable computing machines include DNA sequence matching [106], RSA cryptography [96], text searching [107], fingerprint matching [108], and high speed image processing [109].

3.2.3 Re-configurable Computing in "Bacterium Model"

The research demonstration system introduced in this thesis is an experimental network comprising active nodes based on static re-configurable FPGA's to allow exploration of the management and performance of a network of flexible nodes in an evolving service-demand driven network.

Re-configurability is needed in the system for the flexibility of the network. Since old services may become obsolete and new services may appear, the network must have the flexibility to add new services and abandon old ones. New protocols also need to be introduced into the network over time. Any change of the nodes structures or functions may need reconfiguring of the system.

The demonstration system of "Bacterium Model" simulate an active network made up of active nodes, which are implemented on FPGA-based hardware, and thus is possible to be re-configured according to different design requirements. At the moment "Bacterium

Model" system can only implement static re-configuration, instead of dynamically re-configuration during run time.

3.3 Cellular Computing

3.3.1 Cellular Computing

Early computing technology was dominated by the von Neumann architecture, which is based upon the principle of one complex processor that sequentially performs a single complex task at a given moment [21]. Cellular computing is a computational philosophy that is different from conventional computational architecture and offers the potential of addressing much larger problem instances than previously possible.

Moshe Sipper, one of the most famous experts in this area, has defined the cellular computing as a vastly parallel, highly local computational paradigm with simple cells as the basic processor of computation, and as such it adheres to three principles: simplicity, vast parallelism and locality. The term simplicity means the basic processor used as the fundamental unit of cellular computing, referred to as the cell, is very simple. Unlike a general-purpose processor, which can perform quite complicated computations, the cell can only do a little in itself. Vast parallelism means there are a great number of cells working in parallel at the same time, with the number of cells often measured by the exponential notation 10^x. Locality means the connectivity pattern between cells is local, and a cell can only communicate with a small number of other cells that are physically close by, and the connection lines usually carry only a small amount of information. According to this last principle, any single cell cannot have a global view of the entire system. This means there is no central controller in the cellular computing paradigm unlike in the von Neumann architecture. The three principles are highly interrelated [21]. Only the combining of all the three principles can result in the definition of cellular computing. For example, the attainment of vast parallelism is notably facilitated by the cells' simplicity and local connectivity.

Cellular automata (CA) are the example of cellular computing as they exhibit the three notable features of massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). CA were originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behaviour of complex, extended systems [23]. Cellular automata are mathematical idealizations of physical systems in which space and time are discrete [110]. A cellular automaton consists of a regular grid of cells, each of which can be in one of a finite number of 'N' possible states, updated synchronously in discrete time steps according to a local, identical interaction rule [21]. The state of a cell is determined by the previous states of a surrounding neighbourhood of cells [24][25].

3.3.2 Cellular Computing in "Bacterium Model"

In the developed system, the idea is to design a cellular adaptive network. To do this, the ways of implementing flexible cellular nodes have to be explored. The bacterium inspired solution makes each node within the network responsible for its own behaviour. The system was developed as an active network comprising of connecting cellular nodes (cells). Each node is relatively simple and has the same adaptive function. Each node is local, connecting with only very few other nodes (possibly its neighbour nodes). The network is thus modelled as a community of cellular automata. Each member of the community is selfishly optimising its own (local) state.

A Cellular automata consists of a n-dimensional uniform lattice of sites (cells) which may in principle be infinite in extent [110]. Real networks, like the Internet, include a very large number of nodes. In contrast, the hardware-implementation of the bacterial system developed in this thesis only implements a small number of nodes because of the limitation of the experiment environment in lab. However, this model is scalable and thus extendable (this is essential if it is to be used to model parts of a realistic network). These properties are consistent with the definition of cellular computing. This is why "Bacterium Model" was modelled as a community of cellular automata.

3.4 Concurrent Programming and Parallel Processing

Concurrent programming involves the notations for expressing potential parallelism so that operations may be executed in parallel and the techniques for solving the resulting synchronization and communication problems [28]. Notations for explicit concurrency are a program structuring technique while parallelism is mode of execution provided by the underlying hardware. Thus parallel execution can be implemented without explicit concurrency in the program, such as the case of a distributed system in which many processors are executing in parallel. Also a program contains concurrency can be implemented without parallel execution such as the case when a concurrent program is executed on a single processor. The concurrent operations expressed in the program can only be executed by interleaving executions in the processor. This section introduced briefly the concept of concurrent programming, parallel executing and other related subjects.

3.4.1 Concurrent Programming

A sequential or single threaded program consists of a series of steps that are executed in sequence, one after the other [27]. It specifies sequential execution of a list of statements and the execution is called a process [28]. A concurrent, or multithreaded program is one in which several things can happen simultaneously. It specifies two or more sequential programs that may be executed concurrently as parallel processes. This is important because much of the programming world involves concurrent applications. They are used in operating systems, networking, real-time systems, databases and multimedia systems [111][27]. Concurrent programs are increasingly important, whether these programs are stand-alone programs for a single machine system or used in distributed computing systems.

When discussing concurrent program, several co-related concepts need to be distinguished from each other. These include parallel programs and distributed programs. David W. Bustard gave appropriate definitions to these terms in [28]: a concurrent program defines actions that may be performed simultaneously; a parallel program is a concurrent program that is designed for execution on parallel hardware; a distributed program is a parallel program designed for execution on a network of autonomous processors that do not share main memory.

As a concurrent program contains two or more processes, one important problem involves the synchronization and communication between different processes. Two processes are said to communicate if an action of one process must entirely precede an action of a second process. Synchronization is related to communication. In fact, the communication method is one of the most important characters of concurrent languages.

Concurrent languages have been developed for a long time. A list of such languages can be seen in [30] that includes Occam, Ada, Java, Pascal-FC (Pascal – extension), SDL, MSC, HardwareC, Handel-C etc. The document highlights the strength and weaknesses of a selection of concurrent languages for the specification of the concurrent parts of the process. Here is a brief summary to some of the languages following this document.

Communicating Sequential Processes (CSP) [32] is a formalism developed by Professor Hoare of Oxford University that is a mathematically based method of proving and describing concurrent systems. Occam [26] was developed from the CSP formalism by Hoare. It has directives to determine what should be executed sequentially and what should be executed in parallel. Communication between processes is by channels. The processes communicating synchronise at the communication point. The language supports monitors, but discourages any form of shared variables. Ada was commissioned by the US Government and standardised in 1983 for use in embedded systems. Communication between processes is implemented by defining the interface of the task (which Ada calls a process) and then defining the body of the process. Ada does not support any of the concurrent structures like semaphores, mutexes etc, but provides it's own version of monitors. Java is multithreaded by virtue of its underlying architecture. The threads are scheduled by the Java Runtime system, in a priority based scheduling system. Communication is via method calls and is non-blocking unless declared synchronized. Java has underlying monitors and mutexes, but does not support shared memory.

Handel-C is a concurrent programming language that was designed for compiling programs into hardware implementations [31]. It is based on the ANSI C software programming language standard with extended features being added for parallel realization of computations in a manner that directly supports the concurrent nature and specific features of digital hardware [112]. The parallelism expressed within this language is explicit and conforms to the CSP model. This model allows the users to describe systems as a number of components (processes) which operate independently and communicate with each other over well-defined channels. As the recent popularity of object-oriented programming demonstrates, such an approach fits the structure of many problems extremely well, making

CSP a powerful notation for modelling systems, especially those which are inherently parallel.

While Handel-C provides an alternative method to hardware design compared to other methods of hardware design, such as Schematic Capture methods and Hardware Definition Languages (HDL), it is not designed to be a hardware description language, but a high-level programming language designed for compiling programs into hardware images for FPGAs or ASICs implementation.

As the demonstration system developed in this thesis includes both algorithmic and network/structural concurrency (which was explained in detail in Chapter 4) and is going to be implemented in FPGA based hardware, Handel-C is an appropriate language candidate for describing such a system. The later design and implementation has demonstrated this.

3.4.2 Parallel Processing

Parallel processing involves a wide range including the parallel system in which multi-processors are executing in parallel and sharing the main memory and the distributed system in which a network of autonomous processors that do not share main memory are executing in parallel. As long as a parallel process environment was provided, then parallel executing can be implemented. Parallel processing also involves the concurrent operations found in a hardware circuit. A hardware system may consist of various components executing asynchronously and simultaneously. These components can interact with each other by reading and updating signals [113]. Hardware description language like VHDL [114] provides concurrent statements to model these parallel entities and signals to describe the interactions. Various types of concurrent statements are provided to make modelling easier in VHDL, such as the 'process' statement and the 'component instantiation' statement. In VHDL, concurrency is used to model the way real circuits behave. For example, consider the case of two gates whose inputs change and each evaluates its new output independently of the other. There is no inherent sequencing governing the order in which they are evaluated. Each of these concurrent statements form an asynchronous parallel executing entity that may interact with other such statements to model the behaviour of the design.

This thesis concentrates on the parallel processing operations described by Handel-C language and their implementation on FPGA hardware. The most notable feature of Handel-C is its true parallel processing implemented through the use of directives which instruct the compiler to create multiple, concurrent blocks of code that will be synthesised into concurrent hardware entities [34]. It is possible to explicitly declare blocks of code that are processed in parallel. Handel-C includes commands for using and coordinating this feature in a real-time, hardware environment. The keyword 'par' around a code block means that each statement listed within the block will be executed in parallel (table 3.1). For example, if there are two assignments listed within a 'par' block, they will be executed literally in parallel in a multiprocessing software environment and will be synthesised into concurrent hardware —this is not the time-sliced pseudo parallelism of a conventional microprocessor implementation but rather two specific pieces of hardware build to perform these two assignments. Because FPGA can be configured to have a parallel architecture, or it can perform operations in parallel, this means an algorithm targeted at an FPGA can exploit parallelism.

Sequential Expressions	Parallel Expressions

`{`
`.....`
`a = 1;`
`b = 2;`
`.....`
`}`
	`par`
	`{`
	`a = 1;`
	`b = 2;`
	`}`

This executes the two statements, one after the other sequentially	This executes both statements in parallel

Table 3.1 Comparison between sequential and parallel expression

When expressions are evaluated in parallel, communication between the parallel processes becomes a problem due to synchronisation. Channels are provided in Handel-C to communicate between processes. Figure 3.5 illustrates the communication. If process 1 reaches the point A before process 2 reaches the point B, then process 1 is made to wait at the point A until process 2 reaches the point B in execution and visa versa. This is achieved by the following constructs "put" or "!" and "get" or "?" (Table 3.2). In each case the sender or receiver is made to wait if there isn't a receiver or sender at the other end of the channel.

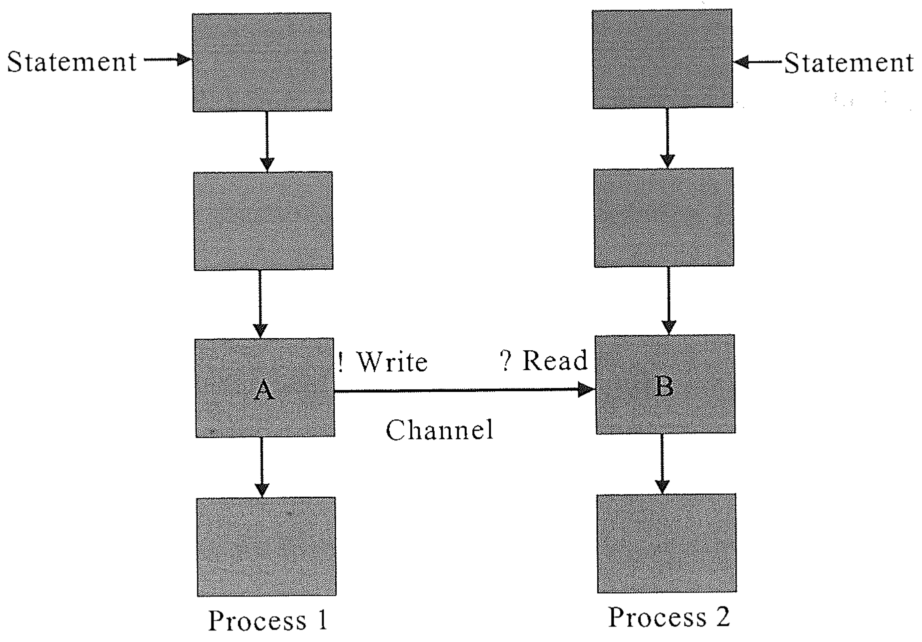


Figure 3.5 Communication between parallel processes

Channel ? Variable	This reads a value from the channel and assigns it to the variable
Channel ! Expression	This writes the value of evaluating the expression to the channel

Table 3.2 Channel communication expressions

It follows that using explicit concurrent or parallel statement in a design will result in parallel executing when targeting into hardware, thus gaining better performance for the implemented design. In the design, all the parallel tasks can be processed or executed simultaneously. Like parallel computing in a parallel or distributed system, physically simultaneous processing involves multiple processing elements (PEs) or independent device operations [29] and gains better performance through parallel processing. In the case of a FPGA, parallel statements or processes means more resources of the chip will be used to generate the separate hardware for each parallel process, i.e., pieces of hardware will be built to perform the parallel processes.

3.5 Conclusion

This chapter reviewed some important concepts, theories and technologies that were involved in the design of prototype adaptive nodes for an active network. As the purpose of the research is the investigation of the management solution for the future active network, the concept of active network was introduced first. Active network is the network where the network node is programmable and can be programmed by any user of the network. Investigation of the management solution for the network results in the study of a GA. GA is one of the branches of the evolutionary computation and is used to model natural processes of evolution using selection, mutation, and crossover operations. It provides an adaptive control solution for many applications. The simulated network was modelled as a community of cellular automata that perform cellular computing and was implemented on FPGA based hardware, which in turn provides the re-configurable computing for the system. The system was described using Handel-C, a concurrent programming language that provides the ability to explicitly declare the parallel processes and thus implement the parallel executing on FPGA-based hardware as this kind of device can be configured to have a parallel architecture and perform operations in parallel.

Chapter 4 Design and Simulation of Bacterium Model

This chapter introduces the design and simulation of "Bacterium Model". The hardware description languages –VHDL and Handel-C were used to describe the system. A comparison was made between these two languages. Handel-C was chosen for the development of the design and emphasis was placed on the design, simulation and hardware implementation.

4.1 VHDL - Hardware Description Language

VHDL is an acronym that stands for VHSIC Hardware Description Language, where VHSIC is another acronym which stands for Very High Speed Integrated Circuits. It arose out of the United States government's VHSIC program. In the mid 1980's the U.S. Department of Defence and the IEEE sponsored the development of this hardware description language with the goal to develop very high-speed integrated circuits. VHDL [115] has become now one of industry's standard languages used to describe digital systems. The language can describe the behaviour and structure of electronic systems, and is particularly suited as a language to describe digital electronic designs that are implemented using ASICs and FPGAs as well as conventional digital circuits [116]. This section briefly introduces the knowledge about VHDL, its concept, application, benefits, and design flow.

4.1.1 Introduction to VHDL

VHDL is a programming language that is used to describe a hardware circuit. It is classified as a Hardware Description Language (HDL). Other HDL languages include Verilog and ABEL (Advanced Boolean Equation Language). Both VHDL and Verilog are powerful languages that are used to describe and simulate complex digital systems. ABEL is less powerful than the other two languages and is less popular in industry. The hardware description is normally used to configure a programmable logic device (PLD), such as FPGA, with a custom logic design.

VHDL is a high-level programming language which allows complex design concepts to be expressed as computer programs which capture the behaviour and/or the structure of a complex electronic circuit [115]. It can be used to take two different approaches to describe hardware: the behavioural and structure methods of hardware description.

The behavioural method describes a system in terms of what it does, or how it behaves, rather than in terms of its components and interconnection between them. A behavioural description specifies the relationship between the input and output signals, and more closely resemble a high-level programming language since it normally includes arithmetic and logic operations and constructs such as IF-THEN_ELSE, WAIT UNTIL, FOR loops. The behavioural method can be further divided into two kinds of styles: Algorithmic and Data flow. The algorithmic approach typically describes the behaviour in terms of a mathematical or signal processing algorithm which is transformed into a digital logic circuit using a synthesis tool. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer Level). The data flow model makes use of concurrent statements that are executed in an event driven manner as soon as data arrives at the input. The dataflow description is typically directly synthesised as parallel logic circuits.

The structural method, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a designed function. It involves the instantiation of components, such as logic gates, flip-flops, and higher-level modules made up of simpler ones. A structural description could be compared to schematic or interconnected logic gates. It is a representation that is usually closer to the physical realization of a system.

Although the VHDL language looks similar to conventional programming languages, there are some important differences between them. A hardware description language is inherently parallel. VHDL provides features (such as concurrent statements) allowing concurrent events to be described. This is important because the hardware being described using VHDL is inherently concurrent in its operation [115].

The general format of a VHDL program is built around the concept of BLOCKS that are the basic building units of a VHDL design. Every VHDL model is composed of an entity block and at least one architecture block. Entity describes the interface for the design by defining the input and output signal of the designed circuit. Architecture describes the internal operation of the design (the behaviour and/or the structure of the model). Figure 4.1 illustrates the structure of a VHDL program.

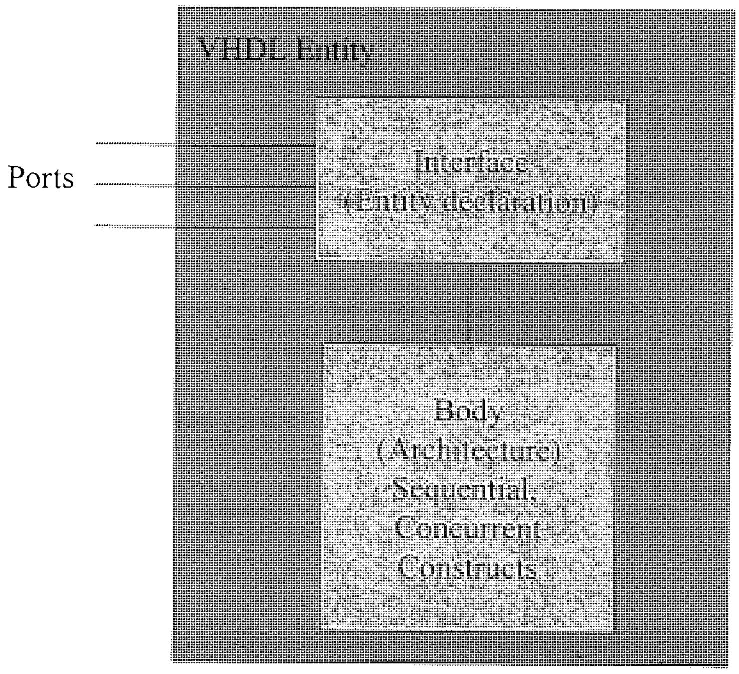


Figure 4.1 The format of VHDL program

After a design is created using VHDL, it can be simulated to verify the design. Simulation can be done at different levels. Behaviour simulation (or function simulation) is the one at the symbolic level that tests if the system's logic works according to design and concept. It is a logic-level simulation of the VHDL code used to validate the functionality against the specification. The designer should make appropriate revisions to the code at this stage if the logic is not right.

The VHDL description of a digital system can be transformed into a gate level implementation. This process is known as synthesis. Synthesis allows implementation-specific factors, such as timing and other influences of actual hardware devices (such as FPGA) to effect the simulation. A simulation after synthesis is called a gate-level simulation and provides a more precise type of check before the design is committed to the real hardware devices. Simulation can even be done after layout providing the specific design environment and tools exist. Generally speaking, VHDL is suitable for use today in the digital hardware design process, from specification through high-level functional simulation, and logic synthesis down to gate-level simulation and implementation on a target device [116].

One important way in VHDL to verify the specified functionality of a design is the use of 'test bench'. Test benches are VHDL descriptions of circuit stimulus and corresponding expected outputs that verify the behaviour of a circuit over time [115]. It is an integral part of any VHDL project and is created in parallel with other descriptions of the circuit. It provides the stimuli for the Device Under Test (DUT) and analyses the DUT's response or stores them in a file. Simulation tools visualize signals by means of a waveform that the designer compares with the expected response. If the waveform does not match the expected response, the VHDL source code has to be corrected by the designer.

4.1.2 Benefits of using VHDL

Using VHDL will bring a lot of benefits for the design [115][116]. First, like any high-level design language, it will improve productivity (shorten time-to-market) when designed using a structured, top-down approach. This is achieved through the development of the re-usable VHDL components or the libraries of commonly used VHDL modules. Second, as a standard language, the rapid pace of development in electronic design automation (EDA) tools and in target technologies bring another benefit when using VHDL. VHDL descriptions of hardware and test benches are portable between design tools, and portable between design centres and project partners [116]. To move a design to a new technology, the designer need not start from scratch or reverse-engineer a specification, just go back up the design tree to behaviour VHDL description and then implement that in the new technology knowing that the correct functionality will be preserved. Third, the behaviour simulation of VHDL can reduce the design time by allowing design problems to be found at an early stage, thus avoid a need to rework designs at lower level.

4.1.3 Design Flow using VHDL

Figure 4.2 shows a simplified design flow using VHDL. It includes both simulation and synthesis. The target chip may be the programmable logic or ASIC chips. Test development should begin as soon as the general requirements of the system are known [115]. In the diagram, VHDL is first used for design entry (it can be used along with other design entry methods such as schematics and block diagrams). After the system is captured, a functional simulation of the VHDL code can be performed to verify the function of the system at the symbolic level (or logic level). Simulation can also be performed after synthesis. This is the time simulation in the diagram. This post-synthesis simulation performs the simulation at gate level and thus provides a more precise verification of the system as it concerns the time characteristic. On the test development side, VHDL test benches can be created to verify that it meets the functional and timing constraints of the specification. These test benches can be entered using a text editor, or other forms of test stimulus information such as graphical waveforms. For accurate timing simulation of post-route circuits, a timing model generation program obtained from a device vendor or third party simulation model supplier may be used. Model generation tools such as this typically generate timing-annotated VHDL source files that support very accurate system-level simulation [115].

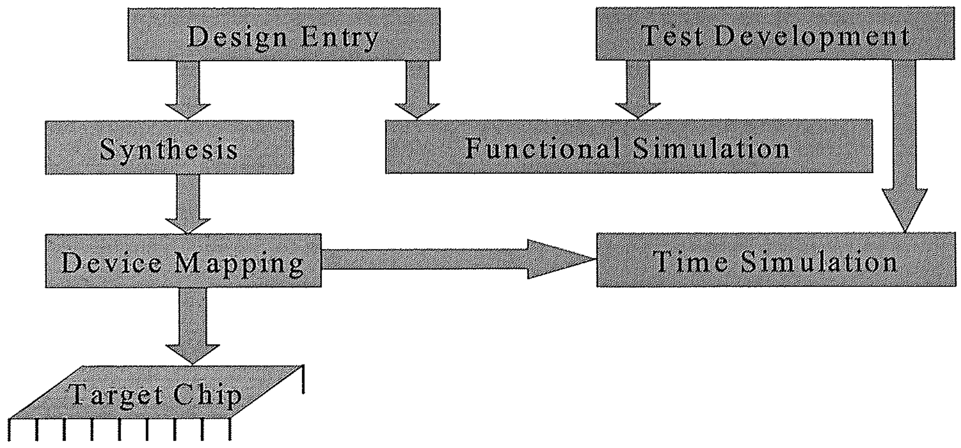


Figure 4.2 A simplified design flow using VHDL

4.2 Handel-C- Concurrent Programming Language

Handel-C is a high level programming language designed to enable compilation of programs directly in to hardware, which is implemented on FPGA or ASICs. The semantics of the language are based on Occam/CSP [117][118] and the syntax is based on the C language [119] with additional extensions to take advantage of the features of hardware. Handel-C provides an alternative method to hardware design, compared to other methods of hardware design such as Schematic Capture methods and traditional Hardware Description Languages (HDL) such as VHDL and Verilog [120].

4.2.1 Introduction to Handel-C

While register-transfer-level (RTL) subsets of HDLs, such as VHDL and Verilog, do provide a functional interpretation of the hardware description to enable the generation of hardware structure at compile time, their parallel nature requires the design engineer to add extra logic for sequential execution [121]. On the other hand, to use a language like VHDL, the designers need to understand the hardware knowledge and therefore was usually used by hardware engineers and not by a software engineer who know much more about traditional software high-level programming language such as C and C++ than hardware description languages such as VHDL and Verilog, whereas the FPGAs and PLDs are evolving into programming systems that require knowledge of both hardware and software. Further, to describe a system on a very high level of abstraction and thus describe the desired function in the briefest possible way, a traditional software programming language is more suitable and flexible than a traditional hardware description language which focus more on the underlying structural detail [121].

Due to these reasons, recent years have seen the development of languages that were based on traditional software programming language such as C and aims to allow designers with a limited hardware background to describe a system to be implemented in hardware. Handel-C is such a programming language designed to enable the compilation of programs into synchronous hardware and for implementing algorithms in hardware. This is because Handel-C closely corresponds with a typical software flow and provides the essential extensions required to describe hardware. The high level flow is geared for programming functionality, but can be compiled directly to hardware. It also facilitates architectural design space exploration, and hardware/software co-design. A high level approach such as the Handel-C methodology provides a level of abstraction that makes it easier for designer to address a larger design space and find the correct solution for making a design smaller and/or faster.

Handel-C is designed to be familiar to programmers, heavily resembling the C programming language but with semantics rooted in Occam. It closely corresponds with a typical software flow and provides the essential extensions required to describe hardware [121]. These mainly include flexible data widths, parallel processing and communications between parallel threads. The language also utilizes a simple timing model that gives designers control over pipelining without needing to add specific hardware definition to the models. The feature of flexible-width variables allows hardware to be created that does not waste resource, i.e. it doesn't make sense to have a 32-bit adder when the maximum number to add is only 15.

The parallelism expressed within Handel-C is explicit and conforms to the CSP Model that was popularised in the Occam programming language [117]. In the CSP model the system can be described as a sets of units of computation (processes) which operate independently and communicate with each other over a named communication path that is called a channel. When both the sending and receiving process rendezvous with one another then communication occurs in a lock-step nonbufferred blocking manner [112]. As the recent popularity of object-oriented programming demonstrates, such an approach fits the structure of many problems extremely well, making CSP a powerful notation for modelling systems, especially those which are inherently parallel.

An important feature of Handel-C is its very simple timing semantics. As the manual says, "Assignment takes one clock cycle, everything else is for free". This means expressions and the control logic for statements take no cycles. Only assignment a value to a variable takes a cycle. But channel communication may take anything from one cycle upwards. Since communication blocks until there are both a reader and a writer, a read or write to a channel must wait until it has a partner. In this case it may take an arbitrary number of clock cycles.

4.2.2 Handel-C and the Hardware Constructs (Synthesis)

Handel-C captures programs at the level of simple transformations on data and movement of data between variables [122]. Following [122], some of the hardware constructs used in the transformation of Handel-C programs to a net-list graph are introduced. First, all variables in the user program are mapped to hardware registers. These are constructed from sets of flip-flops; one flip-flop is built for each bit of a variable in the program.

Second, Handel-C includes an explicit parallel construct, the 'par' statement, which allows parallel execution of blocks of code. 'Par' statements are executed concurrently and synchronized at the block end. When a parallel block of code is encountered, execution flow splits at the start of the block and each branch of the block executes simultaneously. Execution flow then re-joins at the end of the block when all branches have completed. The instruction following the par statement will not be executed until all branches of the parallel block completed [34]. Figure 4.3 shows the parallel executing in Handel-C.

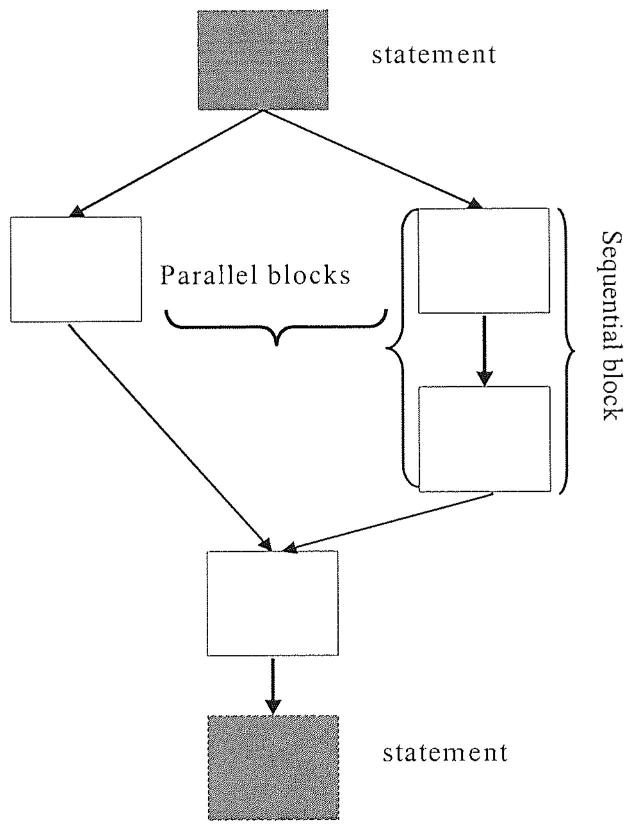


Figure 4.3 Parallel executing of Handel-C

This kind of parallelism is not the time-sliced pseudo parallelism of conventional microprocessor implementation but rather several pieces of hardware built to perform each block of the code, i.e., the hardware for each block listed in the "statement" will be generated.

Following [122], each construct in the Handel-C program maps onto a control circuit in the hardware. Figure 4.4 shows the control construct used for parallel execution of statements using **par {s1, s2, s3}**; This circuit passes control simultaneously to all of the parallel statements s1, s2, s3 to initiate parallel execution. The **par {}** construct terminates only when all of the constituent components have terminated. Thus the parallel control circuit collects together the separate finish signals in a set of flip-flops and produces its own finish signal as soon as the last finish signal is generated.

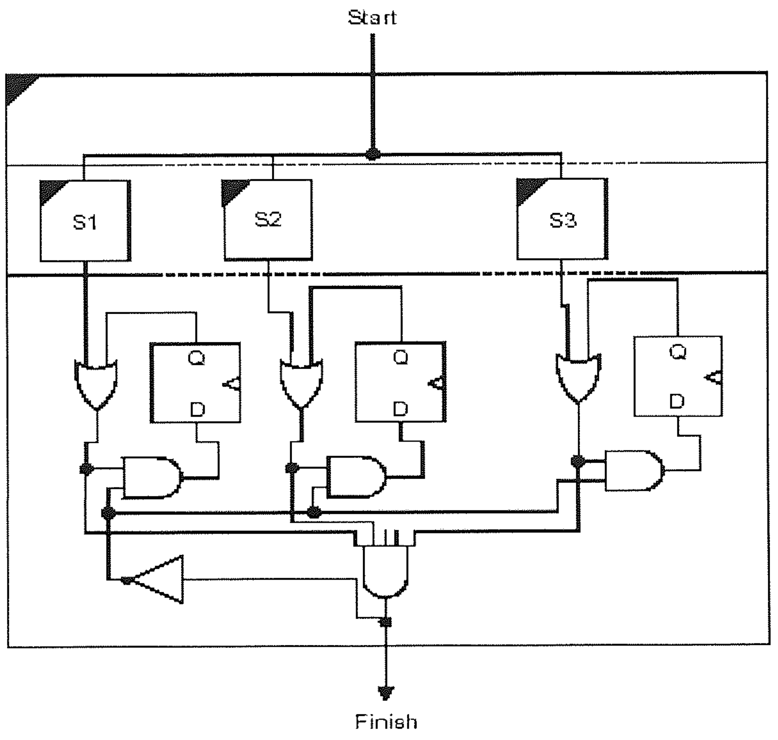


Figure 4.4 Parallel composition

The following code is an example of using "par" statement. Its execution flow is show in figure 4.5.

```
par
{
    x=1;
    y=2;
    z=3;
}
```

When compiled to hardware, three specific pieces of hardware will be built to perform the above three assignments, i.e., the hardware for each assignment will be generated. The control circuit for the assignment statement such as "**R** = **Exp**;" is shown in figure 4.6. The start signal forms the clock enable signal for the destination register of the assignment. At the end of the cycle in which the assignment is scheduled the expression hardware has calculated the new value, which is thus loaded into the destination register.

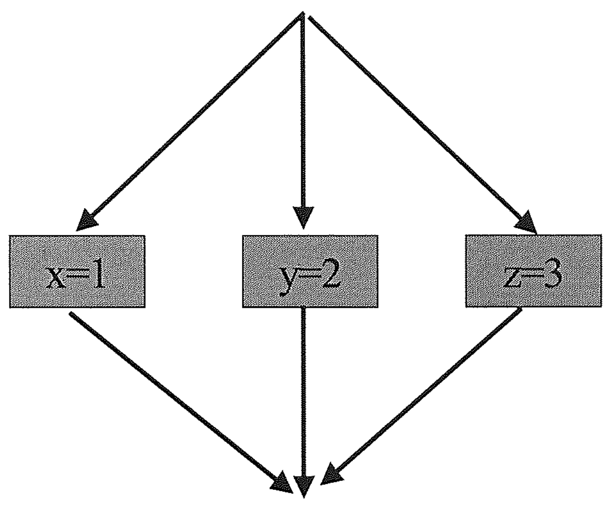


Figure 4.5 Executing flow of a simple parallel code example

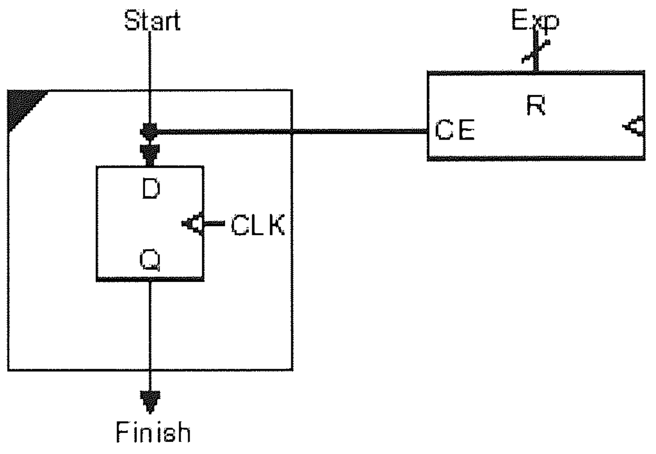
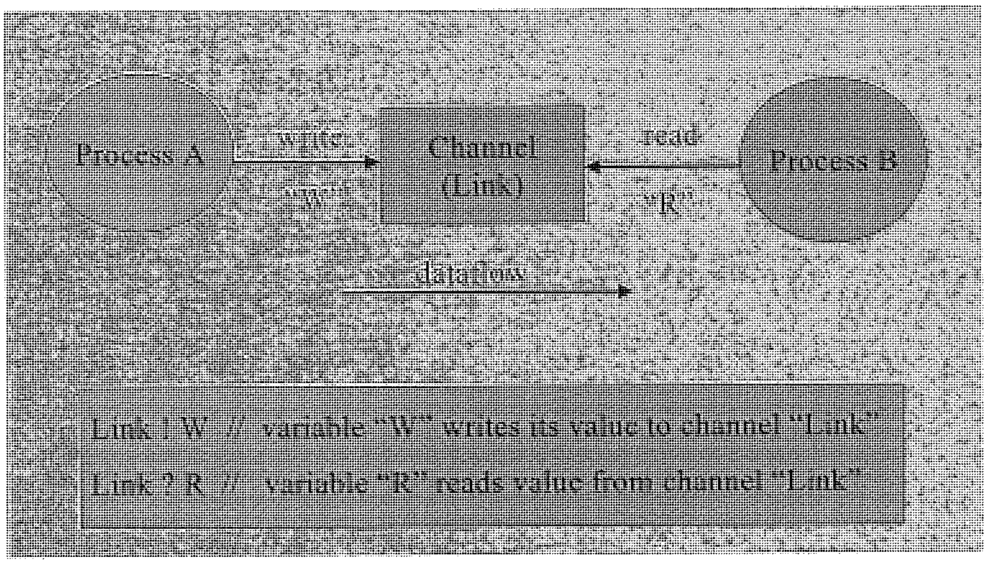


Figure 4.6 Implementation of assignment

Third, parallel blocks can communicate synchronously with each other using channels defined in Handel-C. They are used to communicate between concurrent asynchronous processes. One branch writes to the channel and a second branch reads from it. The communication only occurs when both tasks are ready for the transfer, at which point one item of data is transferred between the branches, thus enforcing synchronisation. There are associated operators that are used to read ('?') from a channel and write ('!') to a channel [15]. Figure 4.7 shows the communication between two parallel branches through a channel. The code example is included in the diagram.



```
Link ! W  //  variable "W" writes its value to channel "Link"
Link ? R  //  variable "R" reads value from channel "Link"
```

Figure 4.7 The channel communication

Figure 4.8 shows the control (synchronization) circuitry for channel input and output [122]. The circuit is the same for the channel input "**c ? var**" and output "**c ! Expr**" commands. Synchronization is achieved by looping back the start signal through a flip-flop and a multiplexor controlled by the transfer signal. The 'Ready' signal goes to the arbitration circuit (figure 4.9) ('ip.rdy' for channel input circuit and 'op.rdy' for channel output circuit) and returns as the 'Transfer' signal to indicate when the partner to this communication is also ready to run. When both circuits are ready to communicate, the circuit below the dashed line in figure 4.8 is activated. This is simply the assignment circuit seen earlier. These diagrams illustrate that channel communication is just distributed, synchronized assignment.

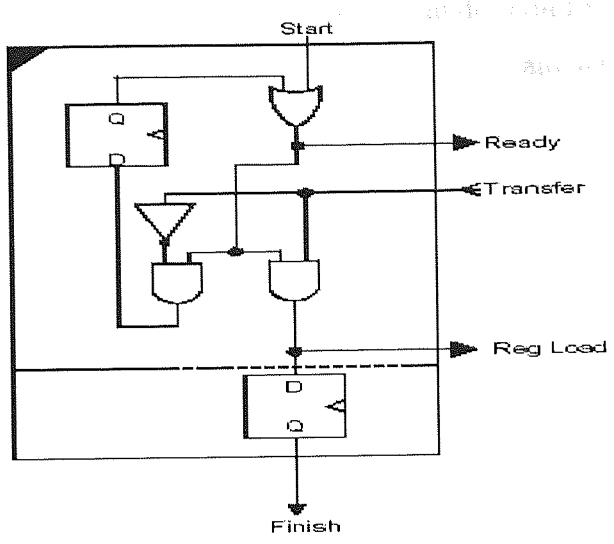


Figure 4.8 Channel communication synchronisation construct

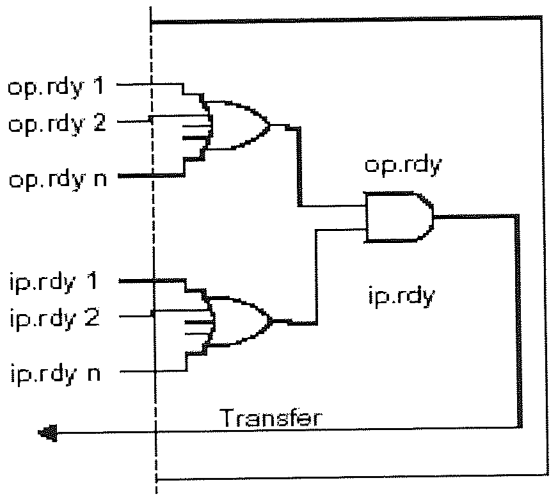


Figure 4.9 Channel communication arbitration construct

4.2.3 Design Procedure Using Handel-C

The Handel-C design procedure is described in Figure 4.10 [120]. First, the user writes the program in Handel-C. The Handel-C complier then compiles and optimises Handel-C source code into a file suitable for simulation [123]. The simulator allows the user to test the program without using real hardware. If the simulation result is not correct, the user needs to debug and modify the source code. Using the simulator tool for debugging, the state of every variable in the program can be displayed at every clock cycle if required, the simulation steps and the number of cycles simulated is under control. Optionally the source code that was executed at each clock cycle as well as the program state may be displayed in order to assist in the debugging of the source code [123]. The next step is to compile the program again for the hardware synthesis. Before that, the program may need to add the necessary interfaces for hardware. The result of re-compile and target a specific hardware is the output of a Netlist file for the FPGA tools to place and route. After placing and routing a configuration file is created to program the FPGA.

This is a simple introduction of the design procedure followed in Handel-C. The later chapter will illustrate the detailed procedure through the development and implementation of "Bacterium Model", from the source code design to the real hardware implementation on specific FPGA devices.

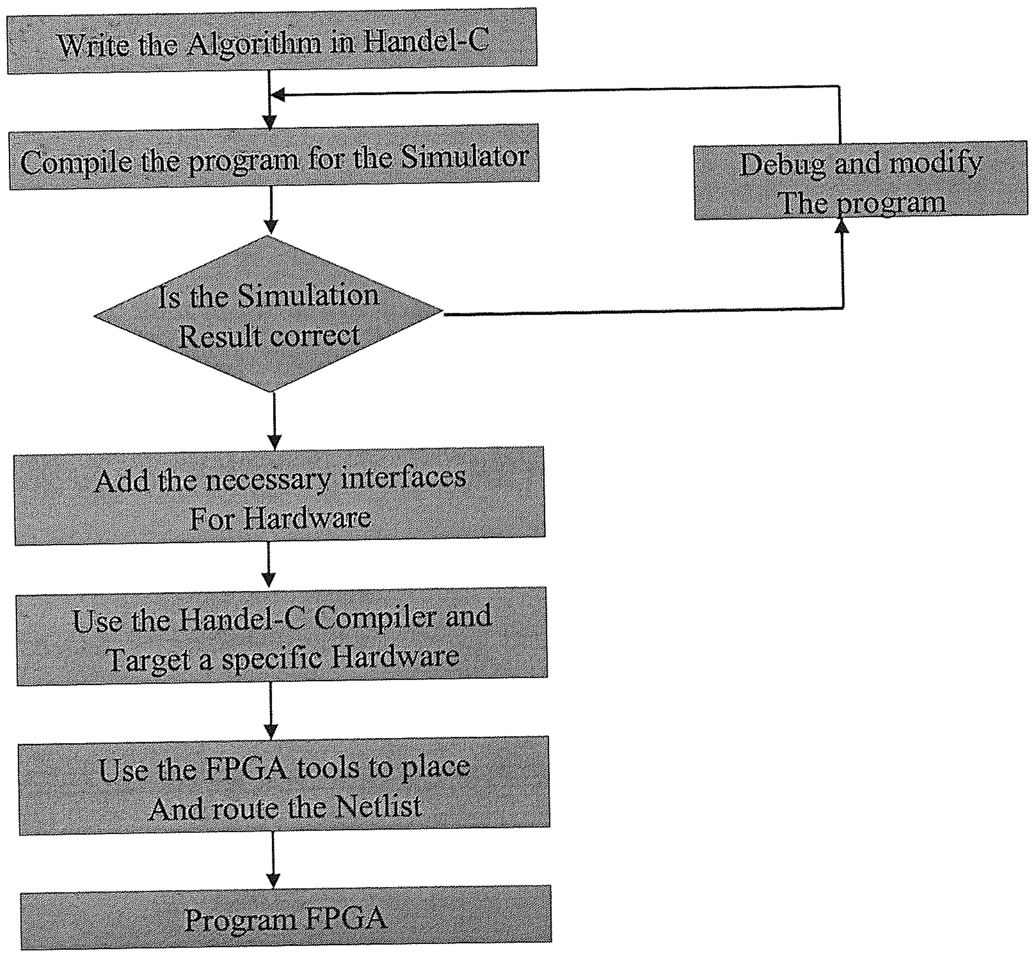

Write the Algorithm in Handel-C
Compile the program for the Simulator
Debug and modify The program
Is the Simulation Result correct
Add the necessary interfaces For Hardware
Use the Handel-C Compiler and Target a specific Hardware
Use the FPGA tools to place And route the Netlist
Program FPGA

Figure 4.10 Design procedure followed in Handel-C

4.2.4 The Benefits of Using Handel-C

The most obvious benefit of using Handel-C is its high-level language based design method rather than a low-level language based method. Handel-C provides a formal and provable way of controlling the levels of design abstraction implied by the description. This is because Handel-C is very similar to the traditional software and thus very flexible for the description. This makes it possible for the designer to control the level of abstraction when describing a system. That's why Handel-C compares favourably against most of the traditional Hardware Description Languages, in terms of ease and flexibility of development. This is important because of the well know fact that system complexity is increasing and the hardware implementations are getting more and more complex. The need for high-level design languages such as Handel-C increases, in order to meet the increasing design requirements, i.e., the development of Handel-C language is one of the approaches to cope with increased complexity.

Handel-C augments the Software-Compiled System Design methodology by allowing hardware designers to take advantage of software design techniques in the implementation of complex algorithms. The Handel-C constructs maintain the designer's ability to get predictable results from the resulting hardware. On the other hand, the language has been designed primarily for use by a programmer and is designed such that the complex design decisions traditionally made at the architectural/hardware level by the hardware engineer do not have to be worried about by the programmer [30]. However, the programmer still needs to have the ability to exploit concurrency, and should not be just familiar with only the traditional 'sequential' way for programming.

In a word, Handel-C provides a familiar language with formal semantics for describing system functionality and complex algorithms that results in substantially shorter and more readable code than RTL-based representations.

4.3 VHDL and Handel-C for Different Aims

Comparing Handel-C with VHDL shows that the aims of these languages are quite different [31]. VHDL is designed for hardware engineers who have the knowledge of hardware. It provides the constructs necessary for complex, tailor made hardware designs. The specialist can specify a single gate or flip-flop built and even manipulates the signals' propagation delays throughout the system. This can be completed through choosing the right elements and language constructs in the right order. Thus the hardware engineer who uses VHDL has to have the knowledge of low-level hardware and have the gate-level effects of every single code sequence in his mind all the time. This requirement distracts the designer's attention from the actual algorithmic or functional subject.

The important advantage of VHDL is that it can accommodate both algorithmic design and the low-level structure design. However, the algorithmic VHDL cannot incorporate concurrency in the natural way that Handel-C does. In VHDL, concurrency is typically implemented using (i) dataflow statements which are intrinsically concurrent, (ii) communicating VHDL processes, and (iii) blocks in a structural VHDL design.

On the contrary, Handel-C is not designed to be used only by the engineer who has the hardware knowledge. As it is a high-level programming language with hardware output, even a software engineer can use it to describe a hardware design. The language doesn't provide highly specialised hardware features and allows only the design of digital, synchronous circuits [31]. It focuses on fast prototyping and optimising at the algorithmic level. The low-level problems are hidden completely from the designer. The gate-level decisions and optimisation are made by the compiler and the designers only need to focus their mind on the task being implemented. Thus the hardware design using Handel-C resembles more to programming than to hardware engineering. In fact this language is developed for programmers who may have no hardware knowledge at all, but need to have the knowledge of concurrency concept, not only the conventional sequential method to program.

4.4 Development of "Bacterium Model" Using VHDL

Chapter 2 introduced the principle of "Bacterium Model" in which a GA based on "bacterium-learning" is used to implement the management of a network containing active and adaptive nodes. To describe such a system, a powerful and flexible hardware description language is needed. VHDL is one of the choices of many hardware description languages to describe the system whose features and benefits have been introduced in the former section. This section addresses the development of "Bacterium Model" using VHDL. The designed system implements a model that contains only one node. More nodes system can be implemented through increasing of the nodes number and adding connections between the nodes.

The "Bacterium Model" is proposed and designed as a system prototype to implement an active network in which a bacterium inspired GA is employed to provide the adaptive management. The main functions of such a model involve creating requests for services and their input to the network, the acceptance and handling of these requests, and the GA based evolution of the rules by which the node decides how to handle the requests (i.e., implementation of the GA). To simplify the design, the VHDL version of the GA only involves the migration algorithm (the mutation algorithm is not included at the moment). Before the system is designed for each node, an analysis of the function structure should be considered.

According to the principle of the model, each node should have an engine to provide a sequence of requests for services, a FIFO queue to store the incoming requests, a mechanism to decide if each incoming requests match one of the node's rules, a processor to process the matched requests, a mechanism to calculate the busyness of the node (how busy it is), a decision making mechanism to decide how to change the rules based on the value of the busyness and the queue length, and a rule set storage. In addition, as the GA of plasmid migration involves the exchange of the rules between nodes through a common-accessed "rule pool", a rule pool storage mechanism should be provided. Based on the above analysis of the system function, the structure of the system for each node can be partitioned as the following function units: a request generate unit, a FIFO queue unit, a request match unit, a request process unit, a busyness calculation unit, a fitness evaluation unit, a local node rule set unit, a rule pool unit and a clock generate unit. The clock generate unit is needed as it provides all kinds of clocks that the system may need to run.

Such a partition of the system function into several sub-functions not only gives each unit a clear and unique function specification, but also provides a clear connection between different units. The following section describes the purpose of each unit and its design in detail.

4.4.1 Request-Generate-Unit

The purpose of this unit is to create the sequence of requests for services for each node. In the model four kinds of requests are named as "A", "B", "C" and "D", and are represented as a vector signal and encoded as follows "00", "01", "10" and "11". As the sequence of requests is a random sequence of these four vectors, the key issue of the design is the generation of a random number that takes the values between "00" and "11" (including these two numbers). One of the random number generators is the Linear Feedback Shift Register (LFSR) method that has been used by Maruyama et al [124]. Figure 4.11 shows the schematic of the LFSR used in this work, where a sequence of 2^n-1 random numbers between 1 and 255, or the corresponding vector between "00000001" and "11111111" was created. Eight D-type flip-flops are concatenated together as shown in the figure 4.11. Two 8-bit vectors are defined in the design. One is the input vector of D, defined as **signal** D: **bit_vector** (7 **downto** 0), another is the output vector of Q, defined as **variable** Q: **bit_vector** (7 **downto** 0). As the figure shows, the input to the first bit of vector D, the value of D0 can be gained through the following equation:

$D0 <= Q3 \oplus Q4 \oplus Q5 \oplus Q7;$ where $\oplus$ is the exclusive-OR operator

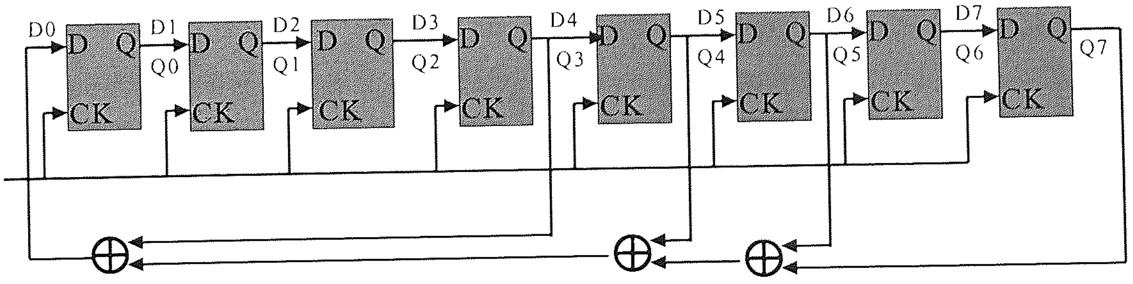


Figure 4.11 A random number generator

Such a circuit will produce a sequence of 2^n-1 (i.e., 2^8-1=255) pseudo random number and the taps Q3, Q4, Q5, Q7 decide the maximal length sequence. As each clock come as an event, the output vector of Q will take the value of a random vector ranged from

"00000001" to " 11111111" (i.e. an integer in the range 1 to 255). As the "Request-Generate-Unit" requires only four types of services, that means a random number sequence with only four numbers, suppose 0, 1, 2, 3 is needed. An obvious way to implement this is to divide the whole range of the number from 1 to 255 into four sub-ranges. Let numbers 1 - 63 belong to the first range, 64 – 127 belong to the second range, 128 –191 belong to the third range and 192 – 255 belong to the fourth range. In such a way, whenever a random number between 1 and 255 is generated, if the number belongs to the first range, a new random number of "0" is generated as the output. In a similar way, three other random numbers of "1", "2" and "3" are generated. As a result, a new sequence of random number that includes only four numbers is created. These four random numbers can be used to represent the four different types of requests for services in the network. Suppose the output port that represent the generated requests is defined as a two bit vector that take the value between "00" and "11" and has a name of "generated-requests", the following example shows the code for implementing the above described function.

```
if    Q < "01000000"   then  generated-requests <= "00";
elsif Q < "10000000"   then  generated-requests <= "01";
elsif Q < "11000000"   then  generated-requests <= "10";
else  generated-requests <= "11";
end if;
```

The frequency of the clock used in this unit will decide the speed of the request generating and thus decide the load of the network. The higher the clock frequency, the higher the network load. If an external fixed clock is used as the clock input, the load will be fixed. To make the load changeable, an extra circuit is needed to generate a request-generating clock with a different frequency. Such a design suppose the requests are injected into the network all the time at the request generating speed, the case of "no request" is not included in the system design.

4.4.2 FIFO-Queue-Unit

The purpose of this unit is to implement a FIFO (first in first out) queue. The elements of the queue are the incoming requests generated from Request-Generate-Unit. In the design the max queue length is defined as a constant "max-queue-length" that has the value of 200 (which is the dimension chosen by the original proposal). An "**array**" type signal named "queue-array" that has the range from "0" to "max_ queue_length-1"is defined to represent the queue. Three other parameters are defined: the current queue length named "current-queue-length" (it may take the value from 0 to 200, 0 means the queue is empty and 200 means the queue is full), the new coming request named "queue-in" (which will become the first element of the array indexed by 0) and the element named "queue-out" that to be output from the queue to the next unit (which is not necessary the last element of the array as the current queue length may not equal to the max queue length. The last element in the array is indexed by max_queue_length-1 and the "queue-out" is the element in the array indexed by current_queue_length-1). Figure 4.12 shows the structure of the queue.

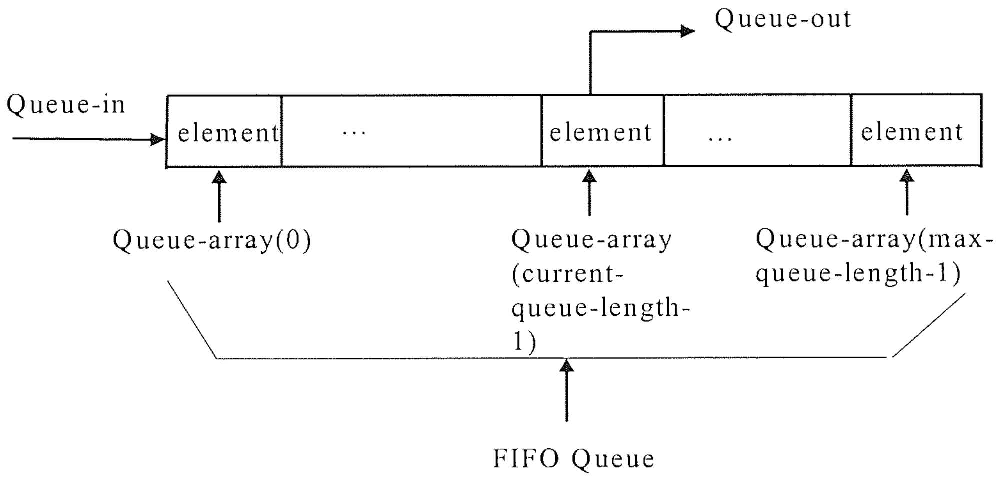


Figure 4.12 FIFO queue

The main functions of a queue include two aspects: an element is written into the queue and read out from the queue. The following description shows how the queue is designed to implement these two aspects of functions.

(1) Whenever there is a new request coming in (an element is put into the queue) and if the current queue length is less than the max queue length (the queue is not full), the current queue length is increased by 1. All the elements of the array will take the value of its previous element one by one and the new coming request becomes the first element of the array. The following example shows the code for implementing the above described function.

```
if current-queue-length < max-queue-length then
```

```
    current-queue-length<=current-queue-length+1;
    for i in max-queue-length-2 downto 0 loop
    queue-array (i+1) <= queue-array (i);
    end loop;
    queue-array (0)<=queue-in;
  end if;
```

(2) Whenever there is a need that a request should be output to the next unit (an element is read out from the queue) and the current queue length is larger than 0 (the queue is not empty), the last element of the current queue becomes the "queue-out" element (request to be output to the next unit), i.e., the "queue-out" will take the value of the last element of the current queue. The current queue length is decreased by 1. The following example shows the code for implementing the above described function.

```
If  current-queue-length >0 then
   queue-out <= queue-array (current-queue-length-1);
   current-queue-length<=current-queue-length-1;
end if;
```

The above method of controlling a FIFO queue used the sequential VHDL constructions of "if-then" and "loop" statements. Its' synthesized logic would be the combinational logic according to [114].

4.4.3 Request-Match-Unit

The purpose of this unit is to compare the request at the front of the queue with all the available rules of the node and see if there is a match. If the request matches one of the rules it will be output to the next unit (Request-Process-Unit) to be processed, if not, the request will be forwarded to a neighbour node.

First of all, the rule has to be expressed in some kind of data type. As each rule involves three aspects (as defined in chapter 2): x, represents the type of service requested, y, the threshold of queue length for comparison and z, the threshold of busyness for comparison. In the design, the rule is defined as a record type that includes three elements, service request type (.request), queue length (.queue-length), and busyness (.busyness). The definition of the rule type is showed as the following example:

```
type rule is record
request: service-type;
queue-length: queue-length-range;
busyness: busyness-range;
end record rule;
```

In the above example, "service-type", "queue-length-range" and "busyness-range" separately defined the date types of the three elements. To decide if a request matches one of the rules will involve the comparison of all these three elements. The direct way of expressing such a behaviour is to use the VHDL sequential '**if**' statements. A symbolic code to implement this is shown as following.

```
If    in-request =  rule.request and
      Current-busyness<rule.busyness and
      Current-queue-length<rule.queue-length
then out-request<=in-request;
else request-to-neighbour<=in-request;
end if;
```

Consider the behaviour from the hardware perspective [114], one three input AND gate will be synthesized for the '**if**' statement and a demultiplexor will be synthesized to assign the value to one of two branches ('**then**' or '**else**') based on the value of the output of the AND gate. Figure 4.13 shows the synthesis for the above code comprised of if-then-else statements. The code expresses the simplest algorithm to implement the behaviour and thus provide the efficiency for the computation.

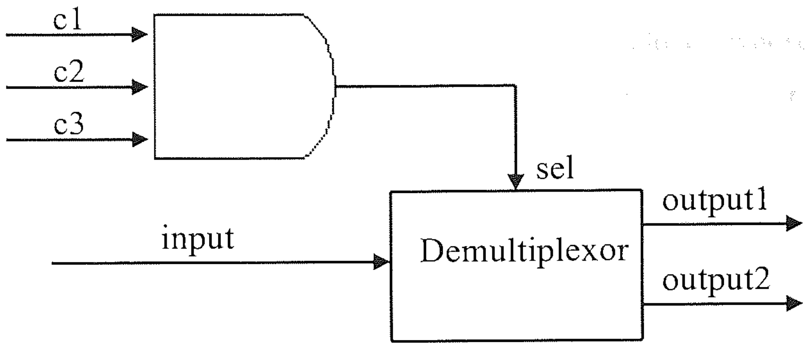


Figure 4.13 Synthesis for the Request-Match-Unit

For the above code, 'c1' can be imaged to "*in-request = rule.request*", 'c2' can be imaged to "*Current-busyness<rule.busyness*" and 'c3' can be imaged to "*Current-queue-length<rule.queue-length*". Similarly, the variable 'input' is imaged to "*in-request*", 'output1' and 'output2' is imaged accordingly to "*out-request*" and "*request-to-neighbour*".

4.4.4 Request-Process-Unit

The purpose of this unit is to act as a processor that is used to process the matched requests coming from the 'Request-Match-Unit'. Since the purpose of the developing system is to demonstrate the efficiency of an adaptive GA on the management of an active network, the kind of processing that is carried out is not important. For example, the processor may just hold the request for some time (represent the process time) and then release or delete it. This is the design adopted in the 'Request-Process-Unit'. The behaviour description is then summarised as a very simple one: accept the coming request, wait for some while (process time) and then output it. In the design, it is defined that there must be no more than four requests being processed in each epoch. This is from the original proposal that decide the number to be four in [11].

The key issue in the design for this unit is the "wait" for the process time of each request and the guarantee that no more than four requests are processed in each epoch. As the direct '**wait**' statement cannot be synthesised, the design used a pre-generated clock named "process-time-clock" whose half-period is the same as the process time of each request (suppose each request service need the same process time). Thus whenever a request is received, it is released (output to a port) as the "process-time-clock" signal changed its value. In a similar way, a pre-generated clock named "epoch-clock" whose half-period is defined as the epoch length is used to make sure at most four requests are processed in each epoch.

The "process-time-clock" and the "epoch-clock", as well as the "request-generating-clock" used in "Request-Generate-Unit" are generated in a unit named "Clock-Generation-Unit". This unit generated all the clocks that were used in the design for different purposes. In addition to the above three clocks, there are other two clocks named "evaluation-clock" and "four-evaluation-clock" also generated in the unit and used in the later processes.
As the behaviour description of this unit is also implemented using the sequential '**if-then-else**' construct as in the previous unit, the synthesis for this unit will still be the combinational logic.

4.4.5 Busyness-Calculation-Unit

The purpose of this unit is to calculate the value of "busyness". The busyness is calculated by combining the busyness at the previous epoch with the busyness for the current epoch in a 0.8 to 0.2 ratio (these numbers again come from the original proposal and were defined in [11]). The equation to calculate the busyness is then expressed as the follows:

*busyness<=**integer** (previous-epoch-busyness*4/5)+*
***integer** (current-epoch-busyness/5);*

4.4.6 Evaluation-Unit

The purpose of this unit is to evaluate the state of the node (busy or idle) as a function of the current queue length and busyness, and thus provides a basis for the later migration algorithm, which result in the exchange of the rules between the nodes through the common accessed "rulepool" environment. The principle of evaluating the node state and the corresponding migration algorithm can be described as follows:

- If the queue length or busyness is above a threshold (50 for both queue length and busyness in the design), a random section of the node's rule set will be copied into the rule pool;
- If the node continues to exceed the threshold for four evaluation periods, it replicates its entire rule set into an adjacent non-active node;
- If the busyness is below a different threshold (10 in the design), a rule randomly selected from the rule pool will be injected into the node's rule set;
- If a node is 'idle' for four evaluation periods, the node will be switched off;

Four output signal are then defined representing the states of the node: "node-state-busy", indicating if the queue length or busyness is above a threshold in one evaluation period ("evaluation-clock" is used here), "node-state-too-busy", indicating if the node has continually exceeded the threshold for four evaluation periods ("four-evaluation-clock" is used here), "node-state-idle", indicating if the busyness is below a different threshold and "node-state-too-idle", indicating if a node has been 'idle' for four evaluation periods. The input signals include the "current-queue-length" from "FIFO-Queue-Unit" that indicates the value of the queue length and the "busyness" from "Busyness-Calculation-Unit" that indicates the value of the busyness. The function of this unit is then to evaluate the state of the node and make it available to the "Local-Node-Rule-Set-Unit" and "Rule-Pool-Unit" which execute the migration of the rules.

The sequential statements of '**if-then-elsif-then-else**' are used for the behaviour description and the hardware synthesis will be complex combinational logic for the conditional statements.

4.4.7 Local-Node-Rule-Set-Unit

The purpose of this unit is to manage the node's rule set. The node's rule has been defined as a record type that was described in the "Request-Match-Unit". To simplify the design, the node's rule set includes only one rule named "current-rule" at the moment. This number is too small in the real situation, but for a one-node system, in which there is no actual exchange of the rule between the nodes, this number is allowable. The management of the rule set involves the exchange of the rule between the node and the rule pool. How the rules are managed depends on the result from the "Evaluation-Unit" which determines the node's current state as one of the followings: "node-state-busy", "node-state-idle", "node-state-too-busy" and "node-state-too-idle". The action of the "Local-Node-Rule-Set-Unit" corresponds to the node state.

(1) If the node state is "node-state-busy", the "current-rule" will be copied to the rulepool;
(2) If the node state is "node-state-too-busy", the "current-rule" (representing the entire set of the active rules) will be copied to the neighbour node's rule set;
(3) If the node state is "node-state-idle", a random selected rule from the "rulepool" will be injected into the node's rule set;
(4) If the node state is "node-state-too-idle", the node will be switched off, i.e., becoming non-active.

Similarly, the sequential statements of "**if-then-elsif-then-else**" are used for the behaviour description in this unit design and the hardware synthesis will be complex combinational logic for the conditional statements.

4.4.8 Rule-Pool-Unit

The purpose of this unit is to implement a rule pool that is accessible to all nodes. As the rule pool is comprised of many rules, it is then defined as "**array**" type with rules being its elements. In the design, the number of the rules included in the rule pool is defined as 16. The definition of the rule pool type is showed as the following expression:

type *rule-pool* ***is array*** *(0* ***to*** *15) of rule.*

The function of the "Rule-Pool-Unit" is different from that of each node and a single rule pool should be included in the network. Each node connects with this unit to exchange rules between the node's rule set and the rule pool. This exchanging is controlled by the evaluation result from the "Evaluation-Unit" of each node. As mentioned in the previous section, when the node state is "node-state-busy", a random selected rule of the node will be copied into the rule pool (as the node's rule set contains only one rule as defined in this simplified design, there is in fact no randomly selection of rules in this part), which means the rule pool unit will have input port signals from the "Evaluation-Unit" of each node. When the node state is "node-state-idle", a rule randomly selected from the rule pool will be injected into the nodes' rule set, which means the "Rule-Pool-Unit" will also have output port signals connecting to each node. The main functions of the "Rule-Pool-Unit" are then implemented using two processes, each of which respond to one of the above two situations. The VHDL description of the function is expressed as the following example:

```
injection: process (node-state-idle) is
begin
if node-state-idle = true then
rule-to-node <= rule-pool (random-number1);
end if;
end process injection;
copy: process (rule-from-node-ruleset) is
begin
rule-pool(random-number2) <= rule-from-node-ruleset;
end process copy;
```

In such a design, when a rule is copied to a rule pool, it will overwrite the original rule if there was one in the rule pool at the moment. As a random selection of the rules exists, processes that produce the random number (based on the same principle as introduced in "Request-Generate-Unit") are also contained in this unit design.

4.4.9 One Node System Design

The entity description of the node is shown in figure 4.14. There are two input ports: "Reset" signal, "Clock" signal and five output ports: "forward-request-to-neighbour", "copy-rules-to-neighbour", "output-processed-request", "copy-rule-to-rulepool" and "state-of-local-node".

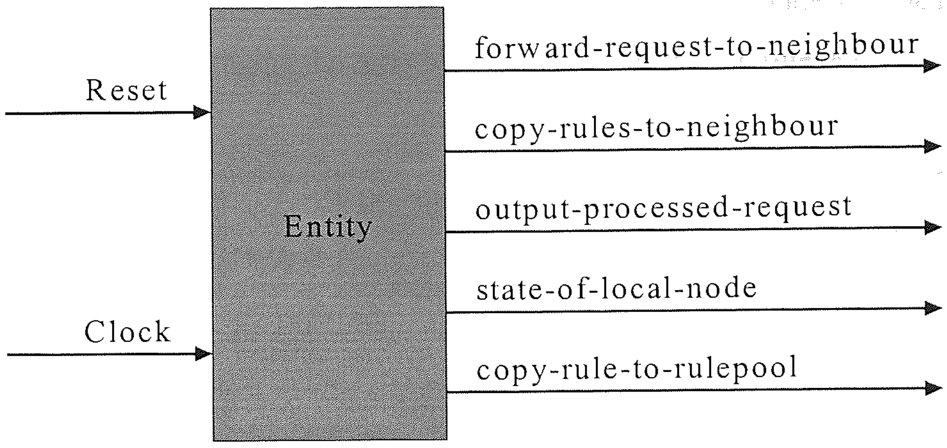


Figure 4.14 The entity description of each node

"Forward-request-to-neighbour" is the signal to forward a request to a neighbour node. This occurs in two situations; firstly, when the node is not active, and secondly when the input request does not match any of the nodes' rules. In the model, the four kinds of requests were represented as a two bit vector signal encoded as "00", "01", "10" and "11". "Copy-rules-to-neighbour" is the vector signal that is used to copy the node's rule set to an adjacent non-active node and activates it. Each element of the vector is a record type rule that includes three items: "X", representing the request, "Y", representing the queue length and "Z", representing the busyness. The code for "X" is the same as the one for the signal "forward-request-to-neighbour". The code width of "Y" depends on the max value of the queue length. In the model the max queue is 200, and the code width is 8, i.e., "Y" is a 8-bit vector signal. Similarly, "Z" is a 7-bit vector signal as the max value of the busyness is 100. Thus width of the signal of "Copy-rules-to-neighbour" will be 2+8+7=17. "Output-processed-request" signal represents the request that have been processed. It has no sense in the simplified model as it is not used any more and thus can be ignored in the design. "State-of-local-node" represents the active or non-active state of the node because the neighbouring nodes will copy their rule set to the local node only when the local node is non-active. It will take the value of "1" for active and "0" for "non-active".

Figure 4.15 showed the function structure of a one-node system in the VHDL design. The diagram shows a simple relationship between the different function units. For a multiple-nodes system, each node has to connect to its neighbouring nodes for the purpose of exchanging the requests and the rules. Each node also has to connect to the rule pool unit for the purpose of exchanging the rules between the node and the rule pool.

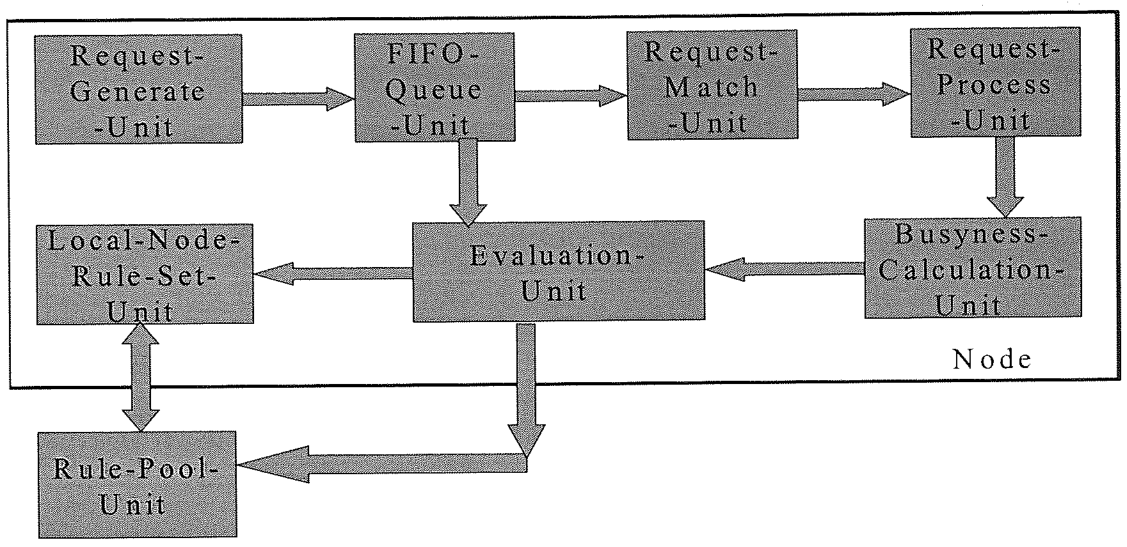


Figure 4.15 The function structure of a one-node system

4.4.10 Simulation and Synthesis

The design of the one-node system was simulated as a very simple model. As it includes only one node, the mechanisms for exchanging requests and rules between nodes cannot implement here. The main functions of the node were simulated using the Xilinx design tool of ModelSim Simulator. Simulation tools visualize signals by means of a waveform that the designer compares with the expected response. In case the waveform does not match the expected response, the designer has to correct the source code. The waveform includes important signals such as input port signals, output port signals and intermediate signals. For example, the "Request-Generate-Unit" entity has one input port and one output port. The input port signal is: "request-generating-clock". The output port signal is: "generated-requests". While simulating the entity, the output port signal was observed and behaved as expected. In this example, the output was a random series of four types of requests coded as "00", "01", "10", "11". In a similar way, the functions of other units were simulated.

Through observation of the waveforms of the involved signals, the simulation results were obtained quickly. Analysis of these results can indicate whether the developed model meets the original design requirements or not. The simulation was performed separately for each of the design unit. To illustrate the result of the simulation, take the example of "Request-Generate-Unit". Figure 4.16 shows the waveforms of the input signal "request-generating-clock" and the output signal "generated-requests". The input signal is on the above and the output signal is at the below. As the output signal is a two bit vector, the sequence of "00", "01", "10" and "11" that represent the random series of generated requests is displayed on the waveform. This figure comes directly from the result of the Simulation.

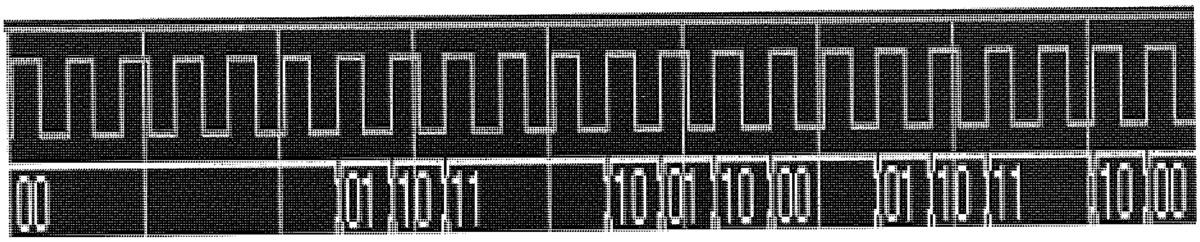


Figure 4.16 Simulation waveform of "Request-Generate-Unit"

From the waveform, it can be seen that in each clock cycle, just at the edge when the clock value changed from '0' to '1', a request is generated. It will be any of the four vectors among "00", "01", "10" and "11". This guarantees four requests are generated randomly at the defined clock speed. In the same way, other unit can be simulated to test the function it implemented. The simulation result shows that the model implements most of the basic functions of one node: generating the random series of requests, evaluating the requests arrived in the input queue on a FIFO principle, processing the matched requests, calculating the busyness, making the evaluation based on the queue length and busyness, managing the node's rule set and communicating with the rule pool (exchanging the rule between the node and the rule pool according to the evaluation result). All these results were showed through the waveform of the signals. The chosen signals were the input/output port signals of each process unit and the important intermediate signals such as the generated requests, the matched requests, the rule copied to the rule pool from the node's rule set, the rule injected from the rule pool to the local node's rule set, the evaluation result of the node state, the busyness or the queue length etc.

The main work of the VHDL design of "Bacterium Model" was concentrated on the code design and symbolic simulation as described above, only parts of the design units were synthesised such as the generating of the random number and the busyness calculation. The whole work was transferred to the Handel-C design of the same model before carrying out the synthesis of all the VHDL design units.

While synthesising, limitation was imposed on the design because some of the syntax was not supported by the Synplify VHDL synthesis tool used to synthesise the design. For example, the busyness calculation unit used an equation as the follows:

*busyness<=**integer** (previous-epoch-busyness*4/5)+*
***integer** (current-epoch-busyness/5);*

However, the operation of "divided by 5" cannot be synthesised by Synplify whereas the "divided by 4" was supported. Therefore, for demonstration purposes the busyness was calculated by combining the busyness at the previous epoch with the busyness for the current epoch in a 0.75 to 0.25 ratio as shown below:

*busyness<=**integer** (previous-epoch-busyness*3/4)+*
***integer** (current-epoch-busyness/4);*

Similar considerations can be applied to the design of the FIFO queue. The design method for FIFO introduced in this thesis was used very seldom. Most of the designs for a FIFO used the traditional method [125] that was proved to be efficient in the software implementation. The general VHDL design of FIFO used dual port memory. Some of the generic VHDL FIFO cores were introduced in [126]. As the synthesis of the FIFO unit was not performed, the question of how efficient the traditional method is compared with the design in this thesis remains an area for future investigation. But one thing is clear. That is the code size of the traditional method is less than the method being used.

The VHDL design code was presented as an appendix in the attached CD-ROM with this thesis.

4.5 Development of "Bacterium Model" Using Handel-C

The above section introduced the VHDL design and simulation of a one-node system of "Bacterium Model". This section describes the design of the same model with more (four and nine) nodes using Handel-C language. The features of Handel-C introduced in section 4.2, particularly the benefits of using Handel-C have demonstrated that it is an attractive language for describing systems that will be implemented in hardware.

4.5.1 Design of "Bacterium Model" Using Handel-C

This section introduces the Handel-C design of "Bacterium Model". It includes the design for both the function implementation and the efficient synthesis.

4.5.1.1 System Function Design using Handel-C

In a real world network, the connection relationships among the nodes are complicated. For demonstration purposes just consider a system in which four nodes are connected in a rectangular grid since it was anticipated that this could be implemented within a single FPGA and would provide a scalable component in a larger network. Figure 4.17 shows how the nodes are connected.

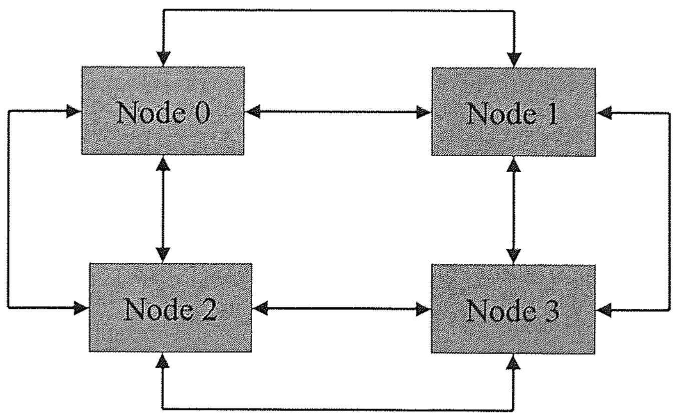


Figure 4.17 Connection relationship between the nodes within the four nodes system

Each node is responsible for its own behaviour, and the simulated network is modelled as a community of cellular automata. Each node has the same structure and performs the same functions listed below.

- Each node has a process of “inject requests” used for the injection of the requests into the node.
- Each node has a process of “receives requests in FIFO queue” for receiving the requests on a FIFO principle.
- Each node has a process of “node rule server” (include active rules and dormant rules) that defines its request handling mechanism.
- Each node has a process of “handle requests” for handling its requests according to its active rules: either processes the request when it matches one of its rules or transfers the request to an adjacent node if there is no match.
- Each node has a process of “GA”, which contains two sub-processes ‘mutation’ and ‘migration’ for the evolving of its rule set.
- Each node communicates with its four adjacent nodes (say up, down, left, right).
- Each node communicates with the commonly accessed ‘rule pool’.

The whole system is then composed of four such nodes and one rule-pool that can be accessed by all nodes. Figure 4.18 shows the hierarchical function structure of the whole system.

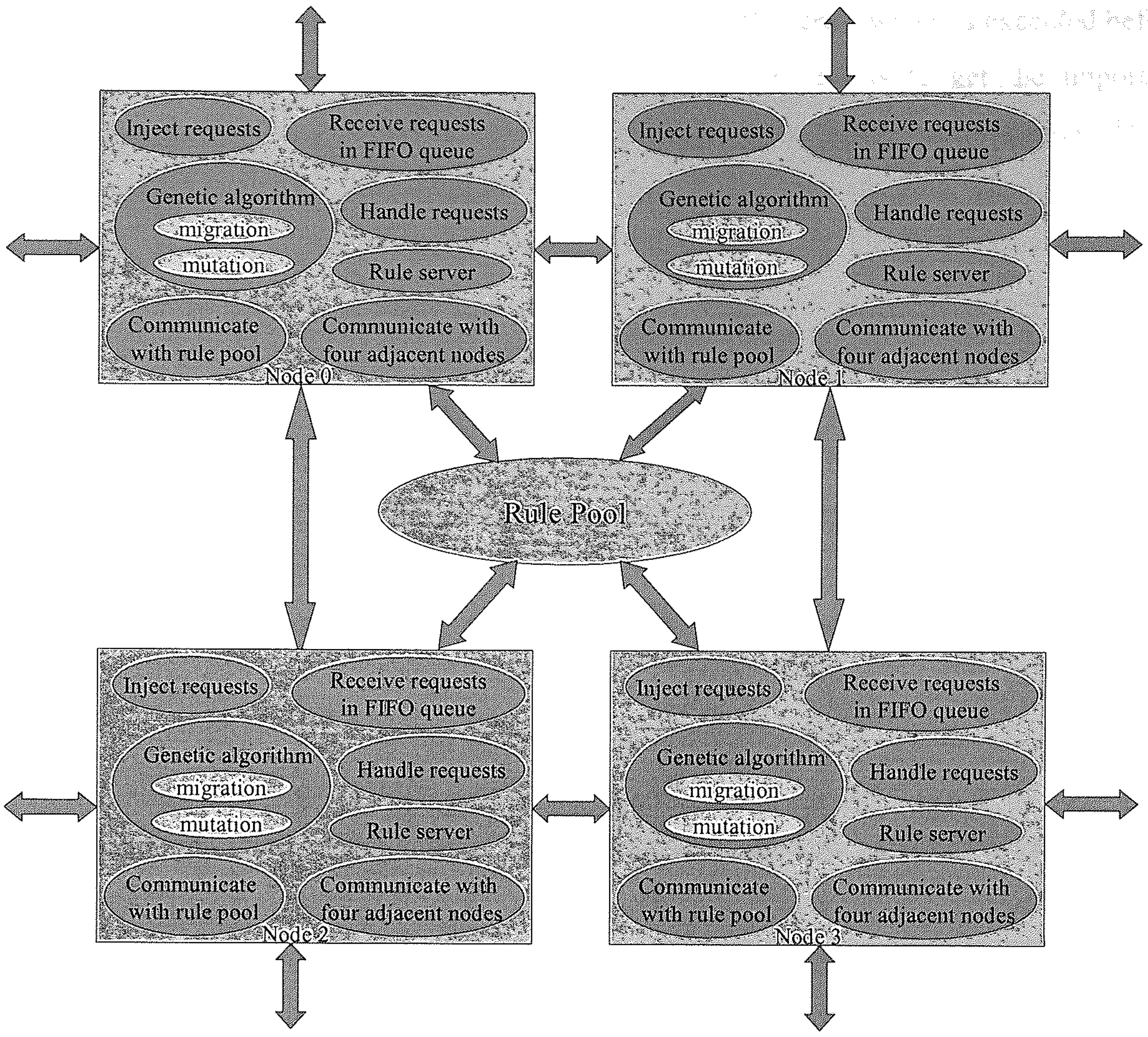

Inject requests
Receive requests in FIFO queue
Genetic algorithm
migration
mutation
Handle requests
Rule server
Communicate with rule pool
Communicate with four adjacent nodes
Node 0
Inject requests
Receive requests in FIFO queue
Genetic algorithm
migration
mutation
Handle requests
Rule server
Communicate with rule pool
Communicate with four adjacent nodes
Node 1
Rule Pool
Inject requests
Receive requests in FIFO queue
Genetic algorithm
migration
mutation
Handle requests
Rule server
Communicate with rule pool
Communicate with four adjacent nodes
Node 2
Inject requests
Receive requests in FIFO queue
Genetic algorithm
migration
mutation
Handle requests
Rule server
Communicate with rule pool
Communicate with four adjacent nodes
Node 3

Figure 4.18 The hierarchical structure of the system

Therefore in the Handel-C design there should be the four processes, one for each node to implement the node's functions and a separate process to implement the function of the rule pool. Figure 4.19 shows the top-level block diagram of the whole system's design.

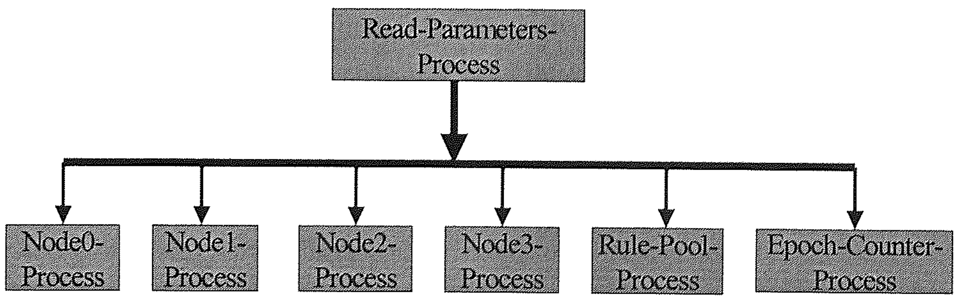


Figure 4.19 Top-level block diagram of the program for the whole system

In the figure, there is a process named "Read-Parameters-Process", which is executed before the following parallel processes. The purpose of this process is to get the important parameters from outside world before the execution of the following processes. These parameters are explained along with the processes in which they are used. The main part of the design consists of six parallel processes. Four of them are the four nodes' processes (Node0-Process, Node1-Process, Node2-Process and Node3-Process). Each node's process contains sub-processes implementing the node's functions. Figure 4.18 shows its structure. The other two processes are the "Rule-Pool-Process" for implementing the rule pool function and the "Epoch-Counter-Process" for counting the number of epoch that has passed since the beginning of the execution. The concept of an epoch is used in the model to represent a period of time (time unit) as measured by the clock cycles in the design. A parameter from the "Read-Parameters-Process" determines how many clock cycles are in each epoch.

Such a structure for the whole system design is applied to gain the most benefit in performance from the target hardware. This is because its inherent parallelism was exploited using Handel-C. As in the real network each node runs concurrently, the process of each node is designed as parallel processes because they are executed concurrently. Similarly, the rule-pool process is executed in parallel with the four nodes' processes as they are designed as parallel processes.

The process of each node is the key part in the system design as it implements most of the main functions listed earlier and showed in figure 4.18. Figure 4.20 shows the lower-level block diagram of the design for each node's process. It is comprised of one separate process " Initialise-Node-Process" and five parallel processes: "Inject-Requests-Process", "Receive-Requests-Process", "Handle-Requests-Process", "Migration&Mutation-Process", and "Node-Rule-Server-Process". The process "Initialise-Node-Process" is executed before the following parallel processes, and initialises the node by defining the node's initial state (active or not active) and initial rule set. This is simply implemented in the Handel-C design by a variable assignment statement which was synthesised as a simple control circuit, and the variables defined in the program were mapped to hardware registers, as described in section 4.2.2. The five parallel processes shown in figure 4.20 implemented the main functions of each node.

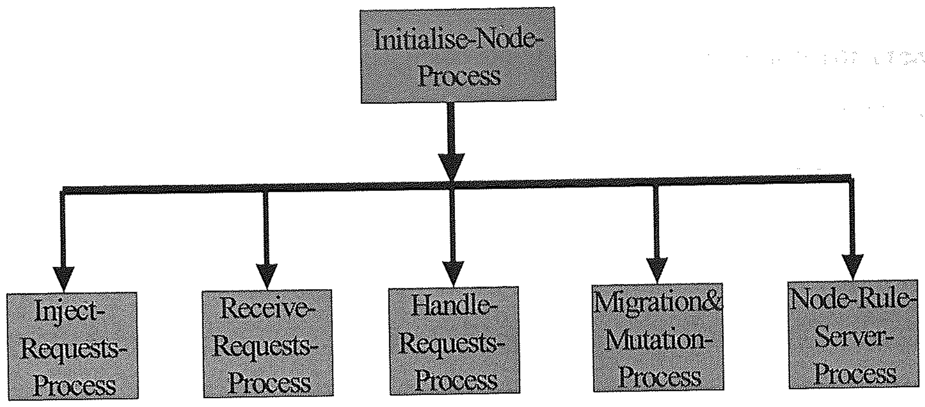


Figure 4.20 The lower-level block diagram of the design for each node

The detailed design for all the parallel processes within each node and the two processes, "Rule-Pool-Process" and "Epoch-Counter-Process" within the whole system are described below. The communication relationships between these processes and the relationship between nodes are also described. Figure 4.21 shows these relationships.

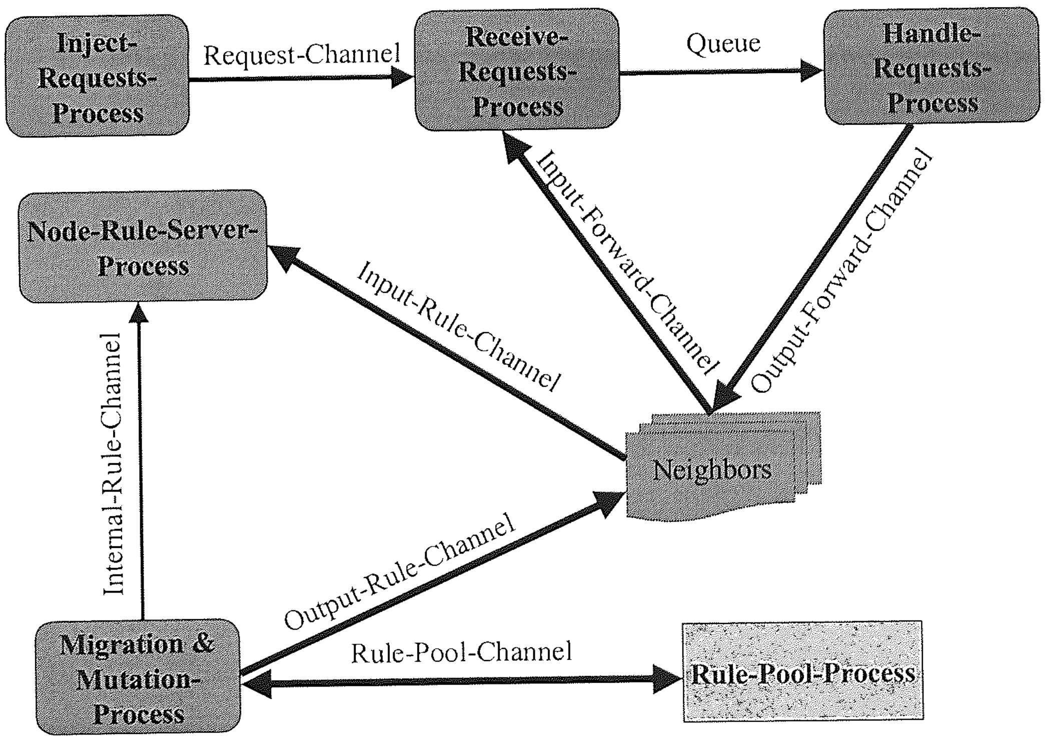


Figure 4.21 Communication relationship between processes

- **Inject-Requests-Process**

The purpose of this process is to implement the function of injection (or creation) of the requests for services. In "Bacterium Model", each node has an engine to create sequences of requests (representing services). A series of random numbers were created to represent the series of the four types of requests. The random number generator applied the same method as applied in the VHDL design (figure 4.11). In the Handel-C design it is implemented as a '**macro procs**', which is one of the macro types defined in Handel-C. The code is much simpler than the earlier VHDL design, containing only two statements. It is showed below where, "Generate-Random" is the '**macro procs**' name, and "Random-Seed" is the variable name representing the created random number.

```
macro procs Generate-Random ( Random-Seed)
{
    Random-Seed<<=1;
    Random-Seed | = 0 @ (Random-Seed [4] ^ Random-Seed [5] ^
                            Random-Seed [6] ^ Random-Seed [8]);
}
```

In there, '@' is the concatenation operator which joins two sets of bits together into a result whose width is the sum of the widths of the two operands. The "Random-Seed" is a nine-bit width variable ranged from 0 to 511. As only four random numbers ranged from 0 to 3 is needed to represent the four types of requests, the created requests variable just take the two least significant bits. This can be implemented using the operator "<-" in Handel-C.

Having the random number, the next step is to create the sequences of requests. Two key issues have to be considered for this part of design. First, the requests have to be created at some rate and this rate has to be changeable. Second, the created sequences of requests have to be forwarded in some way to the next stage for the further process. The Handel-C design solved these two problems in the following way.

In the early description, there is a process named "Read-Parameters-Process" contained in the whole system's design (figure 4.19). The purpose of this process is to acquire some important parameters from outside world such as a user defined data file. The value of one of the parameters was assigned to a variable of "Request-Generate-Time" defined in the program. This variable will decide the rate of request generate. The request-generate rate can be changed easily by updating the value of the parameter in the user data file. For the second issue, the program defined a channel named "Request-Channel" for the forwarding of the created requests to the next stage. The requests were written to this channel and the next stage process read the requests from the same channel. The following code implements the above-described functions.

```
par
{
    for (Clock=0; Clock< Request-Generate-Time; Clock++)
    {
        // take the number of "Request-Generate-Time" clock cycle
    }
    seq
    {
        Generate-Random (Random-Seed); //generate the random number
        Request-Channel ! Random-Seed<- 2; // take the two least significant bits
                                            and write into the "Request-Channel"
    }
}
```

In there, "Clock" is a variable used to count the number of the clock cycle from "0" to "Request-Generate-Time" and thus guarantee the requests are created at the rate of "Request-Generate-Rate", which being the reciprocal of "Request-Generate-Time". This is implemented through a '**for**' statement. Since each assignment statement in Handel-C takes one clock cycle, it will take "Request-Generate-Time" clock cycles to execute the '**for**' statement. The '**seq**' statements contained in its brace first the '**macro procs**' "Generate-Random" to create the random number and then the statement for writing the request variable into the channel of "Request-Channel". As both the '**for**' statement and '**seq**' statement are contained in a '**par**' statement, they are executed in parallel. The number of the clock cycles taken by the '**for**' statement guarantee that requests are created at the rate of "Request-Generate-Time".

- **Receive-Requests-Process**

The purpose of this process is to receive the "requests" and put them into a queue based on a FIFO principle. For each node, the "requests" are received from five sources. One is the "Inject-Requests-Process", which provide the "requests" through writing the "requests" variable into the channel of "Request-Channel", as shown in figure 4.21. The "Receive-Requests-Process" then reads the variable value from this channel. The other four sources of the "requests" are the four neighbour nodes (up, down, left, right). Four forwarding channels are defined for each node. There are actually eight forwarding channels used by each node to communicate with its neighbour nodes. These channels are used to forward a node's "unmatched requests" to its neighbour nodes and by the neighbour nodes to forward their "unmatched requests" to this node, as showed in figure 4.21. They are actually shared between adjacent pairs of nodes. If one node writes the channel, the other reads it, or vice versa. For example,

```
Forward-Channel[Node-ID][Right-Channel] ? Request;
Forward-Channel [Right-Node (Node-ID)] [Left-Channel] ! Request;
```

As a result, the "Input-Forward-Channel" for one node is just the "Output-Forward-Channel" for its neighbour nodes or vice versa. This explains why there are two types of forwarding channels showed in figure 4.21, in which the forwarding channels for inputting the "requests" to the node from its neighbour node are called "Input-Forward-Channel" and the forwarding channels for outputting its "requests" to the neighbour nodes are called "Output-Forward-Channel".

These channels can be declared as 2-dimensional "**arrays**" of channels because each of the four nodes has four forwarding channels. It is then declared as follows:

```
chan unsigned  channel-width    Forward-Channel [4][4]
```

This declares 4 * 4 = 16 channels each of which is "channel-width" bits wide. The width and type of data sent down a channel must be the same type and width as the channel itself [34]. Thus, in the above example the "channel-width" is "2" because the data sent down the forwarding channel is the "request" variable and whose width is "2".

As each of the five sources provides the "requests" independently, in Handel-C design the most appropriate statement for implementing the above function is the "**prialt**" statement [34]. The code is show below:

```
prialt
{       case Request-Channel ? request:
        break;
        case Forward-Channel[Node-ID][Right-Channel] ? request:
```

```
        break;
        case Forward-Channel[Node-ID][Left-Channel] ? request:
        break;
        case Forward-Channel[Node-ID][Up-Channel] ? request:
        break;
        case Forward-Channel[Node-ID][Down-Channel] ? request:
}
```

"**prialt**" selects communication channels depending on the readiness of the other end of the channel. The communication channel that is ready first will execute and data will be transferred over the channel. If two channels are ready simultaneously then the first one listed in the code takes priority. In the code, "request" is the variable representing the data being read from these channels. "Node-ID" represents which node's forwarding channels are being used for the channel communication. Each node has a Node ID. It is a number represent the exact node. For the four nodes system, the four nodes' Node ID are represented respectively as "0", "1", "2" and "3".

After reading the "requests" from one of the five channels, the next step is to put the received "requests" into a queue based on a FIFO principle. In Handel-C design, the queue is implemented as a '**structure**' consisting of two counters (Queue-In and Queue-Out) which are used to test how full the queue is, and an "**mpram**" containing the queued data. Use of an "**mpram**" (means the multi-port memory) allows the queue to be written to and read from in the same clock cycle [127]. The code is as follows:

```
typedef struct
{
        unsigned int Queue-Width Queue-In;
        unsigned int Queue-Width Queue-Out;
        mpram
        {
                wom unsigned Data-Width Q-in [Max-Queue-Length+1];
                rom unsigned Data-Width Q-out [Max-Queue-Length+1];
        } values;
}Queue
```

With the declaration of the queue, the operations to it can be implemented as the following four '**macro expr**', which is one of the Preprocessor forms in Handel-C macros definition:

```
macro expr Increase-Queue-Index (I) = (I+1);  // Increase the queue index;
macro expr Queue-Empty (Queue) = (Queue.Queue-In ==Queue.Queue-Out);
// Decide if the queue is empty;
macro expr Queue-Full (Queue) = (Increase-Queue-Index (Queue.Queue-In) ==
Queue.Queue-Out); // Decide if the queue is full;
macro expr Queue-Length (Queue) = (Queue.Queue-In – Queue.Queue-Out);
// Calculate the queue length;
```

With the declaration of the queue and the related operation, the function of put a "request" item into a queue can be implemented using the following '**macro proc**' of "Queue-Input".

```
macro proc Queue-Input (Queue, Input-Item)
{
        if (!Queue-Full (Queue))
        {
                par
                {
                        Queue.values.Q-In[Queue.Queue-In] = Input-Item;
                        Queue.Queue-In++;
                }
        }
}
```

In the code, "Input-Item" represents the request item that is input to the queue. The max queue length is defined as 32 in the design.

- **Handle-Requests-Process**

The purpose of this process is to implement the function of handling the received requests. The node first take the request item from the FIFO queue and then decide if the request matches one of its rules. If matching, the request is processed. Otherwise, it is forwarded to an adjacent node. The function of taking the request from the FIFO queue is implemented in the following '**macro proc**' of "Queue-Output".

```
macro proc Queue-Output (Queue, Queue-Item, Output-Succeed)
{
        If (Queue-Empty (Queue))
        {
                Output-Succeed = 0;
        }
        else
        {
                par
                {
                        Queue-Item = Queue.values. Q-out [Queue.Queue-Out];
                        Queue.Queue-Out++;
                        Output-Succeed =1;
                }
        }
}
```

In the above code, "Queue-Item" is the variable represents the request item being taken from the queue. And "Output-Succeed" is the variable represents if the action of taking an item from the queue is successful (i.e. if the queue is empty, no item can be taken).

After taking a request item from the queue, the next step is to do the "matching" work. Before introducing the matching of a request to the rules, the declaration of the node "rules" in the Handel-C design has to be presented. As each rule contains three elements, an obvious way to define rules is using the '**structure**'. But to simplify the design, the three elements are defined separately. Each of them can be defined as either an "**array**" or a "**mpram**". But there is difference between these two forms of declaration. To declare an "**array**" is equivalent to declare a number of variables [127]. Each entry in an "**array**" may be used exactly like an individual variable, with as many reads, and as many writes to a different element in the "**array**" as required within a clock cycle. "**rams**", however, are normally more efficient to implement in terms of hardware resources than "**arrays**", but they only allow one location to be accessed in any one clock cycle. Therefore, an "**array**" should be used when the design need to access the elements more than once in parallel and a "**ram**" should be used when efficiency is needed. The use of "**array**" type will occupy more hardware resources, but will provide more flexibility for programming. The declaration of the rules using both the "**array**" method and the "**mpram**" method is showed below:

The declaration of rule in the form of "**array**":

```
unsigned Request-Width  Rule-X [Number-of-Rules];  // X element of the rule
unsigned Queue-Width  Rule-Y [Number-of-Rules];  // Y element of the rule
unsigned Busyness-Width Rule-Z [Number-of-Rules];  // Z element of the rule
```

The declaration of rule in the form of "**mpram**":

```
mpram
{
        ram <unsigned Request-Width> Read-Write-Mem [Number-of-Rules];
        rom <unsigned Request-Width> Read-Only-Mem [Number-of-Rules];
}Rule-X;  // X element of the rule
mpram
```

```
{
        ram <unsigned Queue-Width> Read-Write-Mem [Number-of-Rules];
        rom <unsigned Queue-Width> Read-Only-Mem [Number-of-Rules];
}Rule-Y;  // Y element of the rule
mpram
{
        ram <unsigned Busyness-Width> Read-Write-Mem [Number-of-Rules];
        rom <unsigned Busyness-Width> Read-Only-Mem [Number-of-Rules];
}Rule-Z;  // Z element of the rule
```

In the above code, "Number-of-Rules" represents the number of the rules in the node's rule set. The developed system set this number as 4. The function of matching a request to each rule is then implemented in the following '**macro proc**' of "Match-Rule".

```
macro proc Match-Rule (parameter list...)
{
        Request-Matched =0;
        for (index = 0; index < Number-of-Rules; index++)
        {
                if (Request ==Rule-X [index] &&   // the request type matches
                Queue-Length(Queue)<Rule-Y[index]&&// the queue length matches.
                Busyness < Rule-Z [index]) // the busyness values matches
                Request-Matched =1;
        }
}
```

In the above code, the rules are supposed as declared as "**array**" and the variable of "Request-Matched" is used to express if the request matches the rule. Its value of "1" represents match and "0" means not match. Another variable of "Busyness" represents how busy the node is at the moment. The busyness is calculated in the same way as introduced before in the VHDL design. Suppose the equation is 65*0.8+75*0.2 =67, it can be re-written as 65*4/5+75*1/5=67 or (65*4+75)/5=67. Here, the ratio of 0.8/0.2 is used as the original proposal, not replaced by 0.75/0.25 as in VHDL design because for Handel-C design the division by 5 can be compiled (synthesised) without any difficulty. The calculation of the busyness is then implemented in the following '**macro proc**' of "Calculate-Busyness":

```
macro proc Calculate-Busyness (Cumulative-Busyness, Current-Busyness)
{
        unsigned (Busyness-Width+2) TempBusyess;
        TempBusyness = 0@ Cumulative-Busyness;
        Cumulative-Busyness = (((TempBusyness <<2) + Current-Busyness ) /5 <-
        Busyness-Width;
}
```

In the code, "TemBusyness" is a bit-extended intermediate variable that represent the previous cumulative busyness and "Current-Busyness" represent the current busyness. Its value is calculated by counting the number of requests that are processed (expressed as variable of "Number-of-Processed-Requests) in the current epoch, i.e.

*Current-Busyness = 25 * Number-of-Processed-Requests;*

If the matching result is "no" (the value of the variable "Request-Matched" is 0), that means the request doesn't match any of the node's rules, the request is then forwarded to one of the neighbour nodes. As each node has four forwarding channels (see former section) connecting to its four neighbour nodes, the function of forwarding the requests can be implemented through these channels. As there are four neighbour nodes, the node to which the request is forwarded is decided by the random-number generator process, "Generate-Random", which has been used in the earlier introduced "Inject-Requests-Process" process. The following '**macro proc**' of "Forward-Request" implements the forwarding function.

```
macro proc  Forward-Request (Node-ID, Request, Random-Seed)
{
        Generate-Random (Random-Seed);
        Switch (Random-Seed<-2)
        {
                case Right-Channel:
                Forward-Channel [Right-Node (Node-ID)] [Left-Channel] ! Request;
                break;
                case Left-Channel:
                Forward-Channel [Left-Node (Node-ID)] [Right-Channel] ! Request;
                break;
                case Up-Channel:
                Forward-Channel [Up-Node (Node-ID)] [Down-Channel] ! Request;
                break;
                case Down-Channel:
                Forward-Channel [Down-Node (Node-ID)] [Up-Channel] ! Request;
                break;
                default:
                break;
        }
}
```

Where Right-Node (Node-ID), Left-Node (Node-ID), Up-Node (Node-ID) and Down-Node (Node-ID) are four "**#define**" macros used to represent the ID of the node' four neighbour nodes respectively.

If the matching result is "yes" (the value of the variable "Request-Matched" is 1), i.e., the request matches one of the node's rules, it will then be processed. The node has to spend some time to process each request. The time period the node spends to process each request, "Request-Process-Time", is another parameter from the "Read-Parameters-Process". In the following design, there is no actual process algorithm applied to the request. The program waits for a period of time of "Request-Process-Time" and then increases the number of the processed requests by 1. This is implemented in the following code.

```
for (Clock =0; Clock < Request-Process-Time; Clock++)
{
        Number-of-Processed-Requests ++;
}
```

Where "Clock" is a variable used to count the number of the clock cycle from "0" to "Request-Process-Time". Again, as described before in the "Inject-Requests-Process" description, the executing of the "**for**" statement will take the number of "Request-Process-Time" clocks. That means every "clock-cycle * Request-Process-Time" time later, the value of "Number-of-Processed-Requests" is increased by 1, i.e., a request is processed during this time period. "Number-of-Processed-Requests" is the variable which represents the number of the requests that have been processed during the current epoch. The value is used to calculate the value of variable "Current-Busyness".

To summarise, the "Handle-Requests-Process" implements the following functions: taking the request from the queue; matching the request to the rules; forwarding the un-matched request to the neighbour node and calculating the value of busyness.

- **Migration & Mutation-Process**

The purpose of this process is to perform the GA of migration and mutation for creating the rule diversity. Migration is the process by which rules are exchanged between the nodes through a rule pool structure that is accessible to all the nodes. Rules of the more active nodes are shed or replicated into the rule pool environment and subsequently being absorbed into the rule set of the less active nodes. If migration doesn't help less active nodes increase their fitness they eventually die. The more active nodes have more requests that can be matched to the node's rules and therefore these nodes can process more requests within the fixed time period. Mutation makes the rules to be randomly altered. It is based on the standard mutation operation in the GA. Adaptation of the rules is then acquired through these two algorithms. As the rules define how the node handle the requests for services in the network, the adaptation of the rules thus implements an adaptive management of the network.

The "Migration & Mutation-Process" is therefore comprised of two sub-processes: the 'Migration' sub-process and the 'Mutation' sub-process. This is the most difficult part in the whole system design as the adaptive GA is the core of the whole model. The efficiency of the bacterium inspired solution depends heavily on the efficiency of this algorithm.

Migration is based on the evaluation of the queue length and cumulative busyness. Evaluation is performed every five epochs. If the queue length or busyness is above a threshold (8 for queue length and 50 for the busyness in the design), then the node is regarded to be 'busy' and a random section of the node's rule set is copied into the rule pool (the rule pool structure is introduced in the later "Rule-Pool-Process"). If the node continues to exceed the threshold for four evaluation periods then the node is regarded to be 'too busy' and it replicates its entire rule set into an adjacent node that is not active and activates it. If the busyness is below a different threshold (say 10), the node is regarded to be 'idle' and a rule randomly selected from the rule pool is injected into the node's rule set. If the node is 'idle' for three evaluation periods then the node is regarded to be 'too idle' and its active rules are deleted. If dormant rules exist, these are brought into the active domain. If there are no dormant rules the node is switched off (the node becoming non-active).

Following the above description, the migration process needs to communicate with not only the rule pool process but also its neighbour nodes for exchanging the rules. The type of communication that is needed at any one time depends on the evaluation of the queue length and the busyness at the moment. All these communications are implemented through different channels defined for each node. Each node has a channel called "Rule-Pool-Channel" used to communicate with the "Rule-Pool–Process". Each node also has four other channels used for exchanging the rules (rules migration) with its four neighbouring nodes respectively. These channels are called "Rule-Channel" (as shown in figure 4.21).

Defining four rule channels (up, down, left, right) for each node, there are actually eight rule channels used by each node for migrating the rules between the neighbouring nodes. The rule channels for copying the "rules" to the node from its neighbour node are called "Input-Rule-Channel" and the rule channels for copying its "rules" to the neighbour nodes are called "Output-Rule-Channel". They are actually shared between adjacent pairs of nodes. The "Input-Rule-Channel" for one node is just the "Output-Rule-Channel" for its neighbour nodes or vice versa. This is similar to the situation of forwarding channels introduced in the "Receive-Requests-Process".

With the definition of these channels, the function of migration can be implemented. For example, if the node is regarded to be 'busy', according to the migration principle, a random section of the rule set is copied into the rule pool. This was implemented by writing a random selected rule into the "Rule-Pool-Channel" from which the "Rule-Pool-Process" will read the rule. The selection of a random rule from the rule set was implemented using the "Generate-Random" process. In a similar way, if the node is regarded to be 'too busy', according to the migration principle, the node replicates its entire rule set into an adjacent non-active node. This was implemented by writing all the rule set data to the "Rule-Channel" connecting to the adjacent non-active node and that node in turn reading the rule set data from the same channel later. To decide if an adjacent node is active or not is implemented through a "shake-hand" signal which is used to communicate between the two nodes before the replication of the rule set. The node which is going to replicate its rule set into another node would first send a "shake-hand" signal to that node (by writing the signal to the "Rule-Channel"), to which the target node will respond by sending back a "acknowledgement" signal representing its node state, active or non-active. After receiving the "acknowledgement" signal, the node then decides whether to replicate its rule set to another node or not. The node will try all the four neighbour nodes in turn (in the order of right-down-left-up) until a non-active node is found. If all the four neighbour nodes are active, the replication process stops.

Similarly, if the node is regarded to be 'idle', a rule randomly selected from the rule pool is injected into the nodes' rule set. This was implemented by the Migration process writing a

"Get-Rule-from-RulePool" command signal (which asks for a random rule from the rule pool) to the "Rule-Pool-Channel". After the rule pool receives the command, it writes a randomly selected rule to the "Rule-Pool-Channel" and the "Migration" process reads the rule from the channel. To select a random rule from the rule pool was implemented using the "Generate-Random" process. If the node is regarded to be 'too idle', its active rules are deleted. This was implemented by making the current rules invalid (it assigns a special value to one item in a rule). If the node has dormant rules, they are promoted to be active rules. If there are no dormant rules, the node is switched off by assigning the value of "0" to the node's state variable. The following code briefly illustrates the process of "Migration".

```
macro proc Migration(parameter list...)
        {
                ............ // variable definition
        if (Queue-Length > Queue-Up-Threshold) ||
            Cumulative-Busyness > Busyness-Up-Threshold)
        {
                UpExceedTime++;
                Copy-a-Rule-to-Rule-Pool....
                if (UpExceedTime > 4)
                {
                        UpExceedTime=0;
                        ReplicateRuleSet-to-Neighbour-Node...
                }
        }
        else
        {
                UpExceedTime=0;
        }
        if(Cumulative-Busyness < Busyness-Low-Threshold)
        {
                LowExceedTime++;
                Inject-a-Rule-From-Rulepool-to-the-Node...
                if (LowExceedTime >3)
                {
                        LowExceedTime=0;
                        Deactive-the-Node;
                }
        }
        else
        {
                LowExceedTime=0;
        }
}
```

In the above code, variable "*UpExceedTime*" represents how many times the "*Cumulative-Busyness*" or "*Queue-Length*" has exceed its up threshold ("*Busyness-Up-Threshold*" for the former and "*Queue-Up-Threshold*" for the latter). And variable "*LowExceedTime*" means how many times the "*Cumulative-Busyness*" is below its low threshold (*Busyness-Low-Threshold*).

The implementation of "Mutation" is relatively simple compared with that of "Migration" because it only involves the random alteration of just one value in a single rule. The "Generate-Random" process is used three times as three random numbers are needed to decide (i) which rule in the rule set will be altered, (ii) which item in the selected rule will be altered and (iii) what value will be assigned to the selected item. To prevent any conflict when two processes within a node try to change the rules at the same time, the design prescribed that any changes to the node's rules can only be performed in the process of "Node-Rule-Server-Process". The mutation process cannot change the rules directly. It can only write the new value of the rule to a channel of "Internal-Rule-Channel" that is used to communicate between the two processes of "Migration & Mutation-Process" and "Node-Rule-Server-Process". The latter will read the value of the rule from the channel and assign the value to the rule and thus changes the rule. The code for the process of "Mutation" is briefly illustrated below.

```
macro proc Mutation(parameter list...)
    {
        unsigned Rules-Width Rule;
        unsigned 2 Item;
        unsigned Busyness-Width Value;
        Generate-Random ( Random-Seed)
        Value=Random-Seed<-width (Value);
        par
        {
            Rule= Value <-Rules-Width;
            Item=Value\\(Busyness-Widht-2);
        }
        Internal-Rule-Channel-CMD ! Mutation-CMD
        Internal-Rule-Channel-Data ! Rule;
        Internal-Rule-Channel-Data ! Item;
        Internal-Rule-Channel-Data ! Value;
        }
    }
```

As migration is based on the evaluation result and evaluation is performed every five epochs, the Migration sub-process is then performed every five epochs. The "Mutation" sub-process is also performed periodically in the design. The period time is decided by a variable of "Mutation-Time" whose value is acquired from the "Read-Parameters-Process" and therefore is changeable. A simple illustration of the whole "Migration & Mutation" process code is shown as below.

```
macro proc Migration & Mutation (parameter list...)
{
        ............ //  variable definition
        par
        {
                for (Clock=0; Clock< 5*Epoch-Time; Clock++)
                {
                        // totally take 5*Epoch-Time clock cycle
                }
                Migration (parameter list...);
                If (Mutation-Counter >= Mutation-Time)
                {
                        Mutation (parameter list...);
                        Mutation-Counter=0;
                }
                else
                {
                        Mutation-Counter++;
                }
        }
}
```

In the above code, "Clock" is the variable for counting the clock cycles. It guarantees that migration process is executed every 5*Epoch-Time clock cycle time. "Mutation-Counter" is the variable counting from 0 to "Mutation-Time. Its value is increased by 1 in each clock cycle and returns back to "0" when it gets to "Mutation-Time". This guarantees that mutation process is executed every "Mutation-Time" clock cycle time.

- **Node-Rule-Server-Process**

This process works like a node-rule server and is used to manage the node's rules. It guarantees that any changes to the node's rules can only be performed in this process. No other sub-processes within the node process are allowed to change the node's rules, thus prevent any conflict when two processes try to change the same rule at the same time. The design of "Node-Rule-Server-Process" is described below.

First, as the node may receive the "shake-hand" signal from any of its four neighbour nodes through the four "Rule-Channel", this is implemented in the process as a "**prialt**" and "**case**" statement. If there is a "shake-hand" signal from any of the four "Rule-Channels" then the value of the "acknowledgement" signal (which represent the node state, active or non-active) is written to the same channel. The code is simply shown as below:

```
prialt
{
        case Rule-Channel [Node-ID][Right-Channel] ? Shake-Hand:
        Rule-Channel [Node-ID][Right-Channel] ! Acknowledgement;
        break;
        case Rule-Channel [Node-ID][Left-Channel] ? Shake-Hand:
        Rule-Channel [Node-ID][left-Channel] ! Acknowledgement;
        break;
        case Rule-Channel [Node-ID][Up-Channel] ? Shake-Hand:
        Rule-Channel [Node-ID][Up-Channel] ! Acknowledgement;
        break;
        case Rule-Channel [Node-ID][Down-Channel] ? Shake-Hand:
        Rule-Channel [Node-ID][Down-Channel] ! Acknowledgement;
        break;
}
```

After sending the "acknowledgement" signal, if the node is non-active, it will read the rules data from the "Rule-Channel" because its neighbour node which sent the "shake-hand" signal to it must have written its entire rule set to the same channel after receiving the "acknowledgement" signal. This completes the function of "replicating the entire rule set into an adjacent non-active node". After this, the variable that represents the node's state is assigned to a value of "1", indicating that the non-active node is activated.

Secondly, when the mutation process decides what alteration is going to be made to the selected rule item, it only writes the new value into the channel of "Internal-Rule-Channel" that is used to communicate between the two processes of "Migration & Mutation-Process" and "Node-Rule-Server-Process" (shown in figure 4.21). It is the "Node-Rule-Server-Process" that really changes the value of the selected rule because this process reads the new value from the same channel and then assigned the value to the rule's selected item. This is in accordance with the previous prescription that any changes to the node's rules can only be performed in this process.

- **Rule-Pool-Process**

The purpose of this process is to implement the function of managing the rule pool in the design. Rule pool is a structure which contains a amount of rules that is accessible to all the nodes in the system. Similar to the definition of the rule set of each node, the rule pool can be defined as either "**ram**" or "**array**" as shown in the following:

ram *<****unsigned*** *2> Rule-Pool-X [Rule-Pool-Size]*
ram *<****unsigned*** *Queue-Width> Rule-Pool-Y [Rule-Pool-Size]*
ram *<****unsigned*** *Busyness-Width> Rule-Pool-Z [Rule-Pool-Size]*

or

unsigned *2 Rule-Pool-X [Rule-Pool-Size]*
unsigned *Queue-Width Rule-Pool-Y [Rule-Pool-Size]*
unsigned *Busyness-Width Rule-Pool-Z [Rule-Pool-Size]*

Here, "Rule-Pool-Size" is a constant that represents the maximum number of the rules in the rule pool. The number is set to 8 in the design. The design of the "Rule-Pool-Process" has to be considered along with the migration process. In the migration algorithm, when the node is evaluated as 'busy', it will copy one of its rules to rule pool and when the node is evaluated as 'idle', a rule from the rule pool will be injected into the node's rule set. All these functions are implemented through the "Rule-Pool-Channel" that is used to communicate between the two processes (shown in figure 4.21). Each node has a rule pool channel used to communicate with the rule pool structure. All the nodes' rule pool channels are defined together as an "**array**" of channels with the number of the elements in the "**array**" being the number of the nodes in the system (i.e., 4, in the design). It is shown as below:

chan unsigned *Rule-Width Rule-Pool-Channel [Number-of-Nodes];*

In the migration process, whenever the node is seen to be 'busy', the process first sends a command of "Copy-Rule-to-RulePool" to the rule pool process through writing the value of the command to the "Rule-Pool-Channel" and then writes the value of the rule to the same channel. Whenever the node is seen to be 'idle', the process first sends a different command of "Get-Rule-from-RulePool" to the rule pool process through writing the value of the command to the "Rule-Pool-Channel" and then reads the value of the rule from the same channel. The two commands were assigned with different constant using '**#define**' statement such as follows:

```
#define Copy-Rule-to-RulePool     1
#define Get-Rule-from RulePool    0
```

The design of the rule pool process is then as follows. The "**prialt**" and "**case**" statement is used to decide if there is a command coming from any of the nodes in the system and read the value of the command from the channel. This is implemented in the following code:

```
prialt
{
        case Rule-Pool-Channel [0] ?  Command:  // Node 0 sends the command?
        Node-ID =0;
        break;
        case Rule-Pool-Channel [1] ?  Command:  // Node 1 sends the command?
        Node-ID =1;
        break;
        case Rule-Pool-Channel [2] ?  Command:  // Node 2 sends the command?
        Node-ID =2;
        break;
        case Rule-Pool-Channel [3] ?  Command:  // Node 3 sends the command?
        Node-ID =3;
        break;
}
```

After one of the nodes writes a command to the variable of "Command", the process then use a "**switch**" and "**case**" statement to take different actions according to the different command. If the command is "Copy-Rule-to-RulePool", that means in the migration process the value of the rule has been written onto the "Rule-Pool-Channel", and the rule pool process receives the rule and inserts it into the rule pool structure. If the command is "Get-Rule-from-RulePool", that means the migration process is waiting to read the value of the rule from the "Rule-Pool-Channel", the rule pool process then needs to write the value of the randomly selected rule onto the same channel. The code can be simply shown as below:

```
switch (Command)
```

```
{
        case Copy-Rule-to-RulePool:
        ..........;     // read the "Rule-Pool-Channel"
        break;
        case Get-Rule-from-RulePool:
        ..........;     // read the "Rule-Pool-Channel"
        break;
}
```

To summarise, the communication between each node and the rule pool is implemented through the "Rule-Pool-Channel" and the exchanging of the rules between them is totally contained within the two processes of "Migration-Process" and "Rule-Pool-Process" (shown in figure 4.21).

- **Epoch-Counter-Process**

The purpose of this process is to count the number of "epochs" that has passed since the beginning of the execution. "Epoch" is defined in the design as the measurement period. This is important because the node takes its actions, such as processing the requests, making the measurements of the queue length and busyness periodically within the defined time period: only four requests can be processed per epoch, the evaluation to the queue length and the cumulative busyness is performed every five epochs. The time period each epoch represents is decided by the clock speed and the clock cycles contained in one epoch in the design. The parameter of "Epoch-Time" decides the number of clock cycles in each epoch. Its value is acquired from the "Read-Parameters-Process". The design defines a global variable, "Current-Epoch", to represent the current epoch value. The process is then implemented in the following '**macro proc**' of "Epoch-Counter".

```
macro proc Epoch-Counter (Epoch-Time)
{
        unsigned Clock;
        Current-Epoch = 0;
        Clock =0;
        While (1)
        {
                Clock+=2;
                If (Clock>=Epoch-Time)
                {
                        par
                        {
                                Clock=0;
```

```
                    Current-Epoch ++;
                }
            }
            else delay;
        }
}
```

Where "Clock" is the variable used to count the number of the clock cycles. After the number of "Epoch-Time" clock cycle, the value of the "Current-Epoch" is increased by 1. Thus implement the function of counting the number of epochs that has passed since the beginning of the execution.

- **Read-Parameters-Process**

The purpose of this process is to read several important parameters from the outside world into the design. As showed in figure 4.19, this process is executed before the other parallel processes as the parameters have to be used in these following processes. Part of the parameters has been introduced earlier such as the "Epoch-Time", "Request-Generate-Time", "Request-Process-Time" and "Mutation-Time". There is another parameter which will be introduced in a later section. Through defining their values in a data file and not in the Handel-C code, each time the system is executed using different parameters, it only necessary to re-write the data file and thus avoid the re-compile, re-build and re-place and route procedure associated with hardware implementation.

The design of this process is related to the development environment and tools as the standard functions provided in the Handel-C design suite has to be used to read these parameters. These functions are described in the later chapter after introducing the development platform and design suite.

So far, all the processes within the top-level whole system design (figure 4.19) and the sub-processes within the low-level each node design (figure 4.20) have been introduced. A simple illustration of the code for each node's process (c.f. figure 4.20) is as follows.

```
macro proc Node-Process (parameter list...)
{
        ............... // variable definition
        Initialise-Node-Process (parameter list...);
        par
        {
                Inject-Requests-Process (parameter list...);
                Receive-Requests-Process  (parameter list...);
                Handle-Requests-Process (parameter list...);
                Migration & Mutation-Process (parameter list...);
                Node-Rule-Server-Process (parameter list...);
        }
}
```

And the design code for the whole system (c.f. figure 4.19) is as the follows.

```
void main ()
{
        ............ // variable definition
        Read-Parameters-Process (parameter list...);
        par
        {
                Node0-Process (parameter list...);
                Node1-Process (parameter list...);
                Node2-Process (parameter list...);
                Node3-Process (parameter list...);
                Rule-Pool-Process (parameter list...);
                Epoch-Counter-Process (parameter list...);
        }
}
```

As channels play a vital role in the whole design, all the channels (c.f. Figure 4.21) and their associated communications are listed in table 4.1. These channels can be classified as other intra-node channel or inter-node channel. Intra-node channels, such as "Request-Channel" and "Internal-Rule-Channel", are used to communicate between the sub-processes within the low-level node design. Inter-node channels, such as "Forward-Channel", "Rule-Channel" and "Rule-Pool-Channel", are used for communicating between the processes within the high-level system design. The table shows all the channels' type, name, performed communication and the data passed in the message.

The source code of the Handel-C design for four-nodes implementation is provided in the attached CD-ROM.

Channel Type	Channel Name	Communications Performed	Data passed in the message
Intra Channel	Request-Channel	Communicate between "Inject-Requests-Process" and "Receive-Requests-Process"	Requests for service, in the form of two bit vector: "00", "01", "10", and "11"
	Internal-Rule-Channel	Communicate between "Node-Rule-Server-Process" and "Migration & Mutation-Process"	Rules of the node, in the form of X, Y, Z where X represents the request, Y represents the queue length and Z the busyness
Inter Channel	Forward-Channel	Communicate between nodes for forwarding the requests	Requests for service, in the form of two bit vector: "00", "01", "10", and "11"
	Rule-Channel	Communicate between nodes for copying the rules	Rules of the node, in the form of X, Y, Z where X represents the request, Y represents the queue length and Z the busyness
	Rule-Pool-Channel	Communicate between node and rule pool for exchanging the rules	Rules of the node, in the form of X, Y, Z where X represents the request, Y represents the queue length and Z the busyness

Table 4.1 Channels and communications

4.5.1.2 Design Summary

The above section introduced the design of "Bacterium Model" system in detail concerning the major design components. This section summarises the design difficulties, strategies, and other key issues that were involved in the design.

The key issues that had to be resolved in the design were classified as four major tasks: the generating of the random series of requests, the implementing of a FIFO queue, the GA (mutation & migration) and the output of the resulting data. Most of the design difficulties related to these four processes. Other function units such as the forwarding of the un-matched requests, the receiving of the requests, the matching of the requests and the calculating of the busyness are simple and clear in the design.

The task of generating random series of requests involved the generating of random number. One of the traditional methods is the Linear Feedback Shift Register (LFSR). Maruyama et al [124] referred to the generator as a m-sequence, or maximal sequence. This means that the generator of length *n* generates 2^n-1 numbers. Although only four requests were needed, the generator produced a sequence of 2^8-1 pseudo random numbers as they were also used in other parts of the design such as the mutation algorithm, where the value of the requests (0 to 3), the queue length (0 to 31) and the busyness (0 to 100) could be randomly changed.

An analysis of the effect of using different random number generators (RNG) in a hardware implementation of Genetic Programming (GP) using FPGA was made by Peter Martin in [128] and the hardware systems used RNGs based on Logical Feedback Shift Registers. The paper evaluated different configurations of these generators as well as using a source of true random numbers. Their conclusion was that the simple LFSR can be improved upon by using a generator based on multiple LFSRs. Multiple LFSRs means implementing *n* LFSRs of length *m* and using a single bit from each LFSR at each time step. This can also be done by using a single long LFSR of $n*m$ bits, effectively implementing *n* parallel LFSRs. However, implementing a long shift register in a Xilinx Vertex FPGA is not efficient because, although the look up tables can implement a 16 bit shift register very easily, they are not so well suited to longer shift registers that require more extensive routing resources [128]. They also make it clear that the effect of different RNGs on the performance of a hardware implementation of GP is generally small.

In "Bacterium Model" design, the RNG was used in the GA (GA). The behaviour of GP and GAs has been investigated using different RNGs. Meysenburg and Foster considered the effect of different RNGs on GAs [129] and GP [130]. Their conclusions were that there were no statistically significant differences in the performance of GA or GP when different RNGs were used.

Another difficulty in this part of design is the on-line change of the "Request-Generate-Rate". As this parameter decides the network load, the higher the rate, the higher the network load. The on-line change of the rate means the on-line change of the network load. The design defined an initial value for this parameter in the data file along with other important parameters. Each time the system is executed using different parameters, it only needs to re-write the data file. At the same time, the design defined several time constants (expressed in epochs such as epoch 300 and epoch 500) using the "#define" macro and let the "Request-Generate-Process" change the rate to a different value at the pre-defined time constants. Thus the network load was changed during the system executing.

The implementation of a FIFO queue is another important task in the system design. In the original VHDL design, an awkward method was used (see section 4.4). That method is not explored any further in the present Handel-C design. The present design employed the traditional method of FIFO that has been shown to be effective in the software field. The method was also shown to be effective in [125] when it is implemented using FPGA as it keep logic to minimum. In there, the analysis of the method and the circuit schematic was given to demonstrate its effectiveness.

The most important and difficult part in "Bacterium Model" is the implementation of the adaptive GA of mutation and migration. The first key issue in this part of design is the type definition of the rule, including both the nodes' rule set and the rule pool. Two obvious ways of the rule definition are the "**ram**" or the "**array**". As mentioned before, "**ram**" are normally more efficient to implement in terms of hardware resources than "**array**", but they only allow one location to be accessed in any one clock cycle. In the design, the rules were defined as "**ram**" for the purpose of efficiency.

As the mutation and migration are two independent algorithms, and each may operate upon the same rule at independent times, therefore the design had to avoid the operations on the same rule by these two separate processes at the same time. There are two ways to solve this problem. One is to put the two processes sequentially in the code. Another way is the use of the "Node-Rule-Server-Process" as it is guaranteed that all the changes to the rule can only be performed in this process, not anywhere else. And thus avoids the situation of two processes trying to change the same rule at the same time. The design uses the latter as it is more reasonable and also reliable for the design.

A compromise was made in the design of the mutation algorithm. According to the principle of "Bacterium Model", the mutation involves not only the random alteration of the value of the rule, but also the mutation occurs at a random time point. The Handel-C design compromise this requirement by making the mutation occur at the fixed time period, instead of a random time. This is for the purpose of simplifying the implementation.

Another important issue in this part of design is the use of the channels for the communication between different processes. There are a lot of channels that were used in the design (table 4.1). The channels have to be used very carefully to avoid the deadlock problem. This is because the data transfer through the channel can only complete when both parties are ready for it. If the transmitter is not ready for the communication then the receiver must wait for it to become ready and vice versa. For example, if the Rule-Pool-Process is waiting to read the command from each node's Rule-Pool-Channel, but each node's migration process has not written to the same channel, then the Rule-Pool-Process will have to wait. Suppose this command was used in another channel communication as the written variable, and that communication in turn depends on the previous channel communication. In such a case, a deadlock will occur. The design of "Bacterium Model" managed to avoid such cases through using different channels for different communication purpose.

Another issue that had to be considered in the channel communication is to avoid the single channel dominating the privilege all the time when multiple channels trying to transfer their data to a single variable. Fortunately, the Handel-C language provides the "**prialt**" statement to select one of a number of communication channels depending on the readiness of the other end of the channel. The channel that is ready first will execute and data will be transferred over the channel.

As far as the output of the resulting data was concerned, two ways were used in the different design versions. One is to write the resulting data to the off-chip SRAM and another is to output the data to the device pin by defining the "**bus_out**" interface. Chapter 5 will show the different effect in the system performance (in terms of predicted device resource use and clock speed by Place and Route tool) using these two different methods. The different design versions with different output data type were also compared to determine the effect of outputting only the resulting data for the QoS measurement (the average age of the requests in the network) or both the QoS data and the GA computing data.

4.5.2 Handel-C Programming for Efficient Synthesis

Efficient synthesis of Handel-C program lies in two aspects. First, the program should be written to minimise the use of FPGA resources. Second, the program should be written to provide the time efficiency of Handel-C design, i.e., the implemented design should be fast enough to satisfy the time specification.

One efficient way to reduce the usage of device CLB is to make use of the ram. The following methods illustrate how to use ram effectively. First, try to make full use of the whole width of ram locations if possible, especially if dealing with off-chip and block rams. This makes more efficient use of the storage available and as much data is accessed in one access cycle as possible. Second, the rams should be arranged in such a way that the number of addresses is a power of 2. This is the most efficient way of using address logic. Third, try to avoid using large arrays of registers unless the random access to many elements in parallel is needed because the storage density of array is much lower than for distributed ram. Fourth, try to fully utilize on chip block rams if they are available on the target device. This can significantly reduce the usage of device CLBs. The Handel-C design of "Bacterium Model" made use of all these methods. For example, the rule set for each node, the queue and the rule pool were all defined as rams and the number of addresses is a power of 2.

There are also other ways to reduce the usage of device CLB such as using shift left operator '<<' and shift right operator '>>' for multiplying and dividing by powers of two because multiplication and division can produce a lot of hardware. For example, the process regarding calculating busyness introduced in section 4.5.1.1 contains the following statement:

Cumulative-Busyness=(((TempBusyness<<2+Current-Busyness)/5<-Busyness-Width

This statement includes a division operator '/'. To find out how the Handel-C synthesis tool synthesises the division operator, three different cases were designed to compare the tool's performance in terms of predicted hardware space use and the attainable maximum speed: i) divided by 5 using '/5' as the original algorithm required; ii) divided by 4 using '/4'; and iii) divided by 4 using '>>2'. The first case is expressed as the above statement and the other two cases are expressed as the follows.

Cumulative-Busyness=(((TempBusyness<<2)+Current-Busyness)/4<-Busyness-width
Cumulative-Busyness=(((TempBusyness<<2)+Current-Busyness)>>2 <-Busyness-Width

The performance comparison data for the three synthesised designs is listed in table 4.2. The data is from the implementation result of a 4 nodes "Bacterium Model" after place & route.

	Divided by 5 using '/5'	Divided by 4 using '/4'	Divided by 4 using '>>2'
Number of slices	6,084 out of 19,200 31%	6068 out of 19200 31%	5,606 out of 19200 29%
Number of Slice Flip Flops	2833 out of 38,400 7%	2,833 out of 38,400 7%	2,833 out of 38,400 7%
Number used as LUTs	9,297	9318	8629
Total equivalent gate count for design	107,985	107,895	100,929
Max frequency	17.24 MHz	16.37 MHz	21.73MHz

Table 4.2 Performance comparison using different division expressions

It can be seen from the table that there is less difference between the cases of division by 5 ('/5) and division by 4 ('/4') whereas using right shift operator is much more efficient than using division operator. In this specific 4-nodes "Bacterium Model", which involves a division algorithm in each node, the performance is improved in both the space (6.5 % reduction) and the system speed (20% improvement). From this result, it can be inferred that the Celoxica DK1.1 synthesis tool will not replace the expression of '/4' by '>>2' automatically and thus is liable to improving.

The proper use of '**macro procs**' instead of '**functions**' or vice visa is another way to implement efficiency. Although functions provide for more portable code and are the preferred method of writing in Handel-C, '**macro procs**' may provide an area saving for small blocks of code and do allow for more extensive parameterisation of the code. '**Macro procs**' share datapaths if possible, but require a duplicate control path for each call. Function calls share control path and datapath. In some cases the overhead of a function call may be larger than the duplicated control logic; for example when the subroutine has a small amount of control logic and is only used a few times. The Handel-C design of "Bacterium Model" uses '**macro procs**' instead of '**function**' in most of the cases because the compilation optimisation result showed a better performance (use less resource) when using the former rather than the latter.

As far as the time efficiency of Handel-C hardware is considered, the maximum clock rate should be fast enough for the system performance or real time constraints [127] and the program should be written to reduce the number of clock cycles needed to perform the complex operations.

FPGAs are synchronous devices. That means a clock is used to latch data into registers. In Handel-C all expressions are implemented using combinatorial logic, which if allowed to grow in depth can restrict the maximum frequency the FPGA can be clocked at [131]. This is because of the delays introduced by the combinatorial paths. Therefore, to reduce logic depth, and hence improve on the clock frequency, it is often advantageous to split a complex expression into more but simpler expressions. This usually requires more clock cycles. But by pipelining the operations and with the effective single cycle, throughput can be achieved.

Implementing algorithmic parallelism or pipelining is a frequently used technique in hardware design that reduces the number of clock cycles need to perform complex operation. A pipelined circuit takes more than one clock cycle to calculate any result but can produce one result every clock cycle. The trade off is an increased latency for a high throughput so pipelining is only effective if there is a large quantity of data to be processed: it is not practical for single calculations. This explains why the 'Generate-Random' process and 'Calculate-Busyness' process is not implemented as pipelined circuit although they can be. A high-level pipeline structure is implemented in "Bacterium Model" design. Specifically, in the process of 'Node-Process', the following code actually implemented a pipeline structure because the complex operation from request injection to request handle is divided into three simple operations and each of the simple operations are executed in parallel.

```
par
{
        Inject-Requests-Process (parameter list...);
        Receive-Requests-Process  (parameter list...);
        Handle-Requests-Process (parameter list...);
        ......
    }
```

In addition to pipelining, there are other ways to reduce the logic depth. Certain operations in Handel-C combine to produce deep logic. The methods for reducing the logic depth caused by these operations was summarised below. First, the program should try to avoid division, modulo and multiplication because these operators always produce the deepest logic. Even a single cycle division, mod or multiplier produces a large amount of hardware and long delays through deep logic. Most common division and multiplications can be done with the shift operates or using a long multiplication with a loop, shift and add routine or a pipelined multiplier. Most common modulo operations can be done with the AND operator. Second, the program should try to reduce complex expressions into a number of stages. Consider again the example of the 'Calculate-Busyness' process where the code can be re-written as the follows:

```
macro proc Calculate-Busyness (Cumulative-Busyness, Current-Busyness)
{
        unsigned  (Busyness-Width+2) TempBusyess;
        unsigned   width Temp1;
        unsigned  width Temp2;
        TempBusyness = 0@ Cumulative-Busyness;
        Temp1=TempBusyness<<2;
        Temp2=Temp1+Current-Busyness;
        Cumulative-Busyness =Temp2/5;
}
```

In this example, the operator of multiplication was replaced by left shift operator (T*emp1 =TempBusyness<<2*) and a complex expression of "*Cumulative-Busyness = (((TempBusyness<<2+Current-Busyness)/5*" was reduced into several simple statements. Such a changing will result in a simpler logic being created than a more complex one.

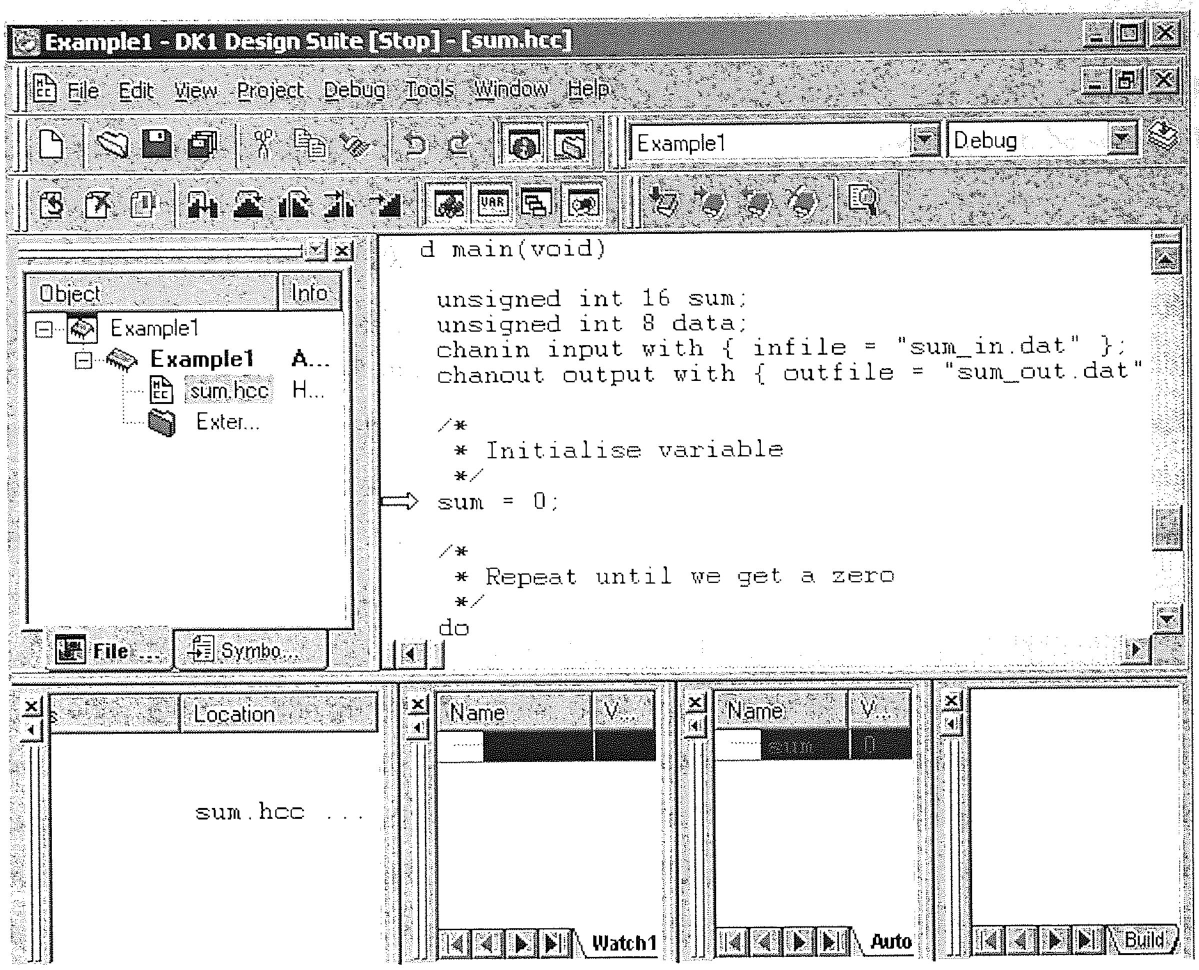
The fast implementation of a complex system depends not only on the fast clock speed, but also on the number of cycles used in the Handel-C program for implementing the system. The basic rule for working out the number of cycles is that 'assignment' and '**delay**' take 1 clock cycle, everything else is free. In all cases, the number of clock cycles can be counted according this simple rule except in the situation of 'channel communication'. Channel communication use one clock cycle if both ends are ready to communicate. If one of the branches is not ready for the data transfer then execution of the other branch waits until both branches become ready. This gives rise to the problem that the execution of channel communication may take any number of clock cycles. In fact, this is also the most difficult part in the design of "Bacterium Model" when considering the time efficiency because a lot of channel communications are used in the design such as the 'Request-Channel', the 'Forward-Channel' and 'Rule-Pool-Channel' et al. It is hard to predict precisely how many clock cycles is needed to complete the channel communications. The only way to know exactly how many clock cycles being taken is through the simulation and experimentation.

4.5.3 Debug and Simulation of Handel-C Design

The Celoxica DK1 integrated development environment (IDE) includes a simulator/debugger to allow the design being tested before committing to hardware [34]. It is a symbolic debugger providing stepwise execution or to break point or cursor. The coding techniques used for debug include substituting simulator channels for hardware interface channels, substituting file input for external channel input and export the contents of variables into files.

In a sequential language such as ANSI-C, one can step through code one line at a time and can stop at an execution point. Because Handel-C is a parallel language, there can be multiple execution points. Where a "**par**" statement is found in the code the execution splits into separate threads, one for each branch of the par statement. The threads execute in parallel: multiple statements may execute on the same clock cycle. While debugging, only one thread can be followed at a time.

Using the debugger will allow all the debug information being displayed in the windows. The variables' window shows the variables that are in use on a specific clock cycle. There is also a watch window. Typing a variable's name in the watch window allows the variable to be displayed constantly. The stack window displays all the functions in the simulated program. All the threads are displayed in the thread window. The clock window displays all clocks. Figure 4.22 shows the Handel-C IDE that is very similar to software IDEs.



```
d main(void)

    unsigned int 16 sum;
    unsigned int 8 data;
    chanin input with { infile = "sum_in.dat" };
    chanout output with { outfile = "sum_out.dat"

    /*
     * Initialise variable
     */
    sum = 0;

    /*
     * Repeat until we get a zero
     */
    do
```

Figure 4.22 Handel-C IDE

Through observing the values of the important variables in the watch window, a clear clue of how the variables change their values during running can be acquired. There are lots of variables that can be listed in the watch window such as the generated requests, the queue length, the busyness, the rule set that includes all the values of each element in each rule and the rule pool values etc. This is important because the result of the algorithm computing is reflected in the change of these values. For example, when a large queue length or busyness value (above the up threshold) is observed, one can expect a migration operation of "copy a rule from the node to the rule pool". This operation really happened as the corresponding change in the rule pool' value is observed in the watch window. A concrete example is presented below to illustrate this.

Suppose the following variables are displayed in the watch window at a moment: node 0's rule set (contains two active rules), its busyness and queue length, and the rule pool values. Say the node's rule set is {{1, 8, 30}, {3, 12, 40}}, the busyness is 60 and the queue length is 12, the rule pool contains the following 8 rules {{0, 4, 25}, {0, 8, 20}, {3, 20, 70}, {2, 10, 65}, {2, 15, 50}, {1, 7, 40}, {0, 15, 50}, {1, 16, 30}}. As both the busyness and the queue length exceed its up threshold (50 for busyness and 8 for queue length), it can be seen that after some while (some clock cycles later), the new rule pool values become {{0, 4, 25}, {3,12,40}, {3, 20, 70}, {2, 10, 65}, {2, 15, 50}, {1, 7, 40}, {0, 15, 50}, {1, 16, 30}}. This means one of the node's rules of {3, 12, 40} has been copied to the rule pool and rewrite the rule pool's original second position. It can even been observed that the random number generated at that time for the random position in the rule pool is exactly 2. This example simply illustrate how the designed functions of the algorithm can be verified through running the program step by step and watching the immediate changes of the related variables.

The other functions can be verified in the same way. These functions include the mutation and other migration operations such as inject a rule from rule pool to a node's rule set, copy a node's whole rule set to a neighbour node, activate the node or deactivate a node. Even the forwarding of the request to a neighbour node and the processing of a matching request can be observed as well. By this way, the whole procedure of the algorithm computing can be traced all along and thus testify if the designed function is correct or not.

On the other hand, Handel-C provides special channels, '**chanin**' and '**chanout**', which are designed to be used with the simulator. The input for a '**chanin**' variable comes from a file, while a '**chanout**' variable can output to the output window or to a file depending on how it is declared. Both these features from Handel-C and DK1 IDE make it possible to output the experimental result in the form of output file. These features can be used in the design to ensure that whenever there is a migration or mutation action during the algorithm execution, the corresponding data is output to a file. This provides another way to test and verify the simulation result through analysing the output data in the data file.

The output items included in the output file are listed in table 4.3. The first column represents the output type. Different identifier codes are defined for different output type, such as "250" for "mutation" code, "251" for "Inject a rule from Rule Pool to a node" code, "252" for "Copy a Rule to Rule Pool " code, "253" for "Deactivate a node " code and "254" for "Activate a node" code. This is listed in the second column. "Node ID" shows which node is involved in the action. "Rule index in a rule set" describes which rule in the node's rule set is involved in the action. "Rule index in rule pool" identifies which rule in the rule pool is involved in the action. "Item in a rule" identifies which item in the rule (request type, queue length or busyness) is involved in the action and "Value of the item" means the value of the involved item. The simulation events information is output as a series of data codes comprising the type of event, the node id, rule id, item in a rule, and the new value.

Output Type	ID Code	Output Data			
Mutation	250	Node ID	Rule Index in a Rule Set	Item in a Rule	Value of the Item
Inject a Rule from Rule Pool to a Node	251	Rule Index in Rule Pool	Node ID	Rule Index in a Rule Set	X
Copy a Rule to Rule Pool	252	Node ID	Rule Index in a Rule Set	Rule Index in Rule Pool	X
Deactivate a Node	253	Node ID	X	X	X
Activate a Node	254	Node ID	Node ID	X	X

Note: "X" in the table means don't care data

Table 4.3 Output type and data

For example, the output data sequence: "250, 0, 1, 1, 4", means that a mutation happened on node 0, it is the first rule in the rule set which mutates, and it is the queue length item which mutates, and the value of the queue length has been changed to 4. Similarly, the output data sequence of "251, 3, 1, 2" means the third rule in rule pool is injected into node 1's rule set to become its second rule. Through the same way, all the data in the data file can be analysed. However, these data only illustrate the resulting data after executing the genetic algorithm. Because the queue length and the busyness is not included in the file, one can not tell why these mutation or migration operations happen at the moment, nor the forwarding requests and the processing of the requests operations. Fortunately, these can be observed in the watch window whiling running the program step by step as has been said before.

By observing and analysing these simulation results, it was shown that the model works just as the design required: mutation and migration is performed whenever the condition is met (based on the evaluation of busyness and queue length). "Forwarding" or "Processing" also happens as the algorithm defined. For example, when a request is inserted to a node's queue and matches one of a node's rule, it can be seen that the request is processed. On the contrary, if the request doesn't match any of the nodes' rules, it is transferred to another node. If a node's queue length or busyness exceeds the up threshold, one of its rules is copied into the rule pool. If the queue length or busyness is less than the below threshold, a rule randomly selected from the rule pool is added into the node's rule set. Handling of the requests ("Forwarding" or "Processing") is not output to the data file but can be watched in the debug window. The debug window can display the executing state of every step.

As the above description, the simulation stage only simulates the workflow of "Bacterium Model" and does not give a QoS measurement. The purpose of simulation is to test and verify the design before implementing on the real hardware. Since the simulation result has verified the design computing, especially the GA, which being the essential of the developed code, this provides the basis for the later hardware implementation. The QoS measurement is performed in the hardware implementation.

Chapter 5 Hardware Implementation of Bacterium Model

This chapter describes the hardware implementation of "Bacterium Model". It includes a brief introduction to the experiment platform and environment. The hardware implementation information in terms of predicted design performance after Place and Route is also presented in this chapter.

5.1 Design Environment and Platform

The design environment used for the hardware implementation is provided by the DK1 design suite from Celoxica which facilitates the design and synthesis of custom hardware from a high-level concurrent software written in the Handel-C language. A FPGA based re-configurable hardware development platform, RC1000 board, also from Celoxica, is used for the prototype design and development. The main implementation device on the board is Xilinx Virtex XCV2000E FPGA. The "Place and Route" tool from Xilinx is also used as part of the design tools.

5.1.1 DK1 Design Suite

The Celoxica DK1 design suite focuses on the design, validation, iterative refinement and implementation of complex algorithms in hardware. It includes built-in design entry, simulation, and synthesis, driven directly by Handel-C [33].

DK1 is an integrated design environment with the look and feel of a software design system, including a GUI (Graphical User Interface) for integrated project management, code editing and source level symbolic debugging. The compiler generates architecture optimised EDIF net-lists ready for input to FPGA and PLD place and route tools. This allows direct generation of prototype hardware, or first generation electronic products, and is claimed to generate significant time saving when compared to traditional design methods.

5.1.2 Re-configurable Hardware Development Platform – RC1000

RC1000 is the re-configurable hardware development platform developed by Celoxica to provides high-performance, real-time processing capabilities and is optimised for the Celoxica DK1 design suite. It was used as the experimental platform of "Bacterium Model".

RC1000 is a standard PCI bus card equipped with a BG560 packaged Xilinx Virtex-E family FPGA with up to 2.5 million system gates. The RC1000 is a plug-in prototyping board with 8Mb of SRAM directly connected to the FPGA in four 32-bit wide memory banks. The memory is also visible to the host CPU across the PCI bus as if it were normal memory. Each of the 4 banks may be granted to either the host SRAM or the board. It is then accessible to the FPGA directly and to the host CPU either by DMA transfers across the PCI bus or simply as a virtual address. The board's detail technical features are provided in Appendix B.

5.1.3 FPGA

FPGA is the main device used for the hardware implementation of "Bacterium Model". FPGAs are programmable logic devices that can be configured after being manufactured. These devices can be used to implement a complete solution or as part of a solution in conjunction with a conventional processor (co-processor). The FPGA technology has evolved rapidly and now offers devices that contain millions of programmable gates. A brief introduction about the FPGA knowledge is provided in Appendix C.

The specific device that was used in the bacterium system is Virtex XCV2000E-BG560 that has 2.5 million system gates on it. The detail specifications of this device are listed in table 5.1.

System Gates	Logic Gates	CLB Array	Logic Cells	User I/O	BlockRAM Bits	Distributed RAM Bits
2,541,952	518,400	80x120	43,200	404	655,360	614,400

Table 5.1 Specification of the Xilinx Virtex XCV2000E-BG560

5.1.4 Place and Route Tools

Place and Route tools are used to map the design, i.e. an EDIF net-list, into a configuration bit-file that can be downloaded onto a FPGA. The Place and Route tool that has been used in the development system is included in Xilinx Foundation 4, which is Xilinx design implementation tool.

5.2 Communication Between Host and FPGA on the Board

Communication between host computer and FPGA on the RC1000 development board plays an important role in the system implementation. This section introduces briefly the general communication methods between host and the FPGA and the particular communication implemented in "Bacterium Model". A detail introduction of the communication methods is provided in Appendix D. The host program development is also introduced in this section.

5.2.1 Communication Methods Between Host and the FPGA

Generally there are three methods of communication between the host and the FPGA [132].

1. Single bit signalling using 2 pins on the FPGA
2. Single byte data transfers using the control/status ports on the RC1000
3. Bulk data transfers using the DMA controller and the banks of SRAM.

Typically, the first method can be used to signal a state to the FPGA or to the host. The second method can be used to send short control messages to the FPGA or short status messages from the FPGA. The third method is recommended for large data transfers. As the development of "Bacterium Model" only uses the last two methods in the communication, more information about the detail implementation of these two methods is provided in Appendix D.

5.2.2 Communication Between Host and FPGA in “Bacterium Model”

The communication between host and the FPGA in “Bacterium Model” has two purposes: to provide the important parameters for the Handel-C design from Host to FPGA such as “Request-Generate-Time” etc. (the parameter that decide the initial value of the network load), and to send back the computing data generated by FPGA execution such as the network performance data to the host.

The Handel-C design parameters are provided by the “Read-Parameters-Process” (shown in figure 4.19), which read the parameters data from the off-chip SRAM. These parameters are initially stored in a data file in the host computer. Before the DMA transfer, the single byte data transfer method is used to synchronize the swapping of ownership of a memory bank. Through the DMA communication these parameters data are first sent to the off-chip SRAM by the host and then read by the FPGA through executing the “Read-Parameters-Process”. These parameters are then assigned to the corresponding variables in the Handel-C program.

In the design, the SRAM of bank0 is chosen for storing the parameters data. Thus, each time the system is executed using different parameters, it only needs to re-write the data file in the host computer and thus avoid the re-compile, re-build and re-place and route procedure of the Handel-C design for the FPGA. Such a solution not only saves the time for prototyping design, but also provides a flexible way for system executing under different parameters as the changes of the parameters in a host data file is much easier than in the Handel-C program for FPGA. Furthermore, this part of communication will not become a bottleneck in the whole system design because the parameters are only read once at the beginning of the system executing.

The second purpose of communication between host and the FPGA in “Bacterium Model” is for outputting the experiment result data to an output data file in the host. While executing, the FPGA writes the result data to the off-chip SRAM for DMA transfer to the host. There are two types of result data. First is the average age (expressed in epochs) data for the QoS measurement and second is the GA computing data. All these data are generated on-line when the system is running in the FPGA.

5.2.3 Host Program Development

The host program for "Bacterium Model" system is designed to perform three functions. The first is to configure the FPGA with the bit file. The second is to communicate with the FPGA on the RC1000 development board. The third is to explain the output result data and make the performance measurement.

To configure the FPGA, the host program calls a function of "**PP1000ConfigureFromFile**", which is included in the host support software, to configure the FPGA directly from a bit file. After the execution of this calling statement the bit-file will be downloaded onto the FPGA.

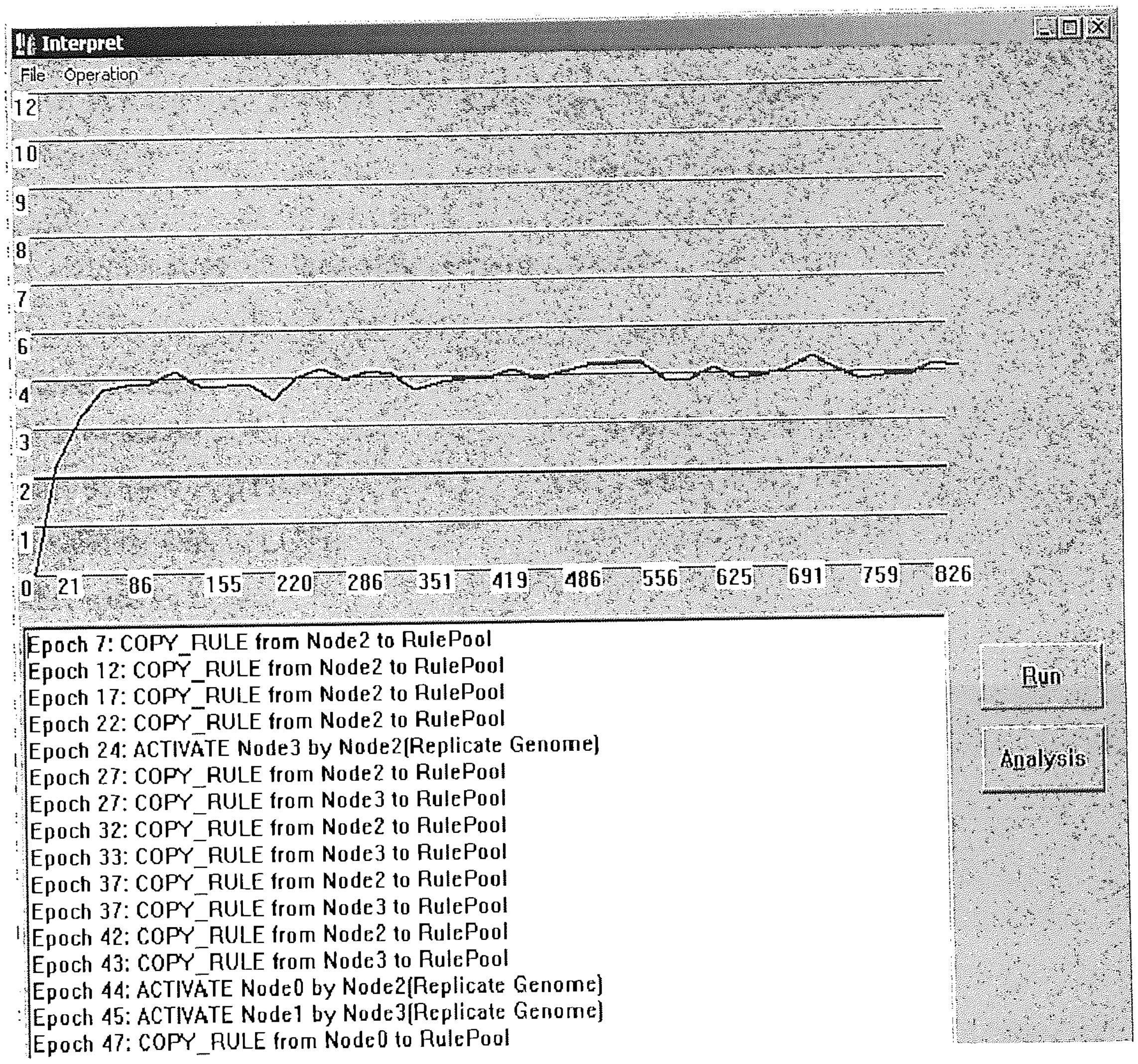
The communication between the host and the RC1000 board has been introduced in the above section. The third function of the host program is to read back the result data, explain them and display in the form of the explanation statement and a QoS performance measurement. As the generated data are written to the data file in the form of a data series, comprising the type ID code followed by the related data, the host program will explain the GA computing data by outputting the explanation statements in the output window and thus give the reader a rough glimpse of the current executing state. Some examples are listed below:

Epoch T1: Copy a rule from Node N1 to Rule Pool
Epoch T2: Active Node N2 by Node N1
Epoch T3: Node N3 has a Mutation

Here, T1, T2, T3 are the time represented in epoch and N1, N2 and N3 is the node ID

It will also produce a graph of X against Y, where X represents the time (epoch) and Y represents the average age (epoch) of all the requests according to the age data. The age data is headed by a special code named "Age-Code". The host program then produces the graph that representing the QoS measurement of the network performance (performance curve).

As a result of the host program executing, a visual window was displayed on the host screen with both the performance curve and the explanation statement within it. Figure 5.1 shows a typical picture of the window that contains the performance curve of a 4 nodes network under a high but stable network load. The host program was developed using Microsoft Visual C++ and runs under the Windows 98 or 2000 environment as a normal executable. The source code can be found on the attached CD-ROM.



```
Epoch 7: COPY_RULE from Node2 to RulePool
Epoch 12: COPY_RULE from Node2 to RulePool
Epoch 17: COPY_RULE from Node2 to RulePool
Epoch 22: COPY_RULE from Node2 to RulePool
Epoch 24: ACTIVATE Node3 by Node2(Replicate Genome)
Epoch 27: COPY_RULE from Node2 to RulePool
Epoch 27: COPY_RULE from Node3 to RulePool
Epoch 32: COPY_RULE from Node2 to RulePool
Epoch 33: COPY_RULE from Node3 to RulePool
Epoch 37: COPY_RULE from Node2 to RulePool
Epoch 37: COPY_RULE from Node3 to RulePool
Epoch 42: COPY_RULE from Node2 to RulePool
Epoch 43: COPY_RULE from Node3 to RulePool
Epoch 44: ACTIVATE Node0 by Node2(Replicate Genome)
Epoch 45: ACTIVATE Node1 by Node3(Replicate Genome)
Epoch 47: COPY_RULE from Node0 to RulePool
```

Figure 5.1 A typical output result display

5.3 Implementation Information

Using Celoxica's RC1000 re-configurable computer platform, both four nodes and nine nodes model was implemented for the research. This section provides the implementation information regarding the occupied hardware resource by the designed system and predicted system clock speed after place and route. A detail introduction about the implement procedure can be found in the Appendix E.

Specifying the EDIF output from the compiler produces a net-list that can be used as input to the "Place and Route" tools. After Place and Route, the information of the design summary, such as the device utilization and design statistics in the system clock speed, is acquired in a log file. Both the information for four nodes and nine nodes implementation are given below.

1) Information for four nodes implementation

Design summary:	
Number of Slices:	6,654 out of 19,200 34%
Number of Slice Flip Flops:	3,073 out of 38,400 8%
Total Number 4 input LUTs:	10,959 out of 38,400 28%
Number used as LUTs:	10,191
Number used for Dual Port RAMs:	288
Number used as 16x1 RAMs:	15
Number used as Shift registers:	4
Number of bonded IOBs:	121 out of 404 29%
Number of GCLKs:	2 out of 4 50%
Total equivalent gate count for design:	115,833
Design statistics:	
Minimum period:	59.23ns (Maximum frequency: 16.88MHz)
Maximum net delay:	15.48ns

2) Information for nine nodes implementation

Design Summary:	
Number of Slices:	19,198 out of 19,200 99%
Number of Slice Flip Flops:	6,596 out of 38,400 22%
Total Number 4 input LUTs:	33,658 out of 38,400 84%
Number used as LUTs:	31,965
Number used for Dual Port RAMs:	648

Number used as 16x1 RAMs:	15
Number used as Shift registers:	9
Number of bonded IOBs:	121 out of 404 29%
Number of GCLKs:	2 out of 4 50%
Total equivalent gate count for design:	309,108

Design statistics:

Minimum period:	70.02ns (Maximum frequency: 14.28MHz)
Maximum net delay:	19.69ns

Handel-C requires that the clock period for a program is longer than the longest path through combinational logic in the whole program. This means that, for example, once FPGA place and route has been completed, the maximum clock rate for the system can be calculated from the reciprocal of the longest path delay in the circuit. For example, the FPGA place and route tools calculate that the longest path delay in the above nine nodes system design is 70.02ns, the maximum clock rate (frequency) that circuit should be run at is then 1/70.02ns = 14.28MHz.

In the Handel-C design, some extra code was added to write the result data to the off-chip SRAM. As the data were written to the SRAM on-line and all the time and were read later by the host, to find out how the extra code for outputting the result data affect the system performance, two experiments were performed. The first experiment was made to compare the system performance under the situations with different amount of output data. Two design versions were run, (i) with only the data for QoS measurement being output, (ii) with both the QoS data and the GA computing data being output. Both of the versions output the data to the SRAM. The second experiment was made to compare the system performance under different output methods, (a) using the original method of writing the data to the off-chip SRAM, (b) by writing the data to the device pin (this was implemented using an interface of '**bus-out**' definition), and (c) using both SRAM method and the device pin. Four versions of design was run for the purpose of comparison: experiment (i) was conducted under the situation of (a), (b) and (c), and experiment (ii) was conducted using (a) only. The predicted performance for all these design versions of a four-node network after place and route were presented in the following table (table 5.2). To simplify these experiments, the mutation process was omitted in the system design.

	Version 1	Version 2	Version 3	Version 4
Output data	QoS data	QoS data and GA data	QoS data	QoS data
Output method	Off-chip SRAM	Off-chip SRAM	Bus-out interface	Off-chip SRAM and bus-out interface
Number of Slices used	6,084 out of 19,200 31%	6,214 out of 19,200 32%	5,955 out of 19,200 31%	6,081 out of 19,200 31%
Total equivalent gate count for design	107,985	110,347	106,254	108,003
Number of bonded IOBs	121 out of 404 29%	121 out of 404 29%	119 out of 404 29%	179 out of 404 44%
Maximum frequency	17.24MHz	18.20MHz	15.45MHz	16.30MHz

Table 5.2 Comparison of performance under different output situations

This table provides a rough picture of how the outputting of the result data affects the system performance. It was seen that the different output methods made little difference to the system performance in terms of resource use (6,084 slices for SRAM and 5,955 for bus-out), but has a difference in the clock speed (17.24MHz for SRAM and 15.45MHz for bus output). The comparison between Version 1 and 2 shows how the amount of the output data affects the system performance. As version 2 outputs both the QoS measurement data and the GA computing data, this kind of comparison gave a clue that how much the outputting of the GA computing data would affect the system performance. Compared with others, version 4 showed a more usage in the bonded IOB. This could be explained as both the output methods were used in the same design.

To find out how each part (process) in the system design affect the performance, one could implement each process as a stand alone design for an FPGA using Handel-C, and record the number of slices used and the maximum attainable clock frequency after place and route. This would give a measurement of the hardware resources needed to implement the process and an indication of the logic depth required. However, the design of "Bacterium Model" only implemented the whole model as an integrated system, and did not isolate each of the tasks (processes) as an independent design for the performance measurement. Only two simple experiments were performed to test how the outputting of the result data affect the performance as listed in table 5.2. Chapter 6 will present more experiments regarding how the process of mutation affect the system performance. More work in this field could be done in the future.

Chapter 6 Experimental Results and Analysis

In this chapter the results generated by the FPGA implementation were analysed to determine the behaviour and performance of the model networks and the efficiency and effectiveness of the hardware implemented GA network management algorithm.

6.1 Explanation of the Experimental Results Output Form

The data generated by the FPGA implementation includes both the data related to migration & mutation which reflects directly the result of the GA process and the data representing the average age of the requests which is used for the QoS measurement. All these data are output to a data file during the execution and analysed by the host program later. The GA related data includes the event ID (as listed in Table 6.1 which forms a superset of Table 4.3), the time (expressed in epochs) when the event happened, and event data corresponding to the event data type and format (as listed in table 4.3).

Data Type		ID Code
Mutation		250
Migration	Inject a rule from rule pool to a node	251
	Copy a rule to rule pool	252
	Deactivate a node	253
	Activate a node	254
QoS Measurement (Age)		249

Table 6.1 ID code for different output data type

For example, consider mutation which is a single action that involves the random alteration of just one value in a single rule. Whenever there is a mutation, the program outputs the mutation event ID code followed by the current epoch time, the involved node identity code, the index of the rule that mutates, the item in the rule that is mutated and the new value of the item after mutation. Thus, if a data series of "250, 64, 1, 2, 0, 1" was generated during the execution, then this can be identified as a mutation which occurred at epoch 64 on node

1, and it is node 1's second rule that was mutated and the rule's first item ("X" item, representing the type of the request) that was changed and the new value for this item after mutation process is "1".

This is very similar to the data explanation introduced in section 4.5.3 when discussing the simulation result of "Bacterium Model", except that here the data is the result of the real hardware implementation, and data representing the time when the event happened was also output.

In the same way, the data related to the migration process can also be explained. Migration is a little more complex as it involves four types of operation as listed in Table 6.1. The generated result for any of the four operations includes the data type ID code, the current epoch time, the involved node ID code and the relative data. For example, if a data series of "252, 78, 0, 1, 3" was generated during the execution, it can be understood as a "Copy a Rule to Rule Pool" operation occurred at epoch 78, and it is node 0's first rule that was copied to the third position of rule pool. Similarly, the data series of "254, 90, 1, 3" means node 3 was activated by node 1 at epoch 90. The data series of "251, 100, 3, 2, 2" means "Inject a rule from Rule Pool to a node" occurs at epoch 100, and it's the rule pool's second rule that was injected to node 3 and became the node 3's second rule. The data series of "253, 120, 2" means node 2 was deactivated at epoch 120.

The performance measurement data was also generated and output during the FPGA execution. These data were headed by an "Age code" (ID code is 249) and followed by the calculated age data. The logics in the FPGA (decided by the FPGA design) store the time (in epoch) when a request was generated, and the age of the request is calculated periodically during the execution (at an evaluation period defined by one of the parameters from the "Read-Parameters-Process"). The host program then averaged the age of all requests generated during the calculation period.

The host program examines the ID code, extracts the corresponding data type, and analyses and interprets the data. The results were displayed in a window containing both the performance curve and the explanation statements, as shown in figure 5.5.

6.2 Experiment Results in the form of Performance Measurement

The real experiment results generated by the FPGA implementation are presented in the form of a data file that contains the long series of data. A typical data file for a four nodes implementation is contained in the attached CD-ROM as an appendix of the thesis. This section presents the results of the QoS performance measurement after host program analysis.

To generate the experimental results an input data file was created before the FPGA execution that provided the values of the important parameters in the "**Read-Parameters-Process**" (section 4.5.1.1) and values of the node's initialisation type in the "**Initialise-Node-Process**" (section 4.5.1.1). Table 6.2 lists the names of the parameters and Table 6.3 lists the names of the node initialisation types. For completeness, one example of this data file (named as configue.dat) is included in the attached CD-ROM.

Name of Parameter	**Meaning**
Epoch-Time	The length of an epoch period (in clock)
Evaluation-Time	The length of an evaluation period (in clock)
Request-Generate-Time	The time that is need to generate a request (in clock)
Request-Process-Time	The time that is need to process a request (in clock)
Update-Age-Time	How often the age data is calculated (in epoch)

Table 6.2 The input parameters

Name of the node's initialisation type	**Meaning**
Node's initial state	Defining the node's initial state
Node's initial rule set	Defining the nodes' initial rule set value
Node's initial busyness	Defining the nodes' initial busyness

Table 6.3 The node's initialisation type

6.2.1 QoS Measurement Results for Four Nodes Implementations

The experiment for four nodes implementation was performed using a range of parameters that covered different states of the network, such as a busy state, an idle state, or a reasonable load situation. The parameters "**Request-Generate-Time**" and "**Request-Process-Time**" in Table 6.2 determine how busy the network will be. If a large value is assigned to "**Request-Generate-Time**", then requests are generated at a low rate and the network will be less busy (i.e. relatively idle). On the other hand, if a larger value is assigned to the "**Request-Process-Time**", then it will take a longer time to process a request and the network will be more busy. Parameters "Epoch-Time" and "Evaluation-Time" define how long an epoch and an evaluation period will be. "Update-Age-Time" defines how often the age will be calculated.

A series of experiments was conducted to determine the QoS measurements for the FPGA implemented 4-node design under the different situations described above. Table 6.4 lists the key parameter values for each experiment. The QoS was measured over time as the load on the network was increased in a series of significant steps, 2X the initial load at epoch 500 and 4X the initial load at epoch 1500. The results of the QoS measurements are shown in Figures 6.1 to 6.6. The data file that contains the parameter values and node's initial configuration data for one of the implementations (corresponding figure 6.1) is put in the attached CD-ROM (named as config.dat) as an appendix.

Parameter	Figure 6.1	Figure 6.2	Figure 6.3	Figure 6.4	Figure 6.5	Figure 6.6
Epoch-Time	40	40	40	40	40	40
Evaluation-Time	200	200	200	200	200	200
Request-Generate-Time	34	32	30	32	25	15
Request-Process-Time	4	4	4	3	4	4
Update-Age-Time	7	7	7	7	7	10

Table 6.4 Parameter values in different experiments (4 nodes)

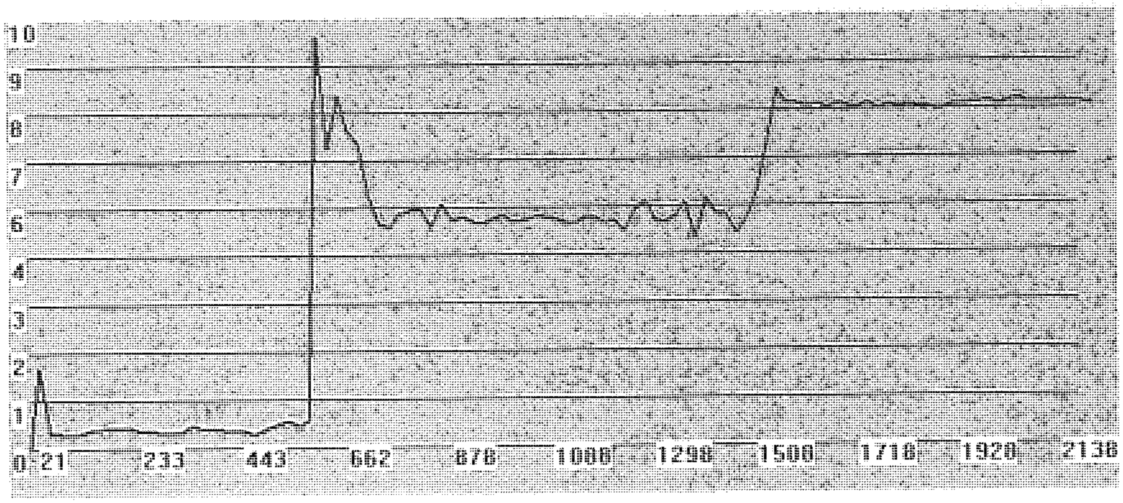


Figure 6.1 Experiment 1: QoS Average time of requests (epochs) v Time (epochs)

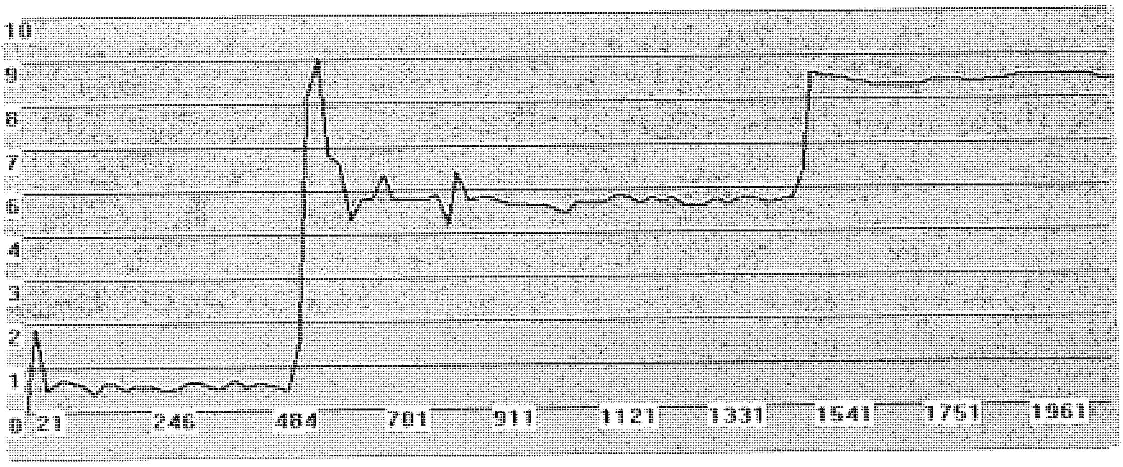


Figure 6.2 Experiment 2: QoS Average time of requests (epochs) v Time (epochs)

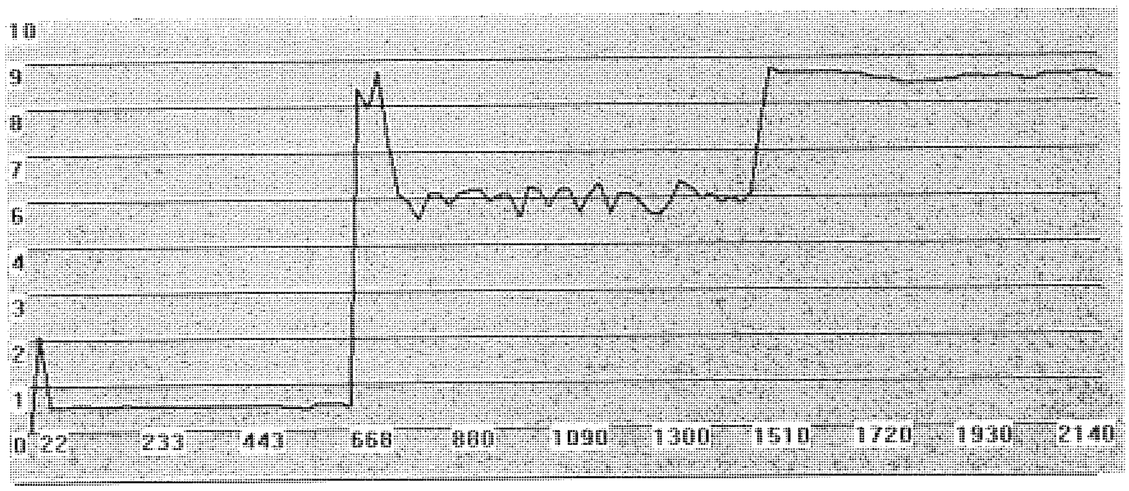


Figure 6.3 Experiment 3: QoS Average time of requests (epochs) v Time (epochs)

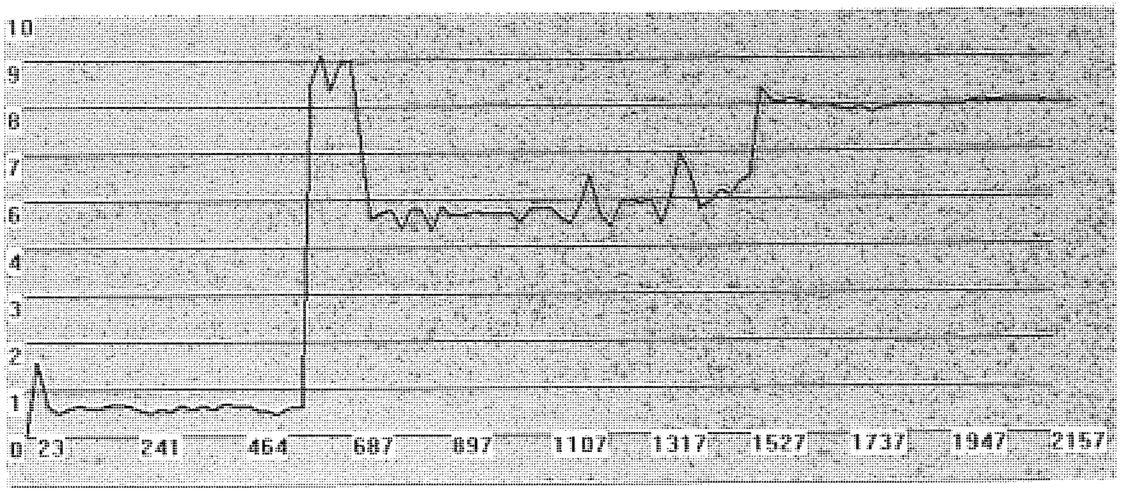


Figure 6.4 Experiment 4: QoS Average time of requests (epochs) v Time (epochs)

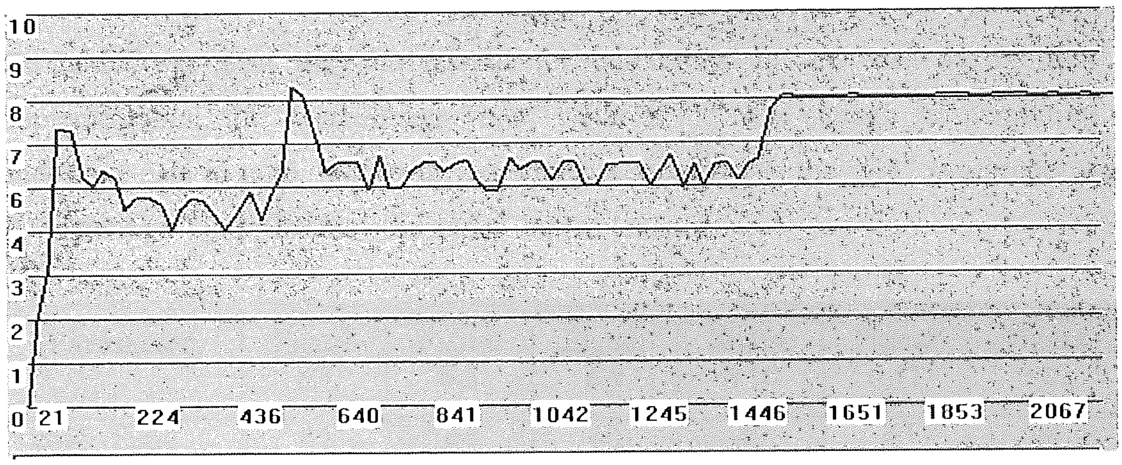


Figure 6.5 Experiment 5: QoS Average time of requests (epochs) v Time (epochs)

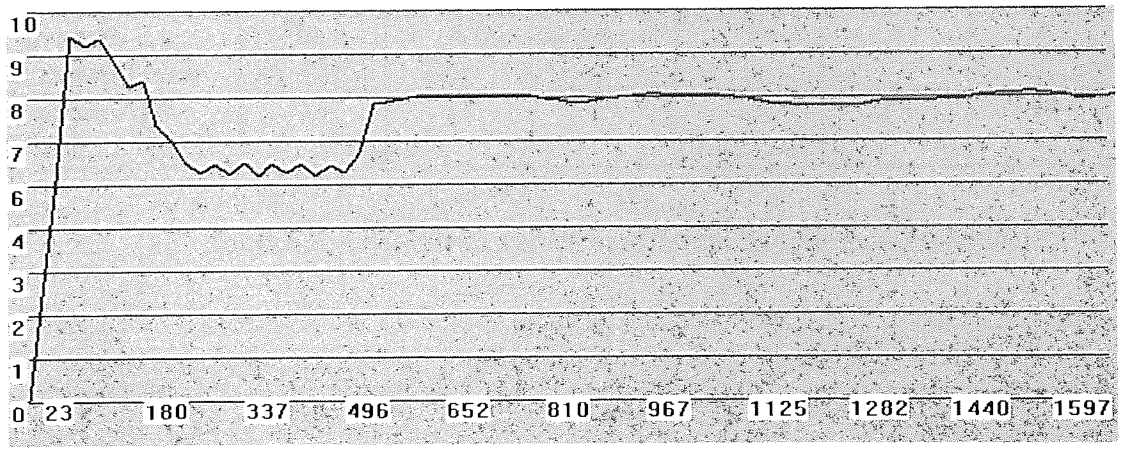


Figure 6.6 Experiment 6: QoS Average time of requests (epochs) v Time (epochs)

6.2.2 QoS Measurement Results for Nine Nodes Implementations

Experiments were also performed for the nine nodes implementation. Table 6.5 listed the key parameter values of all the experiments. Again, the QoS was measured over time as the load on the network was increased in some significant steps, 2X the initial load at epoch 500 and 4X the initial load at epoch 1500. The resulting QoS measurements are shown in Figures 6.7 to figure 6.12.

It is of note that the output of the age data for all the nodes is implemented in one 'macro' process using a "**prialt**" statement. That means that in the case of larger networks the output of all the node's age data can cause a bottleneck in the design. For the nine nodes implementation, if the value of "Update-Age-Time" parameter is defined too small, some of the nodes age data may be missed for outputting. In the case of the four nodes implementation, the "Update-Age-Time" value of "7" was enough for outputting all the nodes age data in time. However, experiments showed that the "15" is a proper value for "Update-Age-Time" for the nine nodes implementation.

Parameter	Figure 6.7	Figure 6.8	Figure 6.9	Figure 6.10	Figure 6.11	Figure 6.12
Epoch-Time	40	40	40	40	40	40
Evaluation-Time	200	200	200	200	200	200
Request-Generate-Time	34	32	30	32	25	15
Request-Process-Time	4	4	4	3	4	4
Update-Age-Time	15	15	15	15	15	15

Table 6.5 Parameter values in different experiments (9 nodes)

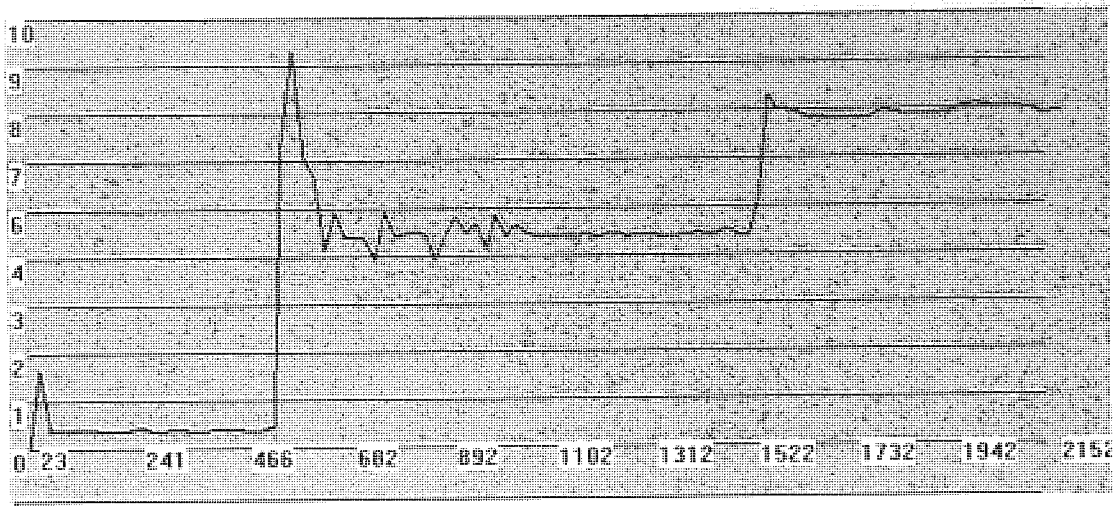


Figure 6.7 Experiment 7: QoS Average time of requests (epochs) v Time (epochs)

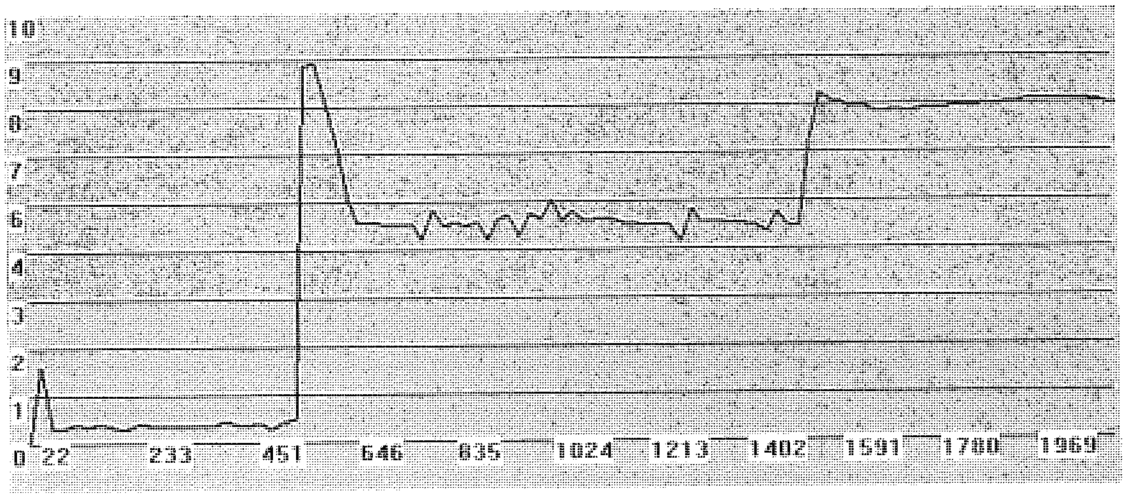


Figure 6.8 Experiment 8: QoS Average time of requests (epochs) v Time (epochs)

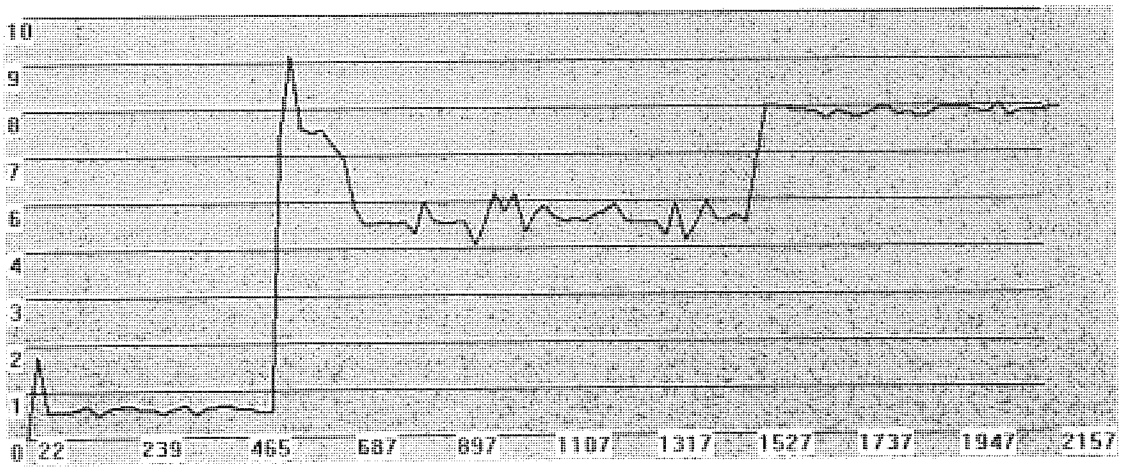


Figure 6.9 Experiment 9: QoS Average time of requests (epochs) v Time (epochs)

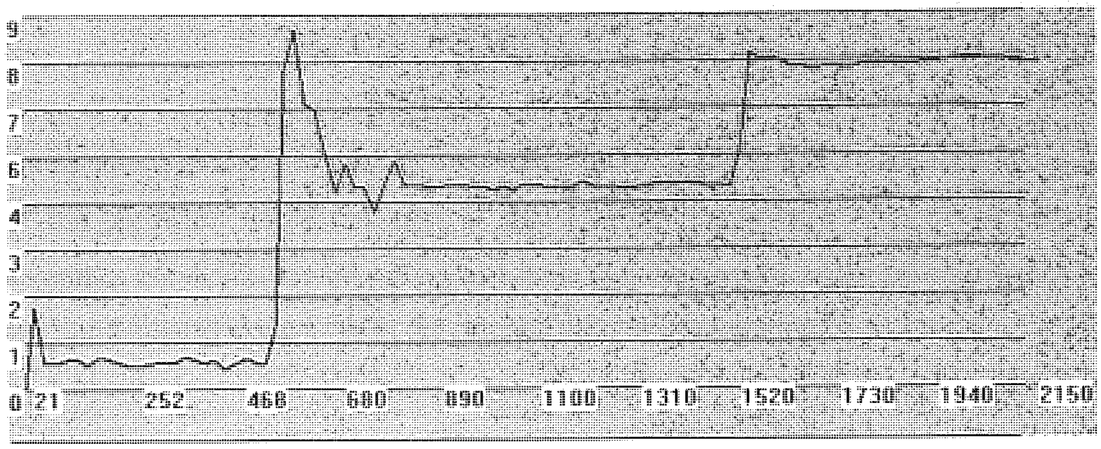


Figure 6.10 Experiment 10: QoS Average time of requests (epochs) v Time (epochs)

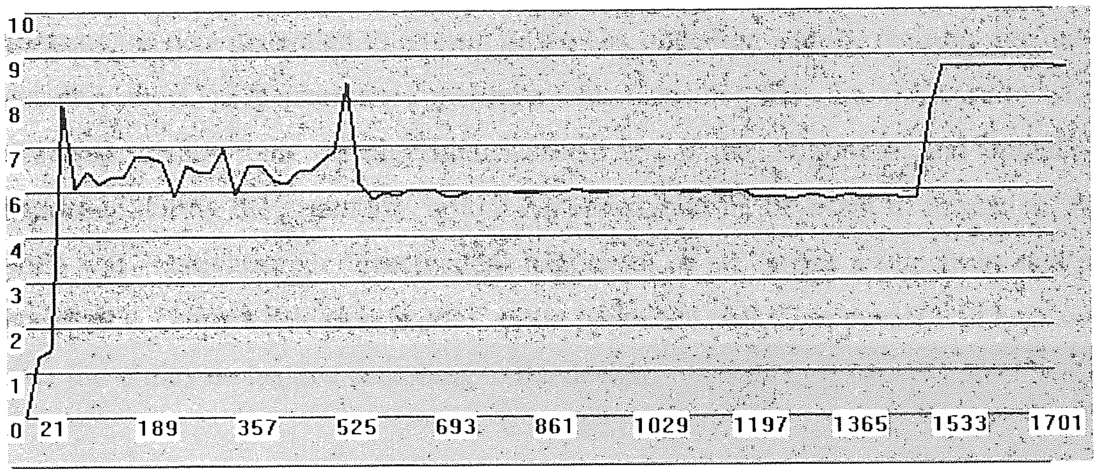


Figure 6.11 Experiment 11: QoS Average time of requests (epochs) v Time (epochs)

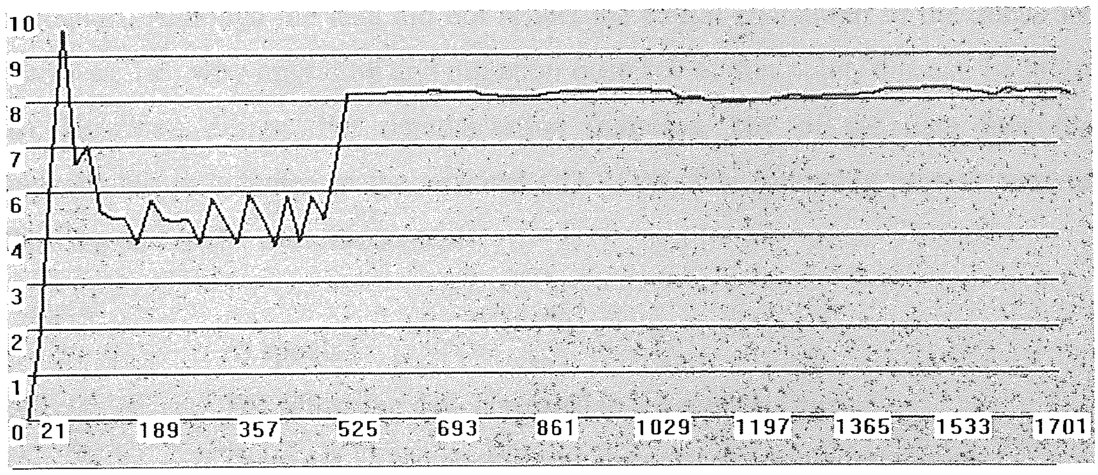


Figure 6.12 Experiment 12: QoS Average time of requests (epochs) v Time (epochs)

6.3 Analysis of Experiment Results

An intensive analysis of the results is presented in this section to demonstrate the efficiency of the hardware implementation of the network management system. The section also examines the difference between the 4-node and 9-node system and compares the results of the implemented systems with the original software simulation.

6.3.1 Analysis of the Hardware Implementation Result

The results generated by the hardware implementation were written into the data file that contains the data generated as a result of the GA operation and the requests age data used to generate the QoS performance measurement. Consider first the GA results which were analysed to show how the algorithm worked, i.e., if the 'migration' and 'mutation' event happened during the execution, how it happened, and in the event of migration, whether a node was 'activated' or 'deactivated' happened, or an "inject a rule from rule pool to a node" or a "copy a rule to rule pool" event happened. All these events will be shown in the data file if they do happen according to the design.

Each run of the implementations generated a separate output data file. Both the 'migration' and 'mutation' events were observed in the output data files. These events include all the five operations listed in table 6.1. That means the GA operation did occur during the FPGA execution. Although the data did not reflect the causal evaluation of the queue length and busyness, i.e., why migration and mutation occurred at that point, this can be inferred from the algorithm design. For demonstration purposes, one of the data files (named as nodeout.dat) is included in the attached CD-ROM. The following present some examples showing part of the data file contents.

```
............
7              // At epoch 7
249            // The following data is related to the requests age
15             // There are 15 requests during the current calculation period
28             // The cumulative age of the 15 requests are 28 epoch
1              // It's node 1's age related data
```

```
…………
20          // At epoch 20
254         // The following data is related to 'activate a node' operation
1           // It is node 1 that will activate another node
3           // It is node 3 that is activated by node 1
………..
50          // At epoch 50
252         // The following data related to a "copy a rule to rule pool' operation
0           // It is node 0 which will copy a rule to rule pool
1           // It is node 0's second rule that will be copied
3           // The rule will be copied and occupied the rule pool's four position
………..
80          // At epoch 80
250         // The following data related to a 'mutation' operation
2           // It is node 2's rule that will be mutated
2           // It is node 2's third rule that will be mutated
0           // It is the first element in the rule that will be mutated
1           // The new value of the element will be 1
………..
```

In another data file, the following data series was observed.

```
………..
30          // At epoch 30
251         // The following data related to 'injection a rule to a node' operation
4           // It is rule pool's fifth rule that will be injected
3           // It will be injected into node 3
2           // The injected rule will occupy the third position in the node 3's rule set
………..
40          // At epoch 40
253         // The following data related to 'deactivate a node' operation
1           // It is node 1 that will be deactivated
……….
```

Although very limited, the above example data series showed that the FPGA based system implemented the designed function of GA. 'Migration' and 'Mutation' operation are performed during the system execution to create the rule diversity. The physical system performs real time concurrent operations with each active node in the system working in parallel and exchanging the rules through a common accessed environment "rule pool". However this kind of analysis to the data file only demonstrate that the 'bacteria' inspired genetic algorithm were implemented in the real hardware. As far as the efficiency of the applied solution to the network management, it explains little.

Fortunately, the QoS measurement result has been acquired to demonstrate the efficiency of the 'bacteria' inspired solution. In section 6.2.1, figure 6.1 to figure 6.6 showed the performance measurement of the four nodes implementations in different situations. Table 6.4 listed part of the configuration parameters in each situation. In addition to these parameters, the initial configuration of each node such as the initial rule set configuration, initial busyness, initial node state (active or not active) will also affect the implementation result in the term of QoS measurement.

For all the QoS measurements shown in figure 6.1 to figure 6.6, the load of the network (decided by the parameter "Request-Generate-Time) was increased to 2X the initial load at epoch 500 and 4X the initial load at epoch 1500. These figures show how the QoS reacts to these increases. From the parameter values listed in table 6.4, it can be seen that figure 6.1 to figure 6.4 shows the situation when the network is relatively idle initially (the average value of "Request-Generate-Time" is 32). Figure 6.5 shows the situation when the network is relative busy initially (the average value of "Request-Generate-Time" is 25), and figure 6.6 shows the situation when the network is very busy (the average value of "Request-Generate-Time" is 15).

In figure 6.1 to figure 6.4, before the load is increased to 2X the initial load at epoch 500, the average age is 0.3 to 0.5 epoch, and after the increase in load the average age increases almost immediately to an average peak level of 9 epochs and then reduces to an average age of 5.8 to 6 epochs. After the load is increased to 4X the initial load at epoch 1500, the age increases again to an average age of 8 to 8.6 epochs and stays at this higher level. In figure 6.5, before the load is increased to 2X the initial load at epoch 500, the average age is 5 epochs. After the increase in load the age increases almost immediately to a peak level of

8.2 epochs and then goes down to an average age of 6.3 epochs. After the load is increased to 4X the initial load at epoch 1500, the age increases again to an average age of 8 epochs and never goes down. In figure 6.6, before the load is increased to 2X the initial load at epoch 500, the average age is 6.3 epochs. After the increase in load, the age increases almost immediately to a peak level of 8 and never goes down. Even when the load is increased to 4X the initial load at epoch 1500 the average age didn't change any more.

From the QoS measurements of figure 6.1 to figure 6.4, it can be seen that after the load is increased at epoch 500, a short period of high requests age (means worsened service) is observed. However, the age returns to a lower lever after a while. This phenomenon can be explained by the experimental result data generated by the FPGA. In all the situations from figure 6.1 to figure 6.6 the initial configuration sets only one node being active, the others are non-active. From the generated data file, it can be seen that when the network load is low at the first stage (before epoch 500), the GA operation of "Inject a rule from rule pool to a node" appears in the data file very frequently. That means the network is idle because the busyness value is low. After epoch 500, when the load is increased to 2X the initial load, the network begins to get busier. Shortly after epoch 500, it can be seen from the data file that all the three other nodes are activated one by one, i.e., the GA operation of "Activate a node" is observed in the data file three times and the related data showed that the activated nodes are just the nodes that are initially non-active. This adaptation, which increases the number of active nodes, results in the average age going down after the short period of increase. Again after epoch 1500, when the load is increased to 4X the initial load, the network begins to get very busy. This can be seen from the data file that from this point, the GA operation of "Copy a rule to rule pool" appears in the data file very frequently for each node. Since all the four nodes are active now, no more nodes can be activated to relieve the busy state of the network. The age is increased again to a high level and never going down.

Figure 6.5 shows the situation when the initial load is relatively large. This explains why the age is higher (about 5 epochs) during the first stage (before epoch 500) compared with that in figure 6.1 to figure 6.4. While after epoch 500, the same phenomenon can be observed as the others, although the average age only goes down to 6.3 epochs because of the higher initial load. Figure 6.6 shows the situation when the network is even busier initially than figure 6.5. As a result the three other nodes are activated very quickly even before the first increase in the load at epoch 500. This explains why in figure 6.6 the age increases to 8 epochs after epoch 500 and never goes down again because no more nodes can be activated to relieve the situation. The data from the resulting data file also proved this.

Thus, from the generated data file, it can be seen that not only the mutation and migration operations are observed in the data file, but also the data is consistent with the QoS measurement results.

Similarly, the QoS measurements for nine nodes implementation shown in figure 6.7 to figure 6.12 can be explained and analysed as below. Again, the network load is increased to 2X the initial load at epoch 500 and 4X the initial load at epoch 1500. These figures show how the QoS reacts to the increases. From the parameter values listed in table 6.5, it can be seen that figure 6.7 to figure 6.10 shows the situation when the network is relatively idle initially (the average value of "Request-Generate-Time" is 32). Figure 6.11 shows the situation when the network is relative busy initially (the average value of "Request-Generate-Time" is 25) and figure 6.12 shows the situation when the network is very busy (the average value of "Request-Generate-Time" is 15).

In figure 6.7 to figure 6.10, before the load is increased at epoch 500, the average age is 0.3-0.5 epoch, after that, the age is increased immediately to an average peak level of 9 epochs and then goes down to an average age of 4.8-5 epochs. After the load is increased to 4X the initial load at epoch 1500, the age is increased again to an average age of 8 epochs and never goes down. In figure 6.11, before the load is increased to 2X the initial load at epoch 500, the average age is 6.5 epochs. After that, the age is increased immediately to a peak level of 8.2 epochs and then goes down to an average age of 6 epochs. After the load is increased to 4X the initial load at epoch 1500, the age is increased again to an average age of 8.8 epochs and never goes down. In figure 6.12, before the load is increased to 2X the initial load at epoch 500, the average age is 5 epochs. After that, the age is increased immediately to a peak level of 8.1 and never goes down. Even when the load is increased to 4X the initial load at epoch 1500 the age didn't change any more.

From the QoS measurements of figure 6.7 to figure 6.10, it can be seen that after the load is increased at epoch 500, a short period of high requests age (means worsened service) is observed. However, the age returns to a lower lever after a while. Again, this phenomenon can be explained by the experimental result data generated by the FPGA. In all the situations from figure 6.7 to figure 6.12 the initial configuration sets only one node being active, the others are non-active. From the generated data file, it can be seen that when the network load is low at the first stage (before epoch 500), the GA operation of "Inject a rule from rule pool to a node" appears in the data file very frequently. That means the network is idle because the busyness value is low. After epoch 500, when the load is increased to 2X the initial load, the network begins to get busier. Shortly after epoch 500, it can be seen from the data file that more nodes are activated one by one, i.e., the GA operation of "Activate a node" is observed in the data file and the related data showed that the activated nodes are just the nodes that are initially non-active. (Depending on the initial load, all the other eight nodes may be activated or only parts of the other nodes are activated. In the case showed in figure 6.7 to figure 6.10, all the other eight nodes are activated after epoch 500). This adaptation in the size results in the average age going down after a short period of increase. Again after epoch 1500, when the load is increased to 4X the initial load, the network begins to get very busy. This can be seen from the data file that from this point, the GA operation of "Copy a rule to rule pool" appears in the data file very frequently for each node. Since all the nine nodes are active now, no more nodes can be activated to relieve the busy state of the network. The age is increased again to a high level and never going down.

Figure 6.11 shows the situation when the initial load is relatively higher. This explains why the age is higher (about 6.5 epoch) during the first stage (before epoch 500) compared with that in figure 6.7 to figure 6.10. While after epoch 500, the same phenomenon can be observed as the others, although the average age only getting down to 6 epochs because of the higher initial load. Figure 6.12 shows the situation when the network is even busier initially than figure 6.11, as a result the initially non-active nodes are activated very soon even before the first increase in the load at epoch 500. This explains why in figure 6.12 the age increased to 8.1 epochs after epoch 500 and never goes down again because no more nodes can be activated to relieve the situation. The data from the resulting data file also proved this.

Still, the mutation and migration operations are observed in the generated data file and the data is consistent with the QoS measurement results. It can be seen from these results that when the network load is increased a short period of worsened service s observed, but as the network adapts in both size and heterogeneity, the QoS return to a more acceptable level.

6.3.2 Comparison between four nodes and nine nodes implementation

Although the same phenomenon can be observed and the same conclusion can be made from both the four nodes and the nine nodes implementation, their QoS measurement results still show some difference between them.

To compare these two implementations results on an equal basis, the same parameter values are provided for both cases (table 6.4 and table 6.5) except the value for "Update-Age-Time". The reason why this parameter chose the different value has been explained before. Fortunately, its value does not affect the QoS measurement too much as it only provides a parameter of how often the age is calculated. However, as the size is different, the initial configuration for the nodes is different. For the four nodes implementation the configuration file provides the configuration data for four nodes only (i.e. the node's initial state (active or not active), each node's initial rule set and initial busyness value). In the case of nine nodes system, the configuration file has to provide the configuration data for nine nodes. For both implementations, there is only one node is active initially, the others are non-active.

From the QoS measurement results (shown in figure 6.1 to figure 6.12), it can be seen that in the initially idle case (figure 6.1 to figure 6.4 for four nodes implementation and figure 6.7 to figure 6.10 for nine nodes implementation), after the load is increased to 2X the initial load at epoch 500, the four nodes implementation has the age immediately increased to a peak level of 9 epoch and after a while returns only to a level of 5.8-6 epochs. However, although the nine nodes implementation immediately increases to the same peak level of 9 epochs, it returns to a lower level of 4.8-5 epochs, which is one epoch lower than for the four node system. In the initially busy case (figure 6.5 for four nodes implementation and figure 6.11 for nine nodes implementation), after the load is increased to 2X the initial load at epoch 500, the four node system has the age immediately increased to a peak level of 8.2 epochs and after a while returns only to a level of 6.3 epochs. However, the nine nodes system returns to a level of 6 epochs from the same peak level of 8.2 epochs, 0.3 epochs lower than the former. In the initially very busy case (6.6 for four nodes implementation and figure 6.12 for nine nodes implementation), as the initial load is very large, all the non-active nodes are activated soon after start up. As a result, the age of the four nodes implementation returns to a low level of 6.3 epochs before the first increase at epoch 500 and the age of the nine nodes implementation returns to a lower level of 5 epochs before the first increase.

From the above comparison, it can be seen that from the QoS point of view, the nine nodes implementation returns to a more acceptable QoS than the four nodes implementation after the load is increased. As the only difference in the design for the two cases is the network size, this contributes obviously to the difference between the two implementations results. This is understandable because the network adapts itself in both size and heterogeneity as a result of the applied genetic algorithm. Large network size means more nodes are available to be activated as the network get busier (the load is increased). The reason why the network size affects the performance while applying the 'bacterium' inspired GA solution is explained in detail in the next section after comparing the results of the former software simulation and that of the hardware implementation.

6.3.3 Comparison of the QoS Measurement Result Between Software Simulation and Hardware Implementation

The QoS measurement result of the software simulation presented in [12] by Ian Marshall et al was showed in figure 2.1, and is repeated here, Figure 6.13, for comparison with the results of the hardware implementation, as shown in figure 6.1 to figure 6.12.

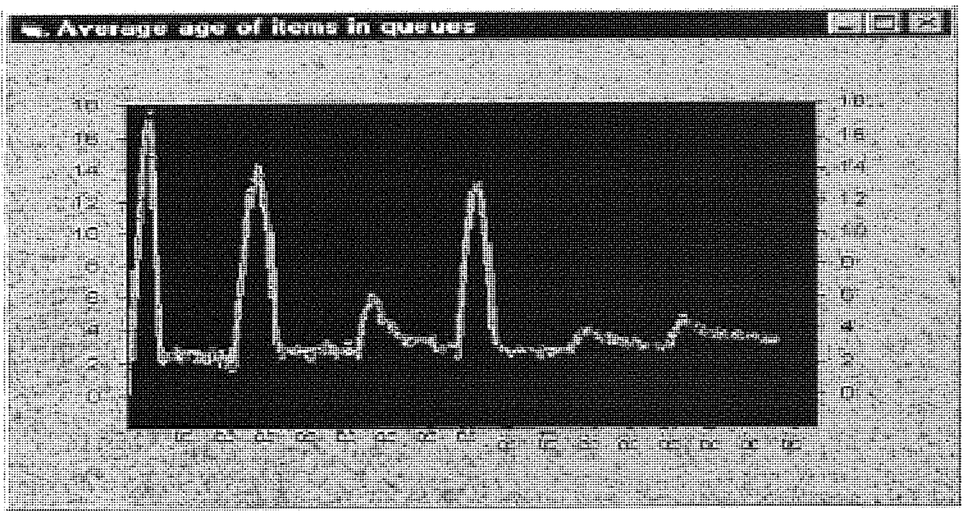


Figure 6.13 QoS level: Average time of requests (epochs) v Time (epochs)

In the terms of QoS measurement result, two distinct differences are observed between the software simulation and the hardware implementation. First, it is seen from figure 6.13 that as the load on the network is increased in a series of significant steps, 1, 4, 12, 30, 45, 60X the initial load, the requests age always returns to a lower level after a short period of high level (worsened service). Even at 60X the initial load, the network is nearing its saturation point, yet performance has degraded very little. Although a short period of worsened service is observed every time when the load is increased, as the network adapts in both size and heterogeneity, it returns to a more acceptable QoS at last.

However, for the four nodes and nine nodes hardware implementation, the QoS measurements were only taken as the load is increased to 2X and 4X initial value. This is because after the load is increased to 2X the initial value, all the nodes in the network become active (the network can only adapt in size after the first increase on load). After that no more nodes are available to be activated, i.e., there is no potential to adapt in size any more. Thus, even after 4X the initial load, the network is becoming too busy to handle more load. On the other hand, if the initial load is set to be very low so that after 2X the initial load it is still not busy enough to make all the nodes becoming active, however, in this situation, the initial active nodes may have been deactivated even before the load is increased.

In addition, it is seen from figure 6.13 that even after the load is increased to 60X the initial load, the age is still below 4 epochs after adaptation. However, the hardware implementation results showed a high age of 8-9 epochs after only an increase of 4X the initial load. That means the software simulation result showed a better QoS measurement compared with the hardware implementation.

This section put emphasis on the reason why there is difference between the software and hardware implementation results. As the applied solution (genetic algorithm) is completely the same for each case, the only reason will lie in the differences in the design. When the hardware is designed, four main changes are made to the originally proposed system configuration that is applied by the software implementation.

First, the size of the designed network, i.e., the number of the nodes designed within the system is different. The software simulation implemented a system of at least 400 nodes within the network, while the hardware implementation only contains 4 or 9 nodes in the network. Second, the size of the FIFO queue of each node, i.e., the max queue length, is different. The software simulation defined a max queue length of 200, while the hardware implementation defined the size as 32. Third, the size of the rule set of each node, i.e., the number of the rules in each node's rule set, is different. The software simulation defined a max number of four active rules for each node, while the hardware implementation only defined two active rules for each node. Fourth, the size of the rule pool, i.e., the available positions in the rule pool is different. Although Marshall et al did not give a detail figure in their document about the size of the rule pool, it can be inferred from the principle that the size is related to that of the network because the rule pool is accessible to all the nodes. The hardware implementation only defined 8 positions in the rule pool. Table 6.6 listed the different system configurations of the two implementations. How and why these different configurations can affect the performance measurement is analysed below.

Configuration / Implementation	Network Size	Queue Size	Rule Set Size	Rule Pool Size
Software Implementation	400	200	4	≅ 400
Hardware Implementation	4 (9)	32	2	8

Table 6.6 Different system configurations between two implementations

The size of the network affects the performance from five aspects. First, when the increased load makes the current active nodes busier, the network adapts in size by activating more nodes. However, in a small size network, such as the 4 or 9 node systems in the hardware implementation, all the nodes will soon become active after an increase in the load and the network will be unable to respond further by activating more nodes. It can only respond through the slower processes of rule migration and mutation (adaptation in heterogeneity, i.e., diversity of the rule set created by the applied GA). This behaviour has been demonstrated through the generated data in the data file for both four nodes and nine nodes implementations.

Second, the size of the network affects the performance in the way that rule diversity is created, i.e. migration and mutation. Migration results in rules from more active nodes being replicated into the rule pool and subsequently being absorbed into the rule set of the less active nodes, i.e., the rules can migrate from one node to the environment and may be absorbed by another node later. Through this way the rules can be exchanged among all the nodes and after a long term make each node more fit than before. From this point of view, the small size of the network means each node only has a small chance to acquire proper rules to make it even more fit because less nodes means less fit nodes and less proper rules.

Third, the small size of the network affects the network performance in the way that the un-matched requests are forwarded in the network through the "Request-Forwarding" process. Suppose a request named "a" is injected into node 0, and doesn't match any rules of this node. The request is then forwarded to one of node 0's neighbours, node 1 or node 2 in four-node system (the connection relationship between the nodes is shown in figure 6.14) that is selected randomly, suppose node 1. If this request doesn't match any rules of node 1 either, it will be forwarded again to one of node 1's neighbours, still selected randomly, suppose node 0 this time. That means the same request could be transferred forwards and backwards between the two nodes for a very long time and not being processed by any of the nodes. This conclusion was proved earlier during the simulation stage as it is easier to trace the behaviour through debugging than in the hardware implementation. This situation will lead to a worse QoS measurement because the average age of the requests will be increased. If there are more nodes in the network this situation could be relieved because more nodes can provide more chance for a request to match one of the node's rule.

Fourth, the small size of the network affects the network performance in the way that the requests are matched to one of the node's rule. When a request does not match any of the node's rules, it will be forwarded to another node. If it does not match any of another node's rules, it will be forwarded again. In another word, the more the rules, the more chance for a request to be matched and the less the chance to be forwarded. But small size network means there are less rules used for matching, and thus provides less chance for a request to be matched and in turn more chance to be forwarded.

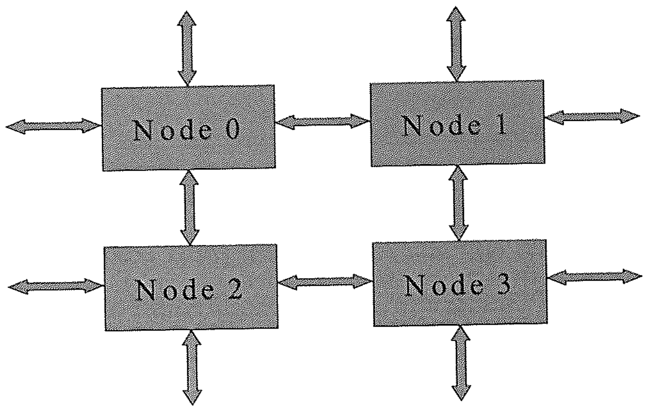


Figure 6.14 Connection relationship between nodes (four nodes system)

Fifth, the small size of the network affects the network performance in the way that parameter values are chosen. For a small size network, it is very difficult to choose a proper parameter value. Take the example of four nodes implementation, if the initial load is chosen to be a little high, then after the first increase in the load (2X the initial load), all the nodes in the network become active. Thus the QoS can never return to a lower level after the second increase (4X the initial load). On the other hand, if the initial load is set to be a little low so that after 2X the initial load it is still not busy enough to make all the nodes becoming active, however, in this situation, the initial active node may has been deactivated even before the load is increased.

These influences of the network size to the performance can also be used to explain the difference between a four-nodes system and a nine-nodes system performance introduced in section 6.3.2.

The size of the FIFO queue defined for each node in the system also affects the network performance. Small value of the max queue length such as "32" defined in the hardware implementation will cause the queue becoming full very soon after the load is increased. According to the design, if the queue is full when a request is insert, this request will be missed for the service by the network. This will lead to a genuinely worse quality of service (QoS) from the customers' point of view. When calculating the age of the requests, if requests are missed then the resulting QoS measurement cannot really reflect the situation exactly.

The size of each node's rule set (number of rules of each node) affects the performance in the same way that small size network affects the performance with respect to matching a request with the rules. Less rules means a request has less chance to be matched to one of the rules and thus lead to more chance to be forwarded. The hardware implementation only defines 2 active rules and 2 dormant rules for each node and the software simulation defines at least 4 active rules and more dormant rules for each node.

The way that the rule pool size affects the performance can be understood as below. The rule pool is the place into which the more active nodes copying their proper rules. More positions in the rule pool mean more proper rules can be stored in it and more chance can be provided for a not active node to absorb them and thus more requests can be matched and processed.

From the above analysis, it can be seen that the larger the size for all the configuration parameters listed in table 6.6, the better the QoS performance measurement. This conclusion explains not only why the software simulation result is better than the hardware implementation, but also why the nine nodes implementation is better than the four nodes implementation.

In summary, for a small size network with small size rule set, small queue length and less positions in the rule pool, it will be very difficult for the requests to be matched and processed. The delay of the requests in the network directly results in the high age and thus a poor QoS of performance measurement. However, at the moment, the hardware implementation can only chose the small value for the above configuration parameters because of the hardware resource limit. While on the other side, it is easier for the software simulation to expand the design.

Except the differences in the system size configurations, other differences such as the definition of the parameter values listed in table 6.2 and the definition of the node's initialisation type values listed in table 6.3 also contributes to the difference between the software and hardware implementations results.

As far as the parameter values and the nodes' initialisation type values are considered, different combinations of different values have different effects on the implementation result. The value for the five input parameters listed in Table 6.2 and all the initialisation types are randomly selected in the experiment. Inspection of Table 6.3 shows that there are 14 initialisation types (this includes '1' node's initial state value, '12' node's rules values (each node has '4' rules and each rule has '3' elements), and '1' node's initial busyness value). Thus each implementation of the four nodes system will involve 61, i.e. (5+4*14) values and the nine nodes system will involve 131, i.e. (5+9*14) values. Therefore it was not possible to cover all the combinations of different values under the limited experiments. Table 6.4 and Table 6.5 lists those parameter values actually used in testing the implemented designs.

Although there are so much difference in the design and performance between hardware implementation and software simulation, the conclusion for the applied solution is still consistent. That is because of the long-term self-stabilising, adaptive nature of bacterium communities, a bacterial type network management algorithm might be a suitable approach to creating a stable network of autonomous nodes. That overall network stability, in terms of QoS, is provided by a set of cells that are acting for their own (not the network's) good. This removes most of the high-level network management problems and thus has a great significance when applied in the real world network.

6.3.4 Analysing the Role and Effect of 'Mutation' in the Design

Mutation is one of the operators used by the general genetic algorithm, worked together with other operators such as crossover. Although mutation plays a role in the GA process, how important its role is continues to be a matter of debate according to [73]. Some refer to it as a background operator, while others view it as playing the dominant role in the evolutionary process.

As mutation involves the random alteration of the candidates, speaking from the theory, the result after mutation may be better in some iteration and worse in others. Actually, there are no mathematical proofs that indicate mutation will eventually produce an optimal solution. No evidence is found to support the assertion that the goal of mutation is to produce the fittest solution. In the general genetic algorithm that using mutation operator, candidate solutions are chosen according to the result of a fitness function. Those who satisfied the fitness criterion were chosen and regarded to be better than others. It is believed that those that are better are more likely to survive and propagate their genetic material from the processes of evolution in nature.

To find out how much influence mutation has exerted on the designed system, an extra experiment was completed that implements the four nodes system without applying mutation operator in the applied GA (i.e., only migration was applied for the diversity of the rules). Four implementations were completed to compare the results under different situations (with and without mutation). Table 6.7 listed the selected parameter values for each of the cases and figure 6.15 to figure 6.18 showed the corresponding QoS measurements. Still, the network load was increased to 2X the initial load at epoch 500 and 4X the initial load at epoch 1500.

Parameter	Fig 6.15 (with mutation)	Fig 6.16 (without mutation)	Fig 6.17 (with mutation)	Fig 6.18 (without mutation)
Epoch-Time	40	40	40	40
Evaluation-Time	200	200	200	200
Request-Generate-Time	42	42	41	41
Request-Process-Time	5	5	6	6
Update-Age-Time	7	7	7	7

Table 6.7 Parameter values in different implementations (4 nodes)

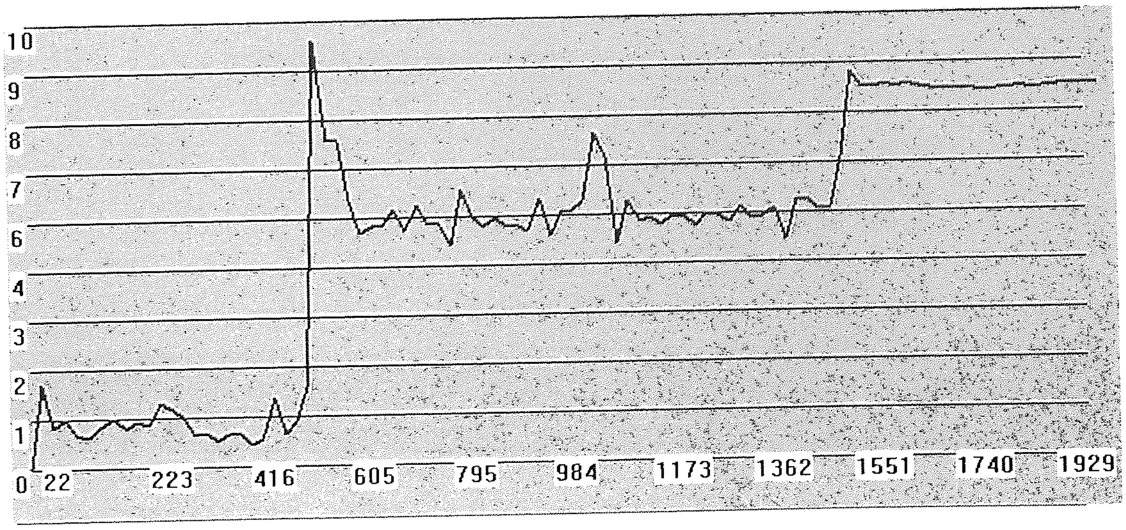


Figure 6.15: QoS with mutation: Average time of requests (epochs) v Time (epochs)

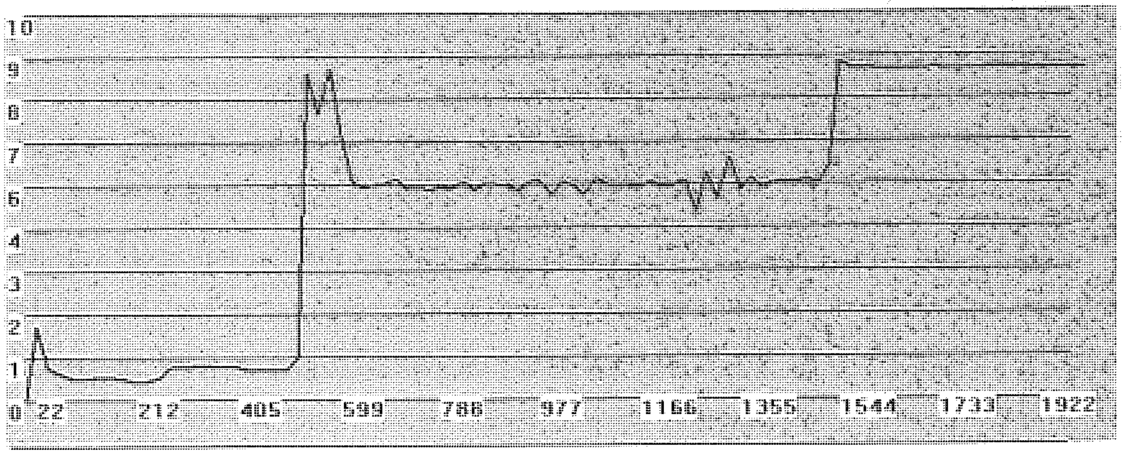


Figure 6.16: QoS with mutation: Average time of requests (epochs) v Time (epochs)

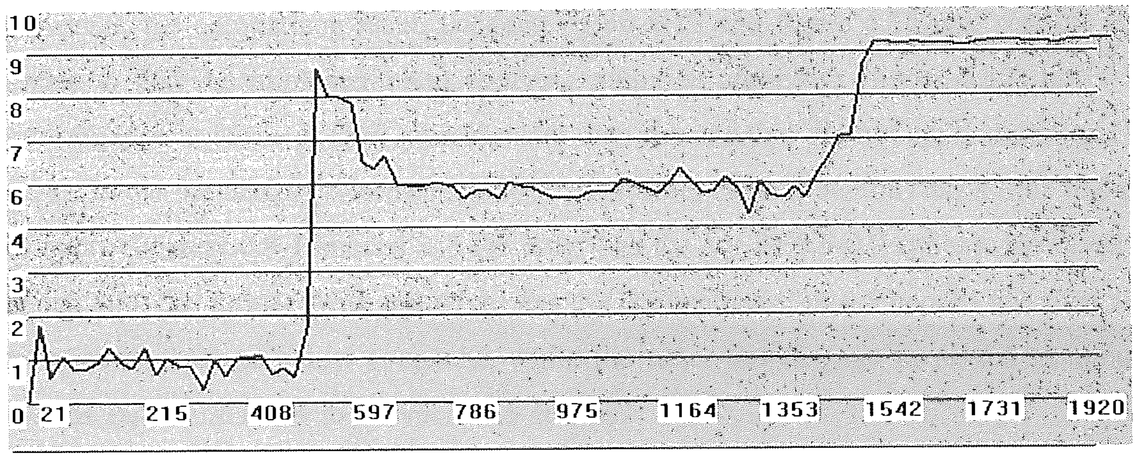


Figure 6.17: QoS with mutation: Average time of requests (epochs) v Time (epochs)

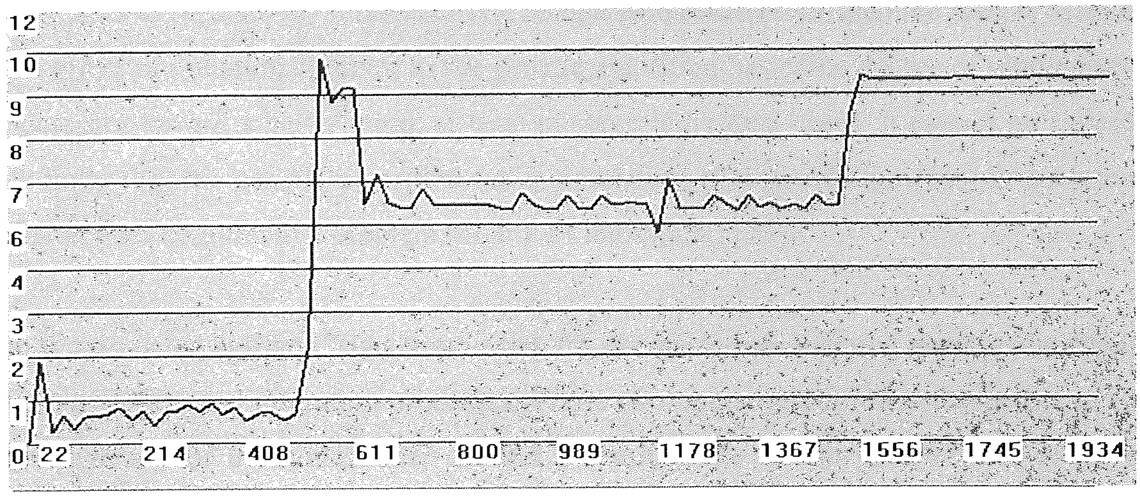


Figure 6.18: QoS with mutation: Average time of requests (epochs) v Time (epochs)

Comparing the QoS measurement result showed in figure 6.15 and figure 6.16, it can be seen that the latter (without mutation) seems even better than the former (with mutation) as in figure 6.16 the age level is more stable and lower. Before the network load is increased to 2X the initial load at epoch 500, the former has an average age of approximately 0.9 epochs and the latter has a lever of 0.7 epochs. After the first increase of the network load, the former has the age increased immediately to a peak level of 9.6 epochs and then returns to an average level of approximately 6 epochs but with a spur up to 7.6 epochs at about epoch 1000. However, the latter has the age increased immediately to a peak level of only 8.8 epochs after the first increase of the network load and also returns to an average level of 6 epochs, and maintains this value with better stability compared with the former.

However, the comparison result between figure 6.17 and figure 6.18 will support another assertion that the implementation applying mutation algorithm (figure 6.17) has a better performance in terms of QoS measurement than the one that did not apply mutation (figure 6.18). In the first stage (before epoch 500), the former (with mutation) has a little higher age of approximately 0.9-1 epochs and the latter has the age of 0.5-0.9 epochs. However, after epoch 500, the former (with mutation) has the age increased to 8.6 epochs and then returns to a lower level of below 6 epochs, while the latter has the age increased to almost 10 epochs and returns only to a level of 6.5 epochs.

More experiments have been completed to compare the result under the different situations of with and without mutation algorithm. But due to space constraints, the thesis only present two cases to indicate the influence. The comparison result confirmed the former opinion that the effect of mutation operator in the genetic algorithm has some degree of randomness, not necessary toward a better trend, at least in some individual cases. It cannot be stressed too strongly that the genetic algorithm (as a simulation of a genetic process) is not a random search for a solution to a problem (highly fit individual) [73].

In the designed system ("Bacterium Model"), the bacterium inspired genetic algorithm uses both mutation and migration processes. These two operations combined together to change the diversity of the nodes' rule set. As a result, some of the good effects of migration process may be changed by the mutation operation. This can be explained and understood though the following example.

Suppose there is a rule injected from the rule pool to a node's rule set. As this rule originally comes from another more active node's rule set (when the node's busyness or queue length is above a threshold, it will copy one of its rules to the rule pool), the injection action itself is one of the migration operations trying to improve a nodes' behaviour in processing more requests. However, if before the new injected rule comes into effect, (i.e., absorbed by the node into its rule set and used for the matching requests process), mutation happened on it, that means the rule was altered randomly, then the migration operation will not have the desired effect. This is possible because mutation operation is completely random. Any alteration is possible at any time. This example just illustrates a possible way that mutation influences the network performance in the special case of "Bacterium Model" (or bacterium inspired genetic algorithm solution).

Except the influence on the network performance, the application of mutation process also has an influence on the system design performance. This is obvious because the adding of mutation process made the designed system even more complex, used more hardware resources (logic) and takes more time to be executed by the FPGA. To find out how the mutation process will increase the used resources and decrease the design performance, table 6.8 lists the predicted resource usage and clock speed after place and route for both the four nodes and nine nodes implementations under the different situations of with and without mutation.

It can be seen from the table that the adding of mutation makes the used total equivalent gate count increase by 6.77% for the four nodes implementation and 5.73% for the nine nodes implementation. And the maximum operating frequency decreased by 2% for four nodes implementation and 7% for nine nodes implementation.

The figure in the table also explained why the current design only implements a maximum network size of nine nodes. Because the nine nodes implementation has used 99% of the total number of slice, any significant extension in the size will not be implemented on the current used platform.

		Implementation with Mutation	Implementation without Mutation
Four Nodes System Implementation	Number of Slice	6,654 (34%)	6,084 (31%)
	Number of Slice Flip Flops	3,073 (8%)	2,833 (7%)
	Number used as LUTs	10,191	9,297
	Total equivalent gate	115,833	107,985
	Maximum Frequency (MHz)	16.88	17.24
Nine Nodes System Implementation	Number of Slice	19,198 (99%)	18,045 (93%)
	Number of Slice Flip Flop	6,596 (17%)	6,056 (15%)
	Number used as LUTs	31,965	29,946
	Total equivalent gate	309,108	291,405
	Maximum Frequency (MHz)	14.28	15.37

Table 6.8 System design performance under the different situations

6.4 Scalability of the Hardware Implementation

Although the current hardware design only implements a four-nodes or nine-nodes system, the design is still scalable. As a nine-nodes system can be implemented in one FPGA chip on a single RC1000 board, by interconnecting two or more development board and thus using more FPGA devices, a larger network system with more nodes can be implemented. For example, connecting four RC1000 boards together with proper interface between the FPGAs can implement 4*9 = 36 nodes based on the current design. As it is not necessary to implement a real large network, such as a network with 1000 nodes for the experiment purpose, the current implementation of "Bacterium Model" system has only four or nine nodes connecting together to make a simple network and was only used as a prototype to demonstrate the feasibility and the efficiency of the proposed management solution for the future active network.

6.5 Summary

This chapter presents and analyses the hardware implementation results of both the four nodes and the nine nodes system. The algorithm performance in terms of the QoS measurement was compared not only between the two hardware implementations but also between the hardware implementation and the software simulation. Analysis of the hardware implementation result and the reason why there is difference between the implementations, especially between software simulation and hardware implementation are also provided.

The hardware implementation demonstrated that the solution based on 'bacteria' inspired genetic algorithm could be implemented in an FPGA-based re-configurable platform and provided an adaptive management for an active network containing active nodes. It also demonstrated the self-stabilising and adaptive nature of the active nodes that employed the designed algorithm. It shows that this biologically inspired algorithm could be a suitable approach to creating a stable network of autonomous nodes in a real network. Although it is still too early to use the solution in a real network, the experimental results have revealed the advanced feature of this active network with active nodes performing cellular adaptive genetic management algorithm.

The prototype system was necessarily constrained in the network size that could be implemented on the current used experiment platform, however the design specifically provides potential for the scalability of the hardware implementation.

Chapter 7 Conclusion

This chapter presents the conclusion for the whole thesis, summarizes the contributions and gives some suggestions for the future work.

7.1 Conclusion

This thesis begins with an introduction of the background of the whole work described here. With the fact that the fast development of the Internet and the increasing demands of the service is bringing a great challenge to the current network, the thesis explains why the traditional network cannot meet the new requirements any more and thus needs radical changes in the structure and management of underlying telecommunications systems. Active network was explored for many years [3] as a new network paradigm for this purpose.

Active network has been shown to address the problems of the traditional networks [4][5][6] such as the difficulty of integrating new technologies and standards into the shared network infrastructure, poor performance due to redundant operations at several protocol layers, and difficulty of accommodating new services with the existing architectural model [3]. This is because the active network is based on the idea that the network can be programmed. That means the network nodes such as the routers or switches have the ability to perform computation through executing the passing code sent by the users and thus implement custom services.

However, the intended flexibility of an active network cannot be fully realised unless it is combined with a highly automated management and control system [8]. This thesis introduced and implemented an adaptive management solution for the active network. It is a novel solution that uses a distributed genetic algorithm inspired by bacterium on the active nodes within an active network to provide adaptive management for the network. The GA used in this solution is inspired by the mechanisms that bacterium use to transfer and share genetic material.

A system model based on this solution was developed and incorporated in a software simulation of a network comprising a number of nodes connected on a rectangular grid. The simulation results showed that this bacterium inspired network management algorithm might provide a suitable approach to creating a stable network of autonomous nodes.

The work introduced in this thesis involves the analysis, design, software simulation and hardware implementation of such a system model. It is a prototype design and implementation of an active network with 'bacterium' type active nodes using GA to implement an adaptive management. The hardware implementation of the prototype system was completed use a re-configurable computing platform based on FPGA processor.

The thesis first introduced the background of the work and thus explains the purpose of the research. A software simulation of the model based on the proposed solution was then introduced. This simulation has been performed by the pioneer researchers in this work field [11] who generated QoS measurements to demonstrate their systems performance. The thesis presented their simulation result for both the purpose of comparison with a new explored algorithm solution and the purpose of demonstrating the effectiveness of the bacterium inspired solution for the future network management.

The new proposed algorithm was based on the original GA and has demonstrated its improvement on the system performance. The new algorithm was developed as a revised version of the traditional GA of "mutation". As "mutation" involves random alteration of part of the "genome", the revised one made a deliberate 'informed' alteration to a rule instead of the random alteration based on some statistic result. That means each time before a mutation was performed, a statistic was made to decide which way the mutation should go. In another word, the revised algorithm process was developed in which the "mutation" is directed towards a desired solution, rather than being completely random.

Chapter 2 introduced the simulation results of both the original algorithm and the new improved one. A difference can be seen in the system performance that demonstrates the advantages of the new improved algorithm. This can be proved from both the QoS (in terms of average age) measurement and the discarding rate measurement. Chapter 2 showed the results of the comparison. It can be seen that the discarding rate was decreased by more than 50 percent with the new algorithm. And the average age was decreased by 20 percent. This introductory work was only taken as far as software simulation. The hardware implementation of the system model introduced later was based on the original solution and is the key task in this thesis.

The thesis also introduced some basic theories and concepts such as evolutionary computation, re-configurable computing, cellular computing, concurrent programming and parallel processing. These theories and concepts were introduced as they are related to the proposed solution and the explored prototype system. For example, the use of a GA in the management solution relates to evolutionary computation and the use of re-configurable FPGA based processors for the hardware implementation involves the re-configurable computing. The whole system was modelled as a community of cellular automata as each node performs cellular computing. The concurrent programming language Handel-C was used to design and describe the system. Finally, the notion of parallel processing is reflected in both of the properties of Handel-C as a software entity and in the concurrent digital electronic FPGA-based design that is synthesised directly from Handel-C.

A system model with four and nine active nodes were described and designed using a concurrent programming language Handel-C and implemented on a FPGA based re-configurable platform. The use of the language Handel-C and an FPGA-based processing element allowed parallelism to be exploited directly in the system design and implementation.

Before the implementation of the explored system on the real hardware based on FPGA, the design and development of the same model using hardware description language was completed and the software simulation was also performed. The prototype system was initially described using a traditional hardware description language VHDL and this was used to implement one node's function only and provided no connection relation between adjacent nodes.

Compared with VHDL, the high-level concurrent program language of Handel-C is more flexible and easier to design a more complex system at a higher abstract level. Handel-C can be used to design a system without the burden of considering the low level structure and is more like a traditional software programming language such as C. With Handel-C, the system was designed as a model that includes four nodes connected on a rectangular grid and was later extended to form a nine-node model.

This thesis introduced the design and the implementation of the model in every detail from code description to the device used. The introduction of the Celoxica DK1 Handel-C design environment and the Celoxica RC1000 development system equipped with a Xilinx Virtex 2000 FPGA provides both a modern, efficient concurrent software design environment and a solution to the implementation problem.

The developed model was then implemented on the FPGA based hardware and the implementation information was provided in terms of hardware resource use and the attainable maximum clock frequency for implementation performance measurement. For the four-node system implementation, the hardware resource was used 34% in terms of the slices number and the maximum clock frequency can attain 16.88MHz. However, the nine-node system implementation used 99% hardware resource and can only attain a maximum clock frequency of 14.28MHz. That's why the one FPGA platform board can only afford a system with at most nine nodes. The experimental results from the FPGA implementation were output and analysed to determine both the effectiveness and efficiency of the explored management solution. Both four-nodes system and nine-nodes system were implemented.

The thesis presents the hardware implementation results including the QoS measurement of the modelled network. The QoS measured the average age of all services on the network. It was measured over time as the load on the network was increased in a series of significant steps. The figures showed how the QoS reacts to the increases. Whenever the network load was increased, a short period of worsened service can be observed, but as the network adapts in both size and heterogeneity, The QoS returns to a better level than the worsened one. The experimented results showed that, because of the long-term self-stabilising and adaptive nature of bacterial communities, a bacterial type network management algorithm might be a suitable approach to creating a stable network of autonomous nodes. The QoS and overall network stability is provided by a set of cells that are acting for their own good. This removes most of the high-level network management problems.

The results in terms of the QoS measurements were compared not only between the four nodes and the nine nodes implementation, but also between the previous software simulation and the hardware implementation. It was seen that the QoS measurement result of the software simulation is better than that of the nine-node system and the latter is better than that of the four-node system. That means the adaptive nature of bacterium type solution was showed in the software simulation result better than in the nine-node system hardware implementation. Similarly, the nature was showed better for the nine-node system than for the four-node system.

The thesis explains why there are differences between them and how these differences are caused. The most important factor that brought the difference was the defined size of the network. The larger the network size, the better the QoS performance. This is because the size of the network played a main role in the way the distributed GA was implemented. For example, when the increased load makes the current active nodes busier, the network has to adapt in size by activating more nodes. In such a case, a large sized network means there are more nodes to be activated and thus result in a better performance. Similarly, the size of the network affects the performance in the way that how the rule diversity is created.

Except these, the parameter values for the FIFO queue size, the node's rule set size and the rule pool size also played a role in the performance difference between the software simulation and the hardware implementation results.

Although the hardware model implemented only four or nine nodes, and this affected the measured performance and QoS, the conclusion is still consistent with the original software simulation. That is a bacterial type network management algorithm might provide a suitable approach to creating a stable network of autonomous nodes because of the long-term self-stabilising and adaptive nature of bacterial communities.

One special experiment was performed to determine how important the mutation algorithm plays a part in the traditional genetic algorithm. As mutation only involves the random alteration of the candidates, thus in theory the result after mutation may be better in some iteration and worse in others. Actually, there are no mathematical proofs that indicate mutation will produce an optimal solution at last. No evidence is found to support the assertion that the goal of mutation is to produce the fitness solution. In the general genetic algorithm that uses a mutation operator, candidate solutions are chosen according to the result of a fitness function. Those who satisfied the fitness criterion were chosen and regarded to be better than others. It is believed that those that are better are more likely to survive and propagate their genetic material from the processes of evolution in nature.

The results were presented and compared in the form of QoS measurements. The comparison result confirmed the opinion that the effect of mutation operator in the genetic algorithm has some degree of randomness, not necessary toward a better trend, at least in some individual cases. It cannot be stressed too strongly that the genetic algorithm (as a simulation of a genetic process) is not a random search for a solution to a problem (highly fit individual) [67].

The current hardware design only implements a four-nodes or nine-nodes system, but the design is still scalable. As a nine-nodes system can be implemented in one FPGA chip on a single RC1000 board, by interconnecting more development boards and thus using more FPGA devices, a larger network system with more nodes can be implemented. The current implementation of "Bacterium Model" system with only four or nine nodes connecting together to make a simple network was only used as a prototype to demonstrate the feasibility and the efficiency of the proposed management solution for the future active network.

7.2 Contributions

The first and most important contribution of the work described in this thesis is the real hardware implementation of a system, which, as an adaptive control management solution for the future network, provided a prototype of an active network model with active nodes performing distributed GA inspired by bacterium. The prototype system was designed and described using Handel-C and was eventually implemented on a re-configurable computing platform to perform evolutionary computation. It is the first time that this network model has been implemented in FPGA-based hardware.

The result demonstrates not only the feasibility of applying the 'bacterial' type management solution on the real network, but also the efficiency of the solution itself. Because of the adaptive feature of the solution based on a biological inspired GA, the model could provide more efficient and flexible management and thus meet the requirement of future complex network.

The QoS was measured for the implemented network model and analysis showed how the explored solution worked for the simple management task like service provision. It also showed how the genetic algorithm functions such as the migration and mutation worked and how they affect the performance. The effect of the migration is obvious and can be testified easily from the figures and the output data result. However, the effect of the mutation cannot be proved so easily. How exactly important of the mutation in the whole genetic algorithm as one of the special operators is still an open topic for research, this thesis only provide a lightly demonstration with some examples of the testing result.

The second contribution of the work described in this thesis is the proposition and testifying of an improved genetic algorithm related to the traditional mutation one. As described in chapter two, a new mutation algorithm based on learning from history was proposed and testified. The principle of the new algorithm was explained and the reason why it may improve the performance was analysed.

It is only because the effect of the traditional mutation cannot be decided exactly that the new algorithm was proposed and which is more based on a reasonable principle. A simulation of the model that employed the new algorithm was performed and the result was compared with the one that used a traditional mutation algorithm. It was shown that the new algorithm has more advantage than the old one in improving the network performance in the terms of QoS measurement.

The third contribution of the work is the development of the host program that was used to communicate with the FPGA on the re-configurable computer board. The host program was used to configure the FPGA, provide the initial parameters and analyse the output result. The host program is developed using VC++ and executed successfully to configure FPGA and communicate with the FPGA board, thus implementing a real re-configurable computing machine.

7.3 Suggestions For Future Work

The work to implement a practical active network with each active node performs cellular adaptive management function using GA is still in its infant stage. There are still a lot of areas that need to be improved and investigated before applied to a real world network.

The first part that needs improving in the design is the 'rule pool' part in the management solution that is applied in the exploring system. Although the network has been implemented as a cellular automata, within which each node only responsible for its own behaviour and performs the distributed GA, the 'rule pool' is still a central concept because it is accessible to all nodes. As long as a central part exist, a bottleneck will happen and the performance will be degraded. Furthermore, to implement a global rule pool is not possible in real world network because of the wide spread of the computer nodes.

Clearly, a way to use a distributed 'rule pool' may be considered. Each small area, like a LAN, could use a 'rule pool' that is connected to a number of nodes within the limited area. It is an alternative solution if each limited area is not too large. An even more efficient solution is to make the 'rule pool' a local concept, instead of a global one. How to implement this on the same FPGA based platform will be one of the most important tasks in future work.

Secondly, the present work only implements a small system with at most nine nodes in the network model. This size of the network is not large enough to benefit from the advantage of the GA, as has been stated in chapter 6. The future work may involve the expanding of the system to contain more nodes. The current used re-configurable computer platform, RC1000 board, has only one FPGA chip on it. In the current design, one chip could only accommodate nine nodes. To implement a 100 nodes system will need at least 12 such platforms to connect together. The current system will then need a hardware interface with outside systems.

Another way to expand the system is to improve the current design and try to make each FPGA chip accommodate more nodes, suppose 24, thus the required number of platforms will be decreased to 4 for implementing a system of 96 nodes. Although this number is still not large enough to represent a real world network, it is large enough for an experiment model under the laboratory conditions. More work has to be done in this area concerning the interface with the outside devices, both in physical and in logical design.

Not only the designed code need to be improved to expand the system, the applied solution itself may also need to be improved to make the algorithm even more efficient for the management task. The methods used for adaptation and evolution is still in its infancy. The relative merits of different flavours of adaptation will need to be investigated. Further, it is also important to devise more systematic methods of assessing performance than the visual analysis of QoS.

Again, the further investigation of the mutation algorithm might provide another important subject for the future work. Two directions could be concerned in this area. First, a more extensive investigation to decide exactly how important the traditional mutation algorithms will have to be done. This may take a huge effort as a statistically significant number of experiments will be needed to prove and testify any conclusion one may make. Second, the new improved mutation algorithm proposed in this thesis was only implemented and testified within the range of computer simulation. No hardware implementation of a system using such an algorithm was performed so far. Thus, a future work may involve implementing a system employing the new algorithm on real hardware based on FPGA to demonstrate its performance.

Still, there is another important job that can be done in the future. As described in chapter 4, a system that applied the same management solution but with a quite simple model was designed using VHDL. The development of a VHDL version of the same model of the full system will provide a basis for the comparison between VHDL design and Handel-C design in many aspects such as the design efficiency, flexibility and performance.

The future work will also involve the investigation of the methods to connect the developed system to the real network and demonstrate its improved performance in the real world network. This is important because it is the last and realistic step towards the real application of the research. Unless these concrete issues were successfully resolved, no future can be seen for the real adoption of the explored management solution.

PUBLICATIONS

Yanfeng Peng, D.J. Holding, K.J. Blow "A Possible Secure Solution for Mobile Agents", Proceddings of 2001 International Conference on Intelligent Agents, Web Technologies and Internet Commerce – IAWTIC'2001.

Q. Gao, D.J. Holding, Yanfeng Peng, K.J. Blow, "Energy Efficiency Design Challenge in Sensor Networks", Proceedings of London Communications Symposium, LCS 2002.

Yanfeng Peng, D.J. Holding, "Hardware Implementation of a Future Network Management Solution", to be submitted.

Yanfeng Peng, D.J. Holding, "Protect the Network from Attacking using Active Network Technology", to be submitted.

Yanfeng Peng, D.J. Holding, "An Investigation to a new genetic algorithm", to be submitted.

REFERENCES

[1] Nelu Mihai, George Vanecek, "New Generation of Control Planes in Emerging Data Networks", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999, pp.144-155.

[2] Chih-Lin Hu, Wen-Shyen E. Chen, "A Mobile Agent-Based Active Network Architecture", 7th International Conference on Parallel and Distributed Systems (ICPADS'00), July 04-07, 2000, Iwate, Japan, pp. 445-453.

[3] David L. Tennenhouse, Jonathan M. Smith, W.David Sincoskie, David J. Wetherall, Gary J. Minden, "A survey of Active Network Research", IEEE Communications Magazine, January 1997, pp. 80-85.

[4] Thomas M. Chen, Alden W. Jackson, "Active and Programmable Networks", IEEE Network, May/June, 1998, pp10-11.

[5] D.Wetherall, U.Legedza, J.Guttag, "Introducing new Internet services: Why and how", IEEE Network, May/June 1998, pp.12-19.

[6] K.Calvert, S.Bhattacharjee, E. Zegura, J. Sterbenz, "Direction in active Networks", Communication IEEE, October 1998, Volume 36, Number10, pp.72.

[7] Amila Fernando, Bob Kummerfeld, Alan Fekete, Michael Hichens, "A New Dynamic Architecture for an Active Network", IEEE OPENARCH 2000, pp.121-127.

[8] Ian W.Marshall, Chris Roadknight, "Adaptive management of an Active Service Network", British Telecom. Technol.J., 18,4, October 2000, pp.78-84.

[9] Marcus Brunner, TIK, ETH Zurich Rolf Stadler, "Service management in Multiparty Active Networks", IEEE Communications Magazien, March 2000, pp.144-151.

[10] Marcus Brunner, "Active Networks and its Management", IEEE International Conference on Networking, ICN'01, July 11-13, 2001 - CREF, Colmar, France.

[11] Chris Roadknight, Ian Marshall, "Future Network Management-A bacterium Inspired Solution", Proceedings of Second International Symposium Engineering of Intelligent Systems. June 27 - 30, 2000.

[12] Chris Roadknight and Ian W. Marshall, "Management of Future Data Networks: An approach Based on Bacterial Colony Behaviour", Artificial Life, December 2001.

[13] Rumeel Kazi, Patricia Morreale, "Mobile Agents for Active Network Management", MILCOM '99 Conference Proceedings.

[14] Alex Villazon, Jarle Hulaas, "Meta-level Management of Active Network Nodes and Services", 2000 IEEE/IFIP Network Operations and Management Symposium (NOMS 2000), "The Networked Planet: Management Beyond 2000", 10-14 April 2000, USA.

[15] Handel-C Language Reference Manual, Version 3.1, Celoxica Limited, 2002

[16] David B. Fogel, "What is evolutionary computation", IEEE Spectrum, Volume 37, Number 2, February 2000.

[17] "Evolutionary Computation: Theory and Applications", edited by Xin Yao (University of Birmingham, UK), Publisher: World Scientific Publishing, ISBN 981-02-2306-4, Nov., 1999.

[18] Holland J.H., "Outline for a logical theory of adaptive systems", Journal of the ACM, 9(3): 297-314, July 1962.

[19] Karthikeya M. Gajjala Purna, Dinesh Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Reconfigurable Computers", IEEE Transactions on Computers, Volume 48, Number 6, June 1999.

[20] http://www.xilinx.com/

[21] Moshe Sipper, "The Emergence of Cellular Computing", Computer, July 1999, pp.18-26.

[22] "Universality and Emergent Computation in Cellular Neural Networks", Radu Dogaru (Polytechnic University of Bucharest, Romania), World Scientific Series on Nonlinear Science, Series A – Vol. 43, ISBN 981-238-102-3, pub. Date: Mar 2003.

[23] J. von Neumann, "Theory of Self-Reproducing Automata", University of Illinois Press, Illinois, 1966, Edited and completed by A.W. Burks.

[24] S. Wolfram, "Universality and complexity in cellular automata", Physica *D*, 10:1-35, 1984.

[25] T. Toffoli, N. Margolus, "Cellular Automata Machines", The MIT Press, Cambridge, Massachusetts, 1987.

[26] C.A.R. Hoare Series editor, "OCCAM Programming Manual", INMOS Limited, Prentice-hall International series in computer science, 1984.

[27] M.B. Feldman, "Software Construction and Data Structures with Ada 95. Copyright 1997, Addison-Wesley Publishing Company (published June 1996; ISBN 0-201-88795-9).

[28] David W. Bustard, "Concepts of Concurrent Programming", SEI-CM-24, April 1990, see http://www.infc.ulst.ac.uk/~dave/cm24.pdf

[29] Jeff Magee, Jeff Kramer, "Concurrency, State Models and Java Programs", see: http://www-dse.doc.ic.ac.uk/concurrency/pdf/ch1.pdf

[30] Francis Cook, "Concurrent Languages in the Heterogeneous Specification", 17 March 2000, see: http://mint.cs.man.ac.uk/Projects/UPC/Languages/ConcurrentLanguages.html

[31] Christian Peter, Oxford University, Computing Laboratory, UK, "Overview: Hardware Compilation and the Handel-C Language", see: http://web.comlab.ox.ac.uk/oucl/work/christian.peter/overview_handelc.html

[32] C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Volume 21, Issue 8, August 1978, pp.666-677.

[33] http://www.celoxica.com/products/tools/dk.asp

[34] Celoxica DK1 Design Suite Learning Series, Ver1.00, Celoxica Limited, March 2001.

[35] Amit Patel, "Active Network Technology-A thorough overview of its applications and its future", IEEE Potentials, Febuary/March 2001, pp.5-10.

[36] Gisli Hjalmtysson, Samrat Bhattachajee, "Control on Demand", IEEE J. Selected Areas in Communications, September 1999.

[37] David L. Tennenhouse, David J. Wetherall, "Towards an Active Network Architecture", Computer Communication Review, Volume 26, Number 2, April 1996.

[38] Lidia Yamamoto, Guy Leduc, "Adaptive Applications over Active Networks: Case Study on Layered Multicast", Proceedings of the First IEEE European Conference on Universal Multiservice Networks, (ECUMN 2000), Colmar, France, October 2000, pp. 386-394.

[39] Thomas M. Chen, "Evolution to the Programmable Internet", IEEE Communications Magazine, March 2000, pp.124-128.

[40] R. Cardoe, J. Finney, A.C. Scott, W.D. Shepherd, "LARA: A Prototype System for Supporting High Performance Active Networks", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999.

[41] David Putzolu, Sanjay Bakshi, Satyendra Yadav, Raj Yavatkar, "The Phoenix Framework: A Practical Architecture for Programmable Networks", IEEE Communications Magazine, March 2000, pp.160-165.

[42] Kustarto WIDOYO, Terumasa AOKI, Hiroshi YASUDA, "A New Network Service Environment Using Active Network", First IEEE European Conference on Universal Multiservice Networks ECUMN'2000, October 2-4, 2000 - CREF, Colmar, France.

[43] A. B. Kulkarni, G.J. Minden, R.Hill, Y.Wijata, A. Gopinath, S. Sheth, F. Wahhab, H. Pindi, A. Nagarajan, "Implementation of a Prototype Active Network", OPENARCH'98, April 1998, pp.30-43.

[44] Danny Raz, Yuval Shavitt, "Active Networks for Efficient Distributed Network Management", IEEE Communications Magazine, March 2000, pp.138-143.

[45] Ian Marshall, Mike Fry, Luis Velasco, Atanu Ghosh, "Active Information Networks and XML", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999, pp.60-72.

[46] David Wetherall, "Active network vision and reality: lessons from a capsule-based system", 17th ACM Symposium on Operating Systems Principles (SOSP'99), December 1999, Published as Operating Systems Review 34(5): pp.64-79.

[47] X. Luo, K. Balakrishnan, M. W. McKinnon, "An Empirical Study of Active Networks", ACM Southeast Conference 2000, April 7-8, 2000.

[48] Raghupathy Sivakumar, Narayanan Venkitaraman, Vaduvur Bharghavan, "The Protean Programmable Network Architecture: Design and Initial Experience", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999, pp.37-47.

[49] Markus Breugst, Thomas Magedanz IKV++ GmbH, "Mobile Agents-Enabling Technology for Active Intelligent Network Implementation", IEEE Network, May/June 1998, pp.53-60.

[50] Smith, J., et al., "SwitchWare: Accelerating Network Evolution", Technical Report MS-CIS-96-38, CIS Department, University of Pennsylvania, 1996, Also available as http://www.cis.upenn.edu/~jms/white-paper.ps

[51] Yemini, Y., da Silva, S., "Towards Programming Networks", FIP/IEEE International Workshop on Distributed Systems, October 1996.

[52] J. E. van der Merwe, I. M. Leslie, "Switchlets and Dynamic Virtual ATM Networks", Proceedings of 5th IFIP/IEEE Int'l, Integrated Network Mgmt., San Diego, CA, May 1997.

[53] Hartman, J., et al., "Liqid Software: A New Paradigm for Networked Systems", Technical Report, Department of Computer Science, University of Arizona, November 1996.

[54] Stamatis Karnouskos, Ingo Busse, Stefan Covaci, "Agent Based Security for the Active Network Infrastructure", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999, pp.330-344.

[55] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, Jonathan M. Smith, "A Secure Active Network Environment Architecture: Realization in SwitchWare", IEEE Network magazine, May/June 1998, pp. 37-45.

[56] M. Brunner, R. Stadler, "The Impact of Active Networking Technology on Service Management in a Telecom Environment", Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston, 1999.

[57] Bharat Bhushan, Jane Hall, "Active Network Challenges to TMN", Active Networks, Proceedings of the First International Working Conference (IWAN '99), Berlin, June 1999, pp.285-298.

[58] Marcus Brunner, Bernhard Plattner, Rolf Stadler, "Service Creation and Management in Active Telecom Networks", Communications of the ACM (CACM), Volume 44, Issue 4, April 2001, pp. 55-61.

[59] Marcus Brunner, Bernhard Plattner, "Management of Active Networks", ICC Workshop on Active Networking and Programmable Networks, Atlanta, 1998.

[60] Yasin Kaplankiran, Alexander Keiblinger, Hermann Tobben, "Active Network Mangement via Agent Technology", Telecommunication Network Intelligence, IFIP TC6 WG6.7 Sixth International Conference on Intelligence in Networks (SMARTNET 2000), September 18-22, 2000, Vienna, Austria.

[61] G. Cabri, L. Leonardi, F. Zambonelli, "Mobile Agent Technology: Current trends and perspectives", Congresso annuale AICA'98, Napoli (I), November 1998.

[62] Y.Z. Tsypkin, "Adaptation and learning in automatic systems", Mathematics in Science and Engineering, Volume 73, Academic Press, 1971.

[63] G. DiCaro, M. Dorigo, "AntNet: Distributed stigmergic control for communications networks", J.Artificial Intelligence Research, September 1998, pp.317-365.

[64] D.A.Fisher, H.F. Lipson, "Emergent algorithms – a new method of enhancing survivability in unbounded systems", Proceedings of the 32nd Hawaii international conference on system sciences, IEEE, 1999.

[65] M. Gregory, B. White, E.A. Fisch, U.W. Pooch, "Cooperating security managers: A peer based intrusion detection system", IEEE Network, Volume 10, Number 1, January 1996, pp.20-23.

[66] L. Lewis, "A case based reasoning approach to the management of faults in telecommunications networks", Proceedings of the IEEE conference On Computer Communications, San Francisco, Volume 3, 1993, pp. 1422-1429.

[67] Heitkoetter, Joerg and Beasley, David, eds. (2001), "The Hitch-kiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)", also available at http://www.faqs.org/faqs/ai-faq/genetic/

[68] Jean-Philippe Rennard, PhD, May 2000, "Introduction to Genetic Algorithm", http://www.rennard.org/alife/english/gavintrgb.html

[69] Sam Hsiung, James Matthews, "An Introduction to Genetic Algorithms", http://www.generation5.org/ga.shtml, updated on 31/03/2000.

[70] Michael A. Harra, Brian E. Boling, Bruce L. Walcott, "Stability Analysis of Genetic Algorithm Controllers",Proc. IEEE SoutheastCon, Tampa, FL, April 1996.

[71] Moshe Sipper, "A brief Introduction to Genetic Algorithms", This document was generated using the LaTeX2HTML translator version 96.1 (Feb. 5, 1996) Copyright 1993, 1994, 1995, 1996, Nikow Drakos, Computer Based Learning Unit, University of Leeds, The translation was initiated by Moshe Sipper on Nov. 13 1996, available on http://www.cs.bgu.ac.il/~sipper/ga_main.html

[72] S. sonea, M.Panisset, "A new bacteriology", Jones and Bartlett, 1983.

[73] S.Golubic, "Stromatolites of Shark Bay", Environment evolution, edited by Margulis and Olendzenski, MIT press 1999, pp103-149.

[74] I. Rechenberg, "Cybernetic Solution path of an experimental problem." Farborough Hants: Royal Aircraft Establishment. Library Translation 1122, August 1965. English Translation of lecture given at the Annuls Conference of the WGLR at Berlin in September, 1964.

[75] L.J. Fogel, A.J. Owens, and M.J. Walsh, "Artificial Intelligence through Simulated Evolution", John Wiley & Sons, New York, 1966.

[76] Holland J.H., "Adaptation in natural and artificial system", Ann Arbor, The University of Michigan Press, 1975.

[77] Peter J. Bentley, Timothy Gordon, Jungwon Kim, Sanjeev Kumar, "New Trends in Evolutionary Computation", the Congress on Evolutionary Computation (CEC-2001), Seoul, Korea, May 27-30, 2001, pp.162-169.

[78] Bentley, P. J., Corne, D. W, "Creative Evolutionary Systems", Morgan Kaufman, Publised August 2001.

[79] The 2001 Congress on Evolutionary Computation (CEC 2001), Introduction on http://scai.snu.ac.kr/cec2001/main.htm

[80] Bentley, P. J., "Exploring Component-Based Representations – The Secret of Creativity by Evolution?", ACDM 2000, University of Plymouth, April 26 –28 , 2000.

[81] Ian C. Parmee, Dragan Cvetkovic, Andrew H. Watson, Christopher R. Bonham, "Multi-objective Satisfaction within an Interactive Evolutionary Design Environment", Evolutionary Computation, Volume 8, Number 2, 2000, pp. 197-222.

[82] Emmeche C., "Garden in the Machine, the Emerging Science of Artificial Life", Princeton University Press, 1994, pp. 114.

[83] M. Mitchell, "An Introduction to Genetic Algorithms", MIT Press, Cambridge, MA, 1996.

[84] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, R. Reed Taylor, "PipeRench: A Reconfigurable Architecture and Computer", Computer, April 2000.

[85] John Villasenor, Brad Hutchings, "The flexibility of configurable computing", IEEE Signal Processin Magazinre, September 1998, pp67-83.

[86] J. Villasenor, W.H. Mangione-Smith, "Configurable Computing", Scientific Am., Volume 276, Number 6, June 1997, pp. 54-59.

[87] "Field –Programmable Gate Array Technology", S.M. Trimberger, ed. Boston: Kluwer Academic, 1994.

[88] Joao M. P. Cardoso, Mario P. Vestias, "Architectures and Compilers to Support Reconfigurable Computing", ACM Corssroads Student Magazine, The ACM's First Electronic Publication, available on http://www.acm.org/crossroads/xrds5-3/rcconcept.html

[89] Olukotun, K. A., Helaihel, R., Levitt, J., Ramirez, R., "A Software-Hardware Cosynthesis Approach to Digital System Simulation", IEEE Micro, Volume 14, November 1994, pp. 48-58.

[90] Athanas, P., Silverman, H., "Processor Reconfiguration through Instruction-Set Metamorphosis: Architecture and Compiler", IEEE Computer, Volume 26, Number 3, March 1993, pp. 11-18.

[91] Barr, Michael, "A Reconfigurable Computing Primer", Multimedia Systems Design, Septermber 1998, pp. 44-47.

[92] S. Brown, J. Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design and Test of Computers, Volume 13, Number 2, 1996, pp. 42-57.

[93] S. Hauck, "The Role of FPGAs in Reprogrammable Systems", Proceedings of IEEE, Volume 86, Number 4, April 1998, pp. 615-638.

[94] D.A. Buell, J.M. Arnold, W.J. Kleinfelde, "Splash2, FPGAs in Custom Computing Machine", IEEE CS Press, 1996.

[95] S. Hauck, "Multi-FPGA Systems", PhD thesis, University of Washington, 1997.

[96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transaction. VLSI, Volume 4, Number 1, March 1996.

[97] D. Smith, D. Bhatia, "RACE: Reconfigurable and Adaptive Computing Enviroment", "Field-Programmable Logic: Smart Applications", New Paradigms and Compilers, Berlin, Springer-Verlag, September 1996, pp. 87-95.

[98] D. Bhatia, P. Kannan, K. Simha, K.M. Gajjala Purna, "REACT: Reactive Environment for Reconfigurable Computing," "Field-Programmable Logic: Smart Applications", New Paradigms and Compilers, R.W. Berlin, Springer-Verlag, August/September 1998.

[99] S. Gehring, S.H.-M. Ludwig, "The Trianus System and Its Application to Custom Computing", "Field-Programmable Logic: Smart Applications", New Paradigms and Compilers, Berlin, Springer-Verlag, September 1996, pp. 176-184.

[100] TSI-Telsys Inc., ACE Card, User's Manual, Version 1.0, 1998.

[101] O.T. Albahama, P. Sethi, J.D. Ullman, "Compilers: Principles, Techniques and Tool", Addison-Wesley, 1986.

[102] S.Singh, P. James-Roxby, "Rapid construction of partial configuration datastreams from high-level constructs using Jbits", FPL 2001 Conference, 27th-29th of August 2001, Belfast, UK.

[103] "Evolvable Systems: From Biology to Hardware", Lecture Notes in Computer Science 1259, Springer-Verlag, 1997.

[104] J. Babb, R. Tessier, M.D. Silvina, Z. Hanono, D.M. Hoki, A. Agarwal, "Logic Emulation with Virtual Wires", IEEE Transaction, Computer-Aided Design of Integrated circuits and Systems, Volume 16, Number 6, June 1997, pp. 609-626.

[105] J. Varghese, M. Butts, J. Batcheller, "An Efficient Logic Emulation System", IEEE Transaction, VLSI Systems, Volume 1, Number 2, June 1993, pp. 171-174.

[106] D.T. Hoang, "Searching Genetic Databases on Splash 2," Proceedings of the IEEE Workshop FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, eds, April 1993, pp. 185-191.

[107] D.V. Pryor, M.R. Thistle, N. Shirazi, "Text Searching on Splash 2," Proceedings of the IEEE Workshop FPGAs for Custom Computing Machines, D.A. Buell and K.L. Pocek, eds., April 1993, pp. 172-177.

[108] N.K. Ratha, A.K. Jain, D.T. Rower, "Fingerprint Matching on Splash2," Splash2, FPGAs in Custom Computing Machine, 1996, pp.117-140.

[109] P.M. Athanas A.L. Abbott, "High-Speed Image Processing with Splash 2," Splash2, FPGAs in Custom Computing Machine, 1996, pp.141-165.

[110] Panagiotis G. Tzionas, "A cellular neural network modelling the behaviour of reconfigurable cellular automata", Microprocessors and Microsystems 25, 2001, pp379-387.

[111] Tamra Kerns, "The Advantages of Multithreaded Applications", Published by EE-Evaluation Engineering.

[112] S. M. Loo, B. Earl Wells, J. Kulick, "Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems", Southeastern Symposium on System Theory, March 17-18, 2002, Huntsville, Alabama, USA.

[113] Krishnan Subramani, Malolan Chetlur, Timothy J. McBrayer, Radharamanan Radhakrishnan and Philip A. Wilsey, Computer Architecture Design Laboratory, Dept of ECECS, TyVIS, A VHDL Simulation Kernel (Documentation for version 1.0).Published by the University of Cincinnati, also available on:
http://www.ececs.uc.edu/~paw/tyvis/doc/node28.html

[114] Sudhakar Yalamanchili, "Introductory VHDL: From Simulation To Synthesis", Prentice-Hall, Inc. Upper Saddle River, New Jersey 07458, 2001.

[115] VHDL Language Guide, Language Overview, Copyright © 2000-2001, Altium Limited, available on:
http://www.acc-eda.com/vhdlref/refguide/language_overview/language_overview.htm

[116] http://www.doulos.co.uk/knowhow/vhdl_designers_guide/what_is_vhdl/

[117] C.A.R. Hoare, "Communicating Sequential Processes", International Series in Computer Science, Prentice-Hall, 1985.

[118] "The occam2 Programming Manual", Inmos, Prentice-Hall, 1988.

[119] B.W. Kernigham, D.M. Ritchie, "The C Programming Language", Prentice-Hall, 1988.

[120] Altaf Abdul Gaffar, "A Survey on the Handel-C Language", Surprise Project 1999, available on:
http://www.iis.ee.ic.ac.uk/~frank/surp99/article1/amag97/

[121] Kjell Torklesson, Vice President, Engineering Celoxica Ltd., "Handel-C backs top-down hardware development", EE Times, Feb. 21, 2002, available on:
http://www.eetimes.com/in_focus/embedded_systems/OEG20020208S0053

[122] "The technology Behind DK1", Application Note AN18, version 1.0, Celoxica Limited, August 2002.

[123] http://www.msc.rl.ac.uk/europractice/vendors/handel.pdf

[124] Tsutomu Maruyama, Terunobu Funatsu, Minenobu Seki, Yoshiki Yamaguchi, Tsutomu Hoshino, "A Field-Programmable Gate-Array system for Evolutionary Computation". IPSJ Journal, Volume 40, Number 5, 1999.

[125] Luis Miguel Brugolaras, SIRE, Madrid, Spain, "FPGA makes simple FIFO", EDN, 10/26/2000, available on:
http://www.e-insite.net/ednmag/index.asp?layout=article&articleid=CA47273

[126] Jamil Khatib, "FIFO, First-In First-Out Memory", this document was generated using the **LaTeX**2HTML translator Version 98.1 release (February 19th, 1998), Copyright © 1993, 1994, 1995, 1996, 1997, Nikos Drakos, Computer Based Learning Unit, University of Leeds. The translation was initiated by Jamil Khatib on 1999-03-28, available on:
http://www.geocities.com/SiliconValley/Pines/6639/ip/fifo.html

[127] "Celoxica DK1 Design Suite On-line help", DK1 Design Suite Version 1.1, Celoxica Limited, 2002.

[128] Peter Martin, "An analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C", GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 837-844.

[129] Mark M. Meysenburg, James A. Foster, "Randomness and GA performance, revisited", Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, Robert E. Smith, editors, Proceedings of the genetic and evolutionary computation conference, Orlando, Florida, USA, July 1999, Morgan Kaufmann, Volume 1, pp. 425-432.

[130] Mark M. Meysenburg, James A. Foster, "Random generator quality and GP performance", Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, Robert E. Smith, editors, Proceedings of the genetic and evolutionary computation conference, Orlando, Florida, USA, July 1999. Morgan Kaufmann, Volume 2, pp. 1121-1126.

[131] Peter Martin, "A pipelined Hardware implementation of Genetic Programming using FPGAs and Handel-C", EuroGP2002, 5th European Conference on Genetic Programming, April 3-5, 2002.

[132] "RC1000 Software Reference Manual", Version 1.3, Celoxica Limited, 2001.

APPENDIX A ABBREVIATIONS AND ACRONYMS

AC	Adaptive Control
AN	Active Network
CA	Cellular Automata
CC	Cellular Computing
CPLD	Complex Programmable Logic Device
CSP	Communicating Sequential Processes
DMA	Direct Memory Access
DUT	Device Under Test
EC	Evolutionary Computation
EE	Execution Environment
EVH	Evolvable Hardware
FPGA	Field Programmable Gate Array
GA	Genetic Algorithm
GP	Genetic Programming
GUI	Graphical User Interface
IDE	Integrated Development Environment
LFSR	Logic Feedback Shift Registers
PLD	Programmable Logic Device
QoS	Quality of Service
RC	Re-configurable Computing
RNG	Random Number Generators
TTL	Time to Live
VAN	Virtual Active Network
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VPN	Virtual Private Network

APPENDIX B TECHNICAL FEATURES OF RC1000

- Full-length 32-bit PCI card
- Supports Xilinx Virtex Series or XC4000XV FPGAs
- Programmable clock 400KHz to 100MHz
- PCI interface:
  - 32-bit, 33MHz
  - 132Mbytes/sec burst
  - Master/slave
- 4 banks of fast asynchronous SRAM with 2Mbytes per bank

APPENDIX C INTRODUCTION OF FPGA

Programmable devices are a class of general-purpose chips that can be configured for a wide variety of applications. Unlike the traditional fully customised VLSI circuits, FPGA represents a technical breakthrough in the sense that they can implement thousand of gates of logic in a single IC and can be programmed (or configured) by users at their site in a few seconds. FPGA based circuits can often give high performance because, unlike a normal general-purpose processor such as a microprocessor which has a fixed sequential architecture, a FPGA can be configured to have a parallel architecture. This means an algorithm targeted at a FPGA can exploit parallelism. Over the past ten years FPGAs have revolutionized the way many systems are designed by providing a low-cost, low risk, low development time, fast-turnaround implementation design approach. These advantages have made FPGA very popular for prototype development, custom computing, digital signal processing, and logic emulation. This field is still expanding as new technologies appear, new architectures are proposed, and new CAD tools are developed to address problems specific to FPGAs.

A typical FPGA has three major configurable elements: configurable logic blocks (CLBs), input/output blocks (IOBs) and interconnects. Logic blocks generally contain look up tables and flip-flops and provide the functional elements for constructing user's logic. Input/Output cells provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. Customized configuration is established by programming internal cells that determine the logic functions and internal connections implemented in the FPGA. The following figure C1 shows the basic FPGA structure.

The FPGA device that is used in "Bacterium Model" system implementation is Xilinx Virtex-E Series FPGA. Vertex-E devices feature a flexible, regular architecture that comprises an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs), all interconnected by a rich hierarchy of fast, versatile routing resources. The abundance of routing resources permits the Virtex-E family to accommodate even the largest and most complex designs.

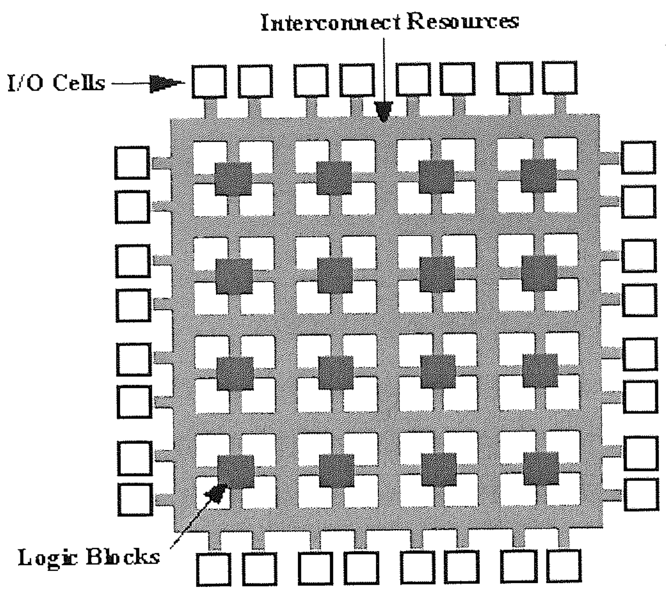


Figure C1 Structure of FPGA

APPENDIX D INTRODUCTION OF THE COMMUNICATION METHODS BETWEEN HOST AND THE FPGA

First, the RC1000 board provides a single byte wide port to support single byte data transfers in either direction between the host and FPGA. This port can be used to send short messages such as control or status bytes between the two parties. The RC1000 host support software provides two functions to implement the communication. The **PP1000WriteControl ()** function will send a byte from the host to the FPGA and the **PP1000ReadStatus ()** function will wait for a byte to be sent by the FPGA. Both functions are blocking and will only return when the operation has completed. While on the other end, the RC1000 Handel-C support software provides two macros of **PP1000WriteStatus ()** that is used to send a byte from the FPGA to the host and **PP1000ReadControl ()** that will wait for a byte to be sent by the host. Both macros are blocking and will only return when the operation has completed. This method of communication is used to synchronize the swapping of ownership of a memory bank.

The RC1000 board also supports bulk data transfers using the DMA. The RC1000 has 4 banks of SRAM fitted. These can be used in communicating between the host and the board FPGA over the PCI bridge (see figure D1). Data can be sent directly between the off-chip RAM on the board and the hosts' memory without involving the host CPU using Direct Memory Access or DMA. DMA is often quicker than allowing the host processors to control the communication because the communication does not have to wait for sufficient CPU power to control the contact. Each bank of the RAM can be granted to either the host or the FPGA (but not both) at any one time. When a memory bank is granted to the host, the DMA controller can transfer data between the host memory and the SRAM memory bank. When a memory bank is granted to the FPGA, it can access the data in the memory to read source data from the host or fill in return data to the host. One of the first two methods (single bit transfer or single byte transfer) can be used to synchronize the swapping of ownership of a memory bank.

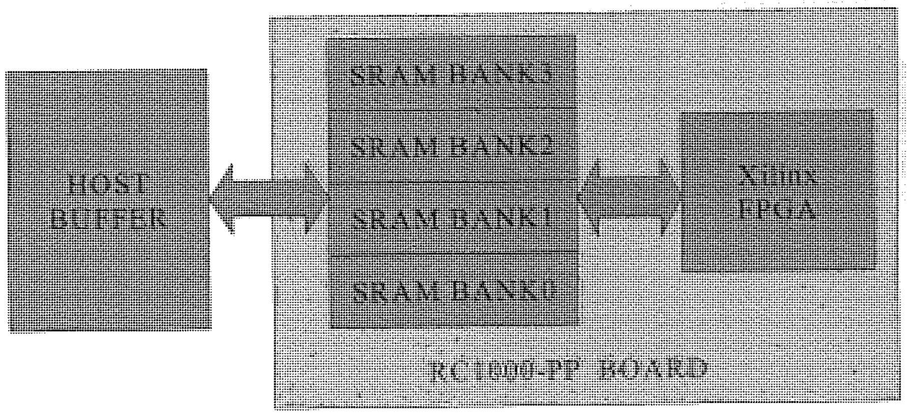


Figure D1 Communication between host and FPGA through off-chip RAM

A typical DMA transfer consists of a number of stages. On the host side, the stage is:

1. Request access to the required memory bank(s)
2. Do the DMA transfer
3. Release the memory bank(s)
4. Pass access control of memory bank(s) to FPGA

The RC1000 host support software provides the following functions to complete the communication: **PP1000RequestMemoryBank ()** function for requesting the memory bank for data transfer, the function will wait until the memory banks have been granted to the host before returning; the **PP1000DoDMA ()** function is called to start the DMA transfer, the function will only return when the DMA transfer is complete; the **PP1000ReleaseMemoryBank ()** function is used to release the ownership of the memory banks. On the other end (the FPGA side), the PP1000 Handel-C support software contains macros to allow the FPGA access to the SRAM banks. The **PP1000RequestMemoryBank ()** macro is used to gain access to the memory bank. The **PP1000ReleaseMemoryBank ()** is used to release the bank. The memory can be accessed using the **PP1000ReadBank#()** and **PP1000WriteBank#()** macro procedures. Figure D2 and figure D3 shows the DMA communication procedure between host and the FPGA, the former for the communication from Host to FPGA and the latter from FPGA to Host.

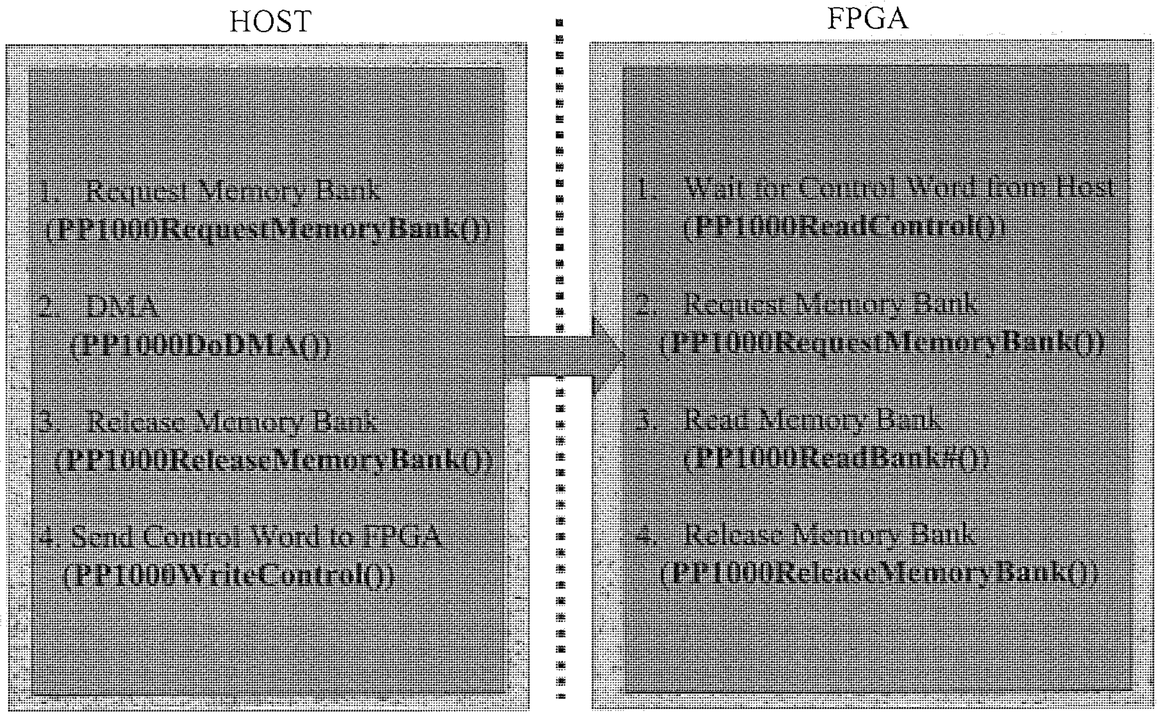

HOST
1. Request Memory Bank (PP1000RequestMemoryBank())
2. DMA (PP1000DoDMA())
3. Release Memory Bank (PP1000ReleaseMemoryBank())
4. Send Control Word to FPGA (PP1000WriteControl())
FPGA
1. Wait for Control Word from Host (PP1000ReadControl())
2. Request Memory Bank (PP1000RequestMemoryBank())
3. Read Memory Bank (PP1000ReadBank#())
4. Release Memory Bank (PP1000ReleaseMemoryBank())

Figure D2 The DMA communication from host to FPGA

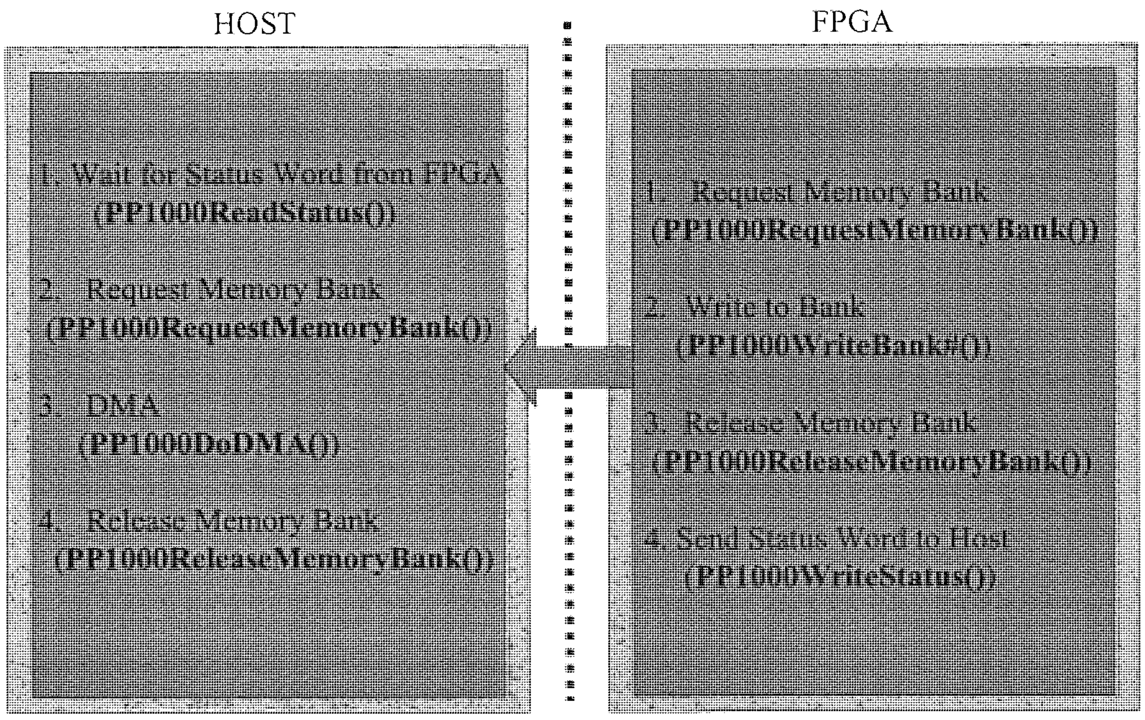

HOST
1. Wait for Status Word from FPGA (PP1000ReadStatus())
2. Request Memory Bank (PP1000RequestMemoryBank())
3. DMA (PP1000DoDMA())
4. Release Memory Bank (PP1000ReleaseMemoryBank())
FPGA
1. Request Memory Bank (PP1000RequestMemoryBank())
2. Write to Bank (PP1000WriteBank#())
3. Release Memory Bank (PP1000ReleaseMemoryBank())
4. Send Status Word to Host (PP1000WriteStatus())

Figure D3 The DMA communication from FPGA to host

APPENDIX E HARDWARE IMPLEMENTATION PROCEDURE

Before targeting Handel-C for hardware, the hardware interfaces was added to the Handel-C code described in section 4.5.1.1. The additional components in the hardware interface declaration include the header file pp1000.h. This contains standard macros and pin definitions for the FPGA mounted on the RC1000. The use of the RC1000 header file requires some additional definitions in the code provided by the #define statements. These include the definition of the set part and set family of the specific FPGA for identifying the hardware being used, the definition of the source clock, a clock divide factor, and the RAM access macros.

The DK1 library provides the configuration of the source clock using the following pre-compiler definitions (table E1):

#define PP1000_CLOCK PP1000_MCLK	The MCLK clock input is set
#define PP1000_CLOCK PP1000_VCLK	The VCLK clock input is set

Table E1 The configuration of the source clock

In the design of "Bacterium Model", the source clock was set to "#define PP1000_CLOCK PP1000_VCLK". The possible settings for the clock division factors are (table E2):

#define PP1000_DIVIDE1	The input clock is not divided
#define PP1000_DIVIDE3	The input clock is divided by 3
#define PP1000_DIVIDE4	The input clock is divided by 4

Table E2 The clock divider setting

In the design of "Bacterium Model", the clock divider setting was set to "#define PP1000_DIVIDE3".

The RAM access macros can be set to access the external SARM either as 8 bit wide or 32 bit wide memory. One of the following lines must appear at the start of the DK1 program (table E3):

#define PP1000_32BIT_RAMS	32 bit access to the memory banks
#define PP1000_8BIT_RAMS	8 bit access to the memory banks

Table E3 The RAM access macros set

In the design of "Bacterium Model", the RAM access macros was set to "#define PP1000_32BIT_RAMS". An example of these #define statements in the code is shown below:

```
#define PP1000_32BIT_RAMS // use 32 bit external ram access
#define PP1000_DIVIDE3 // Handel-C clock is one third the frequency of RC1000-PP clock
#define P1000_CLOCK  PP1000_VCLK // PP1000 clock is PP1000_VCLK
#include <pp1000.h> // Include RC1000-PP support header file
```

As a result of place and route, a bit-file was created. This file was used to configure the RC1000 FPGA.

The FPGA on the board can be configured from 3 possible sources :

- The host over the PCI bus using software utilities or libraries provided.
- Any host PC using a Xilinx Xchecker download cable.
- The on-board serial ROM.

The implementation of "Bacterium Model" used the first method for the configuration. The host program contains the support software that was used to implement the configuration. The FPGA was configured directly from a file through calling the PP1000ConfigureFromFile () Function in the host program.

APPENDIX F   ATTACHED CD-ROM

1. VHDL design of one node "Bacterium Model" (source code)
2. Handel-C design of four nodes "Bacterium Model" (source code)
3. Host program source code
4. Input data file example
5. Output data file example