An Object-Oriented Programming Environment for Parallel Genetic Algorithms

José Luiz Ribeiro Filho

a thesis submitted for the degree of

Doctor of Philosophy in Computer Science

University of London

Department of Computer Science
University College London

1995

Abstract

This thesis investigates an object-oriented programming environment for building parallel applications based on genetic algorithms (GAs). It describes the design of the **G**enetic **A**lgorithms **M**anipulation **E**nvironment (GAME), which focuses on three major software development requirements: flexibility, expandability and portability. Flexibility is provided by GAME through a set of libraries containing pre-defined and parameterised components such as genetic operators and algorithms. Expandability is offered by GAME's object-oriented design. It allows applications, algorithms and genetic operators to be easily modified and adapted to satisfy diverse problem's requirements. Lastly, portability is achieved through the use of the standard *C++* language, and by isolating machine and operating system dependencies into low-level modules, which are hidden from the application developer by GAME's application programming interfaces.

The development of GAME is central to the **P**rogramming Environment for **A**pplications of **PA**rallel **GEN**etic **A**lgorithms project (PAPAGENA). This is the principal European Community (ESPRIT III) funded parallel genetic algorithms project. It has two main goals: to provide a general-purpose tool kit, supporting the development and analysis of large-scale parallel genetic algorithms (PGAs) applications, and to demonstrate the potential of applying evolutionary computing in diverse problem domains.

The research reported in this thesis is divided in two parts: i) the analysis of GA models and the study of existing GA programming environments from an application developer perspective; ii) the description of a general-purpose programming environment designed to help with the development of GA and PGA-based computer programs. The studies carried out in the first part provide the necessary understanding of GAs' structure and operation to outline the requirements for the development of complex computer programs. The second part presents GAME as the result of combining development requirements, relevant features of existing environments and innovative ideas, into a powerful programming environment. The system is described in terms of its abstract data structures and sub-systems that allow the representation of problems independently of any particular GA model. GAME's programming model is also presented as general-purpose object-oriented framework for programming coarse-grained parallel applications.

GAME has a modular architecture comprising five modules: the Virtual Machine, the Parallel Execution Module, the Genetic Libraries, the Monitoring Control Module, and the Graphic User Interface. GAME's genetic-oriented abstract data structures, and the Virtual Machine, isolates genetic operators and algorithms from low-level operations such as memory management, exception handling, etc. The Parallel Execution Module supports GAME's object-oriented parallel programming model. It defines an application programming interface and a runtime library that allow the same parallel application, created within the environment, to run on different hardware and operating system platforms. The Genetic Libraries outline a hierarchy of components implemented as parameterised versions of standard and custom genetic operators, algorithms and applications. The Monitoring Control Module supports dynamic control and monitoring of simulations, whereas the Graphic User Interface defines a basic framework and graphic 'widgets' for displaying and entering data.

This thesis describes the design philosophy and rationale behind these modules, covering in more detail the Virtual Machine, the Parallel Execution Module and the Genetic Libraries. The assessment discusses the system's ability to satisfy the main requirements of GA and PGA software development, as well as the features that distinguish GAME from other programming environments.

To my loved girls

Teresa and Ana

Acknowledgements

A Ph.D. thesis is, in my opinion, the result of a combination of efforts focused on one person, the student. Without the supervisor's advice, institutional sponsorship, friends' encouragement and, most of all, family support, it would be extremely hard, perhaps impossible, for a post-graduate student to successfully conclude such an enterprise. Therefore, it is very difficult for me to express, in these few paragraphs, my gratitude to everyone whose behind-the-scenes work contributed to the completion of this thesis.

In first place, I would like to express my greatest indebtedness to Professor Philip Treleaven, my supervisor. His continuous support, encouragement and concern were extremely important to the conclusion of this thesis. I also thank Professor Treleaven for the unique opportunity he granted me to take part on a European research project. The PAPAGENA project exposed my research to a wide audience and gave me the chance to learn from the feedback it provided. The understanding and experience I have gained in the project go far beyond the frontiers of the pure scientific knowledge I acquired in this thesis.

In the institutional area, I am very grateful to the Brazilian government for the sponsorship of this thesis, through the Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES). I am also obliged to my employer, Universidade Federal do Rio de Janeiro, which supported me in this enterprise, and the Department of Computer Science at University College London, which provided the means for me to pursue this thesis' research.

I am also thankful to Dr. Peter Rounce and my friend Antonio Braga for their invaluable comments and suggestions, while reading the first draft of this thesis.

Very special thanks must go to colleagues and friends who worked in the PAPAGENA project: Cesare Alippi, who first introduced me to the Genetic Algorithms, Jason Kingdon and Laura Dekker. Laura, in special, for her immense patience and tireless disposition to hear and discuss design and implementation details throughout this research. Her constructive criticisms on this thesis' draft were also very much appreciated. I am much thankful, as well, to all PAPAGENA partners and the reviewers of the ESPRIT programme for their invaluable input and recommendations, which led to several improvements in GAME's design.

I must also express my appreciation for the friendship of Celina Motta, Felipe, Priscila, Paulo Rocha, Marco Pacheco and Meyer Nigri, in particular, who gave me exceptional support throughout these years.

Finally, I cannot forget to mention my gratitude for the support I have received from my parents and brothers. In particular Tadeu, who has been hearing my complaints and sorting out our problems in Brazil since we left. However, I feel extremely difficult to put down in words the immense gratefulness I have to my wife Maria Teresa and my daughter Ana Carolina. The love, endless care, concern, support, and encouragement that Teresa gave me during all these years were the true driving forces of this thesis. Ana, with her innocent love, gave me the inspiration and motivation to carry on during the difficult moments of this thesis. Her little smiles were an infinite source of reassuring that renewed my determination to complete the work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The work presented in this thesis represents an attempt to provide software developers with a powerful programming environment for building applications based on genetic algorithms (GAs). This research is based on an analysis of the characteristics of sequential and parallel genetic algorithm (PGA) models, and on a detailed study of several classes of software systems for programming GA applications. The analysis of different GA models revealed a number of important characteristics and requirements. These, added to a selection of features exhibited by existing programming environments, provided the starting point for the definition of a general-purpose programming environment. Three key requirements to support GA and PGA application development emerged from these studies:

- *flexibility* is imperative for the construction of a wide range of GA and PGA-based applications. Software modules such as genetic algorithms and operators, already built for an application (or algorithm) should be made available to be re-used in the construction of new algorithms and applications;
- *expandability* is important to allow any software module to be effortlessly enhanced or adapted to accommodate characteristics and requirements of new GAs or problems;
- *portability* is essential to provide seamless development and execution of applications on a variety of sequential, parallel and distributed computer architectures.

The Genetic Algorithms Manipulation Environment (GAME), presented in this thesis, is the result of the combination of GAs' programming requirements with the principal features of existing programming environments. Moreover, to deliver the required degree of flexibility, expandability and portability, GAME's design is based on modern object-oriented concepts that permits applications built with the system to be executed in a heterogeneous distributed computing environment.

From a strictly practical perspective, GAME has been helping with the dissemination of the genetic algorithm technique through the facilities it offers for rapid configuration of applications and implementation of complex GAs and PGAs. The system is central to the Programming Environment for Applications of PArallel GENetic Algorithms project (PAPAGENA). This is the principal European Community (ESPRIT III) funded parallel genetic algorithms project. PAPAGENA has two main goals: to provide a general-purpose tool kit, supporting the development and analysis of large-scale parallel genetic algorithms (PGAs) applications, and to demonstrate the potential of applying evolutionary computing in diverse problem domains. The system is being developed by University College London, which has the following PAPAGENA partners: CAP Volmac (Holland) and KiQ (England), developing a decision support system for financial organisations; Brainware GmbH (Germany) working on the prediction of stable protein structures and, in conjunction with IfP (Germany), building an economic modelling application to support governmental planning and decisions. Other partners are GMD (Germany), investigating parallel GAs theory and TELMAT Informatique (France), responsible for porting GAME to parallel machines.

1.1. Motivations and Research Goals

Evolutionary computing in general, and genetic algorithms in particular, provide a powerful mechanism to solve problems previously regarded as "intractable". However, due to the stochastic nature of these techniques, the execution of a number of simulations is required before a final solution is obtained. Moreover, the user is often presented with a number of GA-related parameters to configure, before running a simulation. The correct setup for GA parameters, among other factors, has proven to be essential in obtaining fast and accurate results. Nevertheless, parameter tuning is still an empirical operation. The common method to fine tune a GA is based on the execution of the same algorithm, with small variations on the parameter set, in order to identify their influence in the overall performance. This particular situation suggests the use of several versions of the same GA, with different parameter sets, possibly running in parallel.

Parallelism may also be applied to speedup the execution of GAs by simultaneously evolving multiple sub-populations. An increasing number of parallel models for genetic algorithms has appeared in the literature in recent years. Suitable support for developing and running parallel genetic algorithms is therefore one of the most important requirements for a comprehensive tool kit.

An interesting aspect of the GA model relates to its structure. A genetic algorithm comprises a set of *genetic operators*, which are combined and "activated" in a particular way. A genetic operator can be described as a self-contained program module embodying specific knowledge about the actions it has to perform over genetic data structures. Genetic operators are, therefore, independent from any particular GA model, being able to be used in different models. In fact, what characterises a genetic algorithm is the number, type and activation sequence of its genetic operators. This implies that different GAs can be created by simply replacing one or more genetic operators in a standard GA-template. A subtle consequence of this characteristic is the possibility to parameterise a GA in terms of its genetic operators. This approach is very attractive for experimenting with variations of a basic genetic operator, or quickly adapting a GA for different problems. The same consideration can be extended to applications, permitting them to be parameterised in terms of GAs.

Monitoring a GA simulation is important to identify and control anomalies in the GA execution such as early convergence. It is even more important when a number of parallel GAs are working in conjunction to solve a large problem. Performance bottlenecks and runtime problems can be quickly identified by monitoring communication between co-operating processes as well as hardware and operating system signals. The actual amount of information to monitor depends on the size of the problem and is, in many cases, very large. In such applications, text-based displays may complicate even more the analysis of the algorithm's progress. Therefore, a monitoring system combined with a graphical interface to display intermediate results and control the execution, is an essential tool to supervise the GA dynamics.

The above discussion highlights the importance of an in-depth understanding of the GA structure and operation before any attempt to design any software system for helping with the creation of GA-based applications. It also provides the basis for outlining the requirements for such software systems, which would include:

- the ability to execute multiple instances of the same algorithm in parallel,
- facilities for selective reconfiguration of GAs through the substitution of its modules (e.g. genetic operators, string encoding and decoding functions and problem-dependent objective function),
- the ability to implement classes of genetic algorithms and operators independently of problem representation,
- facilities to dynamically monitor and control the GA execution.

Some desirable features to be added are:

- diverse parameterised algorithms and genetic operators grouped into libraries to accelerate the creation of new applications;
- efficiency in executing algorithms, independent of machine and operating system particularities – the same application should be able to run on either scalar or parallel machines. Distributed systems consisting of heterogeneous computing elements, inter-connected by one or more networks, should also be supported.

A growing number of software systems ranging from specialised environments, targeted at particular problem domains, to general-purpose programming environments [73] have been attempting to address these issues. The leading general-purpose GA programming environments are GENESIS from Grefenstette [40], and OOGA from Davis [24], which were created to demonstrate the use of GAs. Other programming systems have already reached a mature stage and are becoming commercial products. Evolver from Axcelis and *MicroGA* from Emergent Behavior are successful examples. Their common aim is to provide an efficient tool to help in applying GAs to complex real-world problems. However, most of the currently available programming environments provide rather simple facilities and little support for programming complex applications. Problem representation is often dependent on specific GA models, as well as the genetic operators used to manipulate their genetic data. In general, it is very difficult to adapt these systems to use algorithms and operators designed for different problem domains. Few systems can be regarded as truly general-purpose tool kits. An example is Splicer (from NASA) [65], which provides some facilities to describe data structures, create new operators and algorithms, and monitor the execution of applications. However, Splicer does not address parallelism. Other general-purpose systems such as PeGAsuS [38] and GAUCSD [85] provide facilities for parallel execution, but applications created with them are not portable across different hardware or operating system platforms.

The lack of a programming environment that could satisfy all the requirements previously listed, and yet serve as a framework for developing real-world applications, experimenting with new GAs, and comparing already existing ones, motivated this research. The main objectives pursued being:

- to provide a flexible programming environment, allowing the user to rapidly configure and run a broad range of GA and PGA-based applications, as well as compare and experiment with new genetic algorithms and operators;
- to define an expandable set of libraries containing parameterised genetic algorithms and operators that could be combined into applications and algorithms, respectively. Furthermore, application developers should be able to easily build new modules or modify existing ones, and incorporate their new implementations into these libraries;
- to permit the development of portable applications by offering hardware and operating system independence. The same application should be able to execute in diverse sequential, parallel or distributed computing environments.

1.2. Research Contributions

The contributions of the research reported in this thesis can be listed along several lines. Firstly, it organises the various types of GA programming environments under a common taxonomy. This effort has been acknowledged by the GA community and is part of a comprehensive survey of GA programming environments published by the IEEE COMPUTER magazine [73]. The major contribution though, is the design and implementation of GAME itself, which can be summarised in the following points:

- the design and implementation of a general-purpose tool kit supporting the essential requirements of GAs and PGAs. GAME combines the most important features of existing programming environments, with enhancements and new concepts, aimed at helping with the creation of complex sequential and parallel applications.
- the definition of genetic-oriented abstractions that enable the representation of complex problems' data structures, independently of any sequential or parallel genetic algorithm model.
- the development of a manipulation engine that operates over the genetic representation – the *Virtual Machine*. GAME's Virtual Machine isolates the application developer from low-level operations such as memory management, via a high-level *Application Program Interface* (API).
- the definition of an object-oriented parallel programming model, as well as the design of its portable communication and process control support – the *Parallel Execution Module.*
- the introduction of an extra level of parameterisation for genetic algorithms and applications that facilitates their construction through the combination of stand-alone executable components and dynamically linked library (DLL) components.

GAME is also the core of the major European Community funded ESPRIT III project, PAPAGENA. As such, the system has been widely distributed to the industry and academia (directly from UCL or electronically through the Internet) since its early versions.

In addition, this research has produced papers published in the proceedings of the IEEE World Congress on Evolutionary Computing [75], the International Conference on Massively Parallel Applications [74], the IMACS World Congress [95], the IEE Workshop on Applications of Parallel Genetic Algorithms [76] and the World Transputer Congress '94 [94]. It also produced 4 research notes that have been published in two books [29,72,77,48]. Finally, a paper on the exploitation of massively parallel genetic algorithms using GAME is due to appear in the journal entitled Simulation Practice and Theory.

1.3. Thesis Overview

This thesis is divided in two main parts. The first part presents an overview of genetic algorithms, emphasising the most important characteristics presented by a variety of GAs, from an application developer point of view. A brief introduction of the GA theory is followed by a description of Holland's "traditional" GA computing model. Variations of this model are also presented, highlighting their common characteristics. Special attention is given to parallel genetic algorithms, where different levels of parallelism are identified. The first part also includes an extensive survey of existing programming environments for programming GAs and PGAs. A taxonomy is introduced, classifying the systems in three major categories:

- Application-oriented systems,
- Algorithms-oriented systems and
- Tool Kits

The second part presents GAME as a powerful environment for programming GA-based applications. The system attempts to cover a broad spectrum of software development needs, ranging from research to industrial applications. GAME can be used to create software applications to solve many real-world problems commonly found in industry, or to perform research experiments with new models for genetic algorithms and operators. It can also be applied as a tool for evaluating and comparing different GAs and PGAs.

The description of the system starts with an overview of its modular architecture. The importance of its object-oriented design is stressed as an effective means for simplifying the creation of complex and sophisticated applications, where a minimum degree of intervention from the user is desired. Special attention is dedicated to the presentation of the genetic-oriented abstractions defined in GAME. They provide the means for representing complex problems and GA independent manipulation of genetic data structures.

The Virtual Machine's description emphasises the necessity to provide the application developer with a general-purpose, genetic-oriented, manipulation engine. The Virtual Machine operation is controlled via a standard application programming interface that isolates the user from low-level tasks such as memory management and certain types of exception handling. Another important element of GAME's architecture is the Parallel Execution Module. It supports the implementation of GAs and PGAs in terms of parameterised *components* (algorithms and operators). Genetic operators, for instance, can be combined in a variety of ways to configure a number of different GAs. This approach is also the basis for parallelising GAs under GAME, since it permits each component to execute as an independent process. Furthermore, GAME's parallel programming model was designed to be insensitive to any particularity of the GA model, permitting it to be applied as a general-purpose method for programming other types of object-oriented parallel applications. All this flexibility could be achieved thanks to the employment of a distributed object-oriented computing philosophy.

The implementation of the whole system is described in terms of its two main sets of *C++* class libraries: the Genetic and the Service libraries. The Genetic libraries maintain hierarchically organised groups of parameterised applications, genetic algorithms and operators. The Service libraries, on the other hand, bring together the various modules of GAME that implement the runtime support for the GA execution (the Virtual Machine), control of parallelism and communications (the Parallel Execution Module), as well as the graphic user interface and monitoring modules.

The assessment of GAME's design and implementation was conducted bearing in mind the three main objectives of this research: flexibility, expandability and portability. Various tests have also been carried out to evaluate the system's performance.

Finally, it is important to stress that this research demonstrates not only the feasibility of designing and implementing a sophisticated programming environment, but also the ability to galvanise research groups and industry into promising new technologies such as evolutionary computing.

1.4. Thesis Organisation

The remainder of this thesis is organised in eight chapters, ranging from a review of GAs and PGAs to a detailed description of GAME's architecture, design and implementation.

Chapter 2 begins with a brief overview of search techniques to introduce genetic algorithms, in the random search class. It then presents a short review of the GA theory, stressing the important characteristics and requirements from end-user and software developers' point of view. Special attention is dedicated to the discussion of parallel GA models presented in the literature.

Chapter 3 examines existing genetic algorithms programming environments. It introduces a taxonomy that classifies the various types of GA programming environments, based on their common features.

Chapter 4 brings together the requirements derived from Chapter 2 and the most important features described in Chapter 3 into the design of a general-purpose programming environment for the construction of GA and PGA-based applications. GAME's architecture is introduced, with a short description of its genetic-oriented abstractions for problem representation, programming model and main modules.

Chapter 5 describes GAME's genetic-oriented abstractions for problem representation and the Virtual Machine (VM) module. The VM design is presented as a problem and GA independent engine, responsible for the manipulation of GAME's genetic data structures. Each of VM's specialised modules, the *Population Manager* and the *Fitness Evaluator*, is described along with its additional ability to parallelise command execution. Finally the Virtual Machine's Applications Program Interface is presented.

Chapter 6 starts with an analysis of three software strategies used to support inter-process communication and control of parallelism. It proceeds with the description of GAME's object-oriented parallel programming model, the design of the Parallel Execution Module (PEM) and its application programming interface.

Chapter 7 discusses the parameterisation of applications and genetic algorithms in the Genetic libraries and the design and composition of the Service libraries.

Chapter 8 assesses the research carried out in this thesis. The main elements are discussed, investigating their strengths and weaknesses. It includes a judgement of the system's ability to fulfil the main requirements for GA programming. GAME's ability to support the creation of GA and PGA models are discussed along with the facilities it provides for expanding its various libraries. The portability of the applications created with GAME and its own portability are also assessed. Finally, various tests are conducted to assess the performance of programs created with GAME, in comparison with other environments (GENESIS) and stand-alone versions.

The final chapter summarises the main results achieved during this thesis investigation. It presents some of the conclusions drawn from the work, and discusses possible future research.

Chapter 2

Genetic Algorithms Revisited

This chapter reviews the genetic algorithms search and optimisation technique. It focuses on the computational models of sequential and parallel GAs to outline relevant characteristics and requirements for the design of a general-purpose programming environment.

2.1. Introduction

A new class of search and optimisation techniques has been increasingly attracting the attention of researchers from many diverse domains. These Evolutionary Techniques are based on simple concepts that are easy to implement – essentially natural evolution mechanisms. Nevertheless, these techniques are very robust and efficient, having outperformed traditional search techniques in many complex problems. Evolutionary Techniques are particularly strong on multi-modal and noisy problems, which are not well suited to traditional numeric techniques. They comprise Genetic Algorithms (GA), Evolutionsstrategie (ES) and Genetic Programming (GP), with the first one being the most well known.

This chapter briefly introduces these techniques, discussing genetic algorithms in more detail. The discussion presented in this thesis approaches the computing models of these techniques from the application developer point of view. It is therefore, not concerned with theoretical aspects or suitability of these techniques to any particular domain or problem. The theory behind these techniques has been evolving since the creation of the Genetic Algorithms and Evolutionsstrategie, in the seventies. Several studies reported in the major conferences of the field [106,107,108,109,110] discuss classes of problems these techniques may be applied to, as well as ways to find suitable representations for different problems.

2.2. Evolutionary Computing

Biologists have been intrigued with the mechanics of evolution, since the evolutionary theory of biological change gained acceptance [24]. Many people are astonished that life at the level of complexity we observe could have evolved in a relatively short time, as suggested by the fossil record.

Natural evolution has also intrigued many scientists not directly involved in biological research. Most of them are interested in understanding the mechanisms used by nature to solve highly complex problems – such as those involving life and environment – and apply them in a variety of human knowledge domains. These ideas inspired people like Holland [47], Rechenberg [71], Schwefel [82] and Koza [51,52,53] among others, in creating search and optimisation techniques that can be grouped under the term Evolutionary Techniques. The materialisation of these techniques into computer programs is then known as Evolutionary Computing.

Each one of these techniques presents their own particularities, but they all have in common the following characteristics:

- The search for the best solution to a given problem is carried out by testing a number of potential solutions (the population) through a mechanism (evaluation function) which returns a measure of their relative performance.
- The population evolves towards the best solution by means of modifications on selected population members. The way modifications are employed mimics nature's asexual and sexual reproduction mechanisms.
- They are all stochastic techniques and their best results are only valid within a user-defined interval of confidence.

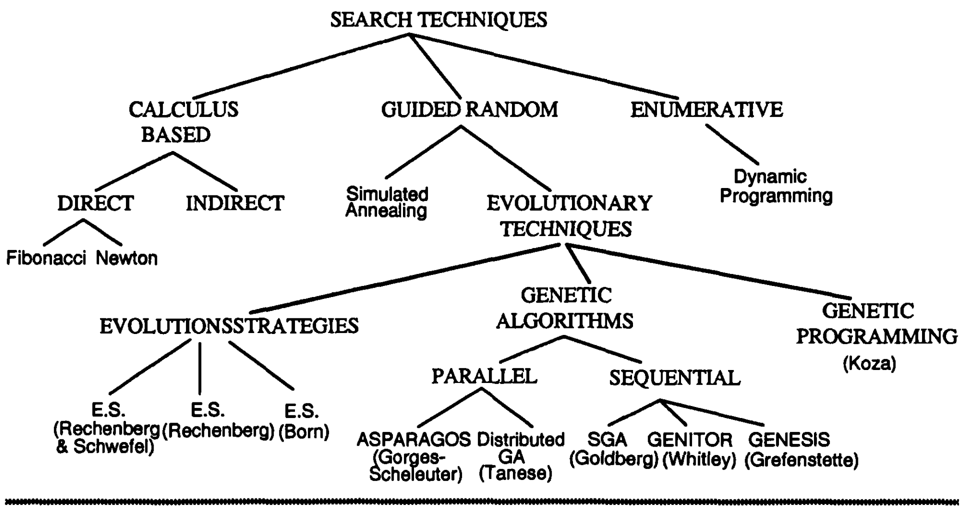
Genetic algorithms are the most prominent of these techniques and have been employed to solve a variety of complex problems [15,21,43,91]. The following sections present an overview of the genetic algorithms technique. Starting with a brief description of classes of search techniques, this chapter proceeds with the description of the traditional GA model, highlighting important issues such as problem representation, algorithm structure and its parallelisation.

2.2.1. Classes of Search Techniques

Search techniques, in general, can be grouped into three broad classes [36] (as illustrated in Figure 2.1): Calculus-based, Enumerative and Guided Random search.

Calculus-based techniques use a set of necessary and sufficient conditions to be satisfied by the optimal solutions of an optimisation problem. These techniques sub-divide into indirect and direct methods. Indirect methods look for local extrema by solving the usually non-linear set of equations resulting from setting the gradient of the objective function equal to zero. The search for possible solutions (function peaks) starts by restricting the search to points with zero slope in all directions. Examples of Direct methods include Newton and Fibonacci. The latter, for instance, seeks extrema by "hopping" around the search space and assessing the gradient of the new point, which guides the direction of the search. This is simply the notion of Hill-Climbing, which finds the best local point by "climbing" the steepest permissible gradient. However, these techniques can only be employed on a restricted set of "well behaved" problems.

Figure 2.1 - Classes of search techniques



Enumerative techniques search every point related to an objective function's domain space (finite or discrete), one point at a time. They are very simple to implement but may require significant computation. The domain space of many applications is too large to search using these techniques. Dynamic programming is a good example of an enumerative technique.

Guided Random search techniques are based on enumerative techniques, but use additional information to guide the search. They are quite general on their scope, being able to solve very complex problems. Two major sub-classes are: Simulated Annealing and Evolutionary Techniques, although both are evolutionary processes. Simulated annealing uses a thermodynamic evolution process to search minimum energy states. Evolutionary techniques, on the other hand, are based on natural selection principles. This form of search evolves throughout generations, improving the features of potential solutions by means of biologically inspired operations. These techniques sub-divide, in turn, into Evolutionsstrategie, Genetic Algorithms and Genetic Programming. Evolutionsstrategie was proposed by Rechenberg [71] and Schwefel [82] in the seventies. They present the ability to adapt the process of "artificial evolution" to the requirements of the local response surface. This means that ESs are able to adapt their major strategy parameters according to the local topology of the objective function [46] – this is very different from traditional GAs. Genetic programming, invented by John Koza, is based on the genetic algorithms model but evolves more complex structures: computer programs. It is the youngest member of the evolutionary techniques family (Koza's landmark book [53] was published in 1992) and has already attracted a vast number of interested researchers, due to the ability of the technique to represent and manipulate complex abstract data structures.

2.3. Genetic Algorithms

A genetic algorithm emulates biological evolutionary theories to solve optimisation problems. A genetic algorithm comprises a set of data elements – the *population* – and a set of biologically inspired operators defined over the population itself. According to evolutionary theories, only the most suited elements in a population are likely to survive and produce offspring, thus transmitting their biological heredity to new generations.

In computing terms, a genetic algorithm maps a problem onto a set of (binary) strings, each string representing a potential solution. The GA then manipulates the most promising strings searching for improved solutions. A GA operates typically through a simple cycle of four stages:

i) creation of a "population" of strings,
ii) evaluation of each string,
iii) selection of "best" strings, and
iv) genetic manipulation, to create the new population of strings.

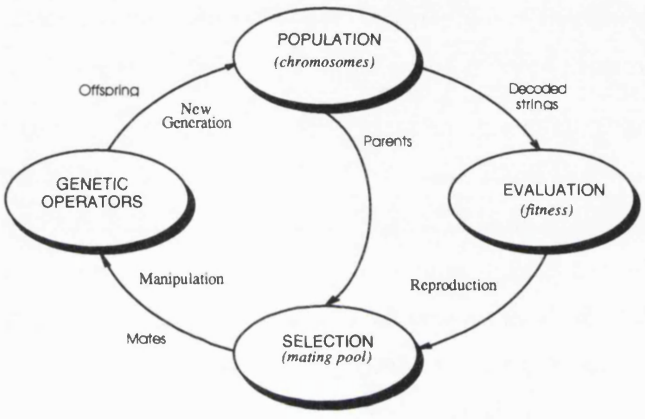
Figure 2.2 shows these four stages using the biologically inspired GA terminology. In each cycle a *new generation* of possible solutions for a given problem is produced. At the first stage, an *initial population* of potential solutions is created as a starting point for the search process. Each element of the population is *encoded*[1] into a string (the *chromosome*), to be manipulated by the *genetic operators*. In the next stage, the performance (or *fitness*) of each individual of the population is *evaluated*, with respect to the constraints imposed by the problem. Based on the fitness of each population member, a *selection* mechanism chooses "mates" for the *genetic manipulation* process. The selection policy is ultimately responsible for assuring survival of the best fitted individuals. The combined evaluation/selection process is called *reproduction*.

The manipulation process employs genetic operators to produce a new population of individuals (*offspring*) by manipulating the "genetic information", referred to as *genes*, possessed by members (*parents*) of the current population. It comprises two operations, namely crossover and mutation. *Crossover* is responsible for recombining the genetic material of a population. The selection process associated with recombination assures that genetic structures, called "building blocks", are retained for future generations. The building blocks then represent the most fitted genetic structures in a population. Nevertheless, the recombination process alone cannot avoid the loss of promising building blocks in the presence of other genetic structures, which could lead to

[1] The encoded form of a problem's candidate solution is called *genotype* and its original (decoded) form called *phenotype*.

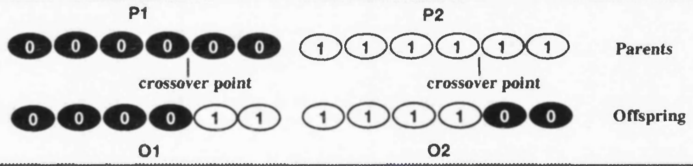
local minima. Also, it is not capable to explore search space sections not represented in the population's genetic structures. The *mutation* operator then comes into action. It introduces new genetic structures into the population by randomly modifying some of its building blocks. It helps the search algorithm to escape from local minima's traps. Since the modification introduced by the mutation operator is not related to any previous genetic structure of the population, it allows the creation of different structures representing other sections of the search space.

Figure 2.2 - The GA cycle



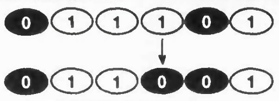
The crossover operator takes two chromosomes and swaps part of their genetic information to produce new chromosomes. This operation is analogous to sexual reproduction in nature. After the *crossover point* has been randomly chosen, the portions of the *parent* strings P1 and P2 are swapped to produce the new *offspring* strings O1 and O2. Figure 2.3 shows the crossover operator being applied to the fifth and sixth elements of the string.

Figure 2.3 - Crossover



Mutation is implemented by occasionally altering a random bit in a string. Figure 2.4 presents the mutation operator being applied to the fourth element of the string.

Figure 2.4 - Mutation



A number of different genetic operators have been introduced since this basic model was proposed by Holland. They are, in general, versions of the recombination and genetic alteration processes adapted to the requirements of particular problems. Examples of other genetic operators are: inversion, dominance, genetic edge recombination, etc.

The offspring produced by the genetic manipulation process constitute the next population to be evaluated. Genetic algorithms can either replace a whole population (generational approach) or only the less-fit members (steady-state approach). The *creation-evaluation-selection-manipulation* cycle is repeated until an ending criterion is reached. A typical GA may include one or a combination of the following ending criteria: maximum number of generations, maximum elapsed simulation time and convergence rate.

Following Holland's traditional GA, many variations of the basic algorithm have been introduced. However, an important and distinctive feature of all GAs is the population handling technique[36]. The traditional GA adopted a generational replacement policy where the whole population is replaced in each generation. Conversely, the steady-state policy used by many subsequent GAs employ a selective replacement for the population. It is possible, for example, to keep one or more individuals within the population for several generations, while they sustain a better fitness than the rest of the population.

This description of the GA computational model reviews the steps needed to design a genetic algorithm. Real implementations however, have to consider a number of problem-dependent characteristics. Problem representation, for instance, is one of the most important factors that determines the success or failure of applying genetic algorithms. A well-defined evaluation function is equally important, since the decision process that selects the population members that will survive is entirely based on the results provided by this function. Other aspects of significance relate to various mechanisms employed to avoid premature convergence and to assess the overall performance of an algorithm. The following sections briefly discuss some of these factors, starting with the various representation and coding methods currently used in many genetic algorithms.

2.3.1. Representation and Coding

Problem representation is one of the key factors for the success of a genetic algorithm. A great deal of study has been conducted in this area, but the issue is still a matter of controversy in the GA community. There are two main groups: those who defend Holland's original binary representation and those who argue that a GA should use the most appropriate representation according to the problem's characteristics and requirements. The latter approach accommodates binary and non-binary (e.g. real value) representations, and are becoming more popular with the use of GAs to tackle increasingly complex problems. This section presents both approaches to stress the importance of supporting them in a general-purpose programming environment.

Binary representation, as originally proposed by Holland, uses the lowest possible cardinality. Holland demonstrated, through the Schema theorem (see Figure 2.5), that fixed-length strings of low cardinality alphabets provide an effective means to search a solution space. Since genetic algorithms work essentially by promoting recombinations of genetic material, the binary representation provides the largest possible number of building blocks, or *schemata* (per population member), to be recombined. The most common encoding method maps the original representation of a problem variable – say an integer – into its binary value. This encoding method, however, is very sensitive to the mutation operator since the effects of this operator are very distinct, depending on the position of the bit being modified. An alteration in a high-order bit produces large variations on the original value, whereas modifications in low-order bits produce only slight variations. This tends to be a problem in the later stages of a simulation, when the population of solutions is closer to the best value. Many researchers adopted Gray encoding methods to overcome this problem. Gray encoding has the property that the binary coding of adjacent values, in the original representation, differs in only one bit. For instance the Gray representation for 7 is 0100 while 8 is coded as 1100. Some researchers [85] adopted this method as standard in their GAs, after performance improvements on the five DeJong [25] test suites were reported in the literature. Goldberg, however, states that Gray encoding may reduce the degree of implicit parallelism as demonstrated by the Schema theorem for the standard binary encoding. Other forms of encoding have also been proposed including adaptive techniques, such as dynamic parameter encoding (DPE) by Schraudolph and Belew [84], which addresses the problem of representing floating-point numbers in a fixed length string, without loosing precision.

Binary strings have been shown to be capable of usefully encoding a wide variety of information. The properties of binary representations for genetic algorithms have been extensively studied, and a good deal is known about the genetic operators and parameters that work well with them [23].

Real Value representation, on the other hand, emerged from the increasingly complex problems being addressed by GAs. It is also the standard representation used by Evolutionsstrategie-based algorithms, and is the only means to effectively represent genetic programming problems. GAs are extremely sensitive to different mappings, which added to the overhead imposed by the string encoding and decoding process, may prevent their use on a large number of real-world optimisation problems. Many industrial problems, for instance, already using traditional techniques, have a well-defined representation and evaluation function. In most cases, these functions would impose constraints on other representations, severely compromising the performance of any GA.

Figure 2.5 - The Schema Theorem

Holland introduced the notion of a *schema* as a collection of genomes that share certain gene values, or *alleles*. For instance, the schema 1**0* represents the set of chromosomes with a 1 at the first locus and a 0 at the fourth locus. A schema group, or *schemata*, provides a way of describing underlying similarities between successful strings. Since each schema is likely to be represented by many strings in the population, it is possible to work out an average score for each. Such explicit calculation is, nevertheless, unnecessary as the process is automatically handled by the selection of whole strings. The net result of the selection process is the increase of the number of strings containing good schemata in the population. Therefore, the number of copies of a schema in the next generation $N_S(g+1)$ can be expressed in terms of its current number of copies $N_S(g)$, its fitness f_S., the average fitness of the whole population $\bar{f}$, and the probability that the schema survives the genetic operators $p(S)$. It is given by the following expression:

$$N_S(g+1) \geq N_S(g)\frac{f_S}{\bar{f}}p(S)$$

Before expanding $p(S)$ into terms that express the influence of the mutation and crossover operators, it is necessary to define the *order* (o_S) of a schema and its *defining length* (d_S). The order of a schema is given by the number of bits that are not represented by the '*' symbol. The defining length gives the distance between two extreme bits that define the schema - note that '*' is not allowed at the extremes. Therefore, the schema 0**01*1 has o_S=4 and d_S=6.

A schema with defining length d_S would be destroyed by the crossover operator (assuming uniform distribution for choosing any point along the string length l) if the crossover point falls between the two extremes of the schema. Hence, the probability to survive crossover is given by:

$$p_c\frac{d_S}{(l-1)}$$

Where:
p_c = probability of applying the crossover operator
d_S = schema defining length
$(l-1)$ = possible positions for a crossover site

If p_m is the probability of applying the mutation operator, then the probability that a schema of order o_S survives mutation (for the typical low values of p_m) is:

$$(1-p_m)^{o_S} \approx 1-o_S.p_m$$

The Schema Theorem can then be finally re-written as follows:

$$N_S(g+1) \geq N_S(g)\frac{f_S}{\bar{f}}\left(1-p_c\frac{d_S}{(l-1)}-o_S.p_m\right)$$

Other problems, specially those using genetic programming, cannot be easily represented with flat, fixed-length, strings. In general, more complex representations are required involving mixed types of data (integers, floating-point, and even binary). Representations used in GP problems, for instance, invariably assume a tree configuration, which has its width and depth modified during simulation, expanding and shrinking many times. Therefore, Radcliffe [69] and others consider that "if there is no benefit to be gained from changing to a *special* genetic representation, it would seem perverse to do so".

From the theoretical point of view, Antonisse [2], Radcliffe [68] and Vose [97], demonstrated that extensions of the Schema theorem can also be applied to real value representation. Their arguments are based on the fact that GAs' search is guided by the quality of the information it collects about the space (through observed schema fitness averages in the population). They have contributed to the demolition of strong beliefs that only low cardinality (binary) representations offer intrinsic parallelism.

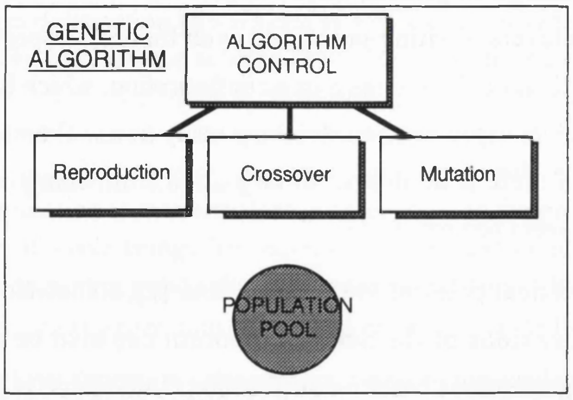
One of the major disadvantages of real value representations is the necessity to define representation dependent genetic operators. This, however, does not seem to be considered a problem by most of the people using this method, since the majority of problems already require genetic operators specially designed to cope with their complexities.

The conclusion of the above discussion is that a general-purpose programming environment must support, at least, binary and some real value representations to satisfy the requirements of diverse problems, using any of the three major evolutionary techniques. It would be even more interesting, however, if the genetic manipulations could take place over a representation-independent data structure. This would allow the user to choose freely the most suitable representation for a problem, and implement problem-independent genetic algorithms and operators.

2.3.2. Genetic Operators

A genetic algorithm has a hierarchical structure where the algorithm (the control level) determines the order and the sequence in which genetic operators are activated. It is also at the algorithm level that the decision to terminate the GA loop is taken. Parallel genetic algorithms add one extra level above the algorithm control level, which is responsible for starting several GAs and collecting intermediate and final results. Figure 2.6 shows a block diagram of the traditional GA. The algorithm level maintains global data – the population pool, population size, etc. – and local data such as ending criteria (max. number of generations, convergence rate and maximum simulation time), current number of generations and elapsed time.

Figure 2.6 - The GA hierarchy



One level below, genetic operators maintain only the (local) information necessary to perform their tasks (e.g. mutation and crossover rates). The traditional GA comprises three genetic operators: reproduction, crossover and mutation, activated in this order.

Reproduction – this operator performs two tasks: firstly it evaluates each member of the population and then, based on the average performance of the whole population, it selects those members that will survive to the next generation. Evaluation is highly dependent on the problem being solved. Some techniques, as explained in section 2.3.3, can be applied to minimise undesirable effects on the simulation performance, due to noisy and time-consuming evaluation functions.

Two important aspects have to be considered in the selection process: the composition of the new population, based on the proportional contribution of each string in the current population, and the number of copies from each selected string that will be present in the new population. The first aspect, known as *selective pressure*, has to be carefully balanced to avoid premature convergence – due to high pressure – or to stagnate the search – due to low pressure. The simplest method of allocating strings to the new population is in proportion to the ratio of their evaluated fitness to the average of the whole population. Therefore, if a particular string has twice the average fitness, it would be expected to be chosen twice as frequently. This method has been applied in most GAs, but it is not suitable for certain types of problems exhibiting large areas of poor performance with localised good spots. In such cases, the fitness of a good string will be far above the average, making it dominate the composition of a next generation. The immediate consequence is loss of diversity in the population, leading to premature convergence of the algorithm. Conversely, “well behaved” problems tend to have most of the population highly rated towards the end of the simulation. Those strings that are slightly better than the average get little selective advantage, and the search stagnates.

Several methods have been reported in the literature that help to avoid both situations [8,24,36]; the most common being "fitness scaling". This method ensures a constant fitness ratio, typically about 2, between the best and the worst strings, thus preventing exceptionally good strings from dominating the next generations. Adaptive methods have also been reported to improve the quality of the results. By dynamically changing the behavior of the selection process according to the best, worst and average fitness, the algorithm adapts itself to different situations presented by the search space.

The second aspect of reproduction entails determining the number of copies of selected strings to be present in the new population. For a GA with fixed population size N, the reproduction operator dictates that future generations must contain the same number of strings, produced by copying (one or more times) selected members of a previous generation. The simplest method, the "roulette-wheel", samples the population N times, with the probability of any member being chosen equal to its relative fitness (whether scaled or not). Therefore, the higher the relative fitness of a string, the bigger the probability of having more copies in the next generation.

It is important to note that the reproduction operator works essentially on the phenotype level, that is, it is not affected by the genetic representation of the problem. Other operators, such as crossover and mutation, act only on the genotype level. This may lead to a premature conclusion that these two operators are representation dependent. The objective of this discussion is to assess the possibility of defining a suitable abstraction for representing a variety of problems (using either binary or real value) that could permit representation-independent implementations of genetic operators.

Crossover – is considered the most important GA operator. In fact, crossover is unique to GAs (Evolutionsstrategie evolves populations only by the means of mutation). Crossover is responsible for promoting useful recombinations of the genetic information that accelerate the search process. It mimics sexual reproduction in which segments of two different strings (the parents) are recombined to form two other strings (the offspring). The standard one-point crossover, as defined by Holland, chooses a random point (based on a normal distribution) between the two extremes of the string. Several variations of this operator have been introduced, the most common used being the *two* or *n*-point crossover, generally applied over a circular string [32].

In general, problems using real value representations are very complex, exhibiting several parameters to be simultaneously optimised. The crossover operators used in these problems work by swapping one or more parameters between two "strings" – the term *individual*, as a reference to a member of the population, seems more appropriate for real value representation and will be used interchangeably with the term *string* in this thesis.

A careful observation of the ways the crossover operator acts over binary and real value representations, suggests a common mechanism for its implementation, independently of the problem representation. Instead of storing the data in the data structure that represents a population member (or string), one can store only references (or pointers) to the actual data. Crossover can then be implemented as a simple swap of pointers; irrespective of the contents of the information being swapped. This is an important step towards the definition of a representation-independent abstraction for programming GAs, ESs and GPs.

Mutation – introduces "unexpected" variations in the population in order to increase its diversity. Without mutation a GA would be constrained to explore only regions of the search space present in the schemata of the first (or initial) generation. If, by chance, the schema of the best solution is not present in the initial population, it would not be possible to find that solution without mutation.

The standard mutation operator works by simply flipping the value of a bit in the binary string. Unlike crossover, which is generally applied at high rates (0.6 to 0.8) to maximise recombination, mutation rates are typically very low (0.005 or less). High mutation rates can be very detrimental, preventing algorithm convergence. This is especially important in the final stages of a simulation, when the population is converging towards the best solution and large genotype variations are undesirable. Some GAs employ adaptive mutation with relatively high rates in the early stages of the simulation, and very low rates towards the end. This approach increases the chance of exploring different areas of the search space in the beginning, when the population needs diversity, without compromising algorithm's convergence.

Mutation operators used in problems with real value representation assume the most varied implementations. Some implementations employ "creeping" techniques, introducing small variations around the current value of the parameter being mutated. Others may produce large variations by replacing the current value by an entirely new one, randomly chosen (generally within certain constraints). Depending on the problem complexity, mutation may even be implemented over a data base of possible values that each parameter is allowed to assume.

The conclusion then is that the choice of mutation operator is extremely dependent on a problem's representation, characteristics and requirements. Nevertheless, since it acts over the genotype, it seems possible to embed the modification information (e.g. mutation rules and parameters) into the representation data structure. This approach is also consistent with Holland's discussion about the mutation operator [47], permitting each member of the population to carry its own (possibly different) mutation information.

A number of other, more sophisticated, operators have been described in the literature and, every year, new variations addressing particular domains and problems appear at GA conferences. A list of traditional, less used, operators includes:

Inversion – chooses a segment of the string and changes the order of its bits, but keeps their original relative position, or *loci,* information. The results of the fitness evaluations are, therefore, the same for the original and modified strings. The objective of this operator is to increase the possibility of exploring different schema configurations. In this sense, although the two strings present the same phenotype, their different genotype will certainly produce different offspring (with different phenotypes) after crossover. The implementation mechanisms for this operator are similar to those used for the crossover operator, i.e., pointer manipulations.

Dominance and Diploidy – These operators work on pairs of strings, which essentially encode the same information but, with slight variations on a bit basis that affect the phenotype value. This means, for instance, that if in a particular *locus* both strings present different bit values, then one of the strings (the dominant) is chosen to be decoded back to its phenotype. Conversely, if they have the same bit values, any of them could be decoded. The objective of this operator is to provide a "memory" of possibly good schema that, otherwise, would be lost due to temporary variations in the genetic material of the population – possibly influenced by a local minimum. Future recombinations of two strings, with recessive genes in the same position, could bring back to the population individuals with good combinations of dominant and recessive genes (in different string positions), displaying even better phenotypes. This discussion can be extended to polyploid representations. The implementation support to this operator can be achieved by allowing the data structures representing the problem to store more than one copy of a given allele (or value), and to embed the dominance criteria.

Sexual determination and differentiation – consist of allowing mating only between strings that carry certain chromosomic characteristics. This allows specialisation of the genetic material to be distributed over different sub-populations, which may result in global better performance. This operator is mostly indicated for multi-modal problems, requiring semi-independent optimisation for each parameter. A more sophisticated version of this operator is called *Niche and Speciation.* DeJong [25] did extensive work in this area, introducing the concept of *crowding factor* as a method for identifying groups of individuals in the population with common characteristics, and selectively replacing them. An individual of one group could only mate in its own group and would be replaced by an offspring with similar characteristics (even from a different group).

Again, both operators rely on extra information embedded in each individual's data structure to perform their actions. Other even less used operators present the same type of requirements for genetic data manipulation. Thus, two characteristics seem to be necessary and sufficient to define a data structure capable of problem-independent representation and manipulation:

- the data structure should maintain a list of references to each piece of genetically relevant information (whether binary or real value); and
- provide the means to embed extra information to be used by genetic operators.

To conclude this section, it is important to stress the self-contained nature of the genetic operators. This is characterised by the lack of information exchange between operators or between an operator and the algorithm that contains it. The algorithm control level (Figure 2.6) can be seen as a template where different operators may be attached, depending on the requirements of the problem to be solved. A subtle consequence of this characteristic is the ability to parameterise genetic algorithms.

2.3.3. Fitness Evaluation

The evaluation function is, perhaps, the most important part of a GA. It has the potential to severely affect the overall performance of the algorithm in terms of quality of results and simulation time.

The evolution process that guides a GA is entirely based on the results provided by the evaluation function. Therefore, an evaluation function that does not accurately represent characteristics and requirements of a problem, will certainly misguide the algorithm's search. Conversely, a very accurate evaluation function, containing too many constraints, will lead to good results, but such a function may prove to be unattainable. Since a GA requires a certain number of members in the population to perform its recombination task, costly evaluation functions are simply impractical. Complex functions produce a strong impact in the total computing time required to evaluate the entire population – possible solutions to this problem may involve parallel computation of the evaluation function.

A compromise solution tries to reduce the complexity of the evaluation function, but without sacrificing the quality of its results. Some techniques are based on the construction of genetic operators that embed the problem's constraints. They try to avoid the creation of "unsuitable" individuals in the population that, otherwise, would expend evaluation time. Another common mechanism is to temporarily store, separately, each individual's evaluation result (preferably into the individual itself). A typical GA requires each member of the population to be evaluated at least twice per generation. Firstly the evaluation function is called to compute the total and average fitness of the population, and then to obtain each member's result, during the selection stage. By storing each evaluation result, when computing the total fitness, the overall time taken by the evaluation stage can be divided by two. This simple mechanism operates as a "cache" for the fitness result. However, a cache entry needs to be invalidated every time a genetic operator modifies the genetic information it represents. The same mechanism can be applied to store the phenotype of each encoded parameter, and avoid multiple calls to the decode function. It is important to note, however, that the caching mechanism may also be affected by some classes of problems that present non-static evaluation functions. A common case being time-dependent fitness functions.

Other aspects related to evaluation such as averaging, noise immunity, scaling and penalty, have been extensively discussed in the literature [61] and will not be addressed in this thesis, since they are not relevant to the analysis of problem representation and algorithm structure.

2.4. Parallel Genetic Algorithms

Parallel processing systems are characterised by the existence of multiple *agents* that co-operate in the execution of a task [9]. Depending on the level of abstraction at which a parallel processing system is described, the notion of an agent may embody several different entities, and therefore imply various patterns of behaviour and characteristics. In a parallel machine, for instance, an agent is naturally identified with one of the processors in that machine. At a higher level of abstraction, however, the system might be a set of processes that constitute a parallel program, and in this case an agent would be identified with one of the processes. From a strictly conceptual point of view, it is irrelevant whether the parallelism is present at the hardware level or at the topmost software system level. In fact, parallel processing agents can be easily identified in as seemingly disparate settings as operating systems and computer networks, for example. According to this concept, a parallel genetic algorithm is considered as a multi-agent model, possibly exhibiting various degrees of interaction among its agents. The analysis of PGA models found in the literature [38,106,107,108,109] reveals three important characteristics for the implementation of agents and their interactions:

- granularity,
- synchronisation and
- type of parallelism.

The concept of granularity relates to the average size of the actions performed by agents (measured in number of executed instructions and used memory) and their degree of interaction (measured in terms of the rate between the time spent for execution and communication). Three granularity levels can be identified in parallel genetic algorithm models:

- *Coarse-grained* PGAs have their agents (whole algorithms) executing almost independently. The interaction among them is very occasional.
- *Medium-grained* PGAs exhibit frequent communication among their agents (genetic operators in this case), however the interval between two consecutive interactions is far longer than the time spent during communication.
- *Fine-grained* PGAs present highly interactive agents (here, individuals from sub-populations). The execution interval between consecutive interactions is short, and may be comparable to the time spent for communication.

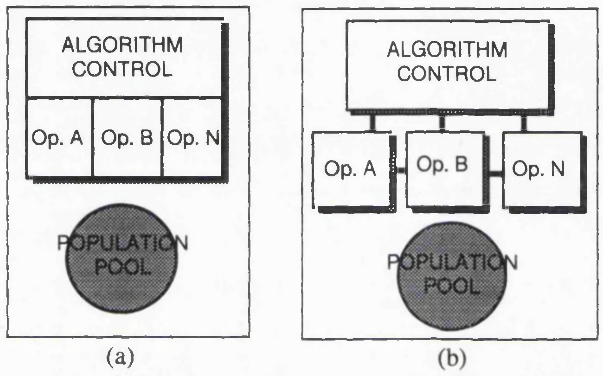
The traditional genetic algorithm is inherently synchronous due to its centralised selection stage. Many parallel models however, overcome this problem by adopting basically two different strategies. In coarse-grained PGAs, selection occurs over sub-populations local to each genetic algorithm instance. Conversely, fine-grained PGAs use distributed selection where a local individual may only be replaced by a migrant if this presents better fitness. In the first case each separate GA still has the synchronous selection stage, but the whole parallel GA is not synchronous. The second strategy is completely asynchronous.

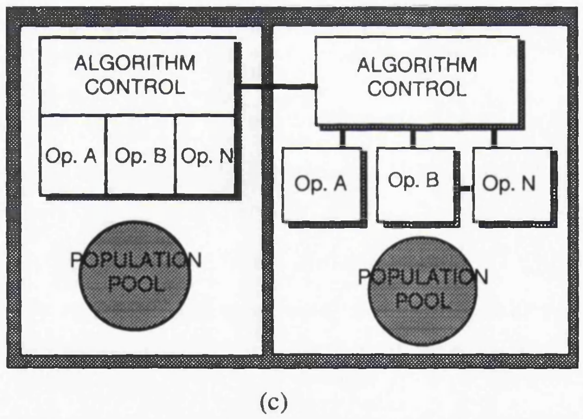
The third important characteristic of parallel genetic algorithms programs relates to the type of parallelism they exhibit. There are two types of parallelism: control parallelism and data parallelism. A program is said to have control parallelism if it could be divided into a number of agents operating independently on different processors (or processes). Problems presented by this type of parallel programs relate to the difficulty in identifying the agents and synchronising their actions. Programs presenting data parallelism take advantage of large amounts of data elements that are independent, or unrelated, and assign processors (or processes) to operate over sub-sets or individual data elements. Both types of parallelism coexist in most parallel genetic algorithm models. Data parallelism in PGAs is always associated with the population structure, and control parallelism is associated with algorithms and genetic operators working as independent agents.

Figure 2.7 shows three models of genetic algorithms with different degrees of interaction. The first model (Figure 2.7a) is a pure sequential GA, like Goldberg's Simple Genetic Algorithm [36]. It has a central control (Algorithm Control), which maintains the global population, the generation counter, the termination criteria and the genetic operators (Op.A ... Op.n). The genetic operators are activated in sequence to evolve the global population, represented by a large set of unrelated data elements. The second model (Figure 2.7b) is a parallel implementation of the same GA, where each genetic operator works as an independent agent, co-ordinated by the algorithm agent. Genetic operators in this model work in a pipeline that is synchronised by, and starts with, the selection operator. As soon as a pair of individuals is selected, the crossover operator is activated to produce modified offspring. These are then passed on to the mutation operator. While crossover is acting on a pair, selection may be choosing a new pair, and mutation modifying another one. This model presents medium granularity, since interactions among genetic operator agents are quite frequent ($([3\ x\ N\ x\ g] \div 2$, where N is the population size and g is the number of generations), but the interval between two consecutive interactions is expected to be longer than the time spent during communication.

The third model (Figure 2.7c) represents a coarse-grained implementation of the same algorithm. A number of similar genetic algorithm agents (sequential or pipelined) execute independently, evolving their own sub-populations. From time to time, these GAs may exchange members of their populations.

Figure 2.7 - Sequential and parallel genetic algorithm models.





Current PGA implementations [33,37,62] can be grouped into three major categories: the panmitic, the island, and the massively parallel GA models. Massively parallel GA models are also called cellular GAs or fine-grain GAs (Table 2.1 lists examples for each category).

Panmitic PGAs are essentially parallel versions of ordinary sequential GAs. They operate over a global population, are normally synchronous and present coarse to medium granularity. Panmitic PGAs are best suited to parallel architectures with shared memory. A parallel implementation of Goldberg's Simple GA, for instance, allocates $n/2$ processors (where n is the population size) to operate over two members of the population at each generation. Other examples of panmitic PGAs include Whitley's Genitor [101] and Eshelman's CHC [35].

Table 2.1 - PGA Models

Panmitic	***Island***	***Massively Parallel***
SGA	I-SGA	Fine-Grained PGA
	GAUCSD	Asparagos
pCHC	PGA	Cellular GA
	Distributed GA	DBGA
Genitor	Punctuated Equilibria	Fine-Grained PGA for Distributed Systems

Island and massively parallel models derive from population genetic theories stating that diversification occurs more naturally in populations with spatial structures. Selection and mating in these PGAs are restricted to neighbourhoods called *demes*. In the island model, the population is subdivided into a number of randomly distributed demes. The sub-populations are processed by independent instances of the same genetic algorithm, which, from time to time may exchange individuals with other algorithms in the same deme. Island PGAs implementations are asynchronous, coarse-grained and have been mostly mapped onto transputer-based MIMD architectures. In its simplest form, pure sequential GAs are replicated and distributed over a number of processors. Since the execution of each algorithm is completely independent, their final results may differ only due to the stochastic nature of GAs and possibly different initial populations. The I-SGA, for example contains in each island a copy of the sequential SGA, operating only over its own sub-population. Most of the island model implementations include a migration operator, which is responsible for "exporting" a single copy of a selected member of the local population to an adjacent population. Another member of the local population is also selected to be replaced by an incoming migrant. These PGAs are based on a ring topology and do not have central selection. GAUCSD [85] can be considered one of the first implementations of an island parallel GA. It is a distributed version of GENESIS, and does not use migration operators. Other implementations are the Punctuated Equilibria PGA from Cohoon [20], Pettey and Leuze's Parallel Genetic Algorithm [66] and Tanese's Distributed GA [92].

Massively parallel models are usually targeted at fine-grained parallel machines. They assign one individual per processing agent (processors in this case), and limit mating to a deme near the individual. In general, demes consist of four individuals, one in each direction in the plane that contains them. Edge elements wrap around, and the whole topology forms a torus. Each individual is processed in parallel at each generation and the offspring replaces the parent, if it has a better fitness. Again, there is no central selection. These PGAs are asynchronous and considered to exhibit fine-grain parallelism, since interactions among neighbour processors are quite high and the ratio between execution and communication can be very small. The preferred parallel architectures are SIMD machines, array processors and connection machines [33] (but some have also proven to be very effective on transputer-based implementations). Examples in this class are the Fine-Grained PGA from Manderick and Spiessens [58], Asparagos from Gorges-schleuter [39], Whitley's Cellular GAs [102], the Distributed Breeder GA [63] from Mühlenbein and the Fine-Grained PGA for Distributed Systems from Maruyama et al. [59].

Parallel genetic algorithms present diverse characteristics ranging from synchronous coarse and medium-grained models, such as panmitic and island, to asynchronous fine-grained models like the massively parallel GA. They also exploit control and data parallelism in various degrees. Most implementations are platform dependent, therefore not easily portable. Consequently, any programming environment intending to cover the broad range of characteristics presented by PGAs and achieve the required level of portability, needs to define a programming model, and a communication and parallel control mechanisms capable of being mapped onto diverse parallel platforms.

2.5. Summary

This chapter has briefly reviewed sequential and parallel genetic algorithms. The emphasis on the topics presented has aimed at better understanding of evolutionary computing, from an application developer's perspective. Special attention was given to the identification of mechanisms that could make possible the definition of genetic-oriented abstractions as well as to parameterise genetic algorithms and operators.

The discussion about problem independent representation, and their relationship with genetic operators, led to the identification of three important characteristics:

- There are two principal types of representation: binary and real value. The binary representation may use various encoding methods whereas real value may comprise different data types. Both alternatives may appear alone or mixed.
- Although traditional GAs present fixed-size representations, recent algorithms and problems based on genetic programming, require more flexible data structures. This implies representation structures capable of dynamically adjusting their size (e.g. width and depth) to support sophisticated recombination operators.
- Some problems require variations in the way genetic manipulations take place over their representation. This characteristic is particularly important to heavily constrained problems, which try to limit the number of unsuitable solutions produced by the genetic operators.

Based on these characteristics, it is possible to derive requirements for the definition of an abstraction that should permit the genetic representation of a wide range of problems. This abstraction should also allow the parameterisation of genetic operators, making them able to adapt their tasks according to particularities of each problem, by means of the extra information embedded into the genetic data structure. The definition of such an abstraction should fulfil the following requirements:

- allow maximum flexibility for the representation of GA, ES and GP problems. This means that mixed binary, real value and any possible variation, as well as non-fixed size representations should be supported.
- separate data contents from data organisation. This translates into being able to modify the organisation of the data without being concerned with their actual contents.
- provide the means to embed problem-dependent information that may affect the behaviour of the algorithm or its operators. Such a property should permit the data structure to store its phenotype value and fitness results, as discussed in section 2.3.3.

Section 2.3.2, discussing the hierarchical structure of GAs, highlighted significant aspects of their implementations. It was possible to identify the self-contained nature of genetic algorithms and genetic operators, based on the following factors:

- the extremely low communication rates between these modules;
- the master-slave structure of algorithms; and
- the asynchronous nature of the whole process.

These observations, added to other characteristics of parallel genetic algorithm models presented in section 2.4, provide the basis for the definition of the programming model, and its communication and task control support, described in Chapter 6.

Chapter 3

Programming Environments for Genetic Algorithms

This chapter reviews programming environments for genetic algorithms. It introduces a simple taxonomy that classifies existing programming environments into three major categories: application-oriented, algorithm-oriented and tool kits.

3.1. Overview

During the last decade genetic algorithms have been used increasingly for solving complex optimisation problems. Their simplicity, robustness and outstanding results have contributed to augmenting the demand for better application development support. A number of research groups and companies in the US and Europe have already produced software tool kits to help with the development of GA-based applications. Some of these systems have even reached mature stage, and are now commercial products, already being used by industry.

The computational model of a genetic algorithm has an enormous potential for the exploitation of control and data parallelism. Nevertheless, the majority of currently available programming environments fail to address this aspect. This chapter reviews existing software systems for programming genetic algorithms applications. This review is part of a comprehensive survey on GA programming environments that was published in the IEEE COMPUTER magazine [73], in an issue dedicated to genetic algorithms[2]. It forms, together with the analysis of GAs and PGAs presented in the previous chapter, the basis for the design of the general-purpose programming environment described in the next chapters.

The review starts by introducing a taxonomy that classifies GA systems into three major categories: application-oriented, algorithm-oriented and tool kits. For each category the design and main features of some of the most important systems are presented and, as a case study, one specific system is examined in more detail.

[2] A copy of the article can be found at the end of this thesis.

3.2. Taxonomy of GA Programming Environments

Genetic algorithms programming environments can be classified according to a taxonomy with three major classes: Application-oriented systems, Algorithm-oriented systems and Tool Kits.

- **Application-oriented systems** are essentially "black boxes" that hide the GA implementation details. Targeted at business professionals, some of these systems support a range of applications; others focus on specific domains, such as finance or scheduling.
- **Algorithm-oriented systems** are programming systems that support specific genetic algorithms. They sub-divide into:

  ⇒ *Algorithm-specific* systems – which contain a single genetic algorithm; and

  ⇒ *Algorithm Libraries* – which group together a variety of genetic algorithms and operators.

Algorithm-oriented systems are often supplied in source code and can be incorporated easily into user applications.

- **Tool Kits** are programming systems that provide many programming utilities, algorithms and genetic operators that can be used in a wide range of application domains. These programming systems sub-divide into:

  ⇒ *Educational systems* – to help novice users to obtain a hands-on introduction on GA concepts. Typically these systems support a small set of options for configuring an algorithm.

  ⇒ *General-purpose systems* – to provide a comprehensive set of tools for programming any GA application.

Table 3.1 illustrates the taxonomy, listing some of the GA programming environments reviewed in this chapter.

Table 3.1 - Classes of GA programming environments

Application Oriented	*Algorithm Oriented*		*Tool Kits*	
	Algorithm-Specific	*Algorithm-Libraries*	*Educational*	*General-Purpose*
EVOLVER	ESCAPADE	*EM*	GA Workbench	EnGENEer
OMEGA	GAGA			GAME
PC/BEAGLE	GAUCSD			*Micro*GA
XpertRule	GENESIS	OOGA		PeGAsuS
GenAsys	GENITOR			Splicer

3.3. Application-oriented Systems

Application-oriented systems are designed for use by business professionals who wish to utilise genetic algorithms in specific applications domains, without having to acquire detailed knowledge of the workings of genetic algorithms.

As seen with expert systems and neural networks, many potential users of a novel computing technique, such as genetic algorithms, are only interested in the applications, rather than the details of the technique. For example, a manager in a trading company may wish to optimise its delivery scheduling. By using an application-oriented programming environment, it is possible, for instance, to configure a particular application for schedule optimisation, without knowing the encoding technique or the genetic operators involved.

Overview

A typical application-oriented environment is analogous to a spreadsheet or word-processing utility. It comprises a menu-driven interface (tailored to business users) giving access to a suite of parameterised modules (targeted at specific domains). Their user interfaces provide menus to configure, monitor and, in certain cases, even assist in programming applications. These systems generally provide good help facilities as well.

Survey

Application-oriented systems follow many innovative strategies. Systems, such as PC/BEAGLE and XpertRule GenAsys, are expert systems using GAs to generate new rules to expand their knowledge base of the application domain. EVOLVER, for instance, is a companion utility for Spreadsheets; and systems like OMEGA, are especially targeted at financial applications.

EVOLVER — is an add-on utility that works within the Excel, WingZ and Resolve spreadsheets on Apple Macintosh and IBM-PC compatible computers. It is being marketed by Axcêlis Inc., who describes it as "an optimisation program that extends mechanisms of natural evolution to the world of business and science applications". The user starts with a model of his system in the spreadsheet and calls EVOLVER from a menu. After filling a dialogue box with the information required (e.g. cell to minimise or maximise) the program starts working, evaluating thousands of scenarios automatically, until it finds an optimal answer. The program runs in background, freeing the user to work in the foreground. When the best result is found, the user is notified and the values are placed into the spreadsheet for analysis. This is an excellent design strategy given the importance of interfacing with spreadsheet in business. In an attempt to improve the system and expand its market, Axcêlis introduced Evolver 2.0 that is being shipped with many tool-kit-like features. The new version is capable of integrating with other applications, besides spreadsheets. Also it offers more flexibility by accessing the "Evolver Engine" from any MS-Windows application capable of calling a Dynamic Link Library (DLL).

OMEGA — the OMEGA Predictive Modelling System, marketed by KiQ Limited, is a powerful approach to developing predictive models. It exploits advanced genetic algorithms' techniques to create a tool that is "flexible, powerful, informative and straightforward to use", according to KiQ. OMEGA is geared to the financial domain and can be applied in the following sectors: Direct Marketing, Insurance Risk (case scoring) and Credit Management. The environment offers facilities for automatic handling of data; business, statistical or custom measures of performance; simple and complex profit modelling; validating sample tests; advanced confidence tests; real-time graphics, and optional control over the internal genetic algorithm.

PC/BEAGLE — produced by Pathway Research Ltd, is a rule-finder program that applies machine-learning techniques to create a set of decision rules for classifying examples previously extracted from a database. It has a module that generates rules by natural selection. Further details are given in the case study section.

XpertRule GenAsys — is an expert system shell with embedded genetic algorithms, marketed by Attar Software. This GA expert system is targeted to solve scheduling and design problems. The system combines the power of genetic algorithms in evolving solutions with the power of rule-base programming in analysing the effectiveness of solutions. Rule-base programming can also be used to generate the initial solutions for the genetic algorithm and for post optimisation planning. Some examples of design and scheduling problems that can be solved by this system are: optimisation of design parameters in electronic and avionics industries, route optimisation in the distribution sector, production scheduling in manufacturing.

Case Study - PC/BEAGLE

PC/Beagle is a rule finder program that examines a database of examples and uses machine learning techniques to create decision rules for classifying those examples, turning data into knowledge. The software analyses an expression via a historical database and develops a series of rules to explain when the target expression is false or true. The system comprises six main modules that are generally run in sequence:

- **SEED** (Selectively Extracts Example Data) puts external data into a suitable format, and may append leading or lagging data-fields as well.
- **ROOT** (Rule Oriented Optimisation Tester) tests an initial batch of user-suggested rules.

- **HERB** (Heuristic Evolutionary Rule Breeder) generates decision rules by Naturalistic Selection, using GA philosophy (ranking mechanisms are also supported).
- **STEM** (Signature Table Evaluation Module) makes a signature table from the rules produced by HERB.
- **LEAF** (Logical Evaluator and Forecaster) uses STEM output to do forecasting or classification.
- **PLUM** (Procedural Language Utility Maker) can be used to convert a BEAGLE rule-file into a language such as Pascal or FORTRAN; in this form the knowledge gained may be used by other software.

PC/BEAGLE accepts data in ASCII format, with items delimited either by commas, spaces or tabs. Rules are produced as logical expressions. The system is a highly versatile package covering a wide range of applications. Insurance, weather forecasting, finance and forensic science are some examples. PC/Beagle requires an IBM-PC-compatible computer with at least 256 Kbytes of RAM and an MS-DOS or PC-DOS operating system, version 2.1 or later.

3.4. Algorithm-oriented Systems

This taxonomy divides algorithm-oriented systems into algorithm-specific systems that contain a single algorithm and algorithm libraries, which group together a variety of genetic algorithms and operators.

3.4.1. Algorithm-specific Systems

Algorithm-specific environments embody a single powerful genetic algorithm. These systems typically have two groups of users: system developers requiring a general-purpose GA for their applications, and researchers interested in the development and testing of a specific algorithm and genetic operators.

Overview

In general, algorithm-specific systems come in source code form and allow the expert user to make alterations for specific requirements. They present a modular structure providing a high degree of modifiability. Their user interfaces are usually rudimentary, and often command-line driven. Typically, these systems have been developed in universities and research centres and their source code is available free over world-wide computer research networks.

Survey

The most well known programming system in this category is the pioneering GENESIS [40], which has been used to implement and test a variety of new genetic operators. In Europe, probably the earliest algorithm-specific system was GAGA. For scheduling problems, GENITOR [100] is another influential system that has been successfully used. GAUCSD allows parallel execution by distributing several copies of a GENESIS-based GA into UNIX machines in a network. Finally, ESCAPADE [46] employs a somewhat different approach – being based on an Evolutionsstrategie – as discussed below.

ESCAPADE — Evolution Strategies **cap**able of **ad**aptive **e**volution — this software package provides a sophisticated environment for building applications using Evolutionsstrategie. ESCAPADE is based upon KORR, Schwefel's implementation of a $(\mu \overset{+}{,} \lambda)$ Evolutionsstrategie. The system provides an elaborate set of monitoring tools to gather data from an optimisation run of KORR. According to the system's author, it should be possible to incorporate a different implementation of an ES or even a GA into the system using its runtime support. The program structure is separated into several independent components that support the various tasks during a simulation run. The main modules are: Parameter Setup, Runtime Control, KORR, Generic Data Monitors, Customised Data Monitors, and Monitoring Support.

During an optimisation run the monitoring modules are invoked by the main algorithm (KORR or some other ES or GA implementation) to realise the logging of internal quantities. The system is not equipped with any kind of graphics interface. All parameters for a particular simulation are passed over as command line options. The output is produced by each data monitor writing their data into separate log files.

GAGA — Genetic Algorithms for General Application — was originally programmed by Hillary Adams, University of York, in Pascal. It is a task-independent genetic algorithm. The user must supply the target function to be optimised (minimised or maximised) and some technical GA parameters, and wait for the output. It is suitable for the minimisation of many "difficult" cost functions.

GAUCSD — This software package was developed by Nicol Schraudolph at the University of California, San Diego [85]. The system is based on GENESIS 4.5 and runs on UNIX, MS-DOS, CrayOs and VMS platforms; but presumes a UNIX environment. It comes with an *awk* script called "wrapper", which provides a higher level of abstraction for defining the evaluation function. By supplying the code for decoding and printing the evaluation function parameters automatically, it allows the direct use of most *C* functions as evaluation functions, with few restrictions. The software also includes a Dynamic Parameter Encoding (DPE) technique developed by Schraudolph, which he claims radically reduces the gene length, while keeping the desired level of precision for the results. Applications created with GAUCSD can be run in background, at low priority, using the *go* command. This command can also be used to execute GAUCSD in remote hosts. The results are then copied back to the user's local directory and a report is produced, if appropriate. If the host is not binary compatible, GAUCSD compiles the whole application in the remote host. Experiments can be queued in files, distributed to several hosts and executed in parallel. The *ex* command will notify the user via *write* or *mail* when all experiments are completed. The experiments are distributed according to a specified loading factor (how many programs will be sent to each host) along with the remote execution arguments to the *go* command. GAUCSD is a very powerful system for parallel simulations.

GENESIS — **GENE**tic Search Implementation System — was written by John Grefenstette to promote the study of genetic algorithms for function optimisation. It has been under development since 1981, and has been widely distributed to the research community since 1985. The software package is a set of routines written in the *C* language. To build their own genetic algorithm, the user has only to provide a routine with the fitness function and link it with the rest of the system. It is also possible to modify or add new modules (e.g. genetic operators, data monitors) and create a different version of GENESIS. In fact, GENESIS has been used as a base for test and evaluation of a variety of genetic algorithms and operators. It was primarily developed to work in a scientific environment offering a suitable software tool for research. It provides a high degree of modifiability and a variety of statistical information on outputs.

GENITOR — **GEN**etic **I**mplemen**TOR** — is a modular GA package containing examples for floating-point, integer and binary representations. Its features include many sequencing operators as well as sub-population modelling. This software package is, in fact, the implementation of the GENITOR algorithm developed by Darrel Whitley [100]. The algorithm presents two major differences from standard genetic algorithms. The first one is the explicit use of ranking. Reproductive trials are allocated according to the rank of the individual in the population rather than using fitness proportionate reproduction. The second difference is that GENITOR abandons the generational[3] approach and reproduces new genotypes on an individual basis. It does so in such a way that parents and offspring can typically coexist. The newly created offspring replaces the lowest ranking individual in the population rather than a parent. This replacement method is called Steady State. GENITOR only produces one new genotype at a time, so inserting a single new individual is relatively simple. Furthermore, the insertion automatically ranks the individual with relation to the existing population — no further measure of the relative fitness is needed.

Case Study - GENESIS

[3] The whole population is replaced in each generation.

GENESIS is the most well known software package for genetic algorithm development and simulation. It is now on version 5.0, which is available from The Software Partnership company. GENESIS runs on most machines with a *C* compiler. The present version runs successfully on both Sun workstations and IBM-PC compatible computers. According to the system's author, the code has been designed to be portable, but minor changes may be necessary for other systems. The system provides the fundamental procedures for *genetic selection, crossover* and *mutation.* Since GAs are task-independent optimisers, the user must provide only an evaluation function that returns a value when given a particular point in the search space.

GENESIS has three levels of representation for the data structures it evolves. The lowest level, or *packed representation,* is used to maximise both space and time efficiency in manipulating data structures. In general, this level of representation is transparent to the user. In the next level, called *string representation,* data structures are represented as null-terminated arrays of *chars*. This structure is made available for users who wish to provide an arbitrary interpretation of the genetic structures, for example, non-numeric concepts. The third level, or *floating-point*representation, is the appropriate level for many numeric optimisation problems. At this level, the user can think of genetic structures as vectors or real numbers. For each parameter, or *gene,* the user specifies its range, its number of values, and its output format. The system then automatically lays out the string representation, and translates between the user-level genes and lower representation levels. The system contains five major modules:

- *Initialisation* – the initialisation procedure sets up the initial population. It is possible to "seed" the initial population with heuristically chosen structures. The rest of the population is filled with random structures. It is also possible to initialise the population with real numbers.
- *Generation* – this is responsible for the execution of the selection, crossover, mutation, and evaluation procedures. It also collects data that are used later to produce several reports.
- *Selection* – this is the process of choosing structures for the next generation from the structures in the current generation. The default selection procedure is a stochastic procedure, which guarantees that the number of offspring of any structure is bounded by the floor and the ceiling of the (real-valued) expected number of offspring. The procedure is based on the roulette wheel algorithm. It is also possible to perform selection based on a ranking algorithm. Ranking helps prevent premature convergence by impeding super individuals from taking over the population within a few generations.
- *Mutation* – after the new population is selected, mutation is applied to each genetic structure in the new population. Each position is given a chance (mutation rate) of

undergoing mutation. If mutation does occur, a random value is chosen from {0,1} for that position. If the mutated structure differs from the original one, it is marked for evaluation.

- *Crossover* – exchanges alleles among adjacent pairs of the first *n* structures in the new population. The result of the crossover rate applied to the population size gives the number of structures to operate on. Crossover can be implemented in a variety of ways. If, after crossover, the offspring are different from the parents, then the offspring replace the parents, and are marked for evaluation.

These basic modules are added to the evaluation function supplied by the user to create the customised version of the system. The evaluation procedure takes one structure as input and returns a double precision value.

To execute GENESIS three programs are necessary: *setup, report* and *ga*. The setup program prompts for a number of input parameters. All the information is stored in files for future use. It is possible to set the type of representation, the number of genes, number of experiments, trials per experiment, population size, length of the structures in bits, crossover and mutation rates, generation gap, scaling window and many other parameters. Each parameter has a default value.

The report program runs the *ga* and produces a description of the algorithm performance. It summarises the mean, variance and range of several measurements, including on-line, off-line and average performance of the current population, as well as the current best value.

3.4.2. Algorithm Libraries

Algorithm Libraries provide a powerful collection of parameterised genetic algorithms and operators generally coded in a common language, and so are easily incorporated into user applications.

Overview

These systems are modular, allowing the user to select a variety of algorithms, operators and parameters to solve a particular problem. Their parameterised libraries provide the ability to use different models (algorithms, operators and parameter settings) to compare the results for the same problem. New algorithms coded in high-level languages, like *C* or *Lisp*, can be easily incorporated into the libraries. The user interface is designed to facilitate the configuration and manipulation of the models, and to present the results in different shapes (tables, graphics, etc.).

Survey

The two leading algorithm-libraries are *EM* and OOGA. Both systems provide a comprehensive library for genetic algorithms, but *EM* also supports simulations using Evolutionsstrategie. OOGA, on the other hand, can be easily tailored for specific problems. It runs in *Common Lisp* and *CLOS* (Common Lisp Object System), an object-oriented extension of the *Common Lisp* language.

EM — *E*volution *M*achine — has been developed by Hans-Michael Voigt, Joachim Born and Jens Treptow [98] at the Institute for Informatics and Computing Techniques in Germany. The *EM* simulates natural evolution principles to obtain efficient optimisation procedures for computer models. The evolutionary methods included in *EM* were chosen to provide algorithms with different numerical characteristics. The programming environment supports the following algorithms:

- Evolutionsstrategie by Rechenberg [71],
- Evolutionsstrategie by Rechenberg & Schwefel [81],
- Evolutionsstrategie by Born [14],
- Simple Genetic Algorithm by Goldberg [36], and
- Genetic Algorithm by Voigt and Born [98].

To run a simulation session the user provides the fitness function coded in the *C* programming language. The system calls the compiler and the linker to produce an executable file containing the selected algorithm and the user-supplied fitness function.

EM uses extensive menus with default parameter settings, data processing for repeated runs and graphical presentation of results (on-line presentation of the evolution progress, one, two, and three-dimensional graphs). The system runs on IBM-PC compatible computers with MS-DOS operating system and uses the Turbo C (or Turbo C++) compiler to generate the executable files.

OOGA — **O**bject **O**riented **G**enetic **A**lgorithm — is a simplified version of the Lisp-based software that has been developed since 1980 by Lawrence Davis. It was mainly created as a support for Davis' book [24] but can also be used to develop and test customised or new genetic algorithms and genetic operators.

Case Study - OOGA

OOGA is a system designed so that each of the techniques employed by a GA is an object that can be modified, displayed or replaced in an object-oriented fashion. The highly modular OOGA architecture makes it easy for the user to define and use a variety of genetic algorithm techniques, by incrementally writing and modifying components in Common Lisp. The files in the OOGA system contain implementations of several techniques used by genetic algorithm researchers, but they are not exhaustive. OOGA contains three major modules:

- *Evaluation Module* which has the evaluation (or fitness) function that measures the worth of any chromosome in the problem to be solved;
- *Population Module* contains a population of chromosomes and the techniques for creating and manipulating that population. There are a number of techniques for population representation (e.g. binary, real number, etc.), initialisation (e.g. random binary, random real, normal distribution, etc.) and deletion (e.g. delete all, delete last, etc.);
- *Reproduction Module* has a set of genetic operators responsible for selecting and creating new chromosomes during the reproduction process. This module allows genetic algorithm configurations with more than one genetic operator as well as its parameters' settings. The system creates a list with the user-selected operators and executes them in sequence. There are a number of genetic operators for selection (e.g. roulette wheel), crossover (e.g. one and two-point crossover, mutate-and-crossover) and mutation. All the parameters, such as bit mutation rate and crossover rate, can be set by the user.

The last two modules are, in fact, a library of several different techniques that enables the user to configure a particular genetic algorithm. When the genetic algorithm is run, the Evaluation, Population and Reproduction modules work together to effect the evolution of a population of chromosomes towards the best solution.

3.5. Tool Kits

Tool kits comprise educational systems for novice users and general-purpose systems with a comprehensive set of tools.

3.5.1. Educational Systems

Educational programming systems are designed for the novice user to obtain hands-on introduction to genetic algorithms' concepts. They typically provide a graphical interface and a simple configuration menu.

Overview

Educational systems are typically implemented on IBM-PC computers for portability and low cost reasons. For ease of use, they have an accessible graphical interface and are fully menu-driven. GA Workbench is one of the best examples of this class of programming environments.

Case Study – GA Workbench

GA Workbench has been developed by Mark Hughes, from Cambridge Consultants Ltd. It is a mouse-driven interactive GA program that runs on MS-DOS/PC-DOS microcomputers. The system is aimed at people wishing to understand and get hands-on GA practice. Evaluation functions are drawn on screen, using the mouse. The system produces run-time plots of GA population distribution, peak and average fitness. Many useful population statistics are also displayed. It is possible to change a range of parameters including the settings of the genetic operators, the population size, breeder selection, etc.

Its graphical interface requires a VGA or EGA graphic adapter and it divides the screen into seven fields:

- A *Command Menu* – this is a menu-bar that has general commands to start or stop a GA execution, as well as let the user enter the target function
- *Target Function Graph* – after selecting the "Enter Targ" command from the command menu, the user inputs the target function by drawing it on a graph using the mouse cursor.
- *Algorithm Control Chapter* – this field is called "chapter" because it can contain several pages, but only one page is visible at a time. It initially displays a page called "Simple Genetic Algorithm". Pages can be flipped through, forwards or backwards, by clicking the left mouse button on the arrows in the top high hand corner of the chapter. Following is a brief description of the available pages:
  - ⇒ *Simple Genetic Algorithm Page* – this page shows a number of input variables used to control the operation of the algorithm. The variable values can be numeric or text strings, and the user can alter any of these values by clicking the left mouse button on the up or down arrows, to the left of each value.
  - ⇒ *General Program Control Variables Page* – this page contains variables related to general program operation rather than a specific algorithm. Here the user can select the source of data for plotting on the output plot graph, set the scale for the X or Y axis, determine the frequency with which the population distribution histogram is updated or seeds the random number generator.

- *Output Variables Box* – this contains the current values of a number of variables relating to the current algorithm. For the Simple Genetic Algorithm, a counter of generations is presented as the optimum fitness value, the current best fitness, the average fitness, the optimum x, current best x, and the average x.
- *Population Distribution Histogram* – this graph shows the genetic algorithm's distribution of organisms by value of x. The histogram is updated according to the frequency set in the general control variables page.
- *Output Graph* – this field is used to display plots of several output variables against time.
- *Axis Value Box* – this box is used in combination with the mouse cursor to read values from any of the graphs described above. When the mouse is moved over the plot area of any graph, it changes to a cross hair and causes the Axis Value box to display the co-ordinate values of the corresponding graph at the point indicated by the cursor.

By drawing the *Target Function*, varying several numeric control parameters, and selecting different types of algorithms and genetic operators, the novice user can practise and have a good idea of how quickly the algorithm is able to find the peak value, or indeed if it succeeds at all.

3.5.2. General-purpose Programming Systems

General-purpose systems are the ultimate in flexible GA programming systems. Not only do they allow users to develop their own GA applications and algorithms, but also provide users with the opportunity to customise the system to suit their own purposes.

Overview

These programming systems provide a comprehensive tool kit, including:

- a sophisticated graphic interface;
- a parameterised algorithm library;
- a high-level language for programming GAs; and
- an open architecture.

Access to the system components is, in general, via a menu-driven graphic interface, and a graphic display/monitor. The algorithm library is normally "open", allowing the user to modify or enhance any module. A high-level language – often object-oriented – may be provided which supports the programming of GA applications, algorithms and operators through specialised data structures and functions. Lastly, due to the growing importance of parallel GAs, some systems provide translators to parallel machines and distributed systems, such as networks of workstations.

Survey

The number of general-purpose systems is increasing, stimulated by growing interest in the application of GAs in many domains. Examples of systems in this category include EnGENEer from Logica Cambridge, *Micro*GA, an "easy-to-use" object-oriented environment for IBM-PCs and Apple Macintoshes, PeGAsuS, a parallel environment, and Splicer, which presents interchangeable libraries for developing applications.

EnGENEer — Logica Cambridge Ltd. developed EnGENEer [78] as an in-house genetic algorithm environment to assist the development of GA applications in a wide range of domains. The software was written in *C* and runs on UNIX systems as part of a consultancy and systems package. It supports both interactive (X-Windows) and batch (command-line) modes of operation. Also a certain degree of parallelism is supported for the execution of application-dependent evaluation functions.

EnGENEer provides a number of flexible mechanisms allowing the developer to rapidly bring the power of GAs to bear on new problem domains. Starting with the Genetic Description Language, the developer can describe, at high-level, the structure of the "genetic material" used. The language supports discrete genes with user-defined cardinality and includes features such as multiple models of chromosomes, multiple species models and non-evolvable parsing symbols, which can be used for decoding complex genetic material.

A descriptive high-level language, the Evolutionary Model Language, is also available to the user. It allows the description of the GA in terms of configurable options including: population size, population structure and source, selection method, crossover type and probability, mutation type and probability, inversion, dispersal method, and number of offspring per generation.

Both the Genetic Description Language and the Evolutionary Model Language are fully supported within the interactive interface (including on-line help system) and can be defined either "on the fly" or loaded from audit files, which are automatically created during a GA run.

Monitoring of GA progress is provided via both graphical tools and automatic storage of results (at user-defined intervals). This allows the user to restart EnGENEer from any point in a run, by loading both the population at that time and the evolutionary model that was being used.

Connecting EnGENEer to different problem domains is achieved by specifying the name of the program used to evaluate the problem-specific fitness function and constructing a simple parsing routine to interpret the genetic material. A library of standard interpretation routines is also provided for commonly used representation schemes such as Gray-coding, permutations, etc. The fitness evaluation can then be run as either a slave process to the GA or via standard handshaking routines. Better still, it can be run on either the machine hosting the EnGENEer or on any, sequential or parallel, hardware capable of connecting to a UNIX machine.

*Micro*GA — marketed by Emergent Behavior, is designed to be used on a wide range of complex problems, while at the same time being small and easy to use. The environment is also designed to be expandable. The system is a framework of *C++* objects and, as such, it is designed so that several pieces are used in conjunction with each other to give the user some default behaviour. Therefore, it goes far from the library concept where a set of functions (or classes) is offered to be incorporated into the user application. The framework is almost a ready-to-use application, needing only a few user-defined parameters to start running. The package comprises a compiled library of *C++* objects, three sample programs, a sample program with an Object Windows Library user interface (from Borland) and the *Galapagos* code generation system. *Micro*GA runs on IBM-PC compatible systems with Microsoft Windows 3.x, using Turbo/Borland C++. It also runs on Apple Macintosh computers.

The application developer can configure his application either using Galapagos or manually. Galapagos is a Windows-based code generator that produces, from a set of custom templates and a little information provided by the user, a complete standalone *Micro*GA application. It helps with the creation of a subclass derived from its "TIndividual" class, required by the environment to create the genetic data structure to be manipulated. The number of genes for the prototype individual, as well as the range of possible values they can assume is requested by Galapagos. The evaluation function can be specified, but the notation used does not allow complex, or non-mathematical fitness functions to be entered via Galapagos. As a result, Galapagos creates a class, derived from TIndividual, which contains specific member functions according to user's requirements.

Applications requiring complex genetic data structures and fitness functions can be defined manually, by inheriting from the TIndividual class and writing the code for its member functions. After creating the application-dependent genetic data structure and fitness function, *Micro*GA compiles and links everything using the Borland C++ or Turbo C++ compiler, and produces an MS-Windows executable file.

*Micro*GA is very easy to use and allows fast creation of genetic algorithms' applications. However, for real applications the user has to understand basic concepts of object-oriented programming and Windows interfacing.

PeGAsuS — is a Programming environment for parallel Genetic AlgorithmS developed at the German National Research Centre for Computer Science. In fact, it is a tool kit that can be used for programming a wide range of genetic algorithms, as well as for educational purposes.

The environment is written in *ANSI-C* and is available for many different UNIX-based machines. It runs on MIMD parallel machines, such as transputers, and distributed systems with workstations. PeGAsuS is structured in four hierarchical levels:

- the User Interface,
- the PeGAsuS Kernel and Library,
- compilers for several UNIX-based machines, and
- the sequential/distributed or parallel hardware

The User Interface consists of three parts: the PeGAsuS script language, a graphical interface and a user library. The user library has the same functionality as the PeGAsuS GA library. It allows the user to define application-specific functions that are not provided by the system library. The script language is used to define application-dependent data structures, "attach" the genetic operators to algorithms and specify the input/output interface.

The Kernel includes the *base* and the *frame* functions. A base function controls the execution order of the genetic operators, manages communication between different processes and provides input/output facilities. This set of functions builds general frames for simulating GAs, and can be considered as autonomous processes. They interpret the PeGAsuS script, create appropriate data structures, and describe the order of the frame functions. Frame functions control the execution of a single genetic operator, and are invoked by base functions. They prepare the data representing the genetic material, and apply the genetic operators to it, according to the script specification. The Library maintains genetic operators, a collection of fitness functions, and input/output and control procedures. It provides the user with a number of validated modules for constructing applications.

Currently, PeGAsuS can be compiled with the GNU C, RS/6000 C, ACE-C, and Alliant's FX/2800 C compilers. It runs on SUNs and RS/6000 workstations, as well as on the Alliant FX/28 MIMD architecture.

Splicer — This software environment was created by the Software Technology Branch of the Information Systems Directorate at NASA/Johnson Space Center, with support from the MITRE Corporation [65]. It is one of the most comprehensive environment currently available, and forms the case study below.

Case Study – Splicer

Splicer presents a modular architecture that includes: a genetic algorithm kernel, interchangeable representation libraries, fitness modules, and user interface libraries. It was originally developed in *C* on an Apple Macintosh and has been subsequently ported to UNIX workstations (SUN3 and 4, IBM RS/6000) using X-Windows. The genetic algorithm kernel, representation libraries, and fitness modules are completely portable. The following is a brief description of the major modules:

- *Genetic Algorithm Kernel* – the GA kernel comprises all functions necessary for the manipulation of populations. It operates independently from the problem representation (encoding), fitness function and user interface. Some of its supported functions are: creation of populations and members, fitness scaling, parent selection and sampling, and generation of population statistics.
- *Representation Libraries* – interchangeable representation libraries are able to store a variety of pre-defined problem-encoding schemes and functions. This allows the GA kernel to be used for any representation scheme. There are representation libraries for binary strings and for permutations. These libraries contain functions for the definition, creation and decoding of genetic strings as well as multiple crossover and mutation operators. Furthermore, the Splicer tool defines the appropriate interfaces to allow the user to create new representation libraries.
- *Fitness Modules* – these are interchangeable modules where fitness functions are defined and stored. It is possible to create a fitness (scoring) function, set the initial values for various Splicer control parameters (e.g. population size), create a function that graphically displays the best solutions as they are found, and provide descriptive information about the problem.
- *User Interface Libraries* – there are two user interface libraries: an Apple Macintosh and an X-Window System user interface. They are event-driven interfaces and provide a graphic output in windows.

Splicer provides basic facilities to build applications using pre-defined genetic operators from its libraries. However, to create a Splicer application for a particular problem, a Fitness Module must be built using the *C* language.

3.6. Summary

This chapter has presented a review of software environments for programming genetic algorithms' applications. A taxonomy has been introduced based on systems' features, types of applications they help to create and their target users. Three major classes of programming environments have been identified: application-oriented, algorithm-oriented and tool kits. Application-oriented systems are targeted at specific domains and therefore do not offer much flexibility. Algorithm-oriented systems help with the creation of new application by providing a set of pre-defined algorithms. However these systems are not concerned with the portability of their code nor the expandability of their functionality. In general, they are provided as bare libraries that an application developer will use to build applications. Any adaptation that a problem may require has to be done in the libraries' source code level, which sometimes is not available. Finally, tool kits provide the ideal environment for developing a wide range of applications.

Most of the reviewed systems offer a comprehensive set of tools to assist with the development and execution of applications. Libraries of algorithms and genetic operators are normally provided, as well as script or configuration languages for setting up applications. The execution may be monitored via graphical interfaces, which can be customised for different applications. These tool kits, although powerful, are not complete. Most of them are not portable and do not offer the required flexibility to represent different problems without major programming effort. Parallel implementations are even more constrained; with the majority of them being restricted to specific platforms. The characteristics, features, strengths and weaknesses observed in all these systems provided the insights for the design of a GA programming environment, which is the main subject of this thesis.

Chapter 4

The GAME System

This chapter presents an overview of the Genetic Algorithms Manipulation Environment. It briefly introduces the system's genetic-oriented abstractions, the programming model, and main modules. It also gives a short description of the PAPAGENA project, which has GAME as its principal development tool.

4.1. Introduction

The characteristics and requirements of GAs and PGAs outlined in Chapter 2, and the various common features of existing programming environments presented in the previous chapter, provided the grounds for the creation of the GA programming environment described in this chapter. The Genetic Algorithms Manipulation Environment (GAME) is also the result of a joint European research project, which brought together university and industry for the development of complex PGA applications.

This chapter presents an overview of the whole programming environment and briefly describes its main modules. Being a complex and sophisticated programming environment, the design and implementation of the various modules of GAME was carried out by a team of three researchers at UCL. This thesis focus on the parts of the system which were the responsibility of the author of this dissertation and comprised:

- the definition of the modular architecture of the system;
- the definition and implementation of GAME's genetic-oriented and problem independent data structures;
- the design and implementation of GAME's genetic manipulation engine: the Virtual Machine;
- the specification of a programming model that enables GAME applications to be dynamically configured;

- the design and implementation of the communication and parallel control module that allows the development of portable sequential and parallel applications;
- and the specification of a hierarchically organised set of libraries to maintain parameterised versions of genetic algorithms and operators.

The next section starts with an overview of GAME's object-oriented architecture. It is followed by the introduction of its representation abstractions and programming model. Then, a summary description of each main module is presented. The chapter ends with a brief overview of the PAPAGENA project, stressing the importance of GAME for the development of real-world applications.

4.2. GAME Architecture

The Genetic Algorithms Manipulation Environment provides a unified framework that makes the development of complex and sophisticated sequential and parallel genetic algorithms applications extremely easy. The design of GAME was mainly driven by the requirements and characteristics observed in a large number of GAs and PGAs. It is also the result of a combination of common features encountered in the majority of general-purpose programming environments. In addition, GAME's innovative object-oriented approach and platform-independent parallel programming model makes the system unique in its class.

GAME addresses all the basic requirements involved in the design cycle of a GA application. It offers problem-independent genetic-oriented data structures, comprehensive programming interfaces and a set of libraries with parameterised versions of a broad range of GAs and PGAs. Its underlying infrastructure provides mechanisms for the manipulation of genetic data structures, simulation monitoring and application execution on a virtual computing environment supporting multiple parallel computation models. GAME is highly customisable and its libraries can be easily expanded with the inclusion of new parameterised modules.

Users can interact with GAME at three distinct levels: application prototyping, module configuration/creation and system customisation. At the *prototyping* level, novice and non-expert users can rapidly configure and execute an application by simply setting up a few parameters in a configuration file. At the *module configuration/creation* level, programmers can combine pre-defined modules from the algorithm and genetic operator libraries to create new applications. Moreover, entirely new modules written from scratch or by modification of source code examples, can be easily integrated into the environment. Finally, the *system customisation* level allows programmers to modify internal modules of the environment, such as the Virtual Machine or the graphic user interface. The interaction at this level is particularly important for porting GAME to hardware or operating system platforms not supported in its original distribution. It is also at the system customisation level that GAME's functionality can be adapted or expanded to suit diverse application requirements. GAME's standard modules, such as the Virtual Machine, can be replaced easily by customised versions thereby providing extra functionality.

4.2.1. Genetic Representation

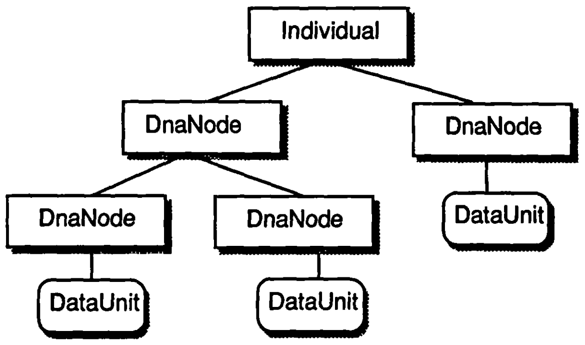
One of the key features of a GA general-purpose programming environment is the ability to support the representation of a variety of problems. The traditional GA requires a problem to be represented as a single string (normally using the binary alphabet). Although effective and easy to manipulate, this representation model restricts the construction of genetic algorithms and operators in many ways, and generally results in problem-dependent implementations.

GAME provides problem-independent representation through a set of abstractions that help with the description of a broad range of genetic data structures. It is possible to represent genetic data structures as simple as ordinary binary strings, or as complex as parse trees of any depth or width (depicted in Figure 4.1), as required in genetic programming applications. GAME's principal genetic-oriented abstraction, the *DNA*, is a *tree node,* which has the ability to store information. It is the result of the combination of two primary objects defined by the system:

- DataUnits and
- DnaNodes

The DataUnit is defined as a class of objects that is capable of storing "genetic information" – the problem's data. Currently, GAME provides DataUnit objects to store most of the native *C++* data types (char, int, long and double) and a special type that stores binary encoded strings. DataUnit types are fully integrated with native data types (internal conversion operators and assignment operator overloading provide transparent conversions between types).

Figure 4.1 - A genetic-oriented representation



The DnaNode object has the ability to "connect" several DnaNode objects. It can also operate as a *container* for one or more DataUnit objects, transforming this particular configuration into a *DNA* object. A DnaNode may connect only to one "parent" node, which in turn, may contain an arbitrary number of other DnaNode objects, configuring a tree data structure. A special class derived from the DnaNode, the Individual, sits on the root of the tree structure and represents a single candidate solution to a given problem. Therefore, any genetic representation in GAME must be formed by an Individual object connected to one or more DnaNode objects (depending on the number of variables, or chromosomes, defined by the problem); the latter containing at least one DataUnit object (a gene). The DnaNode class provides a set of member functions related to node-level operations such as *attachNode*, *detachNode*, *copyNode* and *deleteNode*. It also provides operations over the DataUnit objects it stores such as *readData*, *writeData*, *modifyData*, *copyData* and *deleteData*. The combination of the functionality provided by the DnaNode and DataUnit objects gives an enormous flexibility for describing complex genetic structures containing mixed data types at any level of the tree structure.

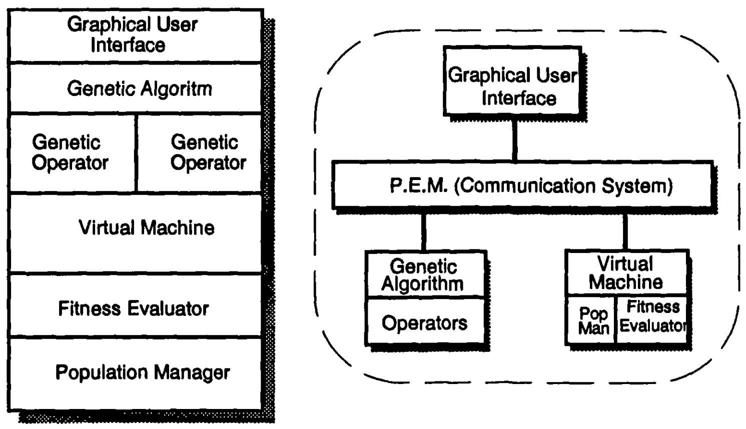
4.2.2. Programming Model

A GAME application is a computer program that co-ordinates the operation of execution units, or active objects, defined as *GAME Components*. Any GAME Component object must be derived from the GameComponent class. This class provides the basic functionality for concurrent execution and inter-process communication. It allows GAME Component objects to execute as independent processes, or active objects, similar to the ACTOR [1] model. GAME's programming model presents the following features:

- GAME applications consist of a collection of GAME Components – typically the application front-end (with the graphic user interface), one or more algorithms (with its operators) and the Virtual Machine – executing sequentially, concurrently or in parallel.
- Components may share the same memory space, the same processor or execute on a different processor or machine. These three options can coexist in the same application (see Figure 4.2), and the actual distribution is dynamically controlled by the components themselves.
- GAME Components communicate asynchronously or synchronously via message passing. Components have the ability to buffer messages in their mailboxes. Messages are queued and collected from a mailbox sequentially, in a first-in-first-out order.
- Messages are objects derived from the MessagePackage class and contain specific commands and parameters for GAME Components. Messages may also be replied with status and other information resulting from the command, on its completion.

- MessagePackage objects can only carry GAME objects derived from the GameStreamObject class. Objects derived from this class have the ability to be represented as a data stream that contains the relevant information to produce exact copies, or clones, of the original object.
- A GAME application must have, at least, an *ApplicationComponent* object, which generally implements the user interface. Application components create one or more *AlgorithmComponent* objects that, in turn, create *OperatorComponent* objects.

Figure 4.2 - Sequential and parallel GAME applications

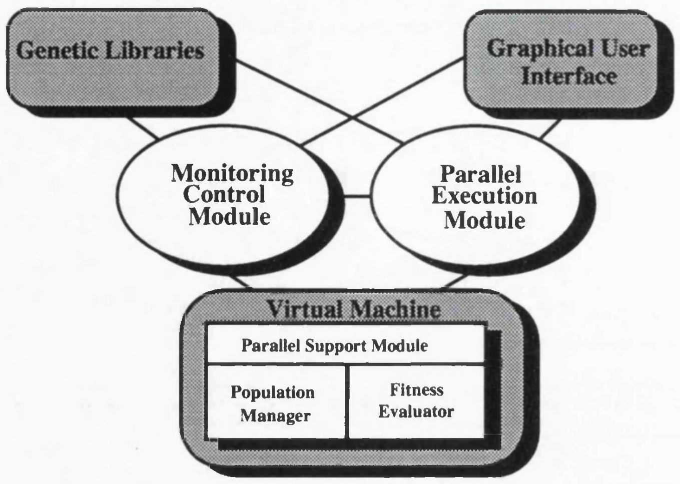


Application components may also create a *MonitorComponent* object, whereas algorithm components typically create a *VirtualMachine* object (also a GAME component) to execute genetic manipulation commands generated by the algorithm itself and its operators.

4.3. GAME Modules

The GAME system is a collection of libraries organised in two major groups: the *Service Libraries* and the *Genetic Libraries*. The Service Libraries contain the modules that form the core of the environment: the Virtual Machine, the Parallel Execution Module, the Monitoring Control Module and the Graphic User Interface. They also contain a collection of classes that implement the genetic-oriented data structures, exception handling, etc. Figure 4.3 presents an overview of GAME's architecture with its main modules.

Figure 4.3 - GAME's modular architecture



The modules in white (Monitoring Control Module, Parallel Execution Module, VM Parallel Support, Population Manager and Fitness Evaluator) represent the parts of the environment runtime libraries that are common to all applications. These modules are not "seem" by the user, being embedded into all applications through various GAME Component classes. Conversely, the Genetic Libraries and the Graphical User Interface (in gray) are modules that contain components which the user may modify, via the application configuration file. These modules are then said to be customisable. The other module represented in gray is the Virtual Machine, which besides being a customisable module (the user can configure the number of PMs and FEs, for instance) also represents an abstraction comprising hidden and more specialised GAME Components, such as the Population Manager and the Fitness Evaluator.

The connections demonstrate the dependence relationship between modules. It can be seen that the two most important modules are the MCM and the PEM, the latter being also used by the former. Being the core of the system, PEM is used for the most basic activity in the application which is communication between its components.

The modular structure adopted for GAME is based on the design of other programming environments, such as the Pygmalion and the Galatea Neural Networks programming environments, also developed at UCL. However, various new concepts and ideas have been introduced and modified to adapt their original design to an object-oriented approach.

4.3.1. Virtual Machine

The Virtual Machine (VM) is the module responsible for maintaining the genetic data structures and providing facilities for their manipulation and evaluation. It isolates the genetic operators and algorithms from dealing directly with the data structures through a set of commands, implemented as a collection of functions, the VM Application Program Interface (VM-API). The Virtual Machine is also capable of providing a certain degree of parallelism, in which several commands can be executed simultaneously, providing parallelism is supported by the host platform. The VM comprises three sub-modules: the Population Manager, the Fitness Evaluator and the Parallel Support Module. The Population Manager maintains the genetic-oriented data structures organised in population pools, and executes genetic manipulation commands over them. The Fitness Evaluator performs the actual evaluation of the genetic data structures and related calculations such as *total, average, highest* and *lowest* fitness. Finally, the parallel support module distributes commands received by the Virtual Machine among (possibly) several copies of the Population Manager and Fitness Evaluator modules.

The VM-API includes commands for creation, elimination, duplication, partial swapping, inversion and modification of genetic data structures. Fitness-related commands permit the computation of each individual's fitness as well as the determination of populations' total and average fitness. The highest and lowest fitness values of a population can also be requested via the API.

4.3.2. Parallel Execution Module

The Parallel Execution Module (PEM) implements a hardware/operating system independent interface that supports multiple parallel computation models. It also provides an API with functions for process initiation, termination, synchronisation and communication. It is responsible for integrating all the GAME Component objects that form an application, with even sequential components relying upon PEM to transport their messages to other GAME components. PEM comprises two layers: (i) the *upper-layer* defines the standard interface functions used by all GAME components of an application; (ii) the *lower-layer* implements the functions that map upper-layer requests into specific platform-dependent support for process control and communication. The design of PEM facilitates its porting to a variety of platforms (with scalar or parallel architectures), by simply replacing its lower layer. Applications created with GAME are then automatically portable across all platforms containing an implementation of PEM.

4.3.3. Genetic Libraries

The Genetic Libraries comprise a collection of modules containing pre-defined and parameterised applications, genetic algorithms and genetic operators. These libraries are hierarchically organised, that is, modules of the Application Library are constructed with modules of the Genetic Algorithm Library that, in turn, are built from modules of the Genetic Operators

Library. New applications and algorithms can be created by simply listing the required modules and their parameters in a configuration file.

4.3.4. Monitoring Control Module

The Monitoring Control Module (MCM) collects and displays – through a graphic user interface – events that happen during a simulation session. GAME Components may be requested to notify the MCM about received or transmitted messages, as well as any modification of DataUnit objects they possess [30]. The level of monitoring can be selected by the user for each GAME Component object in an application. The MCM also has the ability to inform other components of the occurrence of particular events, by keeping internal "lists of interests".

4.3.5. Graphical User Interface

The Graphical User Interface module contains simple graphic widgets that can be used to compose an application front-end. It enables applications to input and output data in a variety of formats. This module includes standard widgets for displaying texts, graphics, dialogue boxes, buttons and charts. It allows different widgets to be associated with various events reported by the monitoring module. The modular design of the graphical user interface permits the integration of the system's basic library with commercial cross-platform or "home-grown" graphic libraries.

4.4. PAPAGENA Applications

The Genetic Algorithms Manipulation Environment was conceived as the central part of the principal parallel GA project funded by the European Commission. The ESPRIT III Programming Environment for Applications of PArallel GENetic Algorithms (PAPAGENA) aimed at disseminating the use of parallel genetic algorithms in complex optimisation and modelling problems. The PAPAGENA project involved many partners including private companies, universities and research centres from Germany, England, Holland and France. Three applications have been developed in the project in different domains namely: finance, bio-informatics and economic modelling.

The financial application, developed by CAP Volmac and KIQ, provides predictive systems to assist financial organisations to optimise their decisions in fields such as credit scoring, insurance risk, or marketing expenditure[42]. Genetic programming is used to construct an algebraic formula that can regenerate, and hopefully predict, a series of training values. Populations of candidate formulas created by the GP are scored on the basis of how well they fit the training set and their ability to predict a validation set. This type of problem evolves highly complex genetic structures represented as parse trees using GAME's genetic-oriented abstractions. In this context, the algorithm must be able to manipulate operators as well as a large number of possible variables. The operators relate to the functions used to construct the algebraic formulae, whereas the variables relate to the possible data fields (e.g., number of credit cards, house price, etc.). Biological-like operators are then defined to directly manipulate the two possible data types (operators and variables). For example, the crossover operator swaps sub-trees at nodes of equivalent data types. Similarly, two distinct forms of mutation are defined to be applied over variables and algebraic operators. Anticipating the system's usage as a financial modelling tool, research concentrates on inducing algebraic formulae from sets of noisy, possibly incomplete, and even contradictory real-world data.

In the bio-informatics domain, an application has been developed at Brainware GmbH to predict stable protein conformations. This problem has the potential to open up a vast new world of drug design and medical treatments. Parallel GAs are being applied to search energetically and structurally favourable protein conformations. Parallelism is fundamental to this application due to the amount of data to be processed for each GA generation. It is exploited in three principal modules, namely: transformation, evaluation and recombination [74]. The transformation module converts proteins' descriptions between polar and Cartesian co-ordinates. Cartesian co-ordinates are commonly used to describe the spatial organisation of protein molecules. However, genetic manipulations are easier to implement using polar co-ordinates. The transformation module was then introduced in the application to accommodate both requirements. Each protein in the GA population is described using polar co-ordinates, which are then converted to Cartesian co-ordinates before undergoing fitness evaluation. The evaluation module contains the objective function that computes the stability of protein conformations. Protein evaluations are extremely time consuming, requiring the use of a large data base of molecules. The data base provides the important characteristics of molecules that are used by the objective function to work out stability of a particular protein conformation. Finally, the recombination module implements the genetic algorithm, which creates new protein conformations by recombining and modifying molecules and spatial organisations of existing proteins. The degree of parallelism required in this application is achieved with the implementation of these modules as GAME Components, and distributing many instances of them among several processors.

The economic modelling application, also developed by Brainware in collaboration with IfP, is targeted at simulating a variety of possible scenarios associated with the current economic changes within Eastern Europe [89]. In this case, GAs are used to mimic the behaviour of complex multi-agent systems, subject to a variety of economical and physical constraints. In essence, an artificial economy is created within the computer, modelled in terms of traditional economic theory, evolution, and principle-based engineering [88]. The application uses three GA components running in parallel to evolve three separate "models" of economic agents: the

Labour-Market, the *Enterprise* and the *Locational* models. Each model is represented by distinct genetic structures using GAME's genetic-oriented abstractions. The results of each module's generation are analysed by the *Global Economic* module that tries to find the best combination of requirements and features of model.

This application is expected to help national and local governments by providing a knowledge basis to assist in the formulation and implementation of effective strategies in many sectors, e.g., investment, industrial location, logistics, etc. It has already been adopted by the Brandenburg State in Germany, as a means for modelling and understanding local labour movements, which have risen considerably since the German unification.

Other PAPAGENA partners were TELMAT Informatique from France and the German National Research Center for Computer Science (GMD). TELMAT was responsible for porting GAME onto their transputer-based parallel machines, whereas GMD provided the theoretical foundations and research support for the development of parallel genetic algorithms' applications.

4.5. Summary

This chapter presented an overview of the GAME programming environment. It introduced the system's genetic-oriented abstractions for the representation of diverse problems and a programming model that helps with the creation of portable sequential and parallel applications. GAME's modular architecture was described, followed by a brief presentation of the system's five main modules. The design and implementation of three of GAME's modules – the Virtual Machine, the Parallel Execution Module and the Service and Genetic libraries – that constitute the main subject of the research reported in this thesis, are described in more detail in the next chapters. Finally, a short description of the PAPAGENA project was presented to highlight the importance of the GAME system in the context of a European project, aimed at solving real-world problems.

Chapter 5

The Genetic-Oriented Representation and the Virtual Machine

This chapter reports on the design and implementation of GAME's Virtual Machine module. It starts by introducing the abstractions and objects that grants GAME the ability to "genetically" represent a broad range of problems. The Virtual Machine, its modules – the Population Manager, the Fitness Evaluator and the Parallel Support – and the VM Application Program Interface are then described.

5.1. Overview

The description of GAME's genetic-oriented abstractions for problem representation and the Virtual Machine module presented in this chapter focuses more on their design than on their implementation aspects. The objective is to give sufficient information about their design to allow other implementations, possibly using even a different programming language. Nevertheless, *C++* class declarations are provided, along with the description of their most important member functions and data.

This chapter starts by explaining the importance of a problem's representation and how GAME facilitates their manipulation via its genetic-oriented abstractions. The following sections show how data of different types are stored, and how the representation structure is organised. One of the sections discusses the problem of addressing large and deep storage units in the representation and presents the solution adopted.

The modular design of the Virtual Machine is then presented. The VM comprises three modules: the Population Manager, the Fitness Evaluator and the Parallel Support. The Population Manager (PM) is responsible for the execution of genetic manipulation commands; the Fitness Evaluator (FE) embeds the problem-dependent objective function and performs related computation (total, average, etc.); and the Parallel Support module controls the execution of many PM and FE instances on parallel platforms. The VM Application Program Interface, its commands and communication objects – VmMsg – that transport them are also described.

5.2. Representing Genetic Data Structures in GAME

One of the most important characteristics of a genetic algorithm is the *genetic data structure* it manipulates. In its simplest form, a GA operates over an encoded string representing a single problem variable (*x*), as shown in Figure 5.1. However, a single variable search is clearly restrictive in terms of the number of problems that can be expressed in this manner. Complex real-world problems generally require optimisation on multi-dimensional spaces. A straightforward way to support these problems would be to simply extend the one-dimensional case by extending the string definition.

Figure 5.1 - A one-dimensional string representation

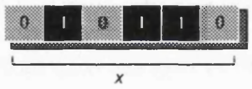
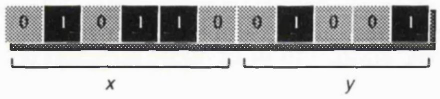


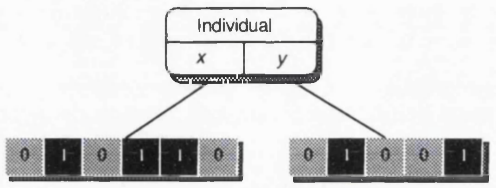
Figure 5.2 depicts the traditional genetic representation for a two-dimensional search, with the variables *x* and *y* encoded in the same alphabet (binary) and concatenated into a single string. Any manipulation of this structure would be an exploration of possible two-dimensional solutions to a given problem.

Figure 5.2 - Concatenating strings on multi-dimensional problems



This two-dimensional representation can clearly be extended to an arbitrary number of dimensions simply by increasing the number of bits in the string. However, in most problem instances, certain practical considerations have to be taken into account. These considerations must include, for instance, the way strings (for all co-ordinates) are interpreted, and the possible restrictions to the range each variable can take. For example, the *x* value could range over the usual binary decoding of 0 to 63, whereas the *y* variable could take values between 97.5 and 99.0. This means that the genetic operators would have to take into account constraints, or restrictions, placed on the interpretations of these chromosomes to prevent invalid strings appearing in the population. A more intuitive approach would be to separate the variables onto different chromosomes. This allows genetic operators to handle constraints more easily. However, it also means that an extra level of abstraction is introduced in the genetic structure in that an *individual* (as a population element) now consists of a set of chromosomes. The full hierarchy is then: a group of genes forms a chromosome; a group of chromosomes forms an individual; a group of individuals forms a population. The individual exists only as an abstraction to facilitate the manipulation of the sets of variables representing a single potential solution to a problem. The new data structure is shown in Figure 5.3.

Figure 5.3 - A bi-dimensional genetic representation



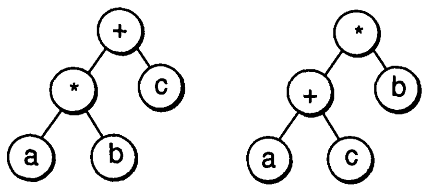
Once the genetic structure has been extended in this way it becomes possible to manipulate it at multiple levels. For instance, the crossover operator can exchange genetic information at the gene level, i.e., sub-strings of the genetic representation of the x parameter. At the chromosome level, crossover could swap an entire parameter (x) while keeping the other (y) fixed. This means that chromosomes can be treated and defined in fundamentally different ways. For example, one chromosome may take a binary representation whereas the other might take real values, or some other discrete encoding. This clearly broadens the scope of the possible GA designs. Another immediate possibility is an overall increase in the number of levels that may be defined for a genetic structure. This allows the representation of even more complex genetic data structures, such as those defined in genetic programming problems.

Genetic programming evolves genetic structures that are themselves computer programs. The objective, besides minimising the number of language primitives and operators applied to implement a task, is to automate program construction. By knowing the possible inputs for a program and the expected outputs, a genetic programming application can produce the required program. The genetic structure manipulated in such cases is a parse tree (usually created by compilers). The tree model is then an extension to the individual/chromosome/gene model, but with an arbitrary number of levels. The genetic operators should be able to manipulate the genetic structure at different levels, not only exchanging sub-trees but aggregating or separating them as well. An example of a simple GP genetic data structure is shown in Figure 5.4, which represents the mutation of the expression *(a*b)+c* which becomes *(a+c)*b* after swapping the branch *+c* with the branch **b* of the parse tree.

This form of genetic representation provides a much more flexible alternative to the traditional single string model. It allows the coexistence of different data types in the same genetic structure and permits genetic manipulations at various levels (chromosomes, genes, etc.).

Furthermore, it simplifies the construction of genetic operators by transferring to the representation structure any possible problem constraint.

Figure 5.4 - A GP tree mutation



5.2.1. GAME's Genetic-Oriented Data Types

The GAME tool kit defines the genetic structures representing a problem in terms of genetic oriented abstractions that conform with the representation model outlined above. The genetic oriented abstractions provided by GAME allow the representation of a variety of ES, GA and GP problems. GAME supports the use of binary (or any other alphabet) and real value representations with an arbitrary number of levels, in a tree-like data structure. The implementation of the genetic-oriented data structures is based on a set of internally defined data types. GAME's data types have the same functionality as the native data types defined by the *C++* language, with some added features that enable data objects to be sent across GAME Component objects. Another feature of GAME data types is the ability to enforce upper and lower limits for the actual values they can store. If no limit is specified for a GAME data type, it assumes the range of possible values of its native language counterpart. For example, the GAME data type used to represent integers, defined as *ga_int*, is able to store the same range of values of a *C++* `int` type. However, if a *ga_int* object is created, as in the example bellow, it allows only values between `-20` and `+20` to be stored. Any other value falling out of that range is automatically "mapped" into a value in the specified range.

Example:

```
ga_int  x(-20,20);  // x only stores values between -20 and 20
```

All GAME data types belong to the DataUnit class. This class defines an abstract type that contains the properties of a GameStreamObject[4] and the basic arithmetic and relational operators supported by the *C++* language. The derived classes contain the variables holding the actual value, as well as the maximum and minimum limits it may assume. For each native language type a corresponding GAME data type is defined. The use of internal *conversion operators* provides interchangeability between native data type values and GAME data type values. The current set of data types may also be easily expanded to support user-defined data

[4] GameStreamObjects provide the functionality to transfer objects like DataUnits across GAME Components.

types. Complex data structures may be encapsulated by classes derived from DataUnit, extending the data types supported by GAME for a particular application. Figure 5.5 lists the declaration of the DataUnit class, and Figure 5.6 shows an example of a GAME data type – the *ga_int* class.

Figure 5.5 - The DataUnit base class

```
class DataUnit : public GameStreamObject
{
public:

//
// Member functions defined by GameStreamObjects
//
  virtual OLT          getObjectLength       (void) = 0;
  virtual hOS          describeObject        (hOS)  = 0;
  virtual hOS          assembleObject        (hOS)  = 0;
  virtual DataUnit*    duplicate             (void)= 0;
  virtual DataUnit&    operator=             (const DataUnit&) = 0;
//
// Aritmetic operators
//
  virtual DataUnit&    operator+=            (const DataUnit&) = 0;
  virtual DataUnit&    operator-=            (const DataUnit&) = 0;
  virtual DataUnit&    operator*=            (const DataUnit&) = 0;
  virtual DataUnit&    operator/=            (const DataUnit&) = 0;
  virtual DataUnit&    operator%=            (const DataUnit&) = 0;
//
// Relational operators
//
  virtual BOOL         operator==            (const DataUnit&) = 0;
  virtual BOOL         operator!=            (const DataUnit&) = 0;
  virtual BOOL         operator>=            (const DataUnit&) = 0;
  virtual BOOL         operator<=            (const DataUnit&) = 0;
  virtual BOOL         operator>             (const DataUnit&) = 0;
  virtual BOOL         operator<             (const DataUnit&) = 0;
};
```

Besides the native *C++* equivalent data types, GAME defines two extra data types: *ga_word* and *ga_binary*. The *ga_word* type is intended to act as a universal type, being able to hold any value that can be stored by any native language data type. It gives an enormous flexibility to the application programmer. But, because it can accommodate virtually any value, its instances may occupy a significant amount of memory. The other data type, *ga_binary*, stores only binary values. It saves memory since all values are stored in a single array containing only the sufficient number of bytes to store the required number of bits. Table 5.1 lists GAME's data types and their native *C++* counterparts.

Table 5.1 - GAME data types

Native C++ Data Types	*GAME Data Types*
unsigned char	ga_uchar
unsigned short	ga_ushort
unsigned int	ga_uint
unsigned long	ga_ulong
char	ga_char
short	ga_short
int	ga_int
long	ga_long
double	ga_double
float	ga_float
▨	ga_word
▨	ga_binary

Figure 5.6 - A GAME data class (ga_int)

```
class ga_int : public DataUnit
{
public:
                        ga_int              (void);
                        ga_int              (const int&, int=0, int=0);
                        ga_int              (const ga_int&);
  virtual               ~ga_int             (void);

  virtual OLT           getObjectLength     (void);
  virtual hOS           describeObject      (hOS);
  virtual hOS           assembleObject      (hOS);

  virtual DataUnit&     operator=           (const DataUnit&);
  virtual hDUT          duplicate           (void);

  virtual void          setRange            (int, int);
  virtual int           getMaxValue         (void);
  virtual int           getMinValue         (void);

  virtual DataUnit&     operator+=          (const DataUnit&);
  virtual DataUnit&     operator-=          (const DataUnit&);
  virtual DataUnit&     operator*=          (const DataUnit&);
  virtual DataUnit&     operator/=          (const DataUnit&);
  virtual DataUnit&     operator%=          (const DataUnit&);

  virtual DataUnit&     operator+=          (const int&);
  virtual DataUnit&     operator-=          (const int&);
  virtual DataUnit&     operator*=          (const int&);
  virtual DataUnit&     operator/=          (const int&);
  virtual DataUnit&     operator%=          (const int&);

  virtual BOOL          operator>           (const DataUnit&);
  virtual BOOL          operator<           (const DataUnit&);
  virtual BOOL          operator==          (const DataUnit&);
  virtual BOOL          operator!=          (const DataUnit&);
  virtual BOOL          operator>=          (const DataUnit&);
  virtual BOOL          operator<=          (const DataUnit&);

  virtual ga_int&       operator=           (const int&);
  virtual ga_int        operator+           (const ga_int&);
  virtual ga_int        operator-           (const ga_int&);
  virtual ga_int        operator*           (const ga_int&);
  virtual ga_int        operator/           (const ga_int&);
  virtual ga_int        operator%           (const ga_int&);
//
// Conversion operator
//
  virtual               operator            int();
//
// Member Data
//
private:

          int           _current_value;
          int           _min_value;
          int           _max_value;
  unsigned int          _mask;
};
```

GAME defines two abstractions that are used to describe the genetic structures to be manipulated by the Virtual Machine. They are implemented by the *Individual* and the *DnaNode* classes. GAME implements genetic representations as tree structures containing an *Individual* object in its root, and an arbitrary number of layered branches (or nodes). Each layer under the root is formed by a number of *DnaNode* objects acting as "connectors" between two layers (see Figure 5.7). A DnaNode object can be connected to an Individual object or another DnaNode object. In fact, both Individual and DnaNode objects are members of a common base class, the *DnaCollection*, which defines their basic properties.

Besides acting as "connectors" between nodes, a DnaNode (or Individual) object may also hold DataUnit objects. In general, DataUnit objects are present in the "leaves" of the tree structure, but they can also be attached to any DnaNode object, at any level.

Figure 5.7 - Simple and complex genetic representations in GAME

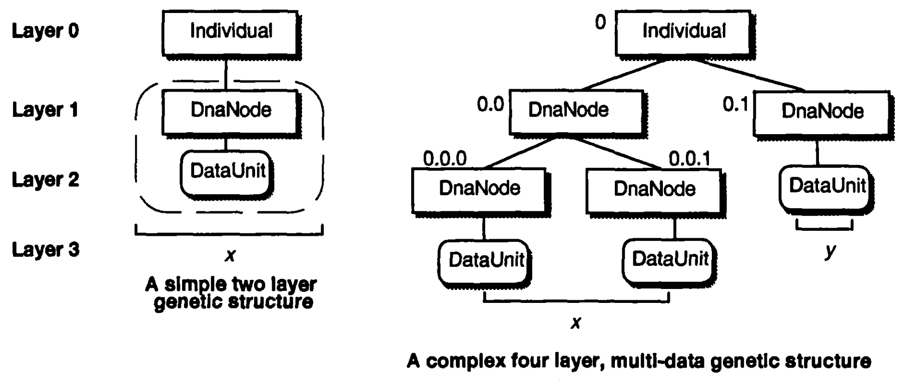


Figure 5.7 shows two examples of GAME's genetic structures. The first genetic structure is the simplest possible in GAME, and is represented by an Individual object that has a single DnaNode object containing a single DataUnit object. This structure corresponds to a problem with only one variable (x).

The second example shows a complex genetic structure that can be described using the genetic-oriented abstractions. The two nodes under the root (Individual) represent two independent variables (x, y) of a problem. It is worth to noting that x and y may use different encoding methods.

Addressing a DnaNode

Nodes in the genetic structure are identified by their addresses. The address of the Individual object (0 in this example) corresponds to its index in the population pool. Sub-node addresses result from the composition of previous nodes' addresses, appended with the index value relative to the node they are connected to. Each node of any Individual object in a population pool is uniquely identified by its *nodepath* address. Thus in the example of Figure 5.7, the x variable (represented by node 0.0) is a chromosome "carrying" two genes (0.0.0 and 0.0.1) whereas the variable y (node 0.1) is represented by a single gene.

Genetic manipulations can occur at any node, meaning that complete flexibility is offered by GAME's genetic structure for operating on genetic types. DnaNodes and DataUnits may be *attached* and *detached*, *duplicated*, *moved* and *deleted* at any node, in any layer. Operations like *gene swaps* and *chromosome swaps* (the whole gene string is swapped at once between two Individuals) are as simple as detaching and re-attaching nodes in the appropriate layers of the genetic structure.

Specifying a particular node in the genetic structure to be swapped with another node, for instance, can be a very complicated operation. Since the genetic structure can be as complex as a problem may require, it could involve a number of nodes and an even greater number of layers for a single genetic structure describing an individual. A *nodepath* is defined as the sequence of indexing items that uniquely specifies the address of a node in a genetic structure. Its first indexing item is the pool handler (`hPOOL`) which indicates the population pool that a particular genetic structure belongs to. It is followed by the individual index in that pool and a sequence of indexing items that identifies the targeted node.

Example:

> Starting from the end, the sequence `20.5.0.1.2` gives the address of the third node on the fourth layer, connected to the second node of the third layer, which is connected to the first node of the second layer of the 5th individual from the pool (20 is the pool handler which is only an identifier for the pool itself, not implying the existence of twenty pools).

The sequence necessary to address a particular node can become quite long as the number of nodes in a genetic structure increases. Also, an addressing sequence may not have a fixed length since GAME permits its genetic structures to be dynamically re-sized. These characteristics make the use of *C++*'s ordinary function call syntax at least uncomfortable. The solution to this problem came with the creation of an "addressing object", the NodePath.

The NodePath class

A NodePath operates as a *fifo* (first-in-first-out) that stores the sequence of addressing items necessary to address a particular node in the genetic structure. The maximum length of the path that can be stored may be specified when a NodePath object is declared. The NodePath class inherits the functionality of the GenericFifo and GenericSet classes, which defines a family of container classes in GAME. It is also a member of the GameStreamObject class (as any member of the GenericSet class) and, therefore, can be carried by message package objects across GAME Components.

A NodePath object is included in any message package delivered to a DnaCollection mailbox. Its contents are then used to "navigate" the message package on the genetic structure until it is finally delivered to the last node specified in the addressing sequence. As the message package navigates throughout the genetic structure, addressing items are extracted from the NodePath object in the same sequence they were inserted. By extracting an address item from the

NodePath object, a node can route the message package to the mailbox of the next node in the addressing sequence. This operation is repeated until no more addressing items are found in the NodePath object, implying that the contents of the message package (a command) must be executed.

The NodePath class shown in Figure 5.8 allows a sequence of bytes, representing nodes' indexes, to be stored into its internal array. The sequence of numbers is inserted using the "=" and "+=" operators. Addressing values are extracted in the same sequence they had been inserted, via the `getNextAddress` member function. The `isLastAddress` member function indicates when no more addressing items are available.

The NodePath class is derived from the abstract type GenericFifo, which defines the functionality of a GAME fifo object.

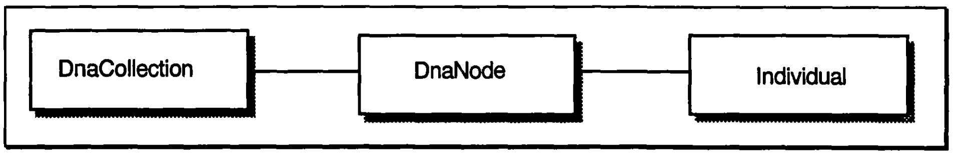
Figure 5.8 - The NodePath class

```
class NodePath : public GenericFifo
{
public:
                           NodePath          (void);
                           NodePath          (WORD); // specify the array size
                           NodePath          (const NodePath&);
  virtual                  ~NodePath         (void);

  virtual OLT              getObjectLength   (void);
  virtual hOS              describeObject    (hOS);
  virtual hOS              assembleObject    (hOS);
  virtual GenericFifo&     operator=         (const GenericFifo&);

  virtual void             clearPath         (void);
  virtual BYTE             getNextAddress    (void);
  virtual BOOL             isLastAddress     (void);
  virtual BYTE             getMaxPathLength  (void);

  virtual void             operator=         (BYTE);
  virtual void             operator+=        (BYTE);
  virtual                  operator          BYTE();
};
```

5.2.2. The Genetic type class hierarchy

The genetic-oriented abstractions used to describe the data structures that represent a particular problem are rooted by the DnaCollection class. As shown in Figure 5.9, a DnaCollection is the base class for the DnaNode and the Individual classes, the latter being a specialised version of the DnaNode class.

Figure 5.9 - Class hierarchy of GAME's genetic-oriented objects



5.2.3. The DnaCollection

The DnaCollection class shown in Figure 5.10, provides the basic functionality for the manipulation of GAME's genetic-oriented data structures. Amongst its most important features is the ability to maintain connections to an unlimited number of other DnaCollection (and derived) objects, and hold many DataUnit objects simultaneously. The DnaCollection class contains the following member data:

```
WORD  _num_nodes
BYTE  _num_variables
```

These are counters holding the current number of DnaCollection type objects connected to this object, as well as the number of DataUnit type objects.

```
WORD  _max_nodes
WORD  _max_variables
```

These two member data are used to store the maximum number of DnaCollection and DataUnit objects that can be connected to this object. These values may be provided when a DnaNode (or Individual) object is declared, via the constructor.

Example:

```
DnaNode n(10,1);        // creates a DnaNode object "n" which can
                        // receive up to 10 connections of
                        // other DnaCollection type objects and
                        // 1 connection of a DataUnit type object.
```

If one (or both) of values in the example are not provided when the object is created, five connections of DnaCollection and one connection of a DataUnit type are assumed by default. The default values may be modified by defining the constants NODE_ARRAY_SIZE and DATA_ARRAY_SIZE in the gameconf.h file.

```
DNA_NODE_STATUS    _status
```

This member data stores the status word of a DnaCollection object. Only two possible status words can be assumed: ATTACHED or DETACHED. They refer to the actual condition of the object, that is, if the object is connected to any other DnaCollection object, its status is ATTACHED otherwise DETACHED. It is used as a mechanism to prevent DnaCollection objects from being connected to more than one genetic structure at the same time.

Figure 5.10 - The DnaCollection class

```
class DnaCollection : public GameStreamObject
{
public:
                        DnaCollection         (WORD, WORD);
                        DnaCollection         (const DnaCollection&);
  virtual               ~DnaCollection        (void);

  virtual OLT           getObjectLength       (void) = 0;
  virtual hOS           describeObject        (hOS)  = 0;
  virtual hOS           assembleObject        (hOS)  = 0;

  virtual DnaCollection& operator=            (const DnaCollection&);
//
// Auxiliary Functions
//
  virtual hDCT          duplicate             (void) = 0;
  virtual MsgPackage&   mailBox               (MsgPackage&);
//
// Status Functions
//
  virtual DNA_NODE_STATUS getNodeStatus       (void);
  virtual void          setNodeStatus         (DNA_NODE_STATUS);
  virtual BOOL          getDataStatus         (BYTE=0);
  virtual MSG_STATUS    setDataStatus         (BOOL, BYTE=0);
//
// Manipulation Functions
//
  virtual MSG_STATUS    attachNode            (WORD, hDCT);
  virtual hDCT          detachNode            (WORD);
  virtual MSG_STATUS    makeNode              (WORD, WORD, BYTE) = 0;
  virtual hDCT          duplicateNode         (WORD);
  virtual MSG_STATUS    deleteNode            (WORD);
  virtual MSG_STATUS    invertNodes           (WORD, WORD=0);
//
// Access Functions
//
  virtual hDUT          readData              (BYTE=0);
  virtual MSG_STATUS    writeData             (hDUT, BYTE=0);
  virtual hDUT          duplicateData         (BYTE=0);
  virtual MSG_STATUS    deleteData            (BYTE=0);
  virtual WORD          getNumNodes           (BYTE=ONE_LEVEL);
  virtual WORD          getMaxNodes           (void);
  virtual BYTE          getNumVariables       (void);
  virtual BYTE          getMaxVariables       (void);

protected:
          void          resetFlags            (LWORD&);
//
// Member Data:
//
          WORD                 _num_nodes;
          WORD                 _max_nodes;
          BYTE                 _num_variables;
          BYTE                 _max_variables;
          LWORD                _valid_data;
          DNA_NODE_STATUS      _status;
          DnaCollection**      _node_aray_ptr;
          DataUnit**           _data_array_ptr;
};
```

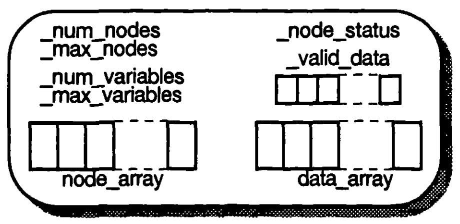
```
DnaCollection**  _node_array_ptr
DataUnit**       _data_array_ptr
```

The list of DnaCollection objects connected to a particular DnaNode or Individual is maintained by an array of handlers (hDCT) created when the object is declared. Since the array is dynamically created (with the specified or default size) only its pointer – *_node_array_ptr* – is declared in the DnaCollection class. The same strategy applies for DataUnit objects connected to a DnaCollection object. The ability to hold more than one DataUnit object offered by the DnaCollection permits, for instance, the definition of polyploid genetic structures. The maximum number of DataUnit objects connected to a DnaCollection object is limited by the number of bits that the *_valid_data* member can contain.

```
LONG    _valid_data
```

The purpose of *_valid_data* is to indicate which of the DataUnits connected to the DnaCollection object is "valid". The meaning of the term "valid" can vary depending on the derived class. For the DnaNode class, for instance, it can be interpreted as the *dominant allele* in a diploid genetic structure. However, it assumes a different meaning in the Individual class (which is explained in another section in this chapter). Each valid ('1') bit in this member data corresponds to a connected DataUnit object. Therefore, the maximum number of DataUnits connections is limited by its number of bits.

Figure 5.11 - A DnaCollection object



The DnaCollection class defines a set of member functions that can be grouped into four categories: status, access, manipulation and auxiliary member functions.

Status member functions are invoked to get or set the status of a node, and also to enquire the validity of its DataUnits. Access member functions are called to read the values of the object's member data such as *_num_nodes*, *_num_variables*, etc. The manipulation group comprises member functions for inserting, removing or copying other DnaCollection objects maintained in a node. It includes member functions like attachNode, detachNode, makeNode, duplicateNode, deleteNode and invertNodes. The auxiliary group defines member functions providing the ability to duplicate a DnaCollection object (duplicate). However, the most important member function of the auxiliary group is the mailBox. It provides the only means for "accessing" all other member functions of DnaCollection objects connected at any level of the genetic structure.

Any public member function of a DnaCollection object may be directly called using the ordinary *C++* syntax. However, the expression necessary to call member functions of objects connected to other objects gets very complicated with the number of levels that a genetic structure may present. A mechanism centralised on a mailbox abstraction was therefore devised to simplify the access to any object connected at any layer of the genetic structure.

The argument of the mailBox member function is a message package object[5] that carries a command, arguments (if necessary) and a NodePath object specifying the address of the DnaCollection object in the genetic structure to which the message is to be delivered. The commands defined for a DnaCollection object, and their related arguments, are listed in section 5.2.6, which describes the DnaNodeMsg class. For the moment it is sufficient to know that a mailbox handles message package objects.

On receiving a message package the mailbox checks if it is addressed to itself. If so, the message package is opened and its command executed by calling the required member function. The same message package is then used to take the status and result of the requested action back to its sender. If the message package is addressed to any of its connected DnaCollection objects, an index is obtained from the NodePath object and the message package passed on to its mailBox. If no further DnaCollection object is found, according to the index specified, the message package is returned with the UNDELIVERED status.

5.2.4. The DnaNode

The DnaNode class is the basic unit for describing a GAME genetic representation. It is derived from the DnaCollection class and simply implements its pure virtual functions.

Figure 5.12 - The DnaNode class

```
class DnaNode : public DnaCollection
{
public:
                            DnaNode             (void);
                            DnaNode             (WORD, BYTE=_DATA_ARRAY_SZ);
                            DnaNode             (const DnaNode&);
  virtual                   ~DnaNode            (void);

  virtual OLT               getObjectLength     (void);
  virtual hOS               describeObject      (hOS);
  virtual hOS               assembleObject      (hOS);

  virtual hDCT              duplicate           (void);
  virtual MSG_STATUS        makeNode            (WORD, WORD=_NODE_ARRAY_SZ,
                                                 BYTE=_DATA_ARRAY_SZ);
};
```

The default values for the node array and the data array created by the DnaNode class are specified by the constants NODE_ARRAY_SIZE and DATA_ARRAY_SZ. These constants may be redefined in the gameconf.h file.

[5] A detailed description of GAME's messaging system, which includes the MessagePackage class, is presented in Chapter 6.

5.2.5. The Individual

The class Individual (Figure 5.13) further specialises the DnaNode class to operate as the root of the genetic structure. As such, it does not maintain DataUnits in the same sense as a DnaNode. In general, the genetic structure of a GA represents an encoded form of a candidate solution to a problem. The encoded genetic structure is referred to as the *genotype* and the original, or decoded form, is referred to as its biological-like *phenotype*. Since Individual objects have the ability to store DataUnit objects, they can use this functionality to store instances of these objects containing the phenotypes. The status flags associated with the DataUnit objects connected to an Individual's data array are used to indicate whether the data being held are valid or not. Any operation executed over an Individual object, which results in any modification of its genetic structure, sets the status flag associated with the DataUnit holding its phenotype as NOT_VALID. When the Virtual Machine receives a command message requesting an individual's phenotype, its status flag is inspected. If a VALID status is returned, a copy of the actual DataUnit object containing the phenotype is produced and given to the requester. Status flags are automatically updated, i.e. set to VALID, when the problem-dependent decode function is called. By keeping an updated copy of the phenotype with its "owner" the overhead resulting from decoding an unchanged genetic structure is avoided.

A similar mechanism is used by the Individual objects to maintain updated copies of their fitness values. Once an individual has been evaluated by the fitness function, the result is stored in a DataUnit object in the individual's *fitness_array*. The next time its fitness value is requested by the application, the stored value is returned, avoiding new fitness evaluations. A fitness evaluation is therefore only required if the genetic structure is modified between two fitness requests.

To provide this extra functionality, new data and function members have been added to the Individual class. Also, the mailBox member function and some of the public member functions that can modify the genetic structure (without using the mailbox interface) such as attachNode, detachNode and deleteNode have been redefined.

Figure 5.13 - The Individual class

```
class Individual : public DnaNode
{
public:
                                Individual          (void);
                                Individual          (WORD, BYTE=_DATA_ARRAY_SZ,
                                                     BYTE=_FITNESS_ARRAY_SZ);
                                Individual          (const Individual&);
  virtual                       ~Individual         (void);

  virtual OLT                   getObjectLength     (void);
  virtual hOS                   describeObject      (hOS);
  virtual hOS                   assembleObject      (hOS);

  virtual MsgPackage&           mailBox             (MsgPackage&);
  virtual MSG_STATUS            attachNode          (WORD, hDCT);
  virtual DnaCollection*        detachNode          (WORD);
  virtual MSG_STATUS            makeNode            (WORD, WORD=_NODE_ARRAY_SZ,
                                                     BYTE=_DATA_ARRAY_SZ,
                                                     FITNESS_ARRAY_SZ);
  virtual MSG_STATUS            deleteNode          (WORD);
  virtual BOOL                  getFitnessStatus    (BYTE=GLOBAL);
  virtual BYTE                  getNumVarFitness    (void);
  virtual hDUT                  readFitness         (BYTE=0);
  virtual MSG_STATUS            writeFitness        (hDUT, BYTE=0);

private:
          BYTE                  _num_var_fitness;
          BYTE                  _max_var_fitness;
          LWORD                 _valid_fitness;
          DataUnit**            _fitness_array_ptr;
};
```

The four member data added to the Individual class are:

`BYTE` *`_num_var_fitness`*

This is a counter for the current number of DataUnit objects holding fitness data, connected to the fitness array.

`WORD` *`_max_var_fitness`*

This member data holds the maximum number of DataUnit objects that can be connected to the fitness array. This value can be specified when an Individual object is declared, via its constructor. The default values provided for these member data may be modified by defining the constants NODE_ARRAY_SIZE, DATA_ARRAY_SIZE, and FITNESS_ARRAY_SIZE in the gameconf.h file.

Finally, it is interesting to note that an Individual's fitness array can maintain more than one DataUnit object, supporting applications requiring multi-fitness evaluations. In such cases, if any of the fitness values kept by an individual is set to NOT_VALID, the getFitnessStatus member function will return FALSE, until all status flags are updated by the Fitness Evaluator module.

5.2.6. Commands to DnaCollection objects

Several commands have been defined for requesting actions to DnaCollection and derived objects. Each command specifies one or more arguments that are gathered in a single message package object. Table. 5.2a and b lists these commands, their arguments, and returned data.

Table. 5.2a - DnaCollection node commands

Message Title	*Arguments*	*Returns*
DN_ATTACH_NODE	hNP - NodePath handle hDCT - DnaCollection handle	MSG_STATUS - status
DN_DETACH_NODE	hNP - NodePath handle	MSG_STATUS - status hDCT - DnaNode handle
DN_MAKE_NODE	hNP - NodePath handle WORD - index to attach new node	MSG_STATUS - status
DN_DUPLICATE_NODE	hNP - NodePath handle	MSG_STATUS - status
DN_DELETE_NODE	hNP - NodePath handle	MSG_STATUS - status
DN_INVERT_NODE_SEQ	hNP - NodePath handle [WORD] - num nodes to invert	MSG_STATUS - status
DN_GET_NUM_NODES	hNP - NodePath handle [BYTE] - ONE_LEVEL/ALL_LEVELS	MSG_STATUS - status WORD - integer value
DN_GET_MAX_NODES	hNP - NodePath handle	MSG_STATUS - status WORD - integer value
DN_GET_NUM_VAR_DATA	hNP - NodePath handle	MSG_STATUS - status BYTE - integer value
DN_GET_MAX_VAR_DATA	hNP - NodePath handle	MSG_STATUS - status BYTE - integer value

Table. 5.2b - DnaCollection data commands

Message Title	*Arguments*	*Returns*
DN_READ_DATA	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status hDUT - DataUnit handle
DN_WRITE_DATA	hNP - NodePath handle hDUT - DataUnit handle [BYTE] - index of the DataUnit	MSG_STATUS - status
DN_DUPLICATE_DATA	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status hDUT - DataUnit handle
DN_DELETE_DATA	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status
DN_GET_DATA_STATUS	hNP - NodePath handle [BYTE] - GLOBAL/index	MSG_STATUS - status BOOL - data status
DN_SET_DATA_STATUS	hNP - NodePath handle BOOL - data status [BYTE] - index of the DataUnit	MSG_STATUS - status
I_GET_FITNESS_STATUS	hNP - NodePath handle [BYTE] - GLOBAL/index	MSG_STATUS - status BOOL - data status
I_GET_NUM_VAR_FITNESS	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status BYTE - integer value

5.2.7. The DnaNodeMsg class

The DnaNodeMsg class, shown in Figure 5.14, defines the messaging object that navigates throughout the genetic structure, delivering commands and their arguments, and bringing back to its sender the execution status and returned data. This class is derived from the MsgPackage class, and inherits all its properties.

Figure 5.14 - The DnaNodeMsg class

```
class DnaNodeMsg : public MsgPackage
{
public:
                          DnaNodeMsg           (void);
                          DnaNodeMsg           (const DnaNodeMsg&);
  virtual                 ~DnaNodeMsg          (void);

  virtual MsgPackage&     operator=            (const MsgPackage&);

  virtual OLT             getObjectLength      (void);
  virtual hOS             describeObject       (hOS);
  virtual hOS             assembleObject       (hOS);

          void            storeMessage         (DNA_NODE_MSG);
          void            storeMessage         (DNA_NODE_MSG, hNP, hDCT);
          void            storeMessage         (DNA_NODE_MSG, hNP, WORD=0);
          void            storeMessage         (DNA_NODE_MSG, hNP, hDUT, BYTE=0);
          void            storeStatus          (MSG_STATUS);
          void            storeStatus          (MSG_STATUS, WORD);
          void            storeStatus          (MSG_STATUS, BOOL);
          void            storeStatus          (MSG_STATUS, hDCT);
          void            storeStatus          (MSG_STATUS, hDUT);
  virtual WORD            getNextAddress       (void);
  virtual BOOL            isLastAddress        (void);

  virtual hDCT            recallNodeHandle     (void);
  virtual hDUT            recallDataHandle     (void);
  virtual WORD            recallMaxNodes       (void);
  virtual WORD            recallNumNodes       (void);
  virtual BOOL            recallDataStatus     (void);
  virtual BYTE            recallMaxVar         (void);
  virtual BYTE            recallNumVar         (void);

protected:
           NodePath       _path;
           hDCT           _node_handle;
           hDUT           _data_handle;
           BOOL           _data_status;
};
```

Some member data have been added to the original set of the MsgPackage class. They enable the DnaNodeMsg class to store specific arguments associated with commands defined for DnaCollection objects. The new member data of this class are:

`NodePath _path`

A NodePath object that carries the sequence of addressing items identifying the node that the DnaNodeMsg object is addressed to. The DnaNodeMsg class provides member functions for reading address items from *_path*.

`hDCT _node_handle`

The *_node_handle* member data stores a handle to a DnaCollection (hDCT) object. Therefore, when a DN_ATTACH_NODE command code is issued, the DnaNode that receives it gets the node handle to be attached from this member data, calling the recallNodeHandle member function. It can also bring a node handle when a DN_GET_NODE command code is issued, for instance.

`hDUT _data_handle`

The *_data_handle* stores a handle to a DataUnit (hDUT) object. Therefore, when a DN_WRITE_DATA command code is sent, the DnaNode that receives it gets the data handle to be written from this variable, calling the recallDataHandle member function. It can also bring a data handle when a DN_READ_DATA command code is sent.

`BOOL _data_status`

This member data is only used to bring the status (VALID, NOT_VALID) associated with a particular DataUnit stored in a DnaNode or Individual object.

Member functions have also been added to provide access to the member data described above. In general, they allow for storing a message command code (the message itself) and its arguments, all in a single call (e.g. storeMessage). Member functions for reading addressing items stored by the *_path* member data have been included as well.

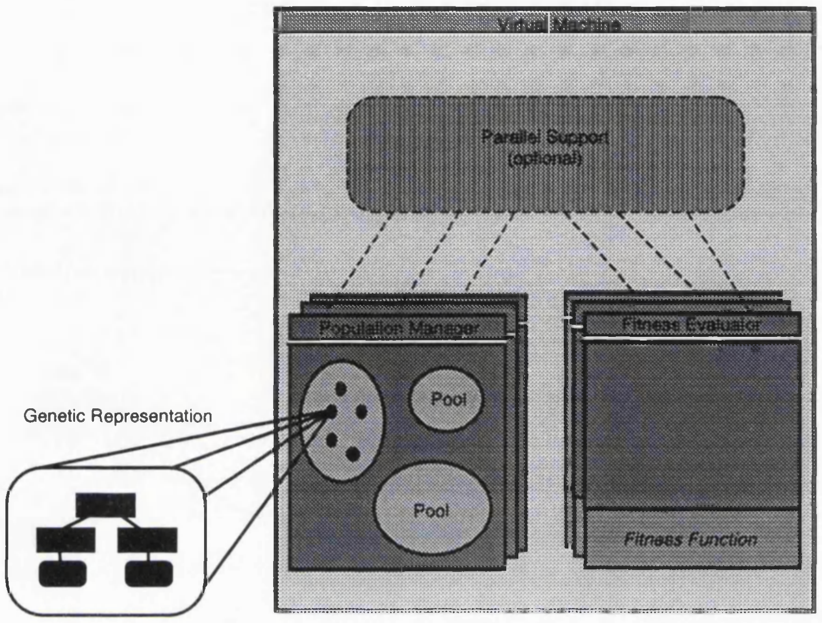
Having described the abstractions and objects defined in GAME that support the genetic representation of problems, the next section explains how these data structures are manipulated by the Virtual Machine module.

5.3. The Virtual Machine

The Virtual Machine (VM) is the module responsible for maintaining the data structures that represent genetic information. It also provides facilities for their genetic manipulation and evaluation. The VM isolates genetic operators and algorithms from dealing directly with the data structures, via a set of commands implemented as a collection of functions — the VM Application Program Interface (VM-API). Furthermore, it offers a certain degree of parallelism, being able to execute commands simultaneously.

GAME's Virtual Machine comprises three modules (see Figure 5.15): the *Population Manager*, the *Fitness Evaluator* and the *Parallel Support*. The Population Manager executes genetic manipulation commands over the data structures kept in its pools. The Fitness Evaluator performs the evaluation of the genetic structures and related computations such as total, average, highest and lowest fitness. Finally, the parallel support module is able to distribute commands received by the VM over one or more instances of the Population Manager and Fitness Evaluator modules.

Figure 5.15 - The Virtual Machine and its modules



The introduction of a separate module for executing genetic manipulations via high-level commands, besides being innovative in this context, facilitates the definition and implementation of representation-independent genetic algorithms and operators. A great deal of flexibility and portability is achieved by preventing these components from directly dealing with data structures, memory management and exception handling.

The design of GAME's Virtual Machine also contemplates other aspects such as modularity and parallelism. By separating manipulations over the data structures from objective function evaluations in two specialised modules, it is possible to modify, adapt or expand any one of these modules with virtually no impact on the other module, or the Virtual Machine itself. Moreover, these modules being defined as GAME Components, can be replicated and executed as independent processes, possibly running on separate processors. This fact alone grants tremendous power and flexibility to the Virtual Machine in parallelising genetic manipulations and fitness evaluations of ordinary sequential applications.

The implementation of the Virtual Machine is straightforward, as shown in Figure 5.16. The various messages, implemented by the VM-API, are received by the Virtual Machine and routed to the appropriate module. Handles of the Population Manager and Fitness Evaluator object instances are kept in the *_pm_array* and *_fitness_array* member data.

Figure 5.16 - The VirtualMachine class

```
class VirtualMachine : public GameComponent
{
public:
                         VirtualMachine       (void);
  virtual               ~VirtualMachine       (void);

protected:

        void             processMail          (VmMsg&);
//
// Member Data
//
        hGC              _pm_array[MAX_PM];          // array of PM handlers
        hGC              _fitness_array[MAX_FITNESS]; // array of Fitness
                                                      // Components' handlers
};
```

5.3.1. Virtual Machine Commands

The Virtual Machine accepts three groups of messages: messages to the VM itself, messages to Population Managers and messages to Fitness Evaluators. The messages are defined as numeric codes allocated in specific ranges. Messages to the Population Manager, for instance, have been allocated in the 0x100 to 0x1FF range. Fitness Evaluator messages occupy the range between 0x200 and 0x2FF, and messages to the VM itself in the range 0x300 to 0x3FF. Any of these groups may be expanded with the definition of new messages, provided the *C++* classes (VirtualMachine, PopulationManager and FitnessEvaluator) that implement their execution are also expanded to process them.

As with any other GAME component, the Virtual Machine interacts with genetic operator and algorithm components by the means of specialised message package objects. Figure 5.17 shows the VmMsg class, which carries to the Virtual Machine all the messages generated by the VM-API.

A VmMsg object is capable of carrying the arguments, status and returned data resulting from a command issued by the VM-API. It provides the required member data to store addressing sequences for up to two genetic structures (as required by swap, copy and move commands), DataUnit indexes, execution status, and other information returned by the VM. A typical VM message would contain a command code, an addressing sequence for a genetic data structure (stored in the *_path* data member) and, possibly, a DataUnit index. On its return, a VmMsg object will bring the execution status (SUCCESS/FAILURE) and, if necessary, the data resulting from the command execution.

Figure 5.17 - The VmMsg package class

```
class VmMsg : public MsgPackage
{
public:

                            VmMsg          (void);
                            VmMsg          (const VmMsg&);
  virtual                   ~VmMsg         (void);

  virtual MsgPackage&       operator=      (const MsgPackage&);

  virtual OLT               getObjectLength(void);
  virtual hOS               describeObject (hOS);
  virtual hOS               assembleObject (hOS);

  virtual void              storeMessage   (GAME_MSG);
  virtual void              storeMessage   (GAME_MSG, hNP, BOOL, WORD=0);
  virtual void              storeMessage   (GAME_MSG, hNP, WORD=0, DOUBLE=0);

  virtual void              storeStatus    (MSG_STATUS);
  virtual void              storeStatus    (MSG_STATUS, WORD);
  virtual void              storeStatus    (MSG_STATUS, hDCT);
  virtual void              storeStatus    (MSG_STATUS, hDUT);
  virtual void              storeStatus    (MSG_STATUS, BOOL);

  virtual hDCT              recallNodeHandle (void); // the node/individual handle
  virtual hDUT              recallDataHandle (void); // the data handle
  virtual BOOL              recallDataStatus (void); // single/global data status
  virtual WORD              getNextAddress   (ADDRESS_TYPE=NODE1);
  virtual BOOL              isLastAddress    (ADDRESS_TYPE=NODE1);
  virtual WORD              recallIndex      (void); // the data index
  virtual hNP               recallNodePath   (ADDRESS_TYPE=NODE1);
  virtual void              insertAddress    (WORD, ADDRESS_TYPE=NODE1);

          void              storePmHandle      (hGC);
          void              storeFitnessHandle (hGC);

          void              storePmId          (BYTE);
          void              storeFitnessId     (BYTE);

          hGC               recallPmHandle      (void);
          hGC               recallFitnessHandle (void);

          BYTE              recallPmId         (void);
          BYTE              recallFitnessId    (void);

// Member Data

protected:

          NodePath          _path[2];      // hold the source (and dest) path(s)
          WORD              _index;        // hold the index of a Data Unit
          hDCT              _node_handle;  // hold the node/individual handle
          hDUT              _data_handle;  // hold the data handle
          BOOL              _data_status;  // hold the single/global data status

          hGC               _pop_man_handle; // hold the PM Component handle
          hGC               _fitness_handle; // hold the Fitness Component handle

          BYTE              _pop_man_id;   // hold the PM index in the VM array
          BYTE              _fitness_id;   // hold the Fitness index in the VM array
};
```

Only two commands are currently defined for the Virtual Machine itself, as shown in Table 5.3 below.

Table 5.3 - VM commands

Message Title	*Arguments*	*Returns*
VM_GET_POP_MAN_HANDLE	[BYTE] - Id of the population manager component	MSG_STATUS - status hGCT - component handle
VM_GET_FITNESS_HANDLE	[BYTE] - Id of the fitness evaluator component	MSG_STATUS - status hGCT - component handle

5.4. The Population Manager

The Population Manager is the VM module responsible for maintaining population pools. Like the VM, it is a GAME component. As such, it interacts with the Virtual Machine by exchanging message package objects. Once the VM receives a message package object, it determines, by looking at the command code, the modules to which the message is being addressed and passes it to the module's mailbox.

Figure 5.18 - The PopulationManager class

```
class PopulationManager : public GameComponent
{
public:
                              PopulationManager      (BYTE);
  virtual                    ~PopulationManager      (void);

protected:

          void               processMail             (PopManMsg&);
          BOOL               checkPoolIndex          (WORD);

// Member Data

          BYTE               _pop_man_id;
          Pool*              _pool_array[MAX_POOL]; // array of pointers to pool
                                                    // objects
};
```

As the Virtual Machine may contain several instances of the Population Manager component, it assigns to each one a unique identifier, or *Id.* These Ids are stored in the VM *_pm_array* and also passed to the associated PM instance. A Population Manager, besides its Id, also keeps a list of population pools in its *_pool_array* member data (see Figure 5.18).

5.4.1. Population Manager Commands

Each command of the Population Manager module specifies one or more arguments that are sent within a VmMsg object. Table 5.4 below lists the Population Manager commands, their arguments, and returned data.

Table 5.4 - Population Manager commands

Message Title	*Arguments*	*Returns*
PM_CREATE_POOL	hNP - NodePath handle hDCT - DnaCollection handle	MSG_STATUS - status
PM_DELETE_POOL	hNP - NodePath handle	MSG_STATUS - status hDCT - DnaNode handle
PM_COPY_POOL	hNP - NodePath handle WORD - index to attach new node	MSG_STATUS - status
PM_GET_POOL_SIZE	hNP - NodePath handle	MSG_STATUS - status
PM_GET_POPULATION	hNP - NodePath handle	MSG_STATUS - status
PM_GET_NODE	hNP - NodePath handle [WORD] - num nodes to invert	MSG_STATUS - status
PM_PUT_NODE	hNP - NodePath handle [BYTE] - ONE_LEVEL/ALL_LEVELS	MSG_STATUS - status WORD - integer value
PM_COPY_NODE	hNP - NodePath handle	MSG_STATUS - status WORD - integer value
PM_MOVE_NODE	hNP - NodePath handle	MSG_STATUS - status BYTE - integer value
PM_DELETE_NODE	hNP - NodePath handle	MSG_STATUS - status BYTE - integer value
PM_INVERT_NODE_SEQ	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status hDUT - DataUnit handle
PM_SWAP_NODES	hNP - NodePath handle hDUT - DataUnit handle [BYTE] - index of the DataUnit	MSG_STATUS - status
PM_GET_NUM_NODES	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status hDUT - DataUnit handle
PM_GET_MAX_NODES	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status
PM_READ_DATA	hNP - NodePath handle	MSG_STATUS - status BOOL - data status
PM_WRITE_DATA	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status BOOL - data status
PM_COPY_DATA	hNP - NodePath handle BOOL - data status [BYTE] - index of the DataUnit	MSG_STATUS - status
PM_DELETE_DATA	hNP - NodePath handle [BYTE] - index of the DataUnit	MSG_STATUS - status BOOL - data status
PM_SWAP_DATA	hNP - NodePath handle BOOL - data status [BYTE] - index of the DataUnit	MSG_STATUS - status

5.5. The Fitness Evaluator

The Fitness Evaluator module, being part of the Virtual Machine, is not directly accessible from the genetic algorithm or operator levels. It is imperative for the Fitness Evaluator to be part of the Virtual Machine, since this represents the most effective way to directly access the genetic data structures. This design requirement is consistent with the desire to keep the implementation of genetic operators and algorithms as independent as possible from problems' characteristics and particularities. However, a Fitness Evaluator must embody the problem dependent objective function. This would normally require direct intervention of the application programmer for its definition and implementation. The Fitness Evaluator is designed to be an "internal class", that is, a module that should not be directly "touched" by the application developer, or user. To accommodate both requirements, a compromise solution allows the "insertion" of a problem-dependent evaluation function, without having to modify the actual implementation of the FitnessEvaluator class (Figure 5.19). The implementation of this solution came via a member function defined in the FitnessEvaluator class – the getFitness function – that calls an external, problem-dependent, FitnessEvaluation function. This strategy implies that the Fitness Evaluator module needs only to be statically linked with the user-defined objective function, before having an instance created by the Virtual Machine. This implementation grants maximum flexibility in defining and implementing problem-dependent objective functions, without directly interfering with the Fitness Evaluator implementation.

Figure 5.19 - The FitnessEvaluator class

```
class FitnessEvaluator : public GameComponent
{
public:
                                FitnessComponent      (BYTE);
  virtual                      ~FitnessComponent      (void);

protected:

           void                 processMail           (FitnessMsg&);

           hDUT                 sumFitness            (FitnessMsg&); // total fitness
           hDUT                 avgFitness            (FitnessMsg&); // average fitness
           void                 highestFitness        (FitnessMsg&); // the highest
           void                 lowestFitness         (FitnessMsg&); // and the lowest

           hDUT                 getFitness            (FitnessMsg&);
           hDUT                 getFitness            (FitnessMsg, WORD);
           WORD                 getPmPopulation       (FitnessMsg&);

// Member data

           BYTE                 _fitness_id;
};
```

The FitnessEvaluator class provides the basic functionality for computing the total, average and finding the highest and the lowest fitness values in a population of genetic data structures.

5.5.1. Fitness Evaluator Commands

A Fitness Evaluator can process up to six different commands with, at least, one argument. Table 5.5 lists these commands, their arguments, and returned data.

Table 5.5 - Fitness Evaluator commands

Message Title	*Arguments*	*Returns*
FITNESS_GET_VALUE	hNP - NodePath handle hDCT - DnaCollection handle	MSG_STATUS - status
FITNESS_GET_STATUS	hNP - NodePath handle	MSG_STATUS - status hDCT - DnaNode handle
FITNESS_GET_SUM	hNP - NodePath handle WORD - index to attach new node	MSG_STATUS - status
FITNESS_GET_AVERAGE	hNP - NodePath handle	MSG_STATUS - status
FITNESS_GET_HIGHEST	hNP - NodePath handle	MSG_STATUS - status
FITNESS_GET_LOWEST	hNP - NodePath handle [WORD] - num nodes to invert	MSG_STATUS - status

5.6. The Parallel Support Module

The Virtual Machine has an optional capability to parallelise the execution of the commands it receives. This is achieved through its parallel support (PS) module. The PS is, in fact, a resource scheduler that is activated if the VM is programmed to operate with more than one instance of the Population Manager or the Fitness Evaluator modules. It enables the Virtual Machine to take full advantage of the data parallelism offered by genetic algorithms. Furthermore, the definition of a standard interface (the VM-API) and the design of the VM and its modules, prompted the exploitation of this form of parallelism.

The design of the parallel support module is based on a farming model, where the Virtual Machine operates as the master and the Population Managers and Fitness Evaluators as slaves. The number of slaves and the actual processors that will host them can be specified by the user in the application configuration file. If activated, VM's parallel support module will look for "idle" instances of Population Manager and Fitness Evaluator modules to deliver messages received by the Virtual Machine. If no idle module is found, the message is queued until a module capable of handling one of the messages in the queue becomes available.

This additional capacity of GAME's Virtual Machine allows transparent exploitation of parallelism in the genetic manipulation and fitness evaluation levels. Moreover, it does so without requiring any special concern with parallelism in the genetic algorithm or operator levels. Genetic algorithms and operators can be implemented as straightforward sequential code and yet benefit from VM's parallel features.

In general, complex problems present very time-consuming fitness evaluations. By simply creating many instances of the Fitness Evaluator module, and distributing them among several processors, a simulation will achieve an enormous execution speedup. GAME's VM offers this possibility without requiring any modification in the application, genetic algorithm or operator's sequential code.

5.7. The VM-API

The Application Program Interface concept provides an easy and simple mechanism to hide the message-driven architecture of GAME. By invoking straightforward *C*-like functions, the user does not have to be concerned with the intricacies involved in creating and passing message package objects across components. The definition of the Virtual Machine API resulted from observations about genetic algorithms' operation, as described in Chapter 2. It provides a comprehensive set of functions (described in Appendix B) that support the implementation of a broad range of genetic operators and algorithms.

Each function generates specific commands that are passed on to the Virtual Machine. The arguments provided by the user on calling VM-API functions are "wrapped" with the command code requesting the required operation into a VmMsg object, which is then dispatched to the VM. The VM-API can be easily expanded with user-defined functions, to make it more suitable for specific applications.

The VM-API is divided into five classes of functions:

- Population Manager functions:
  CreatePool, DeletePool, CopyPool, etc.
- Individual & DnaNode functions:
  GetIndividual, PutIndividual, CopyIndividual, MoveIndividual, KillIndividual,
  GetNode, PutNode, CopyNode, MoveNode, DeleteNode, SwapNodes, etc.
- DataUnit access:
  ReadData, WriteData, CopyData, DeleteData, etc.
- Fitness Evaluation and statistics related functions:
  EvaluateFitness, GetFitness, GetTotalFitness, GetAvgFitness,
  GetHighestFitness, GetLowestFitness, etc.
- Error handling:
  GetErrorStatus, ClearErrorStatus.

The complete set of functions that comprise the VM-API is presented in Appendix B, which describes all the functions, their arguments and returned data.

5.8. Summary

This chapter discussed the design and some implementation aspects of GAME's genetic-oriented representation abstractions and its Virtual Machine module. The genetic-oriented abstractions aims at supporting the representation of a wide range of GA, ES and GP problems. It permits the use of the traditional "flat", binary-encoded and string-based model, as well as real value and multi-levelled representations. Suitable data types (binary, int, float, etc.) have been defined to accommodate mixed representations and facilitate the implementation of problem-independent genetic operators. These data types, embedded into a tree data structure, promote easy and efficient manipulations of genetic representations.

The Virtual Machine concept provides an effective mechanism for problem-independent manipulation of genetic data structures. The separation of problems' data structures from genetic algorithms and operators offers the possibility of designing and implementing standard and re-usable versions of these modules. This approach also provides an enormous flexibility for combining different genetic operators into algorithms, without any regard to the actual problem representation kept by the Virtual Machine. This degree of independence is only possible thanks to the set of access and manipulation functions defined in the Virtual Machine application programming interface. By simply calling VM-API functions, genetic operators and algorithms are kept away from low-level issues like memory management, exception handling or platform dependencies hidden in the Virtual Machine and its internal modules.

Finally, the architecture of the Virtual Machine, comprising distinct modules for genetic manipulation (Population Manager) and evaluation (Fitness Evaluator), facilitates and promotes the exploitation of the data parallelism offered by GAs. The farming model adopted in the design of the parallel support module enables the parallelisation of VM commands without any special requirement in the construction of sequential applications, algorithms and genetic operators.

Chapter 6

GAME's Programming Model and the Parallel Execution Module

This chapter presents a brief review of three software strategies that support inter-process communication and control of parallelism, and introduces GAME's object-oriented parallel programming model. It also describes the design and implementation of the Parallel Execution Module, which supports the programming model.

6.1. Introduction

Parallelisation of complex problems is, on its own, a very large and complex area of the Computer Sciences. The tasks involved in breaking a problem into collaborating processes and co-ordinating their communication and execution require a profound knowledge of the problem at hand, the execution environment and the programming tools to be applied. The success of parallel applications relies upon a careful combination of programming model, implementation tools and parallel architecture. As a consequence of the last statement, one could conclude that parallel applications are not portable across different parallel architectures. In fact, it is much more difficult to port parallel applications then sequential ones. However, common characteristics present in many problems provide the grounds for the definition of "general-purpose" parallel programming models and programming environments. Although commonly called general-purpose, these systems are targeted at broad problem domains like thermal analysis, molecular modelling, fluid dynamics simulations, etc. This domain-based approach gained popularity during the last couple of decades and favoured the development of programming models and tools capable of addressing an even broader range of problems. There are today sophisticated programming environments that provide a variety of tools to help with the design, implementation, debugging and monitoring of a variety of parallel applications. Examples of such programming environments include PVM [90], PRISM [86], ADE [28] and many others.

More recently, a new and exciting area involving parallelisation over distributed heterogeneous platforms is attracting the attention of many researchers and also the industry.

This approach offers the possibility of gathering all available computational resources to work on a particular problem, regardless of the actual machine architecture, network topology and operating system breed. This alternative seems even more attractive, considering the computational power of an increasing number of local-area networks based on cheap and powerful workstations. So far, these enterprise networks have been mainly used to distribute data. However, new technologies based on the object-oriented paradigm are promising to bring true parallelism to ordinary users by distributing and co-ordinating the operation of objects over these heterogeneous networks.

These technologies are central to the parallel tools offered by GAME. The design of GAME's programming model reflects the awareness in adhering to distributed object computing emerging "standards", whereas its implementation fulfils the three main requirements of the programming environment: flexibility, expandability and portability.

The next section presents an overview of common software strategies used in supporting parallelism at different levels and degrees. This brief discussion provides the rationale behind GAME's parallel programming model and its underlying implementation, the Parallel Execution Module, presented in the following sections.

6.2. Parallel Programming Software Strategies

A common problem encountered by developers of parallel applications is the difficulty of porting their programs to different parallel architectures. This task normally implies major changes in the code and sometimes in the program design. In order to overcome some portability problems, operating systems have been equipped with extensions to support parallelism. They partially remove the burden of directly controlling parallel resources from the program developer. In fact, multitasking operating systems offer a natural path to concurrent execution of co-operating agents (processes in this case). Multitasking operating systems running on multiprocessor architectures or on networks of computers (distributed systems), can effectively exploit parallelism by distributing applications' processes over a number of processing units. This type of solution permits applications to be designed independently of the specific parallel hardware. To run the same application on different architectures, without modifications, it should be only necessary to have the same operating system and language/compiler available on the target platform.

The support offered by operating systems, however, is only suitable for certain classes of problems exhibiting medium to coarse-grain parallelism. An application accesses parallel programming resources, such as communication channels, barriers and semaphores, through the operating system. Only the operating system can execute the code that actually controls process creation, communication and synchronisation. This generally implies context switching and many internal administrative operations (scheduling, updating tables and buffers, etc.) being carried out before the code that controls the parallel resource is executed. Therefore, the overhead introduced by the operating system prevents fine-grained parallel applications from effectively benefiting from parallel resources.

Medium and coarse-grained applications present relatively large parallel modules (procedures, functions, etc.). They execute concurrently and request the operating system to communicate or synchronise with other co-operating processes only during a fraction of their entire execution time. In such cases, the overhead introduced by the operating system is diluted. On the other hand, fine-grained parallel applications, generally represented by *loops* manipulating a set of unrelated data elements (e.g. matrix copy or multiplication), execute parallel operations in very short time periods, if compared with the operating system's overhead. These types of applications can only benefit from dedicated hardware that does not require operating system control to perform communication and synchronisation. The conclusion of this discussion is that applications exhibiting multiple levels of granularity, or heavily based on fine grain parallelism, still face portability problems or cannot have their parallel potential fully exploited.

Central to parallel computing are the communication mechanisms and processor support for language features [120]. Message-passing and shared memory systems are two common mechanisms used to implement inter-process communication (IPC) and synchronisation on popular parallel computer architectures. Parallel applications designers might choose among three basic types of software communication and synchronisation mechanisms:

- Language-based,
- Operating system dependent, and
- Language/Operating system independent

Language-based mechanisms have been implemented by extending sequential programming languages with parallel commands, data structures and runtime libraries. Also, parallel languages like *OCCAM* [60] have been specifically created to better exploit features of some parallel architectures.

Many popular operating systems have received extensions and others were entirely designed to support parallelism. Both alternatives offer a number of "services" to supervise communication and control the allocation of parallel resources. Most parallel operating systems provide standard application programming interfaces based on the UNIX specification. This allows a number of applications to be easily ported on to a variety of parallel architectures.

In both cases, an application must be specifically designed to benefit from a particular parallel language, or to exploit the operating system's support for concurrent or parallel execution. Portability problems across different operating systems (or even non-standard implementations of the same operating system) and an increasing demand for higher level programming interfaces, led to the development of language and operating system independent mechanisms to support parallelism. Systems like the Parallel Virtual Machine (PVM), for instance, comprise a library of high-level communication and synchronisation functions, and a runtime module, which supervises and controls interactions among co-operating processes of an application.

The next sections present these three strategies by briefly reviewing examples of software systems that implement them.

6.2.1. Language Based Mechanisms

There are two types of language-based parallel mechanisms: extensions to sequential languages and parallel languages. Libraries and primitives supporting parallelism and communication have been included as extensions to sequential languages such as *FORTRAN* and *C*. In many cases, compilers were specially re-designed to identify and "extract" parallel structures from source code programs, transparently to the programmer. A number of large and complex applications (mainly written in sequential *FORTRAN*) benefited from this approach. They could then be executed on many parallel architectures after recompilation. Conversely, parallel programming languages (e.g. *OCCAM*) offer explicit means for partitioning programs combined with high-level mechanisms for communication and synchronisation.

Object-oriented programming models suggest a natural method for structuring and partitioning parallel applications based on the *active object* concept [103]. Also, the use of messages for object interaction, maps directly into message-passing mechanisms. Two object-oriented languages supporting parallelism are reviewed in this section: *Parallel C++* and *UC++*. *Parallel C++*, from 3L, is targeted at transputer-based machines and extends sequential *C++* with a number of functions grouped into libraries. The *UC++* programming language proposes a machine-independent option for parallel implementations, based on extensions to the standard *C++* language.

3L PARALLEL C++ — is a parallel version of the *C++* language developed by 3L Ltd. that runs only on transputers [104]. The compiler is based on cfront 2.1 (AT&T's *C++* compiler) and the current version does not support parameterised types (templates). *Parallel C++* generates *C* code that is run through a *C* compiler to produce object files. These are then linked with *C* and *C++* libraries to create executable files. Transputer systems are generally attached to conventional front-end hosts like Sun workstations and PCs. A special server program, running in the host, provides the means to load executable code into the transputer system and support I/O operations.

Concurrent processing is provided by a set of additional libraries, rather than through enhancements to the language. *Parallel C++* supports common multitasking and inter-process communication mechanisms through *threads* and *pipes* respectively. It also supports the execution of concurrent processes of the same (or different) programs on multiple processors.

Multitask operations are normally split into I/O tasks, requiring access to the host's operating system facilities, and one or more processor tasks. The latter communicate via a special inter-task communication system known as *channels*. Channels allow tasks to exchange information and are attached to their input and output ports – the interconnections are specified in a separate configuration file. A set of *Parallel C++*'s libraries provides functions to access channels. The language also provides facilities for accessing the host's local memory, including special functions for allocating and de-allocating memory, and transferring data between transputer and host memories. Examples of functions available in the libraries are:

- Functions for initialising, resetting and transferring data via channels

  chan_init, chan_reset,
  chan_in_byte, chan_in_byte_t, chan_out_byte, chan_out_byte_t
  chan_in_word, chan_in_word_t, chan_out_word, chan_out_word_t
  chan_in_message, chan_in_message_t,
  chan_out_message, chan_out_message_t
- Functions that allow processor farm communications

  net_broacast, net_send, net_receive
- Functions to create and manipulate semaphores

  sema_init,
  sema_signal, sema_signal_n
  sema_test_wait
  sema_wait, sema_wait_n
- Functions to generate new threads within a task

  thread_start, thread_create, thread_priority,
  thread_deschedule, thread_restart, thread_stop

One major problem with the runtime libraries of *Parallel C++* is that their functions are not re-entrant. Access to the functions that control parallel resources must be synchronised using a global semaphore variable, which implies that all other threads are locked out of a library, while it is in use.

To write a parallel application, the user should divide it into tasks, each task corresponding to a separate program. After all the tasks of an application (including the special server task for I/O) are compiled and linked, a process configuration file must be created, specifying task allocation and channel interconnections.

Overall, the language offers basic facilities for developing transputer-based parallel applications in *C++*. The library approach is an elegant alternative to extend the language, making it much simpler to port to several parallel architectures. In this case however, no advantage has been taken of *C++*'s features to simplify parallelism. In particular, function overloading is not used to avoid the need for separate channel functions for each data type.

UC++ — is a parallel *C++* language under development by UCL's CoSIDE/COOTS project. The primary aim of the Concurrent Object-Oriented languages Targeting Parallel Systems (COOTS) is to develop a version of the *C++* language that provides concurrence, using object-oriented techniques [103]. The resulting language is suitable for use with a range of MIMD machines. *UC++* is being developed under CoSIDE — an environment supporting parallel object-oriented programming for a variety of languages. The parallel *C++* implementation is based on a set of minimal extensions to the sequential *C++* language, and extensions in the form of class libraries.

UC++ extends the original *C++* language by adding the concept of *active object.* Each active object is associated with a virtual processor and may (potentially) execute member functions in parallel with other active objects. Active objects may be created either statically or dynamically. The computation model adopted for the language design defines an abstract machine with the following characteristics:

- composed by an array of virtual processors, each supporting an active object,
- a global shared address space, and
- a message-passing sub-system, to support active object member function calls.

Two implementation strategies for the language are being pursued: a version directly integrated into the CoSIDE environment, to explore areas such as incremental compilation, and a version implemented using a filter as a front end to an ordinary C++ compiler, to gain the benefits of using a standard compiler and efficient code generation.

6.2.2. Operating System Based Mechanisms

Many mechanisms created to support multitasking on uniprocessor-based operating systems have been mapped on to their parallel versions, taking advantage of the parallel resources offered by the underlying hardware. This approach provides more flexibility to parallel applications, which can benefit from parallel support through operating systems' standard interfaces. The dissemination of popular multitasking operating systems like UNIX, enhanced with parallel extensions, has fuelled the development of a number of parallel applications. The immediate consequence is the availability of the same application for different parallel architectures. Operating system-based parallel support has proven to be particularly suitable for coarse-grained parallel applications.

PARIX and *PAROS* [64] are examples of operating systems offering different approaches to support parallelism. *PAROS* was specifically developed to be a parallel operating system. Its internal structure was designed to accommodate different architectures. *PARIX*, on the other hand, is a UNIX-like operating system including extensions to benefit from a specific parallel architecture, designed and produced by Parsitec.

PARIX — is the operating environment for the Parsytec GC series. It has a user interface based on the UNIX operating system philosophy, with parallel extensions to give access to the resources offered by the Parsitec parallel super-computer. The basic communicating facilities supported by *PARIX* are *virtual links*. A virtual link is a bi-directional, synchronising, non-buffering, point-to-point communication mechanism between two threads of control. Threads can be located in the same or different processor, or server, and a set of virtual links may be combined to define a *virtual topology*. There are two classic types of communication mechanisms available in *PARIX*:

- Synchronous virtual link-bound communication — This is the transputer's native form of communication. Communicating processes are connected via virtual links and synchronised upon communication.
- Asynchronous virtual link-bound communication — Communication can be carried out concurrently with computation. It optimises processor usage, as sending and receiving are performed in the background, while the processor continues executing the application thread. Intermediate buffering is supported at both the sender and receiver sides.

Like many UNIX-compatible operating systems available for parallel machines, PARIX offers parallel extensions, which are specific to support the particularities of the Parsitec hardware. An application built upon its parallel support would have to suffer major modifications in order to be ported to parallel machines with different operating system extensions, and possibly different programming language libraries.

PAROS — PARallel Operating System architecture is an efficient self-hosted, general-purpose parallel operating system for distributed memory parallel machines, sustaining high performance for parallel applications. It has been developed at IMAG – University of Grenoble – as part of the Supernode_II ESPRIT project. One of the major objectives of PAROS is to offer support for various grains of parallelism and communication in parallel applications.

The operating system has been structured as a low-level kernel (PARX) and subsystem environments built on top of it. The PARX kernel provides a reduced set of simple and well-defined basic abstractions that can be used by application and subsystem programmers. It is structured in several layers of Virtual Machines, allowing various interfaces to achieve user level compatibility with many parallel programming models. It currently supports the X/OPEN, UBIK/ASI and PCTE standards.

The implementation of PAROS confines hardware dependencies to its lowest level virtual machine. This approach allows future hardware evolution (or heterogeneous hardware) to be easily integrated. The execution model, which directly supported by the kernel, is based on non-shared memory and synchronous message-passing communications. The process model of PAROS is based upon three levels of abstractions:

- *Ptask* encapsulates a parallel program in execution with the associated control support. It consists of a set of tasks together with some synchronisation and communication protocols.
- *Task* is a logical address space in which several flows of control can execute. A task executes on a given logical processor within a Ptask.
- *Thread* is a sequential flow of control within a task. Threads are controlled as "lightweight" processes.

The use of several threads in a task is intended to support language parallel constructs, as well as to easily implement asynchronous communication at upper Virtual Machine levels, on top of the synchronous native mechanism.

The kernel supports two basic communication objects: ports and channels. Ports can implement a global and flexible many-to-one, protected, system-oriented communication mechanism. It can be used to access subsystems and servers. Channels are fast, one-to-one communication mechanism between threads or tasks within a Ptask. They cannot be used for communication between different Ptasks.

The current generation of operating systems is based on the *microkernel* architecture (similar to PAROS's lowest level Virtual Machine) that was firstly proposed for the MACH operating system from Carnegie Mellon University [96]. Microkernel-based architectures considerably improved the portability and flexibility of recent versions of UNIX-like operating systems such as OSF1 and CHOROUS [67], and is also present in the core of Microsoft's 32-bit operating system Windows-NT. The coming versions of operating systems from IBM, Apple, Novell (now owner of UNIX Systems Laboratories) and Unisys are also following this trend.

6.2.3. Language and Operating System Independent

The previous alternatives contemplate aspects involving parallel programming that are not complementary. It is relatively easy to implement parallel applications based on operating systems support, but they are not able to efficiently exploit fine-grain parallelism. On the other hand, language-based parallel support can effectively give more control to the programmer over the use of parallel resources. However, portability appears as a major problem since parallel languages and compilers are generally tailored to specific machines.

The ideal solution would enable applications to be written independently from any specific parallel support (language or operating system) and, at the same time, efficiently exploit any degree of granularity. This alternative has been pursued by some software systems through the definition of abstract modules acting as interfaces between the application and the underlying parallel support. These modules, sometimes called *mappers,* generally comprise a library with a number of basic functions that implement common parallel and communication mechanisms. A runtime kernel may be present on some implementations to supervise operations that are not directly mapped into operating system or hardware support.

In this case, the actual library implementation establishes the efficiency of the parallel application for a given architecture. For instance, a mapper for a transputer-based machine should provide programming libraries capable of exploiting all the facilities offered by the transputer architecture. The same application, running on workstations in a distributed system, should encounter equivalent libraries mapping its requests on to the parallel support of the host operating system.

Two software systems representing this alternative are MICL and PVM. GMD's *MICL* is a fully machine-independent library used in the PeGAsuS programming environment for PGAs, whereas *PVM* focuses on applications exhibiting coarse-grain components and needs a kernel running on top of the UNIX operating system. PVM's library presents an extensive set of functions that can serve as a model for other parallel support systems in this category.

MICL — is a Machine Independent Communication Library used by the PeGAsuS programming environment at GMD. MICL is based on a port concept. Application's processes are equipped with several ports, each port owning one or more entries. Each port entry can be connected to port entries of other processes. The library, written in ANSI *C*, provides communication functions that act as an interface between the PeGAsuS kernel and machine-dependent communication systems. There are MICL versions for various parallel machines (transputers, Alliant, iPSC) and networks of workstations (SUN, RS6000). MICL provides the following functions:

- WritePort – writes data to a port entry.
- ReadPort – reads data from a port entry.
- InitPortBuffer – creates a buffer for a specified port.
- StopPortBuffer – deletes a buffer of a specified port.
- InitPortASync – makes a port read operation asynchronous.
- StopPortASync – cancels the asynchronous read property on the specified port.
- Synchronise – synchronises all processes possessing ports connected to the specified port.

MICL does not support dynamic process creation and termination. Also, the topology of the communicating processes (PGA islands) is fixed during the application's execution. It is specified with a special configuration language and setup by a master process, defined by their PGA computation model.

PVM — The Parallel Virtual Machine is a software system that enables a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource [90]. The individual machines may be: shared or private memory processors, vector super-computers, or scalar workstations, which may be interconnected by a variety of networks.

The operation of PVM is based on the concept of application components. Usually the term implies a phase or portion of an application that is embodied in a subroutine. However, PVM is a coarse-grained environment targeted at applications that are collections of relatively independent programs. Therefore, a PVM component corresponds not to a phase in the traditional sense, but rather to a large unit of an application. From the system point of view, a component corresponds to an object file that is capable of being executed as a user-level process.

The system comprises libraries of *C* and *FORTRAN* functions to perform process initialisation and termination, communication, and synchronisation via barriers or rendezvous. The libraries act as an interface between application components and a communication kernel (the PVM *daemon*) which actually maps applications requests into the native machine parallel support. The following functions are available in the PVM library:

- Initialisation:
  enrol, initiate, initiateM, whoami.
- Information:
  pstatus, status.

- Sending:

  initsend, putbytes, putncplx, putndfloat, putnfloat, putnint, putnlong, putnshort, putstring, snd, vsnd.
- Receiving:

  probe, probemulti, rcv, rcvmulti, vrcv, vrcvmulti, rcinfo, getbytes,getncplx, getndfloat, getinit, getnlong, getnshort, getstring.
- Synchronisation:

  barrier, ready, waituntil.
- Termination:

  terminate, leave.

The user may optionally control the execution location of specific components. PVM transparently handles message routing and data conversions for incompatible architectures. It has been ported and tested on SUN3, SUN4, CRAY, Alliant, RS6000, TMC CM2, Intel iPSC & RX, Sequent, and Stardent systems.

PVM offers enormous portability for parallel applications across diverse architectures. However, the dependency on its communication kernel (pvmd) imposes a constraint for efficient porting of applications to specialised parallel architectures, such as transputers, that offer hardware mechanisms to implement most of the operations.

Systems like PVM have proven to be very effective in helping with programming and porting parallel applications among a variety of platforms. Nevertheless, a new approach, based on the object-oriented paradigm, is being adopted by the biggest contenders in the computing market. A new generation of object-oriented operating systems supporting distributed active objects is due to appear in the next couple of years. These new operating systems are being re-designed and re-implemented from the ground upwards, according to modern concepts such as microkernel and distributed object philosophies. They will bring powerful abstractions for distributed object-oriented computing, shifting the current program-centred processing concept to the more flexible concept of compound document processing [99]. There are two main model-streams for object-oriented compound documents: OpenDoc and OLE. OpenDoc has been defined by a consortium of large companies including Apple, IBM, Novell (USL), Sun, Xerox, and others. OLE, on the other hand, is solely Microsoft's brainchild. These two models are laying down the foundations for distributed and parallel object-oriented computing. They have already been drawing powerful market forces towards future de-facto standards.

Two different philosophies have been adopted in the underlying process control and communication models of Microsoft's Common Object Model (COM) and IBM's System Object Model (SOM). COM and SOM differ in basic aspects such as language dependence (COM is strongly biased toward *C++* whereas SOM is language neutral), and inheritance control (SOM supports "pure" inheritance and COM adopts an alternative mechanism, similar to *C++* virtual tables, called aggregation by Microsoft). Another important feature is their compliance with the Common Object Request Broker Architecture (CORBA), being designed as an open standard for distributed object-oriented processing (SOM is fully compliant).

This analysis of software alternatives for supporting parallelism demonstrates that only the latter offers the degree of portability and flexibility required by GAME applications. The adoption of a language and operating system independent strategy in GAME, determined the specification of a programming model and the implementation of a communication and task control module – the PEM.

6.3. GAME's Parallel Programming Model

One of GAME's basic principles is to offer facilities to port different GAs and PGAs to a variety of sequential and parallel architectures. In order to support applications exhibiting different degrees of granularity, GAME has adopted a rather pragmatic solution: an event-driven programming model based on active objects communicating via message-passing. This model is supported by a language and operating system independent module — the *Parallel Execution Module.* PEM implements an abstraction that supports multiple parallel computation models. It provides functions for process initiation, termination, synchronisation and communication.

GAME's parallel programming model defines a computer program as a *GAME application.* A GAME application co-ordinates the operation of processing agents, or active objects (from now on referred to as *components*), which provide the basic functionality for concurrent/parallel execution and inter-process communication. Applications can be sequential or parallel. A sequential application consists of a single component embodying all GAME modules (i.e. the graphic user interface, a genetic algorithm, genetic operators and the Virtual Machine). Conversely, parallel applications may be formed by a number of different components communicating via the PEM runtime support. GAME's programming model is inspired by concepts and ideas proposed in some distributed systems programming models such as ACTOR [1,49] and ANSA [93]. It presents the following characteristics:

- GAME applications consist of a collection of components – typically the graphic user interface, one or more genetic algorithms, genetic operators and the Virtual Machine – executing sequentially, concurrently or in parallel.
- A component size is determined by the user (the application developer, in this case) and may be as large as the entire application – sequential application – or as small as an application function (or procedure).

- Components may share the same memory space (*local* components), the same processor, or execute in a different processor or machine (external components). Local and external components can typically coexist in parallel applications. The application and the actual distribution of its components are dynamically controlled by the components themselves.
- The number of components working in the same application in a given time is not fixed. Local or external components may be dynamically added or removed from an application by simply starting or terminating them.
- Components are interconnected by a non-fixed number of bi-directional communication channels.
- There is no fixed interconnection topology, allowing applications to be configured as rings, cubes, trees, etc.
- Components communicate asynchronously or synchronously via message-passing. They have the ability to buffer messages in their *mailboxes*, queuing them to be collected and processed later.
- Messages are special objects created by sender components, containing specific commands and data elements to be processed by receiver components. Messages may be replied with status and other information resulting from the execution of the command it carries.
- Messages are treated as events to be processed by a user-defined function, which is called by PEM whenever a new message is collected from the component's mailbox.

One may immediately conclude, from the above list of characteristics, that GAME's programming model is entirely independent of genetic algorithms' particularities. In fact, it defines a general-purpose parallel programming model and supporting system, that could be used to implement different types of parallel applications.

6.3.1. The Messaging Sub-System

Two techniques for exchanging objects between applications' components have been used in object-oriented programs [22,41]: Object Data Base Management Systems (ODBMS) and object flattening. The most flexible and comprehensive technique is based on the use of ODBMS [17,18,19,31,50,54,56]. In these systems, shared objects are deposited into object-oriented data bases as "persistent" objects. Such data bases control access rights to these objects and provide mechanisms to keep them updated. Objects maintained in an ODBMS are loaded on demand in the memory space of the requesting component, and any hardware/operating system particularity of the component's host machine is dealt with by the ODBMS. The application component only needs to know the "name" of the required object in order to request access to the data base. The major drawback of this approach is the overhead introduced by the ODBMS itself. Persistent objects are, in general, kept as files on disk and non-standard protocols or languages are used to communicate with the data base system.

The second technique is called object flattening [55,57] because objects are "flattened" into a data stream before they are actually transported to the receiver component. Because only data elements held by the object are transferred, the type (or name) of the flattened object must be known by the receiver component, which will then assemble a copy of the original object. The receiver "feeds" the data stream into a local instance of an object of the same type, making a copy of the sender's object. These two techniques could be compared with the methods used for passing arguments to functions. ODBMS could be seen as "calling by reference", and the flattening technique as "calling by value". In the first case, the same object, maintained by the data base, is shared by all application components. In the latter, a local copy of the original object is made by the receiver component. If the second object is modified, the sender must be notified in order to have its copy updated, if necessary. The major advantage of the flattening technique is its ability to use any communication system offered by the underlying operating system. Since objects are flattened into ordinary data streams, any data transfer system could be used to send and receive data streams between two components. This technique is adopted in GAME since it imposes no restrictions on the hardware/operating system platform (provided a communication mechanism is present), and can be easily implemented. Another important aspect is the advantage in the overall performance this mechanism presents, compared with current implementations of ODBMS.

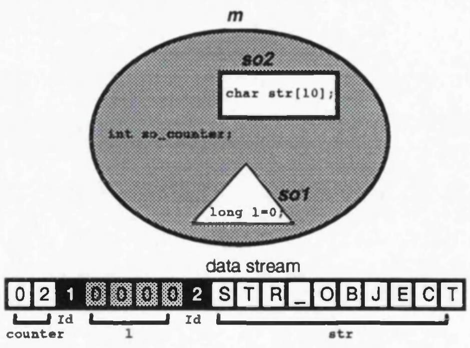
GAME's messaging system defines a class of objects capable of being flattened into a data stream, or sequence of bytes. This mechanism permits complex objects to be passed (or copied) across GAME Components using standard inter-process communication support offered by non-object-oriented operating systems, or hardware communication devices. The GameStreamObject class, shown in Figure 6.1, defines a virtual destructor and three public member functions: getObjectLength, describeObject and assembleObject. The getObjectLength member function returns the length of the object's data stream expressed in number of bytes. The returned value represents the sum of the length of all objects held by a GameStreamObject instance. If a given GameStreamObject contains other GameStreamObjects, their getObjectLength member functions are called in sequence, until the innermost object returns its data stream length. An object data stream length is used by the programmer to allocate a continuous array of bytes, and pass its address as the argument to the describeObject member function. This function "fills" in the array of bytes with the object's data elements. If other GameStreamObjects are contained by the first object, their describeObject member functions are also called in sequence, until the innermost object returns its data stream. Once the object finishes filling in the data stream, it returns a pointer to the next available memory position in the byte array.

Figure 6.1 - The GameStreamObject base class

```
class GameStreamObject
{
public:

  virtual                 ~GameStreamObject              (void);

  virtual LONG            getObjectLength         (void) = 0;
  virtual BYTE*           describeObject          (hOS)  = 0;
  virtual BYTE*           assembleObject          (hOS)  = 0;
};
```

The assembleObject member function performs the inverse task. It receives a pointer to an already filled data stream, with the address of its first byte. The contents of the data stream are then copied into the internal objects in the same sequence used to fill in the data stream. At the end of the assembly operation the object that received the array of bytes is an identical copy of the object that produced the data stream. This mechanism imposes only one condition in order to work properly: the receiver GameStreamObject must be of the same type as the sender, and therefore knows how to assemble the data stream. Figure 6.2 shows an example of a stream object (*m*) and its data stream representation. This particular instance of the GameStreamObject class holds two objects: a long type data element *l* (with value 0) and an array of ten chars with the string "STR_OBJECT". The flattened data stream containing both objects is also presented. It is headed by a counter that indicates the number of objects it contains. Each object also has an identifier (*Id*) prefix, which is only known by the object's flattener and assembler member functions. This explains why only stream objects derived from the same abstract type are capable of rebuilding (or assembling) a copy of the object that produced the data stream.

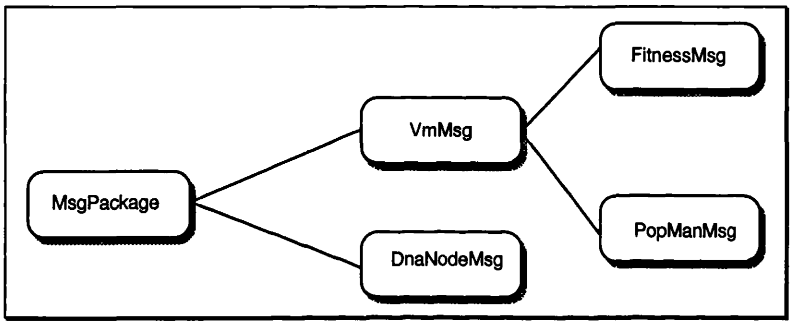
Figure 6.2 - GameStreamObject example



GAME Component objects interact by exchanging commands and data transported by a special class of objects called *message package*. A message package object is an instance of the MsgPackage class, which inherits its properties of data flattening from the GameStreamObject class. Message packages are capable of carrying data elements, commands and execution status. Data elements may be any native *C++* data type such as chars, integers or doubles, or any user-defined object derived from the GameStreamObject class. The standard MsgPackage class provided by GAME can carry a command, a status word (both integers), a char string and two doubles. One of the doubles can be used as four integers, eight chars or as combinations of integers and chars up to the length of a double.

The MsgPackage class is the base for more specialised classes (see Figure 6.3) such as the Virtual Machine message package class (VmMsg), that can carry complex data types like Individuals, DnaNodes and DataUnits.

Figure 6.3 - Message package class hierarchy



All MsgPackage derived classes include, at least, the following member functions:

- storeMessage – called by the sender to insert a command word into the message package object.
- recallMessage – called by the receiver to read the command word brought into the message package.
- storeStatus – called by the receiver to store a status word indicating the result of the execution of the message package command.
- recallStatus – called by the sender to verify command's execution status. Since the command word is always available, the sender does not need to keep a log of the commands sent (this helps with error recovery mechanisms).
- storeByte, storeWord, storeDouble and storeString – called by the sender to pass arguments of commands to the receiver, which may also call them to return information to the sender.
- recallByte, recallWord, recallDouble and recallString – called by both receiver and sender to read information contained into the message package object.

6.4. The Parallel Execution Module

Some basic facilities for inter-object information exchange found in the Parallel Execution Module design have been inspired by distributed object-oriented systems such as PRESTO [12] and DoPVM [44], and recent distributed object computing specifications like IBM's SOM and Microsoft's COM.

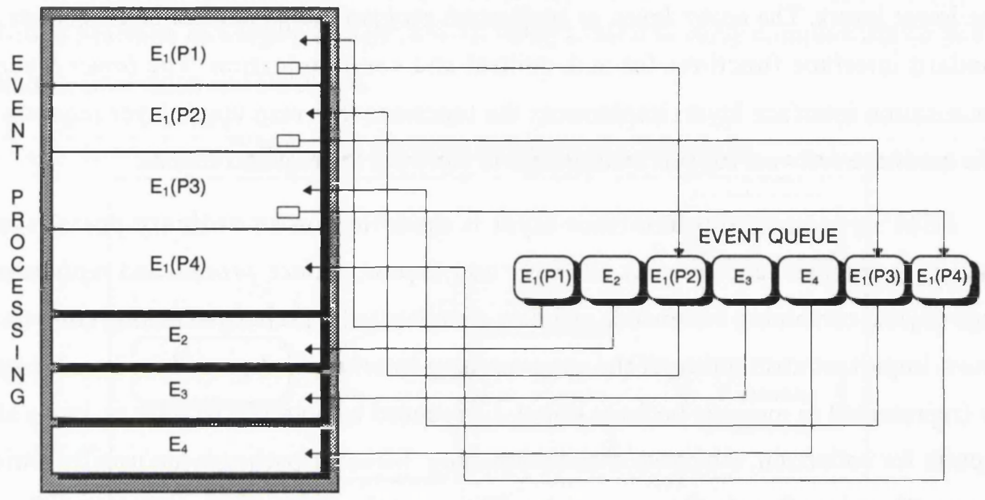
The Parallel Execution Module comprises two layers, as shown in Figure 6.6: the upper and the lower layers. The *upper layer*, or application programming interface layer, defines a set of standard interface functions for task control and communication. The *lower layer*, or communication interface layer, implements the functions that map upper layer requests into specific hardware/software support mechanisms to carry out the required actions.

PEM's programming interface layer is modelled on an ordinary postal system abstraction. It provides functions that allow the user to *post, collect, process* and *reply* message package objects containing commands and data elements to be exchanged among components. The most important abstraction of the programming interface is the *mailbox.* It concentrates *events* (represented as message package objects), produced by a variety of sources, into a single entry point for collection, distribution and processing. Message packages are usually retrieved from a mailbox in a first-in-first-out order. They may be "stamped" with user defined *Id* numbers, and retrieved by the receiver, in any order, based on the Id number.

The operation of PEM is entirely based on the event-driven philosophy. One of the most import characteristics of event-driven programs is the absence of a main thread of control. The parts that form a program (functions and procedures) are activated by the occurrence of asynchronous events. Usually a central routine, or dispatcher, maintains a list of known events and sub-routines to be called on their occurrence. Therefore, a component may receive events produced locally and externally. In the first case a component posts message packages to itself, whereas in the second case, message packages are received via the communication and the user interfaces. A typical problem with event-driven programs appears when a long piece of code is executed by a sub-routine. In such cases, the event queue may overrun and event messages could be lost. A common technique to avoid this type of problem is to split the long code into smaller parts. To ensure the continuity of the execution by all the parts, messages produced by each part are inserted into the event queue, indicating the next part to be executed (see Figure 6.4). This technique may also be used to introduce concurrence in a sequential program.

The management of mailboxes is, therefore very important for efficient PEM implementations. Two approaches may be used for mailbox queue monitoring: continuous monitoring and timed monitoring. The first approach requires the implementation of an infinite loop that permanently verifies the number of messages in the queue (and calls the central dispatcher whenever a new message package arrives). This implementation is most appropriate for pre-emptive multitasking operating systems. The second approach is preferred for non pre-emptive systems. The queue is inspected at certain times only, in general determined by a timer, allowing the operating system and other programs to acquire control of the CPU. In this case however, special mechanisms should ensure that the event queue will be processed at some stage, for instance when overrun is imminent, or when events requiring urgent treatment arrive.

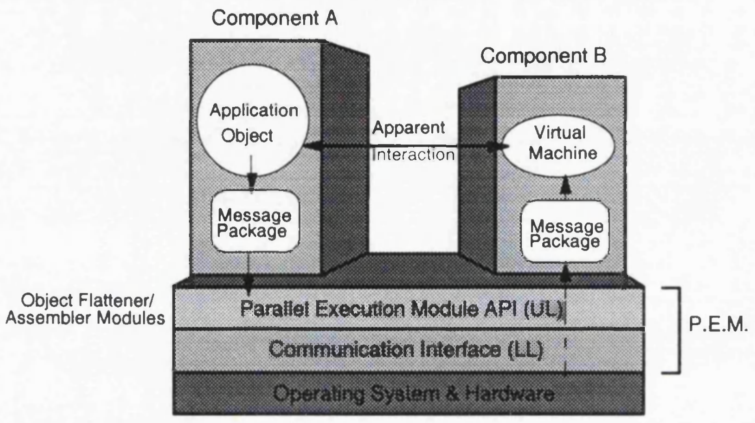
Figure 6.4 - Event driven-processing



PEM defines events in terms of asynchronous and synchronous message packages. A transmission mode for a message package is specified by calling the PostMail function with one of the following three options: NOREPLY, REPLY and WAITREPLY. The NOREPLY and REPLY options are used for asynchronous transmissions. In the first case, a message package is inserted into the receiver's queue, and the control returned to the sender. The second form requires the same message package to be returned to the sender (at some stage later) with status and/or data resulting from its command execution. The control is resumed to the sender as soon as the message package is inserted into the receiver's queue. The WAITREPLY form implements a synchronous transmission, meaning that the execution in the sender is blocked until the same message package returns. Its arrival at the receiver determines immediate processing of the message queue, up to the point where it is inserted.

Typically, when a message package is posted, the sender application's interface functions translate it into a data stream and pass it on to a lower-layer function. This will then use whatever operating system or hardware communication support to send the data stream to the receiver component. An incoming data stream is translated back into a copy of the original message package and inserted into the component's mailbox. Communication between components sharing the same process memory space (local components) are not translated into data streams, but immediately delivered to the receiver's mailbox. From the user's point of view, both methods are transparent, working as if sender and receiver components were interacting directly (see Figure 6.5). Variations on the way messages are posted permit synchronising sender and receiver(s) as well as broadcasting message packages. Global synchronisation is also supported, based on an adaptation of the "barrier" model for an event-driven implementation.

Figure 6.5 - Component's interactions and PEM

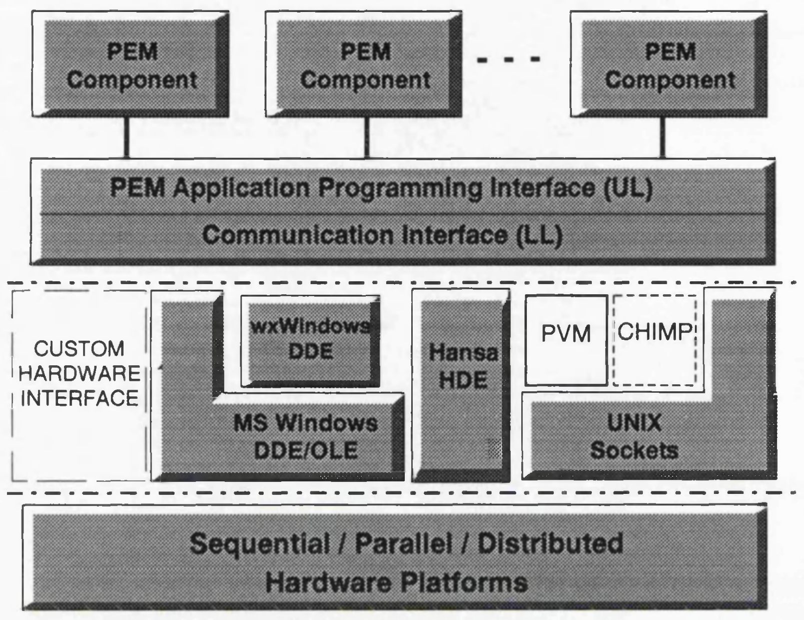

Component A
Component B
Application Object
Apparent Interaction
Virtual Machine
Message Package
Message Package
Object Flattener/ Assembler Modules
Parallel Execution Module API (UL)
Communication Interface (LL)
Operating System & Hardware
P.E.M.

Process control is supported by a set of functions that enable the user to dynamically start and terminate components, as well as open and close one or more communication channels. The complete set of interface functions defined by PEM is presented in the next section and described in detail in Appendix C.

The communication interface layer also defines a set of standard functions. It is intended that only the lower layer needs to be rewritten to port PEM (and GAME) to different platforms. Therefore, upper layer functions should be able to call the same set of lower-layer functions (e.g. SendSync, SendAsync, etc.) that will have different implementations according to the underlying operating system or hardware. Figure 6.6 shows how an application comprising several components is isolated from the host machine's operating system and hardware, through PEM's application programming interface (UL). Various alternatives for the implementation of the communication interface (LL) are also shown. This may directly rely on operating systems' low-level process control and communication support (UNIX sockets, RPC or Microsoft Windows DDE/OLE), or custom hardware interfaces (such as those implemented in language extensions or libraries). Another possibility is to implement PEM's lower layer on top of some more sophisticated software systems like PVM [8], HANSA-HDE [79], CHIMP [16] or wxWindows' DDE [87]. This alternative considerably reduces the complexity of PEM's implementation, but may introduce few undesirable performance constraints. PEM is currently available for the UNIX and MS-Windows platforms over wxWindows (which offers cross-platform implementation support based on the DDE protocol), with HANSA DDE and PVM versions under development.

A transputer-based version has also been implemented by TELMAT, one of the PAPAGENA project partners.

Figure 6.6 - PEM layers and low level support systems

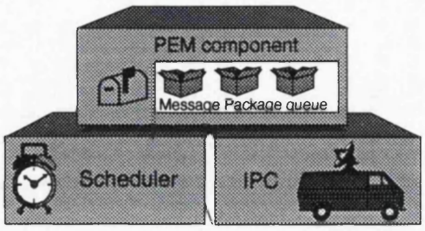

PEM Component
PEM Component
- - -
PEM Component
PEM Application Programming Interface (UL)
Communication Interface (LL)
CUSTOM HARDWARE INTERFACE
wxWindows DDE
MS Windows DDE/OLE
Hansa HDE
PVM
CHIMP
UNIX Sockets
Sequential / Parallel / Distributed Hardware Platforms

6.4.1. PEM Implementation

The implementation of the Parallel Execution Module defines a component in terms of three main objects: the *PEM Component object*, the *Ipc object* and an optional *Schedule object*, as shown in Figure 6.7.

PEM Component — is the principal processing agent, or *component* (as it is called in GAME's programming model). It implements the application programming interface layer specified in the PEM design. There are two types of PEM Component objects: *external* and *local* components. An external PEM Component is an independent program (or process), which can be started automatically by other PEM Components or manually by the user. It maintains an *Ipc* object and, in some implementations, a *Scheduler* object. Conversely, local PEM Components can only be started by external or other local components. They do not possess private memory nor Ipc or Scheduler objects. However, from the implementation point of view, external and local PEM Components perform the same task: they have the same user-defined function to process incoming message packages. They only differ in the way they are handled by compilers (through a macro switch), linkers and loaders. Typically one external component (the parent) may start several local components (child) and share its resources (the Ipc object, the Scheduler and its memory). Local components are, in general, implemented as dynamically linked library modules or local threads. They may also start other local or external components, but are not able to share resources since they are not resource owners.

Figure 6.7 - A PEM "external component"



The PEM Component object implements the application programming interface layer through groups of functions, as presented in Table 6.1. It maintains three main data structures: a message package queue, a message package pool and a data base of known components. Incoming message packages are delivered to the component's mailbox, implemented as a first-in-first-out queue. The mailbox pre-processes some commands (e.g. QUIT) and passes them on to a user-defined function (ProcessMail), where system and user commands are processed. The pool of message packages maintains "free" user-defined message packages. In the component initialisation, the user must provide a prototype of the particular message package object, to be replicated into the pool. Message packages are allocated by the Ipc object to assemble incoming data streams before delivering them to the mailbox. After being processed, message packages are released back to the pool, and are later re-used to assemble new messages. The number of copies of message package objects held in the pool is determined by the user. It should be sufficient to support the maximum number of message packages that a component can accumulate in its queue before processing. A component might receive only one type of message package object, but may send several other types according to the different components it is connected to.

Table 6.1 - PEM Interface functions

Control & Execution	***Communication***	***Synchronisation***
StartComponent	OpenConnection	PostMail
TerminateComponent	CloseConnection	
	PostMail	WaitMail
ProcessMail	HasMail	
ProcessReply	CollectMail	
	ReplyMail	

The third data structure, the component data base, maintains detailed information about child components and connections. Names of components, their communication ports and hosts are kept together with their types (local or external), connection status (no connection, active or inactive) and communication channel identifier (required by the Ipc object). New child components are registered with the data base, which returns a unique handle for the component. This handle is subsequently used to call other functions to obtain information about the component, or post messages. The first two entries in the data base are reserved for information about the component itself and its parent (if it is started by another component).

IPC Object — implements the communication interface layer specified by PEM. A standard set of functions, listed in Table 6.2, is defined to interface with the upper layer. These functions allow the upper layer to request lower layer services independently of any particular implementation of the Ipc object. To further facilitate portability, the object-oriented implementation of both the PEM Component and the Ipc objects permits easy substitution of the latter. Ipc objects are created by PEM Components at run-time. This strategy promotes the integration of new, and possibly different, implementations of these objects into PEM Components. Customised versions of Ipc objects, inherited from its base *C++* class, can immediately replace any other version after re-linking the original application object codes. This strategy also opens up the possibility for more than one Ipc object (with diverse implementations) to be created by a PEM Component, permitting application components to be distributed in heterogeneous computing environments. Besides providing communication services, Ipc objects are also responsible for process control activities such as starting, terminating, loading and unloading external and local components.

Table 6.2 - IPC interface functions

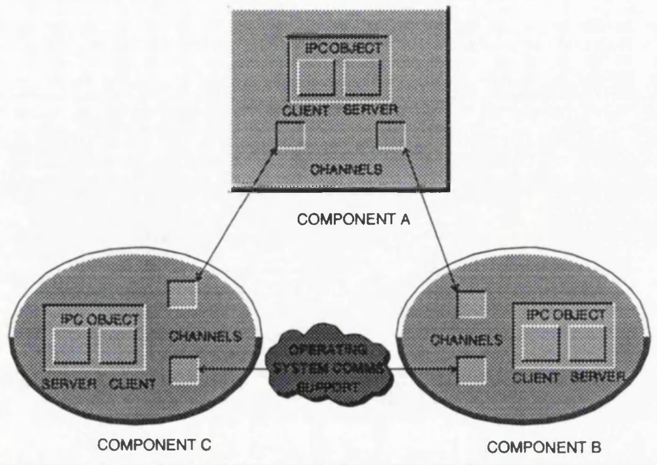
Control	**Communication**
StartLocalComponent	OpenConnection
StartExtComponent	CloseConnection
TerminateLocalComponent	ConnectionStatus
TerminateExtComponent	SendSync
CreatePort	SendAsync
	Reply

The Ipc object operation is based on three other objects it creates: a *client*, a *server*, and *channel* objects (Figure 6.8). The server object is characterised by its communication "port". This is an alphanumeric string and is usually registered with the host operating system, enabling client components to request connections. The client object is called by the Ipc object to request connections to other components. The name of the server communication port and the name of the component (since more than one local component may share an Ipc server) must be provided. Optionally a host name may be given, if the other component is not located in the same machine.

When a server receives a request for connection, it verifies if the name accompanying the request is in the list of local components. If not, the connection is refused, otherwise it requests the Ipc object to create a channel object, and accepts the connection request. On the creation of a channel object, the operating system (or any underlying communication support) is notified.

When the client that requested the connection receives the positive reply, it also asks its Ipc object to create a channel. The implementation of channel objects will then ensure that a bi-directional link between both channels is established.

Figure 6.8 - Application components and their communication objects



There is no limit for the number of connections maintained by an Ipc object. Communication channels are identified by handles kept in a list of connections. The handles are also returned to the upper layer function that requested the creation of a connection. It is stored in the component data base entry that describes the other components, and passed back to the Ipc object when other services are requested (e.g. send message) for that particular connection.

When a channel object is created, it is informed about the PEM Component that "owns" it. This allows the channel to directly deliver an incoming message package to its owner mailbox. In fact, the channel has to request a free message package from its owner's message package pool, before assembling it with the received data stream and inserting it into the message package queue.

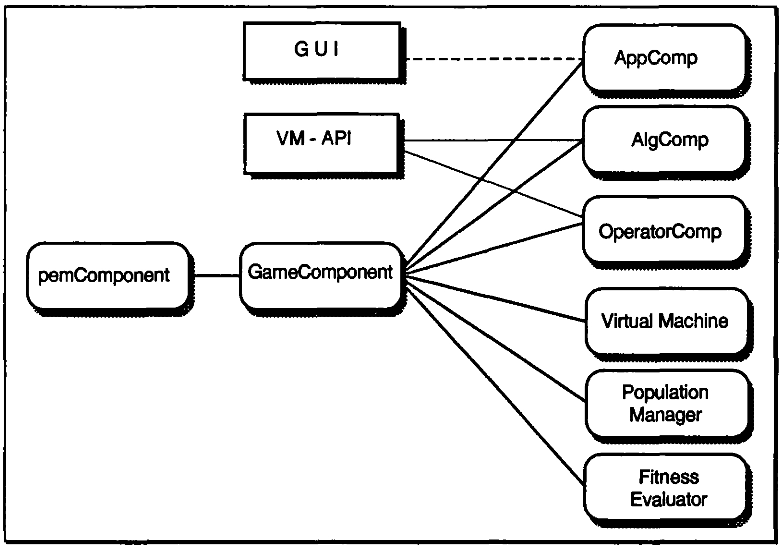
Connections between components in the same process (i.e. sharing the same Ipc object) are treated in the same way, but the flattening/assembling stages are skipped since no operating system (or hardware) intervention is needed.

Scheduler — is an optional object mostly used when local components are implemented as dynamic link libraries – thread-based implementations are usually controlled by operating systems. It controls the concurrent execution of components sharing resources in the same process, i.e., one external component (the parent) and all its the local components. Its simplest implementation is based on a round-robin, non pre-emptive task allocation policy. The scheduler maintains a list of local components and continuously invokes their mailboxes in sequence, granting them an opportunity to process their message queues.

Non pre-emptive multitasking operating systems like Windows 3.x, may require a timer to give the operating system opportunity to acquire control of the CPU. In such implementations, after all components have had a chance to process their message queues, the timer is started and the scheduler passes the control to the operating system. When the timer goes off, the scheduler receives the CPU control and executes one round of mailboxes calls, before restarting the timer. This approach is also suitable for developing and debugging parallel applications on uniprocessor platforms. Since there are no differences between external and local components, from the application developer's point of view, a parallel application could be fully tested (including message passing) in a uniprocessor platform with the help of the Scheduler and local components. In the final (parallel) version, the former local components could be simply re-compiled as external components, and distributed among the processors of the target parallel machine (or distributed system).

The C++ implementation of GAME's Parallel Execution Module defines a class framework (see Figure 6.9) that has the pemComponent class in its base. The pemComponent class provides the functionality specified by PEM's upper layer programming interface. It is inherited by the GameComponent class that further specialises some of its functions, and includes some others to support integration with the graphical user interface and the monitoring control system. Base classes of customised implementations for applications, algorithms and genetic operators are also defined for both local and external types of components. These classes are intended to be used as templates, to be inherited or directly modified by the application developer as explained in see Chapter 7.

Figure 6.9 - GAME components class hierarchy



GAME's programming model and PEM's design allow maximum flexibility for diverse application requirements and cross-platform implementations. Different topologies combining local and external components can be easily set up through a configuration file or alternatively via the user interface. The configuration file example shown in Figure 6.10 is divided in four configuration sections: application, graphic monitor, algorithms and genetic operators. Other sections may include genetic data structures and Virtual Machine configurations. Every section is sub-divided in two parts: component's configuration parameters and component's connections. Configuration parameter entries are specific for each section type. Algorithm parameters, for instance, may include the number of generations and size of their local populations, whereas operator parameters are specific for each particular genetic operator (e.g. mutation and crossover rates).Common to all components are their type (local or external), file name (and path). External components also require a port name (an alphanumeric string). A component wishing to connect to another component in the same application needs only to obtain its port name from the configuration file. The connection configuration sub-sections specify the initial topology of the application, which may be dynamically modified at run-time. The StartComponent and the ConnectToComponent entries specify names of other sections that fully describe a component to be started or connected (as indicated by the dashed lines).

Figure 6.10 - A GAME configuration file example

```
;*****************************************************************
;*                GAME Application - Configuration File
;*****************************************************************
;* Application configuration
;*
[Application]
ComponentPort=4045
TimerInterval=1000
;
; The next two entries specify section names describing
; algorithms to be started or connected
;
StartComponent1=Algorithm1
StartComponent2=Algorithm2
;*
;* Graphic Monitor configuration
;*
[Graphic Monitor]
ComponentType=
ComponentName=C:\GA_App
ComponentPort=4045
ComponentHost=
;*
;* Algorithms Configuration
;*
[Algorithm1] <-
;*
;* Configuration parameters
;*
ComponentType=Local
ComponentName=C:\Gen_Alg
ComponentPort=
ComponentHost=
MaxGenerations=50
PopulationSize=10
ConvergenceRate=0.005
TimeOut=
;*
;* Connection configuration
;*
StartComponent1=Selection
StartComponent2=Crossover
StartComponent3=Mutation
;
[Algorithm2] <-
TimerInterval=100
ComponentPort=1234
ComponentType=External
ComponentName=C:\Gen_Alg
ComponentHost=
MaxGenerations=50
PopulationSize=10
ConvergenceRate=0.005
TimeOut=
StartComponent1=Selection
StartComponent2=Crossover
StartComponent3=Mutation
ConnectToComponent=Algorithm1
;
; Operators Configuration
;
[Selection] <-
ComponentType=Local
ComponentName=C:\TruncSel
TruncationRate=0.7
;
[Crossover] <-
ComponentType=Local
ComponentName=C:\Oneptcros
CrossoverRate=0.6
;
[Mutation] <-
ComponentType=Local
ComponentName=C:\IntMut
MutationRate=0.05
```

6.5. PEM Application Program Interface

The Parallel Execution Module defines two application programming interfaces. The first API contains functions of PEM's upper layer – the user interface layer, whereas the second API contains the interface functions of the lower layer. The functions of the latter are not directly accessible to the application developer, but they are important for porting GAME to other operating systems or custom hardware communication support.

Most of the functions of PEM's upper layer API take a handle to an entry in the component data base as one of their arguments. A data base entry is created when a new child component is started or a new connection is opened to an already running component. In both cases the ComponentDescriptor structure is required (see Figure 6.11). This structure contains information used by PEM to start a component or establish a connection to a component. The user fills in five fields: *component_name, port_name, connection_type, connection_host* and *ipc_buf_sz*. The last two are optional and assume the local host and a buffer of 1 Kbytes, by default.

Figure 6.11 - The component data base entry

```
struct ComponentDescriptor
{
         char*   component_name;       /* path and file name of the component */
         char*   port_name;            /* Server port name                    */
         char*   connection_host;      /* Host where the server is running    */
         PCTYPE  connection_type;      /* Type of connection (LOCAL/EXTERNAL) */
         HANDLE  hIpcComp;             /* Handle provided by the IPC object   */
         HANDLE  hIpcConn;             /* Handle to the IPC channel object    */
         WORD    ipc_buf_sz;           /* Size of the communication buffer    */

         ComponentDescriptor            (void);
         ComponentDescriptor            (const ComponentDescriptor FAR&);
virtual  ~ComponentDescriptor(void);
virtual  ComponentDescriptor& operator= (const ComponentDescriptor FAR&);
};
```

6.6. Summary

The first section of this chapter reviewed three common software strategies used to support parallel programming, namely: language-based, operating system-dependent and language/operating system independent parallel support. The discussion of each one of these strategies was based on currently available programming languages and software systems, and permitted to assess their strengths and weaknesses. This study led to the definition of GAME's parallel programming model and its implementation support – the Parallel Execution Module.

GAME's programming model was then introduced, and its message-passing architecture described. The programming model defined for the system is highly flexible, supporting the implementation of complex parallel applications, based on a language/operating system independent strategy. This alternative was elected since it offers the required degree of portability for applications created with GAME.

The explanation of the messaging sub-system, and its implementation, was followed by the description of the Parallel Execution Module. The design of PEM's internal structure, which comprises two independent implementation layers (the upper and the lower layers), clearly demonstrated the concern with application and system portability. The definition of an application programming interface for the upper layer ensures platform independence for applications. In the same way, an API between the upper and lower layer facilitates the porting of GAME, without affecting applications' design and implementation.

Finally, it is important to stress that GAME's programming model and its underlying support are, together, a powerful tool for the creation of parallel applications. They have been designed to cater for a much broader range of parallel applications and therefore, are not limited to supporting genetic algorithms.

Chapter 7

The GAME Libraries

The GAME Libraries comprise two principal groups of C++ class libraries. This chapter describes the organisation and implementation of the Genetic and Service libraries.

7.1. Introduction

Development environments are complex software systems designed to provide as much flexibility and reliability as possible for application developers. Among the standard set of tools generally available in a programming environment, a collection of runtime libraries constitutes one of the most important assets for software developers. Runtime libraries are normally supplied with a variety of general-purpose functions that help to reduce the time required to implement applications, freeing the programmer to concentrate on the details of the application design. Operations such as formatted input and output, floating-point and complex arithmetic, graphic display, and interactions with the operating system's services are now commonly available in any programming environment. More recently, a new generation of "class libraries" has appeared, based on the object-oriented paradigm. Such libraries offer even more flexibility for the programmer, by allowing the customisation of one or all its member functions and member data, via *inheritance* and *overloading* mechanisms.

Programming environments for applications based on genetic algorithms must also offer a comprehensive collection of libraries, possibly containing parameterised versions of standard sequential and parallel GAs and genetic operators. According to the general model of a GA-based application, it is possible to define a set of parameterised GA libraries. Typically, an application is organised in three hierarchical levels:

- the Application Domain level, comprising domain-specific applications (e.g. finance, telecommunications, etc.);
- the Algorithm Class level, with a variety of algorithms grouped into categories (scheduling, routing, etc.) to be used in applications; and

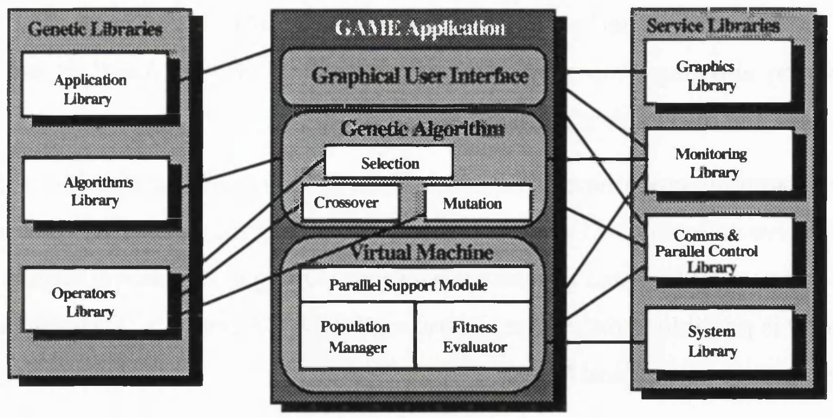
- the Genetic Operator level, encompassing a number of genetic operators to be incorporated into different algorithms.

A second group of libraries containing auxiliary functions to be used in the implementation of GAs and their genetic operators may also be considered. This "auxiliary" group of libraries might include modules for encoding and decoding genetic structures (employing several techniques and alphabets) as well as platform-dependent modules such as random number generators, inter-process communication support, etc.

The GAME programming environment comprises two main library groups, namely the Genetic Libraries and the Service Libraries. These two library groups have been defined according to the structure outlined above. The Genetic Libraries provide parameterised versions of applications, genetic algorithms and operators for a variety of domains. The Service Libraries group, on the other hand, is the repository for the rest of the environment's modules. It comprises the graphic user interface library, the communication and parallel control library, the monitoring library and the system library. The latter contains the Virtual Machine, the genetic-oriented representation and various ancillary functions including string encoding and decoding, memory management, etc. Figure 7.1 shows the relationship between the modules of a typical GAME application and the various libraries of the environment.

The GAME Libraries have been designed to provide the maximum flexibility in combining their modules into algorithms and applications. Furthermore, they can be easily expanded with the inclusion of new modules.

Figure 7.1 - Application and libraries



There is, however, a fundamental difference between these two groups of libraries relating to the way their modules are constructed. The Genetic Libraries contain only GAME Components, i.e., modules that can be either stand-alone executables or dynamically linked (DLLs) with other DLLs or executables. Conversely, the Service Libraries are a collection of modules designed to be statically linked into executables or DLLs like GAME Components. This distinction stems from GAME's programming model, which requires GAME Components to be dynamically interconnected by applications.

Modules from the Genetic Libraries, although including stand-alone executable programs, still fit in the traditional concept of a library module since they cannot provide any useful functionality unless "connected" to other modules (DLLs or other executables). A genetic algorithm component (as an executable module), for instance, needs at least some genetic operators (either as DLLs or stand-alone executables) and the Virtual Machine to perform any useful computation. The major advantage of this library approach is the ability to construct entirely new applications and genetic algorithms without needing to re-compile or re-link any piece of object code. A simple configuration file, as shown in Chapter 6 (Figure 6.10), provides the necessary information to put together parameterised genetic operators, algorithms (= Virtual Machine + genetic operators + parameters) and applications (= algorithms + user interface + parameters).

The next sections discuss in more detail the parameterisation of genetic algorithms and describe some modules of GAME's Genetic and Service Libraries.

7.2. Parameterising GA Applications

A genetic algorithms-based application is generally parameterised in terms of its GA setup. A typical GA setup includes parameters like population size, maximum number of generations, convergence rate, crossover and mutation rates, to cite only the most common. Let us consider two parallel applications, one comprising several instances of the same algorithm (here meaning that each algorithm contains the same set of genetic operators), but with different setups; the other application comprising several instances of different algorithms (defined as containing distinct sets of genetic operators), with possibly different setups[6]. This example suggests another level of parameterisation for GA applications, based on their composition. This means that a parallel application might take genetic algorithms as one of its parameters, and each genetic algorithm might, in turn, take as parameters a list of genetic operators. It is interesting to note that algorithm parameterisation is also applicable to sequential GAs.

However, some genetic algorithms designed to solve certain classes of problems may not "accept" all possible genetic operators. For example, the Genetic Edge Recombination (GER) operator [100] is a special type of crossover used in routing problems such as the travelling

[6] The actual advantages and disadvantages of both alternatives have been briefly discussed in Chapter 2 and the discussion of this subject is beyond the scope of this thesis.

salesperson problem (TSP). It cannot be applied to other classes of problems. Conversely, any GA designed for solving problems of the TSP class may seamlessly use a GER operator.

The parameterisation of an application in terms of its composition is immensely facilitated by GAME's programming model. By defining genetic algorithms and their operators as GAME Components, it becomes extremely easy to combine genetic operators into new algorithms and use them in applications. The restriction imposed by the possible combinations of genetic operators, classes of algorithms and application domains is controlled through the introduction of two different types of parameters, as discussed in the next section.

7.3. The Genetic Libraries

Modules of the Genetic Libraries can accept two types of parameters: configuration parameters and runtime parameters. A *configuration* parameter is defined before the compilation of the library module. Therefore, valid configuration parameters for an application domain will admit only certain classes of algorithms, which in turn, will take only certain types of genetic operators. This mechanism may look restrictive for configuring new applications since it relies on pre-defined classes of genetic algorithms and operators. However, it protects the user from combining components that are not compatible or are unsuitable for solving a problem. If a genetic operator component is not a member of the class of components accepted by an algorithm, its connections will be refused by the latter. This parameterisation mechanism also introduces the concept of *component templates*, which defines applications as place holders for genetic algorithms, and these as place holders for genetic operators. Hence, the larger the number of pre-defined classes of genetic algorithms and operators "configured" in a particular component, the greater will be the potential for creating new applications and algorithms via permutation.

Runtime parameters, on the other hand, are meant to be passed on to already compiled GAME components, either before or during their execution. Initialisation runtime parameters are generally passed to a component before its execution – normally as command-line arguments or in a configuration file. For instance, the network of connections that forms a GAME application, as specified in the application configuration file, is considered as a set of runtime parameters. Other runtime parameters that can be modified during the execution of a component include: number and type of display charts, crossover and mutation rates, etc.

The Genetic Libraries group comprises three libraries: the Applications Library, the Algorithms library and the Operators library. This organisation mirrors the application structure and is supported by a framework of four *C++* classes. The GameComponent class (shown in Figure 7.2) is the base class of the framework and provides the basic functionality to initialise a component and access the services of the Parallel Execution Module. Every time a

GameComponent-derived class is created, the base class constructor executes an initialisation sequence which includes reading the application configuration file to obtain its runtime parameters. This operation may determine the creation of other components such as genetic algorithms and operators. By encapsulating this type of operation in the GameComponent class, the framework ensures an automated and consistent initialisation behaviour for GAME Components. At the same time, it relieves the user from dealing with this type of ordinary operation. Nevertheless, the user may call, at any time, functions such as GetEntryToken, to read a user defined configuration file entry, or StartComponent, to create a new genetic operator.

Figure 7.2 - The Game component class

```
class GameComponent : public pemComponent
{
public:
                GameComponent          (void){}
                GameComponent          (char FAR*, char FAR*, pemIpc FAR* =0);
virtual         ~GameComponent         (void);

protected:

// Member data
        char    cfg_file[256];                /* configuration file name    */
        char    cfg_section[80];              /* configuration file section */
        WORD    hGraphMon;                    /* Graphic Monitor handle     */
        int     gc_handle[MAX_COMPONENTS];    /* Array of handles           */

// Member functions
virtual void    InitComponent          (void);
virtual HANDLE  StartComponent         (ComponentDescriptor FAR&, char* =0);

virtual void    ParseCmdLine           (void);
virtual WORD    GetEntryToken          (char*, char*, WORD);

virtual HANDLE  AllocHandle            (void);
virtual void    FreeHandle             (HANDLE);
virtual void    Quit                   (void);
};
```

The AppComponent class, shown in Figure 7.3, further specialises the GameComponent class to offer the ability to incorporate (and initialise) a graphic user interface via the OnMakeUserInterface member function. It also overrides the GetEntryToken member function to look for application-specific entries in the configuration file. Although the AppComponent class is completely functional, the user may want to derive a new class from it and include a set of member functions to handle events generated by the graphic user interface.

Figure 7.3 - The Application component class

```
class AppComponent : public GameComponent
{
public:
                AppComponent          (int, char**);
virtual         ~AppComponent         (void);

protected:

// Member functions
virtual void    InitComponent         (void);
virtual void    OnMakeUserInterface   (void);
virtual HANDLE  StartComponent        (PCTYPE);

virtual WORD    GetEntryToken         (char*, char*, WORD);
virtual BOOL    OnReceiveMail         (MsgPackage FAR&);
};
```

The third *C++* class provided by the framework is the AlgComponent class (see Figure 7.4) which provides a basic template for standard genetic algorithms. It includes member data to store the ordinary parameters of a GA (PopulationSize, MaxGenerations, TimeOut and ConvergenceRate) and a handle (hVM) associated with the Virtual Machine component created by this algorithm. It also offers member functions to invoke genetic operator components to which it is connected. Since genetic algorithms and genetic operator components are not supposed to interact directly with the user, these classes offer no provision for a graphical user interface, as seen in the AppComponent class. All interactions with the user should occur via the interface provided by the application component (or the graphic monitor module).

Figure 7.4 - The Algorithm component class

```
class AlgComponent : public GameComponent
{
public:
                AlgComponent          (char FAR*, char FAR*, pemIpc FAR*);
virtual         ~AlgComponent         (void);

protected:

// Member data
  WORD          PopulationSize;       /* number of individuals in a pop. */
  WORD          MaxGenerations;       /* max. number of simulation cycles*/
  WORD          GenerationCounter;    /* current simulation cycle        */
  LONG          TimeOut;              /* simulation time limit - ms      */
  DOUBLE        ConvergenceRate;      /* simulation max. tolerance error */

  HANDLE        hVM;                  /* Virtual Machine handle          */

// Member functions
virtual MsgPackage& ProcessMail       (MsgPackage&);
virtual void    InitComponent         (void);
virtual BOOL    OnReceiveMail         (MsgPackage FAR&);
virtual WORD    GetEntryToken         (char*, char*, WORD);

// GA specific member functions

virtual void    Initialize            (void);
virtual void    MainLoop              (WORD);
virtual void    Select                (void);
virtual void    Reproduce             (void);
virtual void    Migrate               (void);
virtual void    Statistics            (void);
};
```

Finally, Figure 7.5 shows the OperComponent class, which provides the functionality to implement genetic operator components. Its member data include a storage space (*ActivationProb*) for the parameter that determines the activation of the operator, such as the mutation rate for mutation operators. It also contains a component handle that uniquely identifies the instance of the Virtual Machine component created by the algorithm that started this operator.

Figure 7.5 - The Operator component class

```
class OperComponent: public GameComponent
{
public:
                    OperComponent         (char FAR*, char FAR*, pemIpc FAR*);
virtual             ~OperComponent        (void);

protected:

// Member data
  DOUBLE            ActivationProb;       /* prob. for activating the operator */
  HANDLE            hVM;                  /* Virtual Machine handle            */

// Member functions
virtual void        Statistics            (void);

virtual MsgPackage& ProcessMail           (MsgPackage&);
virtual void        InitComponent         (void);
virtual BOOL        OnReceiveMail         (MsgPackage FAR&);
virtual WORD        GetEntryToken         (char*, char*, WORD);
};
```

The OperComponent class can be specialised to create a set of standard operators for the Operators library as listed in Table 7.1 below.

Table 7.1 - Standard components of the Operators library

Initialisation	***Selection***	***Crossover***	***Mutation***
Random	Roulette Wheel	Holland Crossover	Bit Flip
Super-Uniform	Truncated Roulette	Two-point	Creeping
Read From File	Deterministic Sampling	Uniform	
	Ranking		
	Reminder Stochastic		
	Tournament		

The three initialisation operators create the starting values for the population of genetic structures. The first operator initialises the population with random values, possibly chosen from a user-defined range. The Super-Uniform method is applied in binary string initialisation and it tosses a biased coin for each bit of each string of the population. Pre-defined values may be provided by the user through the Read From File operator.

The category of selection operators includes the traditional Roulette Wheel, which assigns to each member of the population a percentage of the population's total fitness that is proportional to the individual's own fitness. The Truncated Roulette is a variation of the roulette wheel that restricts the selectable population to some fraction of the best individuals in the current generation. The Deterministic Sampling method will always favour members of the population according to some pre-defined and constant rules. Such an operator may, for instance, take the best 70% of individuals and complete the population with copies of these individuals, again chosen according to certain fixed criteria (even randomness could be one such criterion). Ranking operators will select individuals after ordering the population. These operators generally apply some form of fitness normalisation before sorting the population. The Reminder Stochastic method copies individuals to the next generation according to their integer expectation, or apportionment, in relation to the total fitness of the population. This often leads to some empty places in the new generation, which are filled in by applying the traditional roulette wheel on the fractional parts of the same individuals. Finally, Tournament Selection randomly chooses some number of individuals (often a pair) and select the best from this group. The operation is repeated until the population of the new generation is completed.

The most ordinary crossover operator, the one-point or Holland crossover, divides a genetic string in two parts by randomly choosing a cutting point. It then swaps one of its parts with its counterpart taken from another string. The two-point crossover performs the same swapping operation, but with a section of the strings taken after choosing two different cutting points. The one-point crossover is, in fact, a particular case of the two-point crossover where one of the cutting points falls at one of the ends of the string. An entirely different approach is taken by the Uniform crossover operator. It essentially goes through a pair of strings and determines, on a bit-by-bit basis (by tossing a biased coin) whether or not to swap them.

The mutation operators, apart from the traditional Bit Flip, are in general problem dependent. The Bit Flip operator performs mutation by simply flipping (from 0 to 1, or vice-versa) a randomly chosen bit in the binary string representation. The methods used for real value representations are very diverse, but one usual implementation is the Creeping Mutation. It operates by slightly modifying the original value, adding or subtracting a constant taken from a list, or range, of possible values. A more drastic approach will simply replace the original value with an entirely new one.

A library of genetic algorithms, catering for sequential and parallel implementations of applications, would normally include some of the common GAs and PGAs found in the literature (see Table 7.2). The panmitic models are basically sequential GAs that can also be used in parallel applications, running as completely independent and isolated algorithms. The examples of the island and massively parallel models have been previously described in Chapter 2.

Table 7.2 - GAs and PGAs modules

Panmitic	*Island*	*Massively Parallel*
SGA	I-SGA	Fine-Grained PGA
	GAUCSD	Asparagos
pCHC	PGA	Cellular GA
	Distributed GA	DBGA
Genitor	Punctuated Equilibria	Fine-Grained PGA for Distributed Systems

7.4. The Service Libraries

The *C*++ classes provided by the Genetic Libraries to help with the creation of application, genetic algorithm and genetic operator components are only the front-end that hide the functionality implemented by the Service Libraries. Furthermore, this set of libraries makes available to the user various modules for programming graphical interfaces, monitoring execution and supporting genetic manipulations via the Virtual Machine and its API. The Service Libraries comprise a group of separate libraries, organised according to their functionality:

- Communication and Parallel Control library
- Graphic library
- Monitoring library
- System library

The communication and parallel control library implements the two layers of GAME's Parallel Execution Module as well as the messaging classes (see Chapter 6) and the basic framework of *C*++ classes used to create applications, genetic algorithms and operators, as discussed in the previous section. It also contains the functions offered by the PEM-API. These functions, which are listed in Table 7.3, facilitate component creation, termination and inter-connection at runtime. They also provide support for sending and receiving synchronous and asynchronous messages, "wrapped" into GAME's MessagePackage objects. A detailed description of each function of the PEM-API can be found in Appendix C.

Table 7.3 - Functions of the PEM API

Function	*Arguments*	*Returns*
`StartComponent`	`dComp` - Component descriptor `char*` - Command line args	`HANDLE` - Component handle
`TerminateComponent`	`hComp` - Component handle	
`OpenConnection`	`dComp` - Component descriptor	`HANDLE` - Component handle
`CloseConnection`	`hComp` - Component handle	
`PostMail`	`MsgPackage&` - message `hComp` - Component handle `[WORD]` - NOREPLY/WAITREPLY/ REPLY/BROADCAST `[WORD]` - message id (=0)	`[WORD]` - number of messages successfuly sent
`HasMail`	`[WORD]` - message id (=0)	`[WORD]` - number of messages in the mailbox.
`CollectMail`	`[WORD]` - message id (=0)	`MsgPackage*` - message
`ReplyMail`	`MsgPackage&` - message	`BOOL` - status
`WaitMail`	`WORD` - message id `MsgPackage*` - advise msg `[WORD]` - num. msg. (=1) `[LWORD]` - time-out. (=0)	
`ProcessMail`	`MsgPackage&` - message	`MsgPackage&` - message
`ProcessReply`	`MsgPackage&` - message	
`StartLocalComponent`	`char*` - Local Component name `char*` - Command line args	`HANDLE` - Component handle
`StartExtComponent`	`char*` - Ext. Component name `char*` - Command line args `[char*]` - host name (=0)	`HANDLE` - Component handle
`TerminateLocalComponent`	`dComp` - Component descriptor	
`TerminateExtComponent`	`dComp` - Component descriptor	
`CreatePort`	`char*` - port name	
`OpenConnection`	`dComp` - Component descriptor	`BOOL` - status
`CloseConnection`	`HANDLE` - Connection handle	`BOOL` - status
`ConnectionStatus`	`HANDLE` - Connection handle	`CONSTAT` - conn. status
`SendSync`	`MsgPackage&` - message `HANDLE` - Connection handle	`MsgPackage&` - message
`SendAsync`	`MsgPackage&` - message `HANDLE` - Connection handle	`BOOL` - status
`Reply`	`MsgPackage&` - message	`BOOL` - status

The graphic library groups a number of classes that help with the creation of windows, dialogue boxes, buttons, menu bars, and various 2D and 3D charts (line, bar, etc.). There are currently in the market a number of *C++* class libraries that provide all these objects for cross-platform implementations. The majority of these products can be easily incorporated into this GAME module. Currently, the graphic library is based on the wxWindows [87] classes, which support the MS-Windows and UNIX (OpenLook and Motif) platforms.

The monitoring library is designed to support real-time supervision of GAME components' execution. It works in connection with the graphic library to provide data input and output. Sources of information can be redirected to graphic widgets, files or other devices, through the functions provided by this module.

Finally, the system library groups a variety of modules and functions that include:

- the Virtual Machine and its modules: the Population Manager, the Fitness Evaluator and the parallel support classes;
- the genetic-oriented representation: DnaNode, DataUnit, and related classes;
- the Virtual Machine application programming interface (VM-API); and
- many ancillary functions including: fitness normalisation, string encoding and decoding, memory management and C++ exception handling.

Most of these modules have been presented in Chapter 5. The following tables list various functions of the VM-API. They are divided according to the different groups of operations offered by the VM. Each table presents the function name, their arguments and the result they return, if any. These functions are described in more detail in Appendix B.

Table 7.4 lists the functions of the VM-API that provide operations over population pools. Most pool operations require a pool handle (hPool) that identifies the pool to the Population Manager. A pool handle is returned by the CreatePool and CopyPool functions.

Table 7.4 - VM-API - Pool manipulation functions

Function	***Arguments***	***Returns***
`CreatePool`	`WORD` - pool size `[Individual&]` - prototype	`hPool` - Pool handle
`DeletePool`	`[hPool]` - Pool handle (=0)	
`CopyPool`	`hPool` - Dest. pool handle `[hPool]` - Src. pool handle	`hPool` - Pool handle
`GetPoolSize`	`hPool` - Pool handle	`WORD` - max. size
`GetPopulation`	`hPool` - Pool handle	`WORD` - cur. size

Table 7.5 presents the functions that can be used for manipulations in genetic structures as a unit. Therefore, these functions operate only at the Individual object level. The GetIndividual and GetIndividualValue, for instance, allow the caller to request a copy of an Individual's genotype and phenotype respectively. However, an Individual can only be inserted or replaced in a pool in its genotype form. An individual's phenotype is usually the result of "transformation" over its chromosome data structures.

Table 7.5 - VM-API - Individual manipulation functions

Function	*Arguments*	*Returns*
GetIndividual	NodePath& - index	Individual - the individual
GetIndividualValue	NodePath& - index [WORD] - DataUnit index (=0)	DOUBLE - phenotype
PutIndividual	NodePath& - index Individual& - the individual	
CopyIndividual	NodePath& - Dest. index NodePath& - Src. index [WORD] - num. copies (=1)	
MoveIndividual	NodePath& - Dest. index NodePath& - Src. index	
KillIndividual	NodePath& - index	

The functions defined by the VM-API relating to operations over DnaNodes are listed in Table 7.6. Functions such as SwapNodes and InvertNodes are used to implement genetic operators like mutation and inversion, respectively.

Table 7.6 - VM-API - DnaNode manipulation functions

Function	*Arguments*	*Returns*
GetNode	NodePath& - index	DnaNode - the node
PutNode	NodePath& - index DnaNode& - the node	
CopyNode	NodePath& - Dest. index NodePath& - Src. index [WORD] - num. copies (=1)	
MoveNode	NodePath& - Dest. index NodePath& - Src. index	
DeleteNode	NodePath& - index	
SwapNodes	NodePath& - Dest. index NodePath& - Src. index	
InvertNodes	NodePath& - index [WORD] - num. nodes (=ALL)	
GetNumNodes	NodePath& - index [WORD] - ONE_LEVEL/ALL_LEVELS	WORD - num. nodes
GetMaxNodes	NodePath& - index	WORD - max. nodes

The operations of the Fitness Evaluator module are requested via the VM-API functions listed in Table 7.7. The GetFitness function returns the fitness value of an Individual. It will firstly look at the Individual's fitness cache, if this is invalid, then the user defined fitness function is invoked. The fitness of the entire pool can be computed by calling GetFitness for every member of the pool in a loop construct. However, since this function is implemented in the VM-API as a synchronous message, it prevents VM from taking advantage of parallelism on fitness evaluations. The best program structure to evaluate and read fitness values should use two loop constructs. The first loop calls the EvaluateFitness, whereas the second calls the GetFitness function. Since EvaluateFitness does not return any result, it is implemented in the VM-API as an asynchronous message. This implies that the control is returned to the caller, in the main program, as soon as the message is delivered to the VM. This allows parallel execution of fitness evaluations, providing the VM contains more than one instance of the Fitness Evaluator module.

Table 7.7 - VM-API - Fitness evaluation and related functions

Function	***Arguments***	***Returns***
`EvaluateFitness`	`[NodePath&]` - individual index (=ALL)	
`GetFitness`	`NodePath&` - ind. index `[WORD]` - Multi-fitness index (=0)	`DOUBLE` - fitness value
`GetTotalFitness`	`hPool` - pool handle `[WORD]` - first member (=0) `[WORD]` - last member (=0)	`DOUBLE` - fitness value
`GetAverageFitness`	`hPool` - pool handle `[WORD]` - first member (=0) `[WORD]` - last member (=0)	`DOUBLE` - fitness value
`GetHighestFitness`	`hPool` - pool handle `[WORD]` - first member (=0) `[WORD]` - last member (=0)	`BestIndividual` - struct with individual's data
`GetLowestFitness`	`hPool` - pool handle `[WORD]` - first member (=0) `[WORD]` - last member (=0)	`BestIndividual` - struct with individual's data

The two functions listed in Table 7.8 implement a simple error report mechanism in the VM-API. Every call to any of the other VM-API functions produces an error status that is kept by a global variable. Functions implemented as synchronous messages update the global variable with the MessagePackage status returned. Conversely, functions implemented as asynchronous messages update the API global variable after the message is delivered (or not) to the VM.

Table 7.8 - VM-API - Error functions

Function	***Arguments***	***Returns***
`GetErrorStatus`		`MSGSTATUS` - error code
`ClearErrorStatus`		

The VM-API functions listed in Table 7.9 are used to operate over DataUnit objects. The type of DataUnit object that stores a particular value is defined by the genetic structure. Then, since only DOUBLEs are used by the API functions, the actual values read or written from or to a DataUnit object are converted to its local type via *C++* conversion operators. This approach also facilitates transparent operations (assignments, etc.) between DataUnit types and *C++* types since any DataUnit type is firstly converted to a DOUBLE before any operation takes place.

Table 7.9 - VM-API - DataUnit manipulation functions

Function	*Arguments*	*Returns*
`ReadData`	`NodePath&` - index `[WORD]` - DataUnit index (=0)	`DOUBLE` - value
`WriteData`	`NodePath&` - index `DOUBLE` - value `[WORD]` - DataUnit index (=0)	
`CopyData`	`NodePath&` - Dest. index `NodePath&` - Src. index `[WORD]` - DataUnit index (=0)	
`DeleteData`	`NodePath&` - index `[WORD]` - DataUnit index (=0)	
`SwapData`	`NodePath&` - Dest. index `NodePath&` - Src. index `[WORD]` - DataUnit index (=0)	
`GetNumUnits`	`NodePath&` - index	`WORD` - num. units
`GetDataStatus`	`NodePath&` - index `[WORD]` - index (=GLOBAL)	`BOOL` - status
`SetDataStatus`	`NodePath&` - index `BOOL` - status `[WORD]` - DataUnit index	

7.5. Summary

This chapter presented the organisation of the GAME programming environment in terms of its runtime libraries. It started by showing how the traditional parameterisation of genetic algorithms, based on their setups (population size, ending criteria, crossover and mutation rates, etc.), could be expanded. The newly introduced level of parameterisation allows applications and genetic algorithms to be further parameterised in terms of algorithm components and genetic operator components, respectively. This innovative approach led to the creation of an equally innovative library concept comprising stand-alone executables and modules that can be dynamically linked to other modules. Applications and genetic algorithms may then be easily built by simply specifying the required GAME Components from the Genetic Libraries in a configuration file.

The construction of the Genetic Libraries' modules is supported by a framework of *C++* classes that encapsulate the required functionality for dynamic connection and inter-component communication. The AppComponent, AlgComponent and OperComponent classes provide specialised support for the implementation of applications, genetic algorithms and operators. GAME Components created from these classes are grouped into three libraries – the Applications library, the Algorithms library and the Operators library – which are jointly called the Genetic Libraries.

The second main group of libraries of GAME is called the Service Libraries. The modules of these libraries (always statically linked) are used to build GAME Components. It comprises the Communication and Parallel Control library, the Graphic library, the Monitoring library and the System library. A brief description of each one of these libraries was given, which was followed by a list of GAME's application programming interface functions (PEM-API and VM-API).

Chapter 8

Assessment

This chapter assesses the work developed in this thesis in each of its principal investigation topics, which focuses in GAME's ability to provide a flexible, expandable and portable environment for the development of GA applications. Each major GAME module described in previous chapters is individually assessed, as well as the integral system. The assessment also includes performance evaluations and comparisons with test programs built with other systems.

8.1. Review of Objectives

The research reported in this thesis comprised the design and implementation of an object-oriented programming environment aimed at facilitating the development of genetic algorithms applications. The combination of the flexibility offered by a modular design with facilities to expand and modify modules provided by an object-oriented implementation, plus platform-independent parallelism, resulted in a unique software tool. The diverse characteristics and requirements presented by real-world applications, as seen in the PAPAGENA project, and the desire to assist with the creation of a broader spectrum of applications, were the main drivers for the comprehensive set of features presented by GAME.

The assessment of the system focuses on three main requirements for a general-purpose programming environment, as explained in Chapter 2:

- *flexibility*– to create and configure new applications. A modular design, combined with parameterised libraries and a simple configuration language should provide the users with the required flexibility.
- *expandability* – to accommodate an ever increasing number of different problems. The object-oriented design and implementation of GAME's modules should facilitate their customisation to provide any extra functionality that could be required.
- *portability*– to allow applications and library modules to be ported on to several hardware and software platforms. To accomplish this objective, GAME's implementation relied entirely on what has been widely acknowledged as a "standard" definition for the *C++* language. Special language constructs and compiler-dependent

class libraries have been avoided in favour of portable classes, implemented with basic *C++* language primitives. Moreover, the commitment to a higher degree of portability determined the creation of a specific module to control parallelism and communication, the Parallel Execution Module. This module conferred on GAME the desired degree of portability, making it independent of any particular implementation of parallel *C++* compilers.

The three items listed above resulted from an extensive research on genetic algorithms applications' characteristics and requirements, as described in Chapter 2. They are also derived from features observed in many genetic algorithms programming environments presented in Chapter 3. As a consequence of these considerations and PAPAGENA applications' requirements, GAME's design represents one of the most flexible and modularised programming environment for GAs and PGAs.

8.2. Assessment of GAME's Modules

This section assesses each of the main GAME modules reported in this thesis. The Virtual Machine, the Parallel Execution Module and the GAME libraries are firstly assessed separately. The last section gives an in-depth assessment of the integral environment, which includes performance comparisons based on applications built with GENESIS, GAME and by stand-alone, *C*-based, test programs.

8.2.1. Virtual Machine

The design of the Virtual Machine, and the genetic-oriented data structures it manipulates, were motivated by the necessity to provide GAME with a module that could perform a standard set of operations over the genetic representation of diverse problems. The Virtual Machine should allow manipulations of the problem's genetic data structures, without being concerned with their contents. To achieve the required degree of problem representation independence and, at the same time, be capable of performing meaningful operations over the data structures, two classes of objects for problems' genetic representations have been created: the DnaNode and the DataUnit. The DnaNode class of objects, combined with the DataUnit derived classes, provide GAME with simple, but effective, abstractions that permit a broad range of genetic representations. They offer enough flexibility to represent a range of problems using simple flat and fixed binary strings or complex tree structures, comprising several layers of mixed data types; the latter being normally encountered in genetic programming representations.

The object-oriented design and implementation of these two class families permit easy expansion of their functionality, without affecting the Virtual Machine or any of its internal modules. The definition, by the user, of the problem's genetic representation is done through a simple alphanumeric string. The representation string specifies the DnaNode connections as well as the number and type of DataUnits each node should contain. As an additional feature of these classes, and to comply with GAME's communication protocol, any object derived from these classes can be made persistent. This means that a simulation could be interrupted at any time, with its data structures saved to disk, and re-started later, from the point it was stopped. This degree of flexibility, expandability and ease of use is not found in any currently available GA programming environment.

The modular design of the Virtual Machine isolates genetic manipulations from the algorithm and operator's implementations. This design approach permits algorithms and operators to be implemented with high-level manipulation commands, independent of the actual problem representation. The concept of the Virtual Machine as a module that centralises all the genetic operations is central to the design of a fully parameterised genetic library.

The set of commands defined in the Virtual Machine application programming interface (VM-API) resulted from a comprehensive study of the most commonly used genetic operators. The various types of genetic operators can be grouped in two main classes: operators that act over the data structure organisation (like crossover) and operators that modify the contents of the data structure (like mutation). These characteristics determined the design of GAME's genetic-oriented representation data structures and the definition of the VM-API commands. The Virtual Machine commands perform operations over the problem's population of genetic data structures on several levels. There are commands that operate over entire pools (create, copy, delete), individuals (create, copy, move, delete) and their sub-parts (swap, invert, modify, read, write, delete, move). The comprehensive command set of the Virtual Machine should cover the implementation of the majority of the evolutionary operators. Furthermore, they can be easily expanded and adapted due to GAME's object-oriented design.

The Virtual Machine's specialised internal modules grant GAME another unique feature, which allows applications originally designed for sequential execution to benefit from parallelism. VM's ability to replicate the Population Manager and the Fitness Evaluator modules, under the control of its parallel support module, ensures parallel operations over the population pools.

Finally, the use of GAME's genetic-oriented data structures in conjunction with the Virtual Machine concept and its application programming interface, represents effective mechanisms to implement a variety of parameterised genetic operators and algorithms.

8.2.2. Parallel Execution Module

The creation of the Parallel Execution Module largely resulted from a complete lack of standard and compatible implementations of the *C++* language for parallel platforms. As seen in Chapter 6, the PEM is best suited for supporting medium and coarse-grain parallelism, without compromising portability. The module supports GAME's programming model, which was designed to provide the high degree of portability required for building parallel applications. PEM implements a communication and task control library that can be ported onto different operating systems and hardware platforms with minor programming effort. The main objectives of the Parallel Execution Module are:

- to support the implementation of a variety of parallel applications (including those not related to GAs);
- to provide a smooth path for the development and debugging of parallel applications on sequential machines, before porting them to the target parallel platform; and
- to enable applications and algorithms to be built or reconfigured, without recompilation.

The PEM offers high-level communication and process control commands, implemented as an application programming interface and embedded into a set of *C++* classes. Some of these classes have been specialised to provide a standard framework for the development of sequential and parallel applications, algorithms and genetic operators. The user only needs to derive his own application-dependent classes and include the code for some pre-defined member functions. All the task and communication management have been implemented and are controlled according to PEM's layered design specifications.

The programming model defines application modules in terms of *local* and *external components,* providing flexibility to configure applications according to their requirements and available resources (pre-built algorithms, genetic operators, graphical interface, etc.). Local and external application components can be combined to better exploit the execution environment. It is possible, for instance, to have the application component (which would typically contain the user interface) running on a graphic workstation, and the algorithm (as an external component) and genetic operators (as local components to algorithms) running on a different host. In addition, the Fitness Evaluator module could be running on a fast parallel machine. The actual benefit of spreading algorithms, genetic operators and other GAME modules among different processes and processors depends, essentially, on the communication overhead introduced by the chosen topology. This particular feature of GAME grants the user the ability to define the best configuration for the application dynamically, according to the hardware resources at hand.

The communication system and its object-oriented implementation offer a high-level abstraction for inter-component communication and provide the required functionality for multiple task execution and control. It supports synchronous and asynchronous bi-directional communication as well as process synchronisation. Its layered design ensures application portability since only the lower-layer, not seen by the components, needs to be ported across different platforms. Current and on-going implementations of PEM include Microsoft Windows 3.1, Sun OS 4.01 and a custom version for transputers implemented by TELMAT Informatique.

8.2.3. GAME Libraries

The Genetic Algorithms Manipulation Environment comprises two principal sets of libraries: the Genetic Libraries and the Service Libraries. The design and organisation of these libraries aim at facilitating the configuration of existing applications and assisting with the creation of new application components. They cater for novice and experienced users through three levels of interaction with the programming environment. At the first level, novice users should be able to configure and execute an application by simply combining components from the genetic libraries, and setting up their parameters in a simple configuration file. The second level allows experienced users and programmers to create new components by either modifying or expanding the functionality of existing components. The inheritance mechanism of GAME's object-oriented design and *C++* implementation is paramount to users in this level. Lastly, the third level enables experienced programmers to expand, adapt or port the entire programming environment to support new features and execute applications on different platforms.

The library structure provided by GAME has been designed to support an ever increasing number of genetic operators and algorithms. The organisation of Genetic Libraries, in particular, makes the management of algorithms and genetic operator components extremely easy. Components can be added to or removed from the libraries by simply copying or deleting them in the appropriate directory, therefore dispensing with the use of a librarian utility. Moreover, a new component becomes immediately available to any algorithm or application, with no need for re-compilation or re-linking. This singular flexibility derives from GAME's programming model, which defines application components in terms of stand-alone executables (external components) or dynamically linked modules (local components).

The portability of applications, algorithms and genetic operators is guaranteed by the Parallel Execution Module with its machine-independent communication and task control support.

8.3. Assessment of the Integral GAME

This section assesses the interaction of GAME's modules in applications built with the programming environment. The overall assessment of the programming environment is based on the following criteria:

- suitability of the programming environment for the creation of diverse and complex applications,
- design and implementation, and
- performance

The first criterion judges the facilities provided by GAME to assist with the construction of various types of genetic algorithms' applications. Essentially, it tries to assess how general-purpose the tool kit is. The second criterion looks at the system's architecture and design in connection with techniques and technologies currently employed in software development. Finally, the third criterion assesses the performance in terms of execution times of applications built with GAME, in relation to other GA systems.

8.3.1. Suitability for Applications' Requirements

GAME's genetic-oriented data structures allow the representation of a broad range of problems such as those encountered in the PAPAGENA project. The possible representations include fixed and non-fixed binary, real value (with mixed data types) and combinations of both. Real value representations, for instance, are used by PAPAGENA's protein folding and economic modelling applications. The financial modelling application, on the other hand, employs a non-fixed and irregular data structure usually associated with genetic programming.

The system's object-oriented design permits coarse and medium-grain parallelisation of genetic algorithms through the replication of its internal components. Several instances of genetic algorithms, genetic operators and Virtual Machine components can be created to execute in parallel.

The extensive use of application programming interfaces (e.g. VM-API and PEM-API) provides application developers with an effective set of functions suited for the various requirements presented by different applications. The VM-API, for instance, supports the implementation of genetic manipulations required by the majority of genetic operators, via a comprehensive set of specialised commands and functions. Furthermore, the GA-independent definition of GAME's programming model, the Parallel Execution Module and its API, permit the use of this module in an even broader range of parallel applications.

Therefore, the comprehensive set of tools and features presented by GAME provide the system with the required attributes to assist with the development and execution of complex applications.

8.3.2. Design and Implementation

The design and implementation of all GAME modules reflect the concern with the three most important concepts of object-oriented design and programming: data encapsulation, inheritance and polymorphism. The set of classes designed for representing problems' data structures (DnaNode and DataUnit) provide a suitable abstraction, based on the genetic algorithms terminology. These abstractions offer the required functionality for problem-independent genetic manipulations. They allow genetic operators to work over different levels of abstract structures, which include: *individuals*, as whole sets of problems' parameters to be optimised, and *chromosomes*, *genes* and *dna* objects as sub-sets of those parameters, with different degrees of aggregation. The classes designed for representing a problem's genetic data structures in GAME have been implemented strictly according to object-oriented programming principles and *C++* language specifications. Therefore, they permit functional extensions and modifications via inheritance mechanisms. Also, any member function defined in these classes can be activated via a common base class.

The same discussion applies to the design of the modules responsible for the manipulation and evaluation of the genetic-oriented data structures: the Virtual Machine, the Population Manager and the Fitness Evaluator modules. These modules provide the adequate abstractions for parallelisation of genetic manipulations (via modularisation and task specialisation), and can be easily extended or modified via standard object-oriented procedures.

Finally, it is important to stress the compliance of GAME's design with leading technologies. The programming model implemented by PEM, with its message-driven communication system, conforms with the most recent concepts for distributed object computing. Both have been largely inspired and influenced by object-oriented computing models such as CORBA, COM and SOM.

8.3.3. Performance

A compromise between performance and generality is normally very difficult to achieve, in particular, for complex and sophisticated programming environments. Dedicated, stand-alone applications most often outperform applications created with flexible and general-purpose objectives. However, a programmer should be able to balance a number of requirements, usually including performance, flexibility, generality, expandability and portability, when choosing a programming environment. Furthermore, the programmer should identify the best compromise according to the particular characteristics and requirements of each problem.

This section assesses the performance (in terms of execution times) of applications created with GAME. Comparative results are presented for test programs built with GAME, GENESIS and stand-alone versions written directly in the *C* language. The programs, assessment methodology and simulation environment have been chosen to provide meaningful comparisons of the results. This restricted the range of applications that could be created with GAME to those also supported by GENESIS, and easily implemented in a stand-alone program. Nevertheless, the results collected permitted the elaboration of a simple analytical model for classifying general-purpose GA programming environments, based on the time spent by an application's evaluation function.

Methodology

GAME's performance assessment involved the execution of the same set of programs created in three different ways. The first set was created with GENESIS. This programming environment was chosen because of its popularity and availability. The second group of test programs was created with GAME, and the third, by directly coding them as stand-alone *C* programs. Although stand-alone *C* programs cannot be compared with programming environments in terms of development facilities, they usually represent the best alternative in terms of performance. The total execution times and individual module's execution times were measured and compared.

The main interest is to compare the programming environments by means of their applications' performances (as opposed to assessing a genetic algorithm's performance). Therefore, the tests tried to use as much as possible the same algorithm and parameter setup. Some internal aspects of GENESIS, for instance, are not directly under the application programmer's control, and introduced small differences in the way some operators have been implemented and executed (e.g. selection). The actual function being optimised was also not considered important, but its computational time, as a measure of its influence on the total simulation time. Furthermore, all possible input and output routines were disabled (including displays), in order to eliminate any possible interference caused by different implementations of these normally time-consuming tasks.

The same group of programs was executed many times, with different setups, in order to identify the influence of the following parameters in the total and partial execution times:

- Number of Generations – because we were not interested in measuring how fast an algorithm finds the best result, but on how long each module executes, the programs were left to run for a fixed number of generations. The tests included 44, 74, 550 and 1000 generations to cover for small, medium and large simulations.

- Population size – a standard GA usually has a population of 50 individuals. The programs also used 200 individuals to compare the overhead caused by copying, moving and modifying standard and large populations.
- Representation – simulation results were measured for binary representations only. These should be considered the worst case, compared with real value representations, since they require an extra encoding and decoding steps. Another reason for this option is that GENESIS only supports binary and "flat" representations. The tests used 5 and 30 bit representations.
- Evaluation – the judgement of the evaluation time impact over the total execution time is extremely important since most of the complexity of a problem being solved by a GA is embedded in the evaluation function. Simple problems were represented by fast evaluation functions like x^2. Real-world problems, however, usually present very complex and time-consuming evaluation functions. Since for these types of problems we are mainly interested in their evaluation time (and not the actual complexity of their implementation), the fitness function was simulated by introducing a delay of one second in the evaluation time.

In all test cases the crossover rate was 0.6 and the mutation rate 0.001, which are the most commonly used values. The total execution time and each major GA module execution time (including initialisation, main loop, selection, evaluation, crossover and mutation) were measured in absolute and relative times for every set of parameters.

Simulation Environment

The tests were conducted on an IBM-PC machine with a 66 MHz i486 cpu. All applications were compiled with Borland C++ 3.1 and executed with its profiler tool under MS-DOS 5.0. This provided a standard platform the measurement of absolute cpu execution times.

Results

The next tables and figures show the results, measured as the mean of ten simulations for each one of the four parameters (number of generations, population size, problem representation and evaluation time). Table 8.1 shows the total execution times, in seconds, for the SGA program running with four different numbers of generations.

Table 8.1 - Total execution times(sec) x Number of generations

Generations	Genesis	SGA (C)	SGA (GAME)
44	0.0661	0.1014	193.181
74	0.0913	0.1533	330.0096
550	0.5779	1.2526	2412.3599
1000	1.0446	2.0861	4368.0932

The following tables exhibit the results for a population of 50 individuals running for 44, 74, 550 and 1000 generations. Each individual was represented as 5-bit binary string.

Table 8.2 - Total and partial times (sec) for 44 generations

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0017	0.0009	0.1341
Main Loop	0.0219	0.017	0.0028
Selection	0.0273	0.0647	173.88
Evaluation	0.0051	0.0019	14.726
Crossover	0.0092	0.0065	3.2432
Mutation	0.0009	0.0104	1.1949
Total	0.0661	0.1014	193.181

Table 8.3 - Total and partial times (sec) for 74 generations

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0017	0.0009	0.134
Main Loop	0.0227	0.0186	0.0028
Selection	0.0449	0.1074	297.38
Evaluation	0.0058	0.003	24.726
Crossover	0.0149	0.0061	5.7568
Mutation	0.0013	0.0173	2.01
Total	0.0913	0.1533	330.0096

Table 8.4 - Total and partial times (sec) for 550 generations

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0017	0.0009	0.1341
Main Loop	0.1047	0.2042	0.0028
Selection	0.3333	0.8222	2172.1
Evaluation	0.0196	0.0235	182.79
Crossover	0.1087	0.0692	42.411
Mutation	0.0099	0.1326	14.922
Total	0.5779	1.2526	2412.3599

Table 8.5 - Total and partial times (sec) for 1000 generations

Module	*Genesis*	*SGA (C)*	*SGA (GAME)*
Initialisation.	0.0017	0.0009	0.0134
Main Loop	0.1838	0.2829	0.0028
Selection	0.6103	1.4349	3931.32
Evaluation	0.0323	0.0429	332.28
Crossover	0.1991	0.0848	77.361
Mutation	0.0174	0.2397	27.116
Total	1.0446	2.0861	4368.0932

The above figures demonstrate that most of the results obtained with GAME are two to three orders of magnitude higher than those obtained with GENESIS and the stand-alone *C* program. However, looking at the partial results it is possible to draw the following conclusions:

- Although most of GAME's partial results are higher than those found in the other systems, the main loop always took less time.
- GAME's operators presented consistently higher execution times than the other systems. However, when compared with the main loop results, they indicate that most of the overhead was introduced by the communication protocol between the main module and the Virtual Machine.
- The selection operator always presented the highest result in all the systems. This demonstrates the overhead introduced by this operator when copying data structures from the parent's pool to the offspring's pool. These results seem particularly critical for GAME's selection operator due to the complex objects provided by the system to support a broad range of problem representations.

Figure 8.1 - Relative execution times for 44 generations

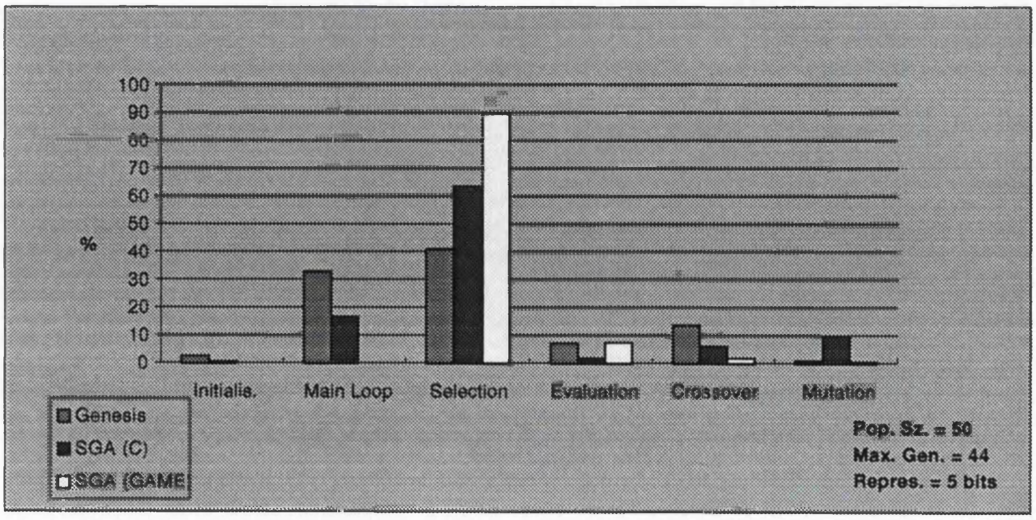
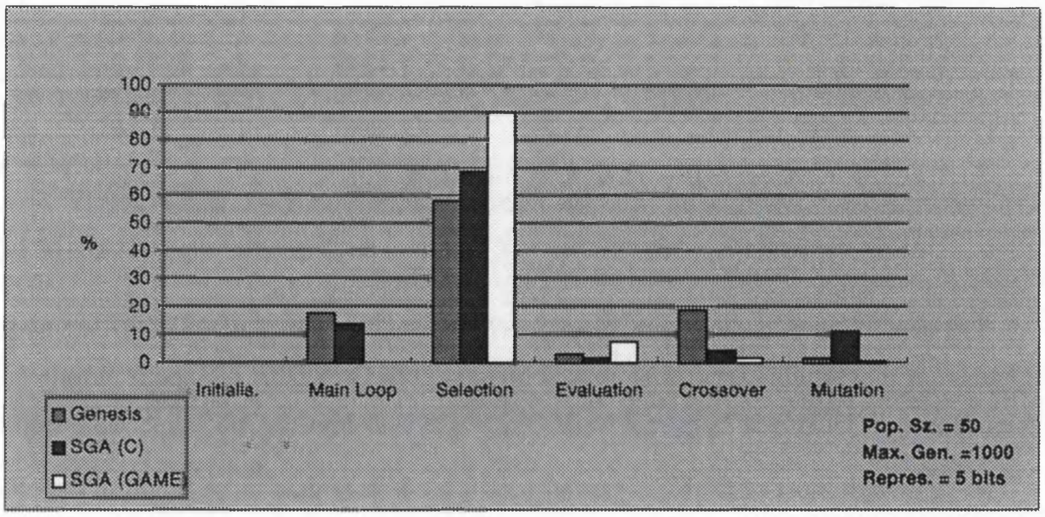


Figure 8.2 - Relative execution times for 1000 generations



- The ratio between each operator and the total execution time (as seen in the Figure 8.1 and Figure 8.2) is nearly constant, indicating that the execution time increases linearly with the number of generations. This also indicates that the number of generations does not affect any operator in particular.

The next tables present the results obtained when the number of individuals in the populations varied and the number of generations was kept constant (= 44). The simulations used 50 and 200 individuals, represented as 5-bit strings.

Table 8.6 - Total and partial times (sec) for a population of 50 individuals

Module	*Genesis*	*SGA (C)*	*SGA (GAME)*
Initialisation.	0.0017	0.0009	0.1341
Main Loop	0.0219	0.017	0.0028
Selection	0.0273	0.0647	173.88
Evaluation	0.0051	0.0019	14.726
Crossover	0.0092	0.0065	3.2432
Mutation	0.0009	0.0104	1.1949
Total	0.0661	0.1014	193.181

Table 8.7 - Total and partial times (sec) for a population of 200 individuals

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0056	0.0025	0.2659
Main Loop	0.0385	0.0495	0.0029
Selection	0.1077	1.1044	681.44
Evaluation	0.0184	0.0073	20.505
Crossover	0.0375	0.0159	6.6783
Mutation	0.0027	0.0427	2.356
Total	0.2104	1.2223	711.2481

The conclusions for these tests are the same as those shown previously. They demonstrate that variations in the number of generations and in the population size do not modify the timing relation between the genetic algorithm and its genetic operators. The same tests were conducted for 74 generations and presented similar relative figures.

Table 8.8 and Table 8.9 show the total and partial execution times for a simulation of 74 generations, with a population of 50 individuals represented as 5 and 30 bits.

Table 8.8 - Total and partial times (sec) for a 5 bit representation

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0017	0.0009	0.134
Main Loop	0.0227	0.0186	0.0028
Selection	0.0449	0.1074	297.38
Evaluation	0.0058	0.003	24.726
Crossover	0.0149	0.0061	5.7568
Mutation	0.0013	0.0173	2.01
Total	0.0913	0.1533	330.0096

Table 8.9 - Total and partial times (sec.) for a 30 bit representation

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0293	0.0059	0.1889
Main Loop	0.061	0.3693	0.0029
Selection	0.2244	1.9667	376.92
Evaluation	0.3833	0.0127	35.81
Crossover	0.0802	0.3106	11.59
Mutation	0.0319	0.0727	2.0121
Total	0.8101	2.7379	426.5239

These results highlight the influence of the a problem's representation on a GA. The total execution time figures show that GENESIS became 9 times slower, and the *C* program became 18 times slower, when the representation changed from 5 to 30 bits. However, the change of representation had a much smaller impact on GAME. The total execution time was reduced by a factor of only 1.3. This can be explained by the facilities provided the genetic-oriented data structures and the Virtual Machine for manipulating complex representations. Figure 8.3 and Figure 8.4 present the results of the same tests, but in relative figures.

Figure 8.3 - Relative execution times for a 5-bit representation

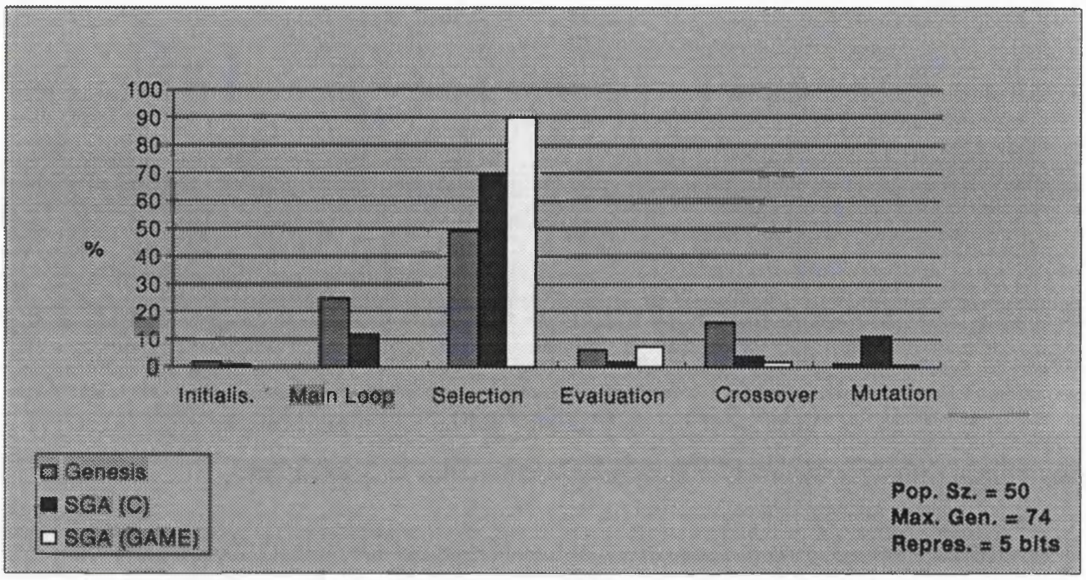
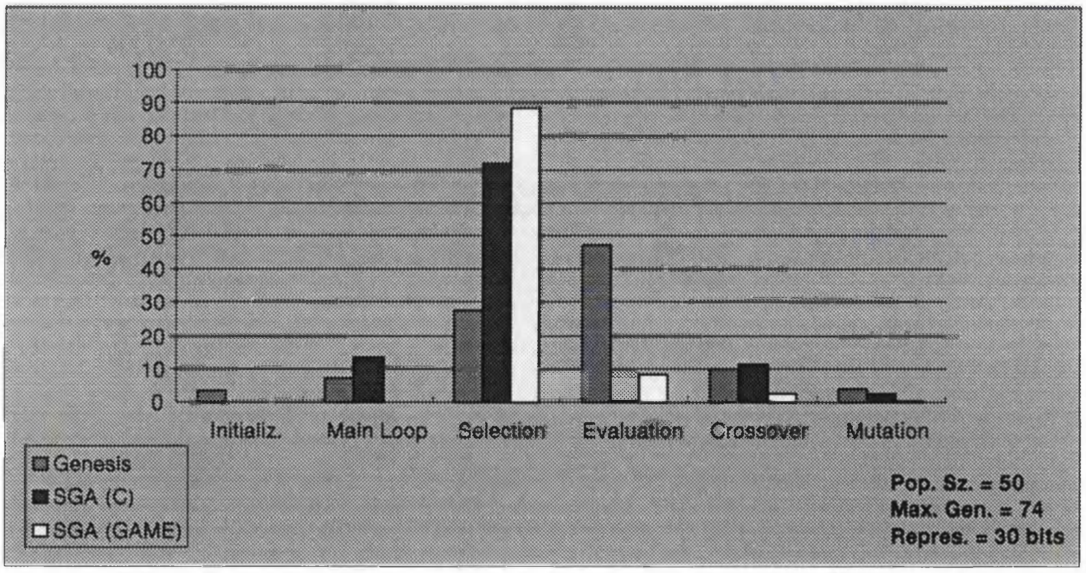


Figure 8.4 - Relative execution times for a 30-bit representation



The next tests were designed to assess the impact of the problem-dependent evaluation function on the overall simulation performance. The important aspect of these tests is the time spent by the evaluation function, and not the particular type of function being used. Table 8.10 and Table 8.11 present the results for the test programs running with a fast fitness evaluation (x^2)

and a slow fitness evaluation, which spends one second. The GA had 50 individuals in the population, represented as 30-bit binary strings, and was left to run for 44 generations.

Table 8.10 - Total and partial times (sec.) for a simple (fast) evaluation function

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0077	0.0022	0.1893
Main Loop	0.0008	0.1898	0.0028
Selection	0.3264	0.0684	223.99
Evaluation	0.0377	0.0018	21.456
Crossover	0.0124	0.0448	6.6249
Mutation	0.0251	0.0587	1.1982
Total	0.4101	0.3657	253.4612

Table 8.11 - Total and partial times (sec.) for a complex (slow) evaluation function

Module	Genesis	SGA (C)	SGA (GAME)
Initialisation.	0.0085	0.0018	0.1901
Main Loop	0.4533	0.60	0.0029
Selection	0.0591	0.0682	224.42
Evaluation	1286.7	2149.3	1235.1
Crossover	0.0392	0.0425	6.7297
Mutation	0.0306	0.0103	1.2002
Total	1287.291	2150.023	1467.643

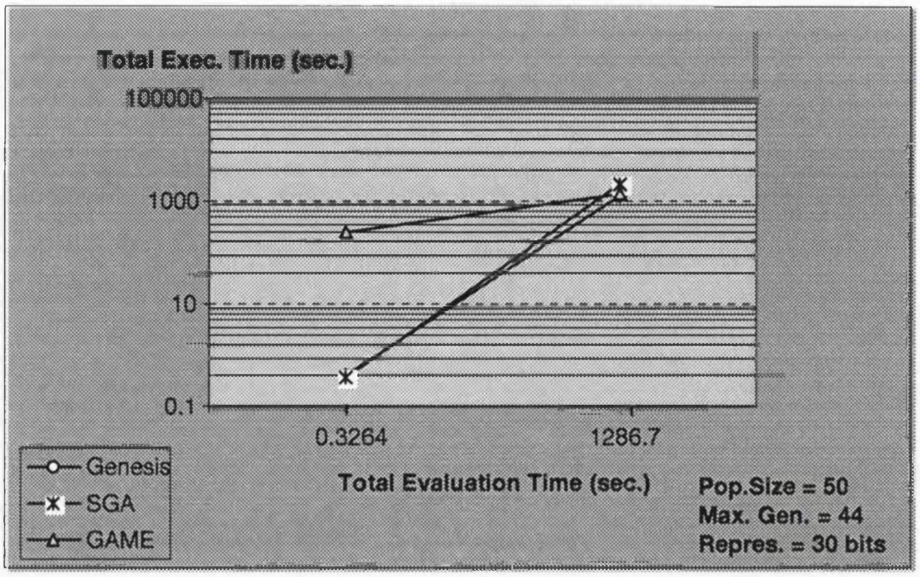
The above figures demonstrate that, for a certain class of problems, even sequential applications built with GAME can offer the same performance as less sophisticated programming environments. Figure 8.5 gives a better picture of the data presented in these tables. It compares the test programs' total execution time with theirs (total) evaluation time. The figure shows that GAME applications present poor performance for "fast" evaluation functions, but similar performance for "slow" evaluation functions. This is a remarkable result, considering the overhead introduced by GAME's underlying infra-structure for communication, task control and for the implementation of its genetic-oriented data structures. This same overhead explains the system's poor performance for fast evaluation functions. However, as soon as the evaluation time exceeds the overhead threshold GAME applications perform similarly to the others.

It is clear from these and previous tables that the selection stage dominates the GA execution on applications created with GAME. The GA selection involves mainly copying genetic structures from the parents' pool to the mating pool. This corresponds to replicating a hierarchy of DnaCollection and DataUnit objects in GAME. Therefore, the duplicate member functions of Individual, DnaNode and DataUnit objects are invoked for every selected member in the population.

The duplication of a population member is essentially the duplication of a list structure organised as a tree. Therefore, the use of special architectures, such as the Kernel System of the

SPAN project [80], which implements a co-processor [27] dedicated to list manipulations, have the potential to dramatically improve GAME's performance.

Figure 8.5 - Total x Evaluation time



Analytical Model

From the numerical results presented in this section, it is possible to derive a simple analytical model for an application simulation time. This model should permit one to easily determine the class of problems that a particular programming environment can successfully address, as a function of the problems' evaluation time. With this model it should also be possible to calculate the overhead introduced by all three systems and explain GAME's results.

The total simulation time (S_T) of a genetic algorithm can be expressed as a sum of the separate execution times of its operators, plus the main algorithm loop execution time.

$$\boxed{S_T = i_T + a_T}$$

Where:

S_T = total simulation time

i_T = initialisation time

a_T = total algorithm time, which corresponds to the total time spent in the main loop

The initialisation time i_T is a function of two parameters: the number of individuals in the population P and the problem representation R, which may require encoding or other operations.

$$i_T = f_1(P,R)$$

The time spent in the GA main loop, a_T, is a function of the number of generations G and the total time spent by the algorithm's operators o_T.

$$a_T = f_2(G, o_T)$$

Expanding o_T in terms of each operator's total simulation time:

$$o_T = r_T + c_T + m_T$$

with $r_T = e_T + s_T$ representing the total time spent by the reproduction operator, which is equal to the sum of the total evaluation and selection times. Then, the time spent by the algorithm's operations in one generation (o_t) is given by:

$$o_t = [(\rho \times e_t) + s_t + (p_c \times c_t) + (p_m \times m_t)] \times P$$

Where:

$e_t = f_f(P, \rho)$, $1 \geq \rho > 0$, representing the time saved by techniques like fitness caching,

$s_t = f_s(P)$, giving the total time for selection,

$c_t = f_c(p_c, P)$, giving the total time spent by crossover (p_c = crossover probability)

$m_t = f_m(p_m, P)$, giving the total time spent by mutation (p_m = mutation probability)

Then, the final expression for the total simulation time is given by:

$$\boxed{S_T = (i_t \times P) + (l_t \times G) + \{[(\rho \times e_t) + s_t + (p_c \times c_t) + (p_m \times m_t)] \times G \times P\}}$$

with i_t corresponding to the time required to initialise one individual, l_t representing the time of a single pass in the main algorithm loop, G the number of generations and P the population size.

The time spent by any genetic operator on a single computation (ϖ_t) can also be expressed in terms of the operator's intrinsic execution time (E_t), the time taken to manipulate the data structure (M_t) and the communication time, i.e., time spent to "activate" the operator (C_t).

$$\boxed{\varpi_t = E_t + (\alpha \times M_t) + (\beta \times C_t) \text{ with } \alpha \geq 0 \text{ and } \beta \geq 1}$$

In most simple programming environments, like GENESIS, $M_t, C_t, \alpha \to 0$ and β is usually 1. Thus the previous expression can be approximated by:

$$\omega_t \cong E_t$$

Complex parallel applications, however, must consider the other two components since the time taken for manipulating large genetic structures, and "activating" an operator in another process or machine cannot be disregarded.

Considering again the total simulation time; once a set of operators and the problem's representation are chosen, and the major parameters of the GA (p_c,p_m,P,G) are fixed, the total simulation time will then depend only on the time spent by the evaluation function (e_t), and what may now be called the ***algorithm intrinsic time*** (τ). This represents the initialisation time for a single population member added to the time spent by the algorithm to perform one generation cycle (excluding the evaluation time). Thus, the total simulation time can be rewritten as:

$$\boxed{S_T = [(\rho \times e_t \times G) + \tau] \times P}$$

with

$$\tau = i_t + [l_t + s_t + (p_c \times c_t) + (p_m \times m_t)] \times G$$

or, in terms of the total evaluation (e_T) and the total algorithm intrinsic time (T):

$$\boxed{S_T = e_T + \mathrm{T}}$$

The above expression permits the separation of the algorithm's intrinsic simulation time from a problem-dependent evaluation function time. Moreover, it allows us to assess the impact of the algorithm's overhead in the simulation time and, therefore, establish the minimum evaluation time that can be successfully supported by a programming environment.

An ideal simulation program would have $\mathrm{T} \longrightarrow 0$ or $e_T >> \mathrm{T}$ in order to minimise the algorithm's simulation overhead. Since, for practical reasons, the first assumption is not feasible, we will assume that $e_T >> \mathrm{T} \approx e_T = (10 \times \mathrm{T})$. Thus, by knowing an algorithm's intrinsic simulation time it is now possible to estimate the minimum time that an evaluation function could spend to have the total simulation time practically unaffected by the algorithm's overhead. Then, the minimum evaluation time $e_{t\,\mathrm{min}}$ is given by:

$$\boxed{e_{t\,\mathrm{min}} = \frac{10 \times \{i_t + [l_t + s_t + (p_c \times c_t) + (p_m \times m_t)]\}}{\rho}}$$

with $1 \geq \rho > 0$.

It is important to remember at this point that the algorithm's intrinsic simulation time is also a function of the problem representation, which appears as an additive component ($\alpha \times M_t$) in the expression for the simulation time of genetic operators. From these equations and the data collected for the simulations performed with GENESIS, GAME and the *C* version of the SGA, it is possible to tabulate the minimum evaluation time required in order to "eliminate" the overhead introduced by each one of the systems considered. Table 8.12 presents e_{tmin} for 5 and 30-bit representations, with 50 individuals in the population and 74 generations.

Table 8.12 - Minimum evaluation time for 5 and 30-bit representations (millisecs.)

Representation	*Genesis*	*SGA (C)*	*SGA (GAME)*
5-bit	0.2	0.4	810
30-bit	1.1	7.3	1055

To use GENESIS, for instance, a fitness function only needs to spend 1.1 milliseconds (for a 30-bit representation), whereas the GAME application requires a fitness function spending, at least, 1.05 seconds.

From Figure 8.5 and Table 8.12 it is possible to conclude that applications created with GAME are able to compete with GENESIS, or even stand-alone *C* code, in a category that comprises complex real-world problems, which commonly exhibit time-consuming fitness functions. These costly evaluation functions easily exceed the minimum evaluation time required by GAME. The above figures, for instance, position GAME for a class of applications that would normally require a minimum of four hours of simulation, running on a 486/66 machine.

At this point, it is important to note that parallel tests, although important for the assessment of GAME, could not be performed. The main reasons were:

- The current implementation of the PEM runs on MS-Windows platforms and therefore allows only sequential and concurrent execution of GAME Components. One of the tasks of TELMAT, in the PAPAGENA project, was to port GAME onto a parallel architecture based on the T9000 transputer. However, the task was not completed since the T9000 was not commercially available by the end of the project. It was expected that the transputer-based version of GAME would provide performance data to assess at least one parallel implementation.

- The option of running test programs on a distributed environment (e.g. a network of Sun workstations) was also considered. This option, however, would require the creation of special tools for gathering information. In addition, several issues including network bandwidth and interference, usually found in multi-user distributed systems, would have to be taken into account in the performance analysis. These tasks, alone, would demand programming effort and time beyond what is feasible for this thesis.

Furthermore, comparing results with other GA parallel systems is very difficult for the following reasons:

- Parallel programming environments for GAs are, in general, not easily available, and most of them are dedicated to particular parallel platforms,

- The current literature reporting on parallel genetic algorithm does not provide enough information about applications' performance related to different programming environments. The reports normally compare the results of genetic algorithms, focusing on their ability to "find" a solution (i.e. their convergence rate) faster than the others. They do not provide execution times figures for each GA stage, nor the total execution time on a per generation basis. There seems to be no concern, at the moment, with the influence of a particular programming environment, language or implementation technique over the applications' performance in terms of execution speed.

Nevertheless, the results presented include part of the overhead introduced by GAME'S communication and task control module. Since all application components (algorithms, operators, Virtual Machine, etc.) rely on GAME's Parallel Execution Module for communication, the compulsory presence of PEM's upper-layer imposes the same communication costs on parallel and sequential applications. This design feature, however, immensely facilitates the porting of sequential applications created with GAME onto parallel platforms. Only the communication overhead introduced by the underlying operating system and parallel hardware was not considered in these tests.

Another important item in the assessment of programming environments relates to the application's development time. This is one of the strongest points of GAME, due to the modular structure of its applications. Building new applications and algorithms, using components of the Genetic Libraries, is extremely easy. It typically requires only the creation of a configuration file. It is also not difficult to customise parts of an application (e.g. user interface). In general, few components of an application need to be modified. Even writing a whole application from ground up is facilitated by GAME's application program interfaces, and source code examples. As a measure of GAME's application development time, it took a student less than two moths to develop a medium-sized prototype application. After this period, the student was able understand and use most of the facilities provided by GAME.

Finally, the results obtained from the series of performance tests presented in this chapter demonstrate that GAME is well suited to very complex applications. GAME's object-oriented design and implementation, added to sophisticated data structures, provide enormous flexibility for programmers to build and configure complex real-world applications. Moreover, the system further extends the concept of portability by allowing parallel applications to be developed and debugged on sequential platforms.

Chapter 9

Conclusions and Future Work

This final chapter presents a summary of this thesis and draws some conclusions about the research and its results. It also gives some directions for future work.

9.1. Thesis Summary

The primary goal of this thesis was to present the design and implementation of a general-purpose programming environment to help with the construction of genetic algorithms applications. In pursuing this objective, the Genetic Algorithms Manipulation Environment was designed and implemented based on investigations of GAs' and PGAs' main characteristics and requirements (Chapter 2). GAME's design also resulted from a comprehensive study of currently available GA programming environments (Chapter 3), and requirements from applications developed within the PAPAGENA project. The conclusions drawn from these investigations pointed to three principal requirements for a GA tool kit:

- *flexibility*– to allow the construction of applications with parameterised versions of GAs and PGAs. The programming environment should make possible the combination of existing software modules, like genetic operators and algorithms, to compose new applications and algorithms.
- *expandability* – to facilitate adaptations and enhancements of any software module in order to provide GAs with new characteristics and diverse problems' requirements.
- *portability*– to provide the ability to execute seamlessly the same application (and its parameterised modules) on a variety of heterogeneous sequential, parallel and distributed computer architectures.

The realisation of these requirements, which turned into the main objective of this research, was made possible due to the adoption of an object-oriented design and a modular architecture for GAME (Chapter 4). Issues like problem-independent manipulation and genetic representation were solved with the definition of genetic-oriented abstractions and the creation of a self-contained module – the Virtual Machine. The VM isolates data manipulation from the actual GA implementation, via a comprehensive application programming interface (Chapter 5).

The definition of a programming model based on distributed object computing, implemented using a language/operating system independent parallel support strategy, was fundamental to maximise flexibility and portability of applications built with GAME (Chapter 6). The concept of autonomous *components*, as defined by GAME's programming model, introduced a new parameterisation level for applications and genetic algorithms. Applications can be parameterised in terms of algorithm components, and these in terms of genetic operator components. This also determined the creation of a new type of runtime library that, unlike traditional libraries, consists of stand-alone executables and dynamically linkable modules. Parameterised versions of genetic algorithms and operators from the Genetic Libraries can be quickly combined into applications and algorithms, respectively (Chapter 7). A simple configuration file specifies the inter-connections between these components as well as other ordinary parameters these components may require.

Finally, GAME's assessment (Chapter 8) exposed the system's strengths and weaknesses. It showed that GAME's design and implementation fulfil the three main objectives of this thesis. Moreover, the performance results demonstrated that the system is able to compete successfully with less sophisticated programming environments (therefore capable of creating faster applications), in the class of highly complex applications. The results obtained for sequential simulations also highlighted GAME's potential for efficient exploitation of parallelism. Another important aspect of a programming environment relates to the ability to speedup application's development. In this area, GAME's object-oriented design, and its framework of specialised *C++* classes, provide the user with a powerful set of tools aimed at fast and efficient application construction.

9.2. Research Contributions

The major contribution of this research is the design and construction of the GAME programming environment itself. However, various issues and problems that surfaced during its design and implementation required the adoption, combination and creation of solutions, which also turned out to be significant contributions of this research. A list of the most important contributions must include:

- *Classification of existing GA programming environments* according to a taxonomy comprising three principal categories: application-oriented systems, algorithm-oriented systems and tool kits. Algorithm-oriented systems further sub-divide into algorithm-specific and algorithm libraries, whereas tool kits sub-divide into educational and general-purpose systems.

- *Definition of a set of genetic-oriented abstractions* that enable the genetic representation of a broad range of problems.
- *Creation of a problem-independent genetic manipulation engine* that relieves application developers from being concerned with low-level issues such as memory management, exception handling, etc.
- *Definition of a parallel programming model* based on modern technologies that can exploit parallelism via distributed object-oriented systems, possibly executing on heterogeneous computing environments.
- *Introduction of an extra level of parameterisation for genetic algorithms* and applications, which facilitates the construction of these modules through the combination of DLL-based and stand-alone executable components.

Finally, it is important to stress that GAME's wide distribution, its role in the major EC funded GA project, and the articles published in many conferences, largely contributed to the dissemination of the genetic algorithms technique.

9.3. Future Work

The current implementation of GAME can be used for the development of complex real-world GA and PGA applications. Although, being a prototype system, GAME still needs enhancements in some of its modules as well as better implementations for a few others. Future work also include some research topics. Therefore, the work to be carried out on GAME can be divided into three areas: system enhancements, implementation improvements and research. The latter aimed at broadening the scope of the environment.

Enhancements tasks include:

- Expansion of the Genetic Libraries to increase the number of parameterised versions of GAs, PGAs and genetic operators.
- Implementation of the communication and parallel control library on top of the PVM system. The current library is based on a *C++* implementation of the MS-Windows DDE protocol, which has also been made available for UNIX platforms (on top of the sockets interface). The PVM version will allow GAME applications to run on a wide range of parallel environments, including networks of workstations, vector machines and massively parallel architectures.

- Integration with various commercial graphical user interface builders to facilitate the adoption of the system in conjunction with other programming environments.

Improvements currently required by the system are:

- Simplification of the Individual and DnaNode classes. The current implementation supports features such as unlimited number of node connections, which impact the overall performance of genetic manipulations. This is particularly apparent in the selection stage of the GA, when these objects are replicated.
- The implementation of the Population Manager module needs to be modified to make DnaNode addressing more efficient. The current PM addresses inner DnaNodes in its population pool via recursive calls to DnaNode's maibox member function. A more efficient method should enable PM to locate the pointer for the required DnaNode object directly.
- Another possibility for improving PM's performance is to create two types of Population Manager. One type would only be able to manipulate genetic structures with fixed size (width and depth). The other type would support problems requiring variable or re-sizeable genetic structures (as in the current implementation). Since a large portion of GAs use fixed-size representations, a PM dedicated to this class of algorithms would offer much better performance.

In the research arena, the parallelism provided by GAME's programming model, and PEM implementation, needs to be compared on various parallel platforms. It is also important to gauge how scaleable parallel applications created with the system are. The assessment of these two topics will provide a better picture of the range of applications that can benefit from GAME's parallel features.

Other extensions of this work would include the incorporation of different types of Virtual Machines. The self-contained nature of GAME's Virtual Machine and the use of a dedicated application programming interface facilitates this task. Virtual machines implementing Neural Network algorithms, Fuzzy Logic engines and Simulated Annealing processing are some of the possibilities to turn GAME into a hybrid parallel programming environment.

9.4. Final Remarks

In these last words, it seems appropriate to stress one point of this research that, for various reasons, could not be made more explicit in the body of a thesis report, but deserves some more consideration. It relates to the actual connection of this research with a major European funded project.

The development of a comprehensive programming environment like GAME, in the context of a project such as PAPAGENA, demanded a number of tasks, with some of them not normally found on a typical Ph.D. research. For instance, planning and co-ordination are key elements in the success of this type of project, which is generally run to a tight timetable. Pragmatic solutions and compromises are often preferred over speculative investigations, in order to increase efficiency and not jeopardise the work being carried out by project partners. Also, permanent interaction with partners, either in the specification stages or during the implementation, is essential to collaborative research. It provides, for both parties, the type of feedback that would never be available on an independent research. Finally, periodic assessment of the work ensures the maintenance of high quality standards for the research. Typically, the development of a project like PAPAGENA is assessed every six months by a panel of independent specialists nominated by the ESPRIT programme. The project review exercise, besides monitoring the accomplishment of tasks, provides an invaluable input for the project by means of the recommendations and suggestions delivered by the reviewers. Moreover, it validates, on a six-monthly basis, the work being carried out by everyone.

References

[1] Agha, G. “ACTORS: A Model of Concurrent Computation in Distributed Systems”, MIT Press, Cambridge, Massachusetts — 1986.

[2] Antonisse, J. “ A New Interpretation of the Schema Notation that Overturns the Binary Coding Constraint” in *Proc. of the Third International Conf. on Genetic Algorithms*, pp.86-91, J. David Schaffer Ed., Morgan Kaufmann Pub., (San Mateo), CA — 1989.

[3] Bäck, T. “Optimal Mutation Rates in Genetic Search” in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 2-9,. Stephanie Forrest — 1986.

[4] Bäck, T. & Hoffmeister, F. “Adaptive Search by Evolutionary Algorithms”.in *Model of Selforganization in Complex Systems (MOSES)*, v. 64, pp.156-163, Werner Ebeling, M. Peschel, and W. Weidlich Ed., Akademie-Verlag, Berlin — 1991.

[5] Bäck, T. & Hoffmeister, F. “Extended Selection Mechanisms in Genetic Algorithms” in *Proc. of the Fourth International Conf. on Genetic Algorithms*, pp. 92-99, Richard K. Belew and Lashon B. Booker Ed., Morgan Kaufmann Pub., San Diego, CA — 1991.

[6] Bäck, T., Hoffmeister, F. & Schwefel, H.P. “A survey of Evolutionsstrategies” in *Proc. of the Fourth International Conf. on Genetic Algorithms*, pp. 2-9, Richard K. Belew and Lashon B. Booker Ed., Morgan Kaufmann Pub., San Diego, CA — 1991.

[7] Bäck, T., Rudolph, G. & Schwefel, H.P. “Evolutionary Programming and Evolutionsstrategies: Similarities and Differences” *in Proceedings of the 2nd Annual Conf. on Evolutionary Programming*, pp. 11-22, David B. Fogel and W. Atmar Ed., Evolutionary Programming Society, La Jolla, San Diego, CA — 1993.

[8] Baker, J.E. “Reducing Bias and Inefficiency in the Selection Algorithm” in *Proc. of the Second International Conf. on Genetic Algorithms and Their Applications*, pp. 14-21, John J. Grefenstette Ed., MIT, Cambridge, MA — 1987.

[9] Barbosa, V.C., *Massively Parallel Models of Computation*, Ellis Horwood Pub., Chichester — 1993.

[10] Bastani, F., Hilal, W. & Ivengar, S.S.“Efficient Abstract Data Type Components for Distributed and Parallel Systems” in *COMPUTER*, IEEE Computer Society Press, Los Alamitos, v. 20, n. 10, pp. 33-44 — 1987.

[11] Beguelin, A. *et al.* “A Users’ Guide to PVM Parallel Virtual Machine” in ORNL/TM-11826, Oak Ridge National Laboratory, Mathematical Sciences Section — 1991.

[12] Bershad, B. N., Lazowska, D. & Levy, H. M. “PRESTO: A System for Object-Oriented Parallel Programming” in *Software-Practice and Experience*, v. 18, n. 8, pp. 713-732, John Wiley, Chichester — 1988.

[13] Bertoni, A. & Dorigo, M. “Implicit Parallelism in Genetic Algorithms” in Genetic Algorithm (Research Note) Artificial Inteligence 61, pp. 307-314, Elsevier Science Publishers B.V. — 1993.

[14] Born, J. & Bellman, K. “Numerical Adaptation of Parameters in Simulation Models by Using Evolution Strategies” in *Molecular Genetic Information Systems: Modelling and Simulation*, pp. 291-320, K. Bellmann Editor, Academie-Verlag, Berlin — 1983.

[15] Bamlette, M.F. & Bouchard, E.E. " Genetic Algorithms in Parametric Design of Aircraft" in Handbook of Genetic Algorithms, pp. 109-123, Lawrence Davis Ed., Van Nostrand Reinhold Pub., NY — 1991.

[16] Bruce, A. *et al.* "CHIMP and PUL: Support for Portable Parallel Computing", *in EPCC-TR93-07*, Edinburg Parallel Computing Centre, The University of Edinburg — 1993.

[17] Butterworth, P., Otis A. & Stein J. "The GEMStone Objetc Database Management System" in *Communications of the ACM*, v. 34, n. 10, pp. 64-77, ACM Press, New York — 1991.

[18] Cheng, W. K. "Distributed Database Management Systems" in *Journal of Object-Oriented Programming*, v. 6, n. 1, pp. 69-74, SIGS Publications, New York — 1993.

[19] Churin, J. & Forgey, S. " Growing Your Own Distributed OODB" in *Object Magazine*, v. 2, n. 5, pp. 67-70, SIGS Publications, New York — 1993.

[20] Cohoon, J.P., Hedge, S.U., Martin, W.N. and Richards, D. "Punctuated Equilibria: a Parallel Genetic Algorithm" in *Proc. of the Fourth Int. Conf. on Genetic Algorithms*, pp. 148-154, Richard K. Belew Ed., Morgan-Kauffman, San Mateo, California — 1987.

[21] Cox Jr., L.A., Davis, L. & Qiu, Y. "Dynamic Anticipatory Routing in Circuit-Switched Telecommunications Networks"in Handbook of Genetic Algorithms, pp. 124-143, Lawrence Davis Ed., Van Nostrand Reinhold Pub., NY — 1991.

[22] Daniels, J. & Cook, S. " Strategies for Sharing Objects in Distributed Systems" in *Journal of Object-Oriented Programming*, v. 5, n. 8, pp. 27-36, SIGS Publications, New York — 1993.

[23] Davis, L. & Steenstrup, M. "Genetic Algorithms and Simulated Annealing" in *Research Notes in Artificial Intelligence*, Morgan-Kauffman, San Mateo, California — 1987.

[24] Davis, L., *Handbook of Genetic Algorithms*, Lawrence Davis Ed., Van Nostrand Reinhold Pub., NY — 1991.

[25] DeJong, K. "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", doctoral dissertation, Univ. Michigan, Ann Arbor, Mich. — 1975.

[26] DeJong, K. "Adaptive System Design: A Genetic Approach", in *IEEE Transactions on Systems, Man and Cybernetics*, v. 10, n. 9, pp. 566-574 — 1980.

[27] Dzikowski, J. "Improving the Performance of Architectures Containing Random Access List Structured Memory", *in Ph.D. Thesis*, submitted to the Department of Computer Science, University College London — 1995.

[28] Dekker, K. & Rehmann, R. "ADE – An Application Development Environment for Transparent Use of Scalable Parallel Architectures" in *Programming Environments for Parallel Computing*, pp. 77-86, AFIP Transactions A11, N. Topham, R. Ibbett & T. Bemmerl Ed., North-Holland Pub., Amsterdam — 1992.

[29] Dekker, L. & Ribeiro Filho, J.L. "The GAME Virtual Machine Architecture" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 93-110, J. Stender Ed., IOS Press, Amsterdam — 1993.

[30] Dekker, L. & Ribeiro Filho, J.L. "GAME Graphic Monitoring System Specification" in *ESPRIT project 6857 - PAPAGENA, WP4.3-TR/93/10*, Department of Computer Science, University College London — 1993.

[31] Deux, O. *et al.* "The O2 System" in *Communications of the ACM*, v. 34, n. 10, pp. 34-48, ACM Press, New York — 1991.

[32] Dorigo, Genetic Algorithms: the State of the Art and Some Research Proposal", in report n. 89-058, Dipartimento di Elettronica, Politecnico di Milano — 1989.

[33] Dorigo, M., Maniezzo, V. " Parallel Genetic Algorithms: Introduction and Overview of Current Research", Dipartimento di Elettronica, Politecnico di Milano — 1991.

[34] East, I. R. & Macfarlane, D. "Implementation in OCCAM of Parallel Genetic Algorithms on Transputer Networks" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 43-63, J. Stender Ed., IOS Press, Amsterdam — 1993.

[35] Eshelman, L. "The CHC Adaptive Search Algorithm: How to Have a Safe Search When Engaging in Nontraditional Genetic Recombination" in *Foundations of Genetic Algorithms and Classifier Systems*, pp. 265-283, Gregory J. Hawlins Ed., Morgan-Kauffman, San Mateo, California — 1991.

[36] Goldberg, D.E."Genetic Algorithms in Search, Optimisation & Machine Learning ", Addison-Wesley, Massachusetts — 1989.

[37] Gordon, S. & Whitley, D. "Serial and Parallel Genetic Algorithms as Function Optimizers" in *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pp. 177-183, S. Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

[38] Gorges-Schleuter, M. "Genetic Algorithms and Population Structures - A Massively Parallel Algorithm", *Ph.D. Thesis*, Computer Science Department, University of Dortmund — 1990.

[39] Gorges-Schleuter, M. "ASPARAGOS: An Asynchronous Parallel Genetic Optimisation Strategy" in *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 422-427, J. David Schaffer Ed., Morgan-Kauffman, San Mateo, California — 1989.

[40] Grefenstette, J.J. "GENESIS: A System for Using Genetic Search Procedures" in *Proc. of the 1984 Conference on Intelligent Systems and Machines*, pp. 161-165 - 1984.

[41] Guttman, M., King, J. A. & Matthews, J. " A Methodology for Developing Distributed Applications" in *Object Magazine*, v. 2, n. 5, pp. 55-59, SIGS Publications, New York — 1993.

[42] Haasdjik, E., Walker, R., Barrow, D. & Gerrets, M. "Genetic Algorithms in Business" in *Genetic Algorithms in Optimisation, Simulation and* Modelling, pp. 157-184, J.Stender, E.Hilebrand & J.Kingon Ed., IOS Press, Amsterdam — 1994.

[43] Harp, S.A. & Smad, T. "Genetic Synthesis of Neural Network Architecture" in *Handbook of Genetic Algorithms*, pp. 202-221, Lawrence Davis Ed., Van Nostrand Reinhold Pub., NY — 1991.

[44] Hartley, C. & Sunderan V. S. "Concurrent Programming with Shared Objects in Networked Environments", Department of Mathematics and Computer Science, Emory University, Atlanta — 1992.

[45] Hockney, R.W. & Jesshope, C.R. *Parallel Computers - Architecture, Programming and Algorithms*, Adam Hilger Ltd, Bristol Publisher — 1981.

[46] Hoffmeister, F., *The User's Guide to Escapade 1.2: A Runtime Environment for Evolution Strategies*, Dept. of Computer Science, Univ. of Dortmund, Germany — 1991.

[47] Holland, J. H., *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press — 1975.

[48] Kingdon, J., Ribeiro Filho, J. & Treleaven, P. "The GAME Programming Environment Architecture" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 85-92, J. Stender Ed., IOS Press, Amsterdam — 1993.

[49] Kafura, D., Mukherji, M. & Lavender, G. "ACT++ A Class Library for Concurrent Programming in C++ Using Actors" in *Journal of Object-Oriented Programming*, v. 6, n. 6, pp. 47-55, SIGS Publications, New York — 1993.

[50] Kim, W. "Architectural Issues in Object-Oriented Databases" in *Journal of Object-Oriented Programming*, v. 6, n. 2, pp. 29-38, SIGS Publications, New York — 1993.

[51] Koza, J.R. "Hierarchical Genetic Algorithms Operating on Populations of Computer Programs" in *Eleventh International Joint Conf. on Artificial Intelligence (IJCAI-89)*, pp. 768-774, N. S.Sridharan Ed., Morgan Kaufmann Publishers, Detroit, MI — 1989.

[52] Koza, J.R "Genetically Breeding Populations of Computer Programs to Solve Problems in Artificial Intelligence" in *Proceedings of the 1990 IEEE International Conference on Tools with Artificial Intelligence (TAI'90)*, IEEE Computer Society Press, Herndon, VA,— 1990.

[53] Koza, J.R., *Genetic Programming: On Programming Computers by Means of Natural Selection and Genetics*, The MIT Press, Cambridge, MA — 1992.

[54] Lamb, Charles *et al.* "The ObjectStore Database System" in *Communications of the ACM*, v. 34, n. 10, pp. 50-63, ACM Press, New York — 1991.

[55] Laurent, P. & Silverio N. "Persistence in C++" in *Journal of Object-Oriented Programming*, v. 6, n. 6, pp. 41-46, SIGS Publications, New York — 1993.

[56] Loomis, M. S. "Distributed Object Databases" in *Journal of Object-Oriented Programming*, v. 6, n. 1, pp. 20-23, SIGS Publications, New York — 1993.

[57] Loomis, M. S. "Making Objects Persistent" in *Journal of Object-Oriented Programming*, v. 6, n.6, pp. 25-28, SIGS Publications, New York — 1993.

[58] Manderick, B. Spiessens, P. "A Massively Parallel Genetic Algorithm: Implementation and First Results" in *Proc. of the Fourth Int. Conf. on Genetic Algorithms*, pp. 279-286, Richard K. Belew Ed., Morgan-Kauffman, San Mateo, California — 1991.

[59] Maruyama, T., Hirose, T. and Konagaya, A. "A Fine-Grained Parallel Genetic Algorithm for Distributed Parallel Systems" in *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pp. 184-190, S. Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

[60] May, D. & Taylor, R., "OCCAM – An Overview" in *Microprocessors and Microsystems*, v. 8, n. 2, pp. 73-79, Butterworth & Co Pub. — 1984.

[61] Mitchell, M., Forrest, S. & Holland, J.H. "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance" in *Toward a Practice of Autonomous System: Proc. of the First European Conf. on Artificial Life*, pp. 245-254, F. J. Varela and P. Bourgine, Ed., MIT Press, (Cambridge), MA — 1991.

[62] Mühlenbein, H. "Evolution in Time and Space - The Parallel Genetic Algorithm" in *Foundations of Genetic Algorithms and Classifier Systems*, pp. 316-337, Gregory J. Hawlins Ed., Morgan-Kauffman, San Mateo, California — 1991.

[63] Mühlenbein, H. & Schilerkamp-Voosen, D. "Optimal Interaction of Mutation and Crossover in the Breeder Genetic Algorithm" in *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pp. 648, S. Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

[64] Muntean, T., Gonzales, N., Langue, Y. "A Parallel Operating System Architecture – PARX kernel approach for the Supernode_II project (P2528)", University of Grenoble, IMAG-Laboratoire de Genie Informatique — 1991.

[65] NASA – Johnson Space Center. "Splicer – A Genetic Tool for Search and Optimization", in *Genetic Algorithm Digest*, v. 5, n. 17 — 1991.

[66] Pettey, C.B., Leuze, M.R., and Grefenstette, J.J. "A Parallel Genetic Algorithm", in *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 398-405, J. David Schaffer Ed., Morgan-Kauffman Pub., (San Mateo), CA — 1989.

[67] Pountain, D. "The Chorous Microkernel" in *Byte*, v. 19, n. 1, pp. 131-136, McGrawHill Pub., Peterborough — 1994.

[68] Radcliffe, N.J. "Forma Analysis and random respectiful recombination" in *Proc. of the Fourth Int. Conf. on Genetic Algorithms*, pp. 222-229, Richard K. Belew Ed., Morgan-Kauffman Pub., (San Mateo), CA — 1991.

[69] Radcliffe, N.J. "Non-linear Genetic Representations" in *Parallel Problem Solving from Nature 2*, pp. 259-268, R. Männer and B. Manderick Ed., Elsevier Science Pub., (Amsterdam) — 1992.

[70] Radcliffe, N.J. & Surry, P.D. "The Reproductive Plan Language RPL2: Motivation, Architecture and Applications" in *Genetic Algorithms in Optimisation, Simulation and Modelling*, pp. 65-94, J.Stender, E.Hilebrand & J.Kingon Ed., IOS Press, Amsterdam — 1994.

[71] Rechenberg, I. "*Evolutionsstrategie: Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*" [Evolutionary strategy: Optimisation of Technical Systems According to the Principles of Biological Evolution], Frommann-Holzboog Verlag, Stuttgard, Germany — 1973.

[72] Ribeiro Filho, J.L. "GAME's Library Structure" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 111-116, J. Stender Ed., IOS Press, Amsterdam — 1993.

[73] Ribeiro Filho & Treleaven, P. "Genetic-Algorithms Programming Environments" in *IEEE COMPUTER*, v. 27, n.6, pp. 28-43, IEEE Computer Society, Los Alamitos, CA — 1994.

[74] Ribeiro Filho, J.L., K. Tout & U. Tiedemann "GAME: A Tool Kit for Exploiting Massively Parallel Genetic Algorithms" in *International Conference Massively Parallel Processing, Applications and Development*, Elsevier Science, Delft, The Netherlands — 1994.

[75] Ribeiro Filho, J.L. & Treleaven, P. "GAME: A Framework for Programming Genetic Algorithms Applications" in *Proceedings of the First IEEE Conference on Evolutionary Computing - IEEE Congress on Computational Intelligence*, pp. 840-845, IEEE Service Center, Piscataway, NJ — 1994.

[76] Ribeiro Filho, J.L "The GAME System" in *IEE Coloquium on Applications of Genetic Algorithms*, London — 1994.

[77] Ribeiro Filho, J.L. & Treleaven, P. "GAME's Parallel Programming Model" in *Genetic Algorithms in Optimisation, Simulation and Modelling*, pp. 111-154, J. Stender, E. Hillbrand & J. Kingdon Ed., IOS Press, Amsterdam — 1994.

[78] Robbins, G. "EnGENEer – The Evolution of Solutions" in *Proceedings of the 5th Annual Seminar on Neural Networks and Genetic Algorithms*, London — 1992.

[79] Rocha, P., Khebbal, S. & Treleaven, P. "A Framework for Hybrid Intelligent Systems" in *Proc. of the First New Zealand International Two-Stream Conf. on Artifical Neural Networks and Expert Systems*, pp. 206-209, IEEE Computer Society Press, Los Alamitos — 1993.

[80] Rounce, P. & Delgado, J. "SPRINT and DICE: Architectures within the ESPRIT SPAN Project" in *IEEE MICRO*, v.10, n.6, pp. 24-27 & 88-97 — 1990.

[81] Schwefel, H.P. "Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie" in *Interdisciplinary Systems Research*, v. 26, Birkäuser, Basel — 1977.

[82] Schwefel, H.P., *Numerical Optimization of Computer Models*, John Wiley, Chischester — 1981.

[83] Sharp, O. "Dynamic Linking under Berkeley UNIX", in *Dr.Dobb's Journal*, v. 18, n. 5, pp. 40-44, Pittsfield, Massachusetts — 1993.

[84] Schraudolph, N.N. & Belew, R.K. "Dynamic Parameter Encoding for Genetic Algorithms"in *CSE Technical Report #CS90-175*, Computer Science & Engineering Department, University of California, San Diego — 1990.

[85] Schraudolph, N.N. & Grefenstette, J.J., *A User's Guide to GAUCSD 1.2*, Computer Science & Engineering Department, University of California, San Diego — 1990.

[86] Sistare, S. *et al.* "Data Visualisation and Performance Analysis in the PRISM Programming Environment" in *Programming Environments for Parallel Computing*, pp. 37-51, AFIP Transactions A11, N. Topham, R. Ibbett & T. Bemmerl Ed., North-Holland Pub., Amsterdam — 1992.

[87] Smart, J., *Reference Manual for wxWindows 1.50: A Portable C++ GUI toolkit*, Artificial Intelligence Applications Institute, University of Edinburgh — 1993.

[88] Stender, J., Addis, T. & Spenceley, E. "Principle-based Engineering and Economic Modelling" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 117-128, J. Stender Ed., IOS Press, Amsterdam — 1993.

[89] Stender, J. "Standort 200: Local Modelling of the Brandenburg Area Using Genetic Algorithms" in *Genetic Algorithms in Optimisation, Simulation and* Modelling, pp. 219-259, J.Stender, E.Hilebrand & J.Kingon Ed., IOS Press, Amsterdam — 1994.

[90] Sunderam, V.S. "PVM: A Framework for Parallel Distributed Computing", Department of Math and Computer Science, Emory University, Atlanta — 1989.

[91] Syswerda, G. "Schedule Optimization Using Genetic Algorithms" in *Handbook of Genetic Algorithms*, pp. 332-349, Lawrence Davis Ed., Van Nostrand Reinhold Pub., NY — 1991.

[92] Tanese, R. "Distributed Genetic Algorithms" in *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 434-440, J. David Schaffer Ed., Morgan-Kauffman, San Mateo, California — 1989.

[93] Tonks, H. "Improving Traditional Object Models" in *Objects in Europe*, v. 1, n. 1, pp. 13-17, Supplent to SIGS publications, T. Durham Editor, SIGS publications, London — 1994.

[94] Tout, K., Ribeiro Filho, J. L., Mignot, B. & Idlebi, N. "A Cross-platform Genetic Algorithms Environment" in *Proceedings of the World Transputer Congress '94*, pp. 79-90, De Gloria A., Jane M.R., Marini D. Ed., IOS Press, Amsterdam — 1994.

[95] Treleaven, P., Nigri, M. & Ribeiro Filho, J.L. "Programming Environments for Neural Networks and Genetic Algorithms" in *Proceedings of the IMACS World Congress*, Atlanta, USA — 1994.

[96] Varhol, P.D. "Small Kernels Hit It Big" in *Byte*, v. 19, n. 1, pp. 119-128, McGrawHill, Peterborough — 1994.

[97] Vose, M.D. "Generalising the Notion of Schema in Genetic Algorithms" in *Artificial Intelligence* — 1991.

[98] Voigt, H. M., Born , J. & Treptow, J., *The Evolution Machine Manual -V 2.1*, Institute for Informatics and Computing Techniques, Berlin — 1991.

[99] Wayner, P. "Objects on the March" in *Byte*, v. 19, n. 1, pp. 139-150, McGrawHill, Peterborough — 1994.

[100] Whitley, D. & Kauth, J. "GENITOR: A Different Genetic Algorithm" in *Proc. of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118-130, Denver, CO — 1988.

[101] Whitley, D. "The Genetic Algorithm and Selective Pressure: Why Rank-Based Allocation of Reproductive Trials is Best" in *Proc. of the Third Int. Conf. on Genetic Algorithms*, pp. 116-121, J. David Schaffer Ed., Morgan-Kauffman, San Mateo, California — 1989.

[102] Whitley, D. "Cellular Genetic Algorithms" in *Proc. of the Fifth Int. Conf. on Genetic Algorithms*, pp. 658, S. Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

[103] Roberts, G.A, Winder, R. & Wei, M. "COOTS - UC++: Concurrent Object-Oriented C++" in *RN/90/68* - Department of Computer Science, University College London — 1990.

[104] *Parallel C++ User's Guide*, 3L Limited, Livingston, Scotland — 1991.

[105] *Programming Enviroments for Parallel Computing*, Topham, N., Ibbett, R. & Bemmerl, T. Editors, IFIP Transactions, North-Holland — 1992.

[106] *Proceedings of the International Conference on Genetic Algorithms*, Morgan-Kauffman, San Mateo, California — 1985.

[107] *Proceedings of the Second International Conference on Genetic Algorithms*, Morgan-Kauffman, San Mateo, California — 1987.

[108] *Proceedings of the Third International Conference on Genetic Algorithms*, J. David Schaffer Ed., Morgan-Kauffman, San Mateo, California — 1989.

[109] *Proceedings of the Fourth International Conference on Genetic Algorithms*, Richard K. Belew Ed., Morgan-Kauffman, San Mateo, California — 1991.

[110] *Proceedings of the Fifth International Conference on Genetic Algorithms*, S Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

Bibliography

Genetic Algorithms

1 . Davis, L. & Steenstrup, M., *Research Notes in Artificial Intelligence,* Morgan-Kauffman, San Mateo, California — 1987.

2 . Davis, L., *Handbook of Genetic Algorithms,* Lawrence Davis Ed., Van Nostrand Reinhold Pub. — 1991.

3 . Goldberg, D.E."Genetic Algorithms in Search, Optimisation & Machine Learning ", Addison-Wesley, Massachusetts — 1989.

4 . Holland, J. H., *Adaptation in natural and artificial systems,* Ann Arbor: The University of Michigan Press — 1975.

5 . Koza, J.R., *Genetic Programming: On Programming Computers by Means of Natural Selection and Genetics,* The MIT Press — 1992.

6 . *Parallel Genetic Algorithms: Theory and Applications,* J. Stender Editor., IOS Press — 1993.

7 . *Genetic Algorithms in Optimisation, Simulation and* Modelling, J.Stender, E.Hilebrand & J.Kingon Editors., IOS Press — 1994.

8 . Rich, E. & Knight, K., *Artificial Intelligence,* McGraw-Hill — 1991.

9 . *Computational Intelligence Imitating Life,* Zurada, J., Marks, R. & Robinson, C. Editors, IEEE Press — 1994.

10. *Foundations of Genetic Algorithms and Classifier Systems,* Gregory J. Hawlins Ed., Morgan-Kauffman — 1991.

11. *Parallel Problem Solving from Nature 2,* R. Männer and B. Manderick Editors, Elsevier Science Pub. — 1992.

12. *IEEE COMPUTER,* v. 27, n.6, IEEE Computer Society — 1994.

13. *Proceedings of the International Conference on Genetic Algorithms,* Morgan-Kauffman, San Mateo, California — 1985.

14. *Proceedings of the Second International Conference on Genetic Algorithms,* Morgan-Kauffman, San Mateo, California — 1987.

15. *Proceedings of the Third International Conference on Genetic Algorithms,* J. David Schaffer Ed., Morgan-Kauffman, San Mateo, California — 1989.

16. *Proceedings of the Fourth International Conference on Genetic Algorithms,* Richard K. Belew Ed., Morgan-Kauffman, San Mateo, California — 1991.

17. *Proceedings of the Fifth International Conference on Genetic Algorithms,* S Forrest Ed., Morgan-Kauffman, San Mateo, California — 1993.

18. *Proceedings of the First IEEE Conference on Evolutionary Computing - IEEE Congress on Computational Intelligence,* pp. 840-845, IEEE Service Center — 1994.

Object-Oriented Design and Programming

1. Booch, G., *Object Oriented Design with Applications*, Benjamin Cummings — 1991.
2. Bronnenberg, W. *et al.* "DOOM: A Decentralized Object-Oriented Machine" in *IEEE MICRO*, IEEE Computer Society Press, Los Alamitos pp. 52-67 — 1987.
3. Coad.,P. & Yourdon, E., *Object-Oriented Design*, Yourdon Press, Prentice Hall Publisher — 1991.
4. Gorlen, K., Orlow, S. & Plexico, P., *Data Abstraction and Object-Oriented Programming*, John Wiley — 1990.
5. Meyer, B., *Reusable Software - The Base Object-Oriented Component Libraries*, Prentice Hall — 1994.
6. "Object-Oriented Programming", *Dr. Dobb's Journal*, vol. 17, issue 10 — 1992.

C/C++ & Windows Programming

1. Clark, *Windows Programming Guide to OLE/DDE*, Sams Publishing — 1992.
2. Coplien, J., *Advanced C++, Programming Styles and Idioms*, Addsion Wesley — 1992.
3. Eckel, B., *C++ Inside & Out*, Osborne McGraw-Hill — 1993.
4. Ellis, M. & Stroustrup, B., *The Annotated C++ Reference Manual*, Addison Wesley — 1991.
5. Hansen, T., *The C++ Answer Book*, Addison Wesley — 1990.
6. Heller, M., *Advanced Windows Programming*, John Wiley — 1992.
7. Meyers, S., *Effective C++*, Addison Wesley — 1992.
8. Plauger, P., *The Standard C++ Library*, Prentice Hall — 1994.
9. Porter, A., *C++ Programming for Windows*, Osborne McGraw-Hill — 1993.
10. Porter, A., *The Best C/C++ Tips Ever*, Osborne McGraw-Hill — 1993.
11. Shapiro, J., *A C++ Toolkit*, Prentice Hall — 1991.
12. Teale, S., *C++ IOStreams Handbook*, Addison Wesley — 1993.
13. Vilot, M., *C++ Programming Power Pack*, Sams Publishing — 1993.
14. Watson, M., *Portable GUI Development with C++*, McGraw-Hill — 1992.
15. *Proceedings of the C plus C++ International Conference*, Boston University Conference Office, London — 1992.
16. *Borland C++ 3.0 - Tools & Utilities Guide*, Borland International Inc. — 1991.
17. *Borland C++ 3.0 - Libraries Reference*, Borland International Inc. — 1991.
18. *Borland C++ 3.0 - Programmers Guide*, Borland International Inc. — 1991.
19. *Borland C++ 3.0 - User's Guide*, Borland International Inc. — 1991.
20. *Turbo Debugger 3.0 - User's Guide*, Borland International Inc. — 1991.
21. *Microsoft Windows User's Guide V 3.1*, Microsoft Corp. — 1992.

22. Maguire, S., *Writing Solid Code*, Microsoft Press. — 1993.

23. McConnel, S., *Code Complete - A Practical Handbook of Software Construction*, Microsoft Press — 1993.

24. Stitt, M., *Debugging - Creative Techniques and Tools for Software Repair*, John Wiley — 1992.

25. C/C++ Programming", *Dr. Dobb's Journal*, vol. 18, issue 8 — 1993.

26. "Software Design and Testing", *Dr. Dobb's Journal*, vol. 19, issue 2 — 1994.

27. "Cross-Platform Development", *Dr. Dobb's Journal*, vol. 19, issue 3 — 1994.

Parallel & Distributed Systems

1. Barbosa, V.C., *Massively Parallel Models of Computation*, Ellis Horwood Pub., Chichester — 1993.

2. *Parallel C++ User's Guide*, 3L Limited, Livingston, Scotland — 1991.

3. Stevens, W.R., *UNIX Network Programming*, Prentice Hall — 1990.

4. "Operating Environments", *Dr. Dobb's Journal*, vol. 18, issue 5 — 1993.

5. "Microkernels and Operating Systems", *Dr. Dobb's Journal*, vol. 19, issue 5 — 1994.

Appendix A

Published Work

1. Ribeiro Filho & Treleaven, P. "Genetic-Algorithms Programming Environments" in *IEEE COMPUTER*, v. 27, n.6, pp. 28-43, IEEE Computer Society, Los Alamitos, CA — 1994.

2. Ribeiro Filho, J.L., K. Tout & U. Tiedemann "GAME: A Tool Kit for Exploiting Massively Parallel Genetic Algorithms" in *International Conference Massively Parallel Processing, Applications and Development*, Elsevier Science, Delft, The Netherlands — 1994.

3. Ribeiro Filho, J.L. & Treleaven, P. "GAME: A Framework for Programming Genetic Algorithms Applications" in *Proceedings of the First IEEE Conference on Evolutionary Computing - IEEE Congress on Computational Intelligence*, pp. 840-845, IEEE Service Center, Piscataway, NJ — 1994.

4. Ribeiro Filho, J.L "The GAME System" in *IEE Coloquium on Applications of Genetic Algorithms*, London — 1994.

5. Ribeiro Filho, J.L. & Treleaven, P. "GAME's Parallel Programming Model" in *Genetic Algorithms in Optimisation, Simulation and Modelling*, pp. 111-154, J. Stender, E. Hillbrand & J. Kingdon Ed., IOS Press, Amsterdam — 1994.

6. Tout, K., Ribeiro Filho, J. L., Mignot, B. & Idlebi, N. "A Cross-platform Genetic Algorithms Environment" in *Proceedings of the World Transputer Congress '94*, pp. 79-90, De Gloria A., Jane M.R., Marini D. Ed., IOS Press, Amsterdam — 1994.

7. Treleaven, P., Nigri, M. & Ribeiro Filho, J.L. "Programming Environments for Neural Networks and Genetic Algorithms" in *Proceedings of the IMACS World Congress*, Atlanta, USA — 1994.

8. Ribeiro Filho, J.L. "GAME's Library Structure" in *Parallel Genetic Algorithms: Theory and Applications*, pp. 111-116, J. Stender Ed., IOS Press, Amsterdam — 1993.

9. Dekker, L. & Ribeiro Filho, J.L. "The GAME Virtual Machine Architecture" in *Parallel Genetic Algorithms*, pp. 93-110, J. Stender Ed., IOS Press, Amsterdam — 1993.

10. Kingdon, J., Ribeiro Filho, J. & Treleaven, P. "The GAME Programming Environment Architecture" in *Parallel Genetic Algorithms*, pp. 85-92, J. Stender Ed., IOS Press, Amsterdam — 1993.

Appendix B

VM-API

This appendix lists and describes the functions of GAME's Virtual Machine Application Program Interface. Each function is identified by a heading showing its name, which is followed by a summary description of its task. The declaration of each function as well as the description of their arguments and returned data are also provided.

Pool Manipulation Functions:

CreatePool

Function Make a new population pool.

Syntax `hPOOL CreatePool(WORD pool_sz, [Individual& prototype = 0]);`

pool_sz	The population initial size.
prototype	A handle to a prototype, or model, of the genetic structure that will be used to initialise the population pool.

Remarks **CreatePool** is used to create an empty population pool on the virtual machine's population manager module. Its first argument specifies the number of genetic structures the pool should accommodate. The second argument is optional, and is a prototype genetic structure that should be used to initialise the pool.

Note that the prototype genetic structure should not contain any DataUnit object, unless the population is meant to be initialised with the very same copy of the genetic structure, including its DataUnit objects. This option is intended for those applications having fixed genetic structures. In such cases, the pool can be created with many "empty" copies of the genetic structure, and later the initialisation operator has only to "send" the DataUnit objects to be connected to each genetic structure in the pool.

Return Value If no errors occurs, **CreatePool** returns a handler which uniquely identifies the newly created pool. This handler must be used to identify this particular pool on any other function that may involve pool or genetic structure manipulations. Otherwise NULL is returned.

DeletePool

Function Delete a population pool.

Syntax

```
WORD DeletePool([hPOOL pool_handle=0]);
```

pool_handle	A handle which uniquely identifies a VM pool (optional).

Remarks **DeletePool** is normally used at the end of a "generation processing" on a *generational* GA. This type of genetic algorithms dictate that the *best* genetic structures should survive to reproduce. They can be copied to a new pool to reproduce and suffer genetic modifications. In such cases, the old pool containing the parents of the new generation may no longer be needed. The single, optional, argument for this function is a pool handle. If not given, all pools of a virtual machine are deleted.

Return Value If no errors occurs, the number of pools deleted is returned, otherwise it returns NULL.

CopyPool

Function Make a copy of a population pool, including its contents.

Syntax

```
hPOOL CopyPool(hPOOL src_handle, [hPOOL dest_handle = 0]);
```

src_handle	The population pool handle which identifies the pool that will be copied.
dest_handle	The optional population pool handle, if already created, identifying the pool into which the contents of the source pool will be copied.

Remarks **CopyPool** is used to make copies of population pools, including all its genetic structures and their DataUnit objects. The first argument is a pool handler which specifies the source pool. The second argument is optional and if not given, indicates that a new pool must be created before the copy is done; otherwise the Individuals of the source pool are copied into the specified destination pool.

Return Value If no errors occurs, **CopyPool** returns a handler which uniquely identifies the newly created pool. This handler must be used to identify this particular pool on any other function that may involve pool or genetic structure manipulations. Otherwise NULL is returned.

GetPoolSize

Function Request the maximum number of genetic structures that a population pool can accommodate.

Syntax

```
WORD GetPoolSize(hPOOL pool_handle);
```

pool_handle	A handle which uniquely identifies a VM pool.

Remarks **GetPoolSize** is used to obtain the maximum number of genetic structures (Individuals) that a particular population pool can accommodate. This value is the argument of the CreatePool function.

Return Value If no error occurs, this function returns the maximum number of genetic structures that a population pool can maintain, otherwise NULL is returned.

GetPopulation

Function Request the current number of genetic structures maintained by a virtual machine.

Syntax

```
WORD GetPopulation([hPOOL pool_handle=0]);
```

pool_handle	A handle which uniquely identifies a VM pool.

Remarks **GetPopulation** requests the current number of genetic structures (Individuals) maintained by a virtual machine pool. If its optional argument is given, it returns only the current number of individuals in the pool specified. In general, the size (population) of a pool does not vary during a GA simulation.

Return Value If no error occurs, this function returns the current number of genetic structures in a population pool, otherwise NULL is returned.

Individual Manipulation Functions:

GetIndividual

Function Get a copy of an Individual object from a pool.

Syntax

```
Individual GetIndividual(NodePath& node_path);

DOUBLE GetIndividualValue(NodePath& node_path,
                          [WORD data_unit_index=0]);
```

node_path	The NodePath object. This object should contain the individual's pool handle and its index in that pool, in this order.
data_unit_index	An optional argument which specifies the DataUnit index for the *phenotype* value associated to one of the Individual's chromosomes

Remarks **GetIndividual** is used to get a copy of a genetic structure (an Individual's *genotype*) maintained by a population manager pool, or its *phenotype* value (the second format). Its first argument is a NodePath object which should contain two addressing items: the pool handle and the Individual's index in the pool. The second argument is optional and specifies the index of the phenotype value stored by the Individual.

Return Value If no error occurs, this function returns a copy of the Individual object or, in the second format, the value of its phenotype.

PutIndividual

Function Insert a copy of an Individual object into a pool.

Syntax

```
void PutIndividual(NodePath& node_path,
                   Individual& genotype);
```

node_path	The NodePath object. This object should contain the individual's pool handle and its index in that pool, in this order.
genotype	The Individual object that will be copied into the specified pool, in the specified position.

Remarks **PutIndividual** is used to insert (or overwrite) an Individual object into a pool maintained by a population manager. The first argument is a NodePath object which should contain two addressing items: the pool handle and the Individual's index. The second argument the Individual object to be copied into the pool.

Return Value No value is returned by this function.

CopyIndividual

Function Copy Individual objects.

Syntax

```
void CopyIndividual(NodePath& src_node_path,
                    NodePath& dest_node_path);
```

src_node_path	The NodePath object that contains the addressing sequence to the *source* Individual object. This object should contain the individual's pool handle and its index in that pool, in this order.
dst_node_path	The NodePath object that contains the addressing sequence to the *destination* Individual object. This object should contain the Individual's pool handle and its index in that pool, in this order.

Remarks **CopyIndividual** is used to make copies of genetic structures (Individual objects) which may be located in any of the population manager's pools. Individuals can be copied into the same pool or between different pools. The first argument is a NodePath object handler which specifies the *source* Individual, the second argument specifies the "address" of the *destination* Individual. In general, this function is used to make copies of the *best* Individuals (parents) from a population to the next generation pool - the reproduction phase of a GA. The copied Individuals then undergo genetic modifications leading to the new population (offspring).

Return Value No value is returned by this function.

KillIndividual

Function Remove an Individual object from a pool.

Syntax

```
void KillIndividual(NodePath& node_path);
```

node_path	The NodePath object. This object should contain the individual's pool handle and its index in that pool, in this order.

Remarks **KillIndividual** is used to remove an Individual from a population manager's pool. A *Steady State* GA, for instance, replaces only one individual in a population (or a subset) and then re-evaluates the whole population. This function is mostly indicated for this type of GAs.

Return Value No value is returned by this function.

MoveIndividual

Function Move an Individual object in the same or between pools.

Syntax

```
void MoveIndividual(NodePath& org_node_path,
                    NodePath& dest_node_path);
```

org_node_path	The NodePath object that contains the origin path of the Individual object. This object should contain the individual's pool handle and its index in that pool, in this order.
dst_node_path	The NodePath object that contains the addressing sequence to the *destination* Individual object. This object should contain the Individual's pool handle and its index in that pool, in this order.

Remarks **MoveIndividual** is used to move genetic structures (Individual objects) in any of the population manager's pools. Individual objects can be moved between two different pools or in the same pool, by modifying its index identifier in the population. The first argument is a NodePath object handler which specifies the *original* addressing sequence of the Individual object to be moved, the second argument specifies its *destination* path. This function can be used to implement *migration* operators in a multi-pool GA.

Return Value No value is returned by this function.

DnaNode Manipulation Functions:

GetNode

Function Get a copy of a DnaNode object from a pool.

Syntax

```
DnaNode GetNode(NodePath& node_path);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that uniquely identify the particular DnaNode object.

Remarks **GetNode** is used to get a copy of any genetic structure from a population manager pool. Its single argument is a reference (handle) to a NodePath object which should contain the full addressing sequence that uniquely identifies a DnaNode object.

Return Value If no errors occurs, this function returns a handler to a copy if the DnaNode object requested, otherwise NULL is returned.

PutNode

Function Insert a DnaNode object into a genetic structure maintained by a pool.

Syntax

```
void PutNode(NodePath& node_path, DnaNode& dna_node);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that uniquely identify the particular DnaNode object.
dna_node	The DnaNode object that will be copied into the genetic structure of the pool and Individual object, as specified in the first argument.

Remarks **PutNode** is used to "write" a copy of a DnaNode object into any population manager's genetic structure. Its first argument is a NodePath object that should contain the full addressing sequence that identifies the place where the DnaNode object is to be connected. If another DnaNode object is already connected to the specified location, it is deleted and replaced by the new one.

Return Value No value is returned by this function.

CopyNode

Function Copy DnaNode objects between two genetic structures.

Syntax

```
void CopyNode(NodePath& src_node_path,
              NodePath& dest_node_path);
```

src_node_path	The NodePath object that contains the addressing sequence to the *source* DnaNode object.
dst_node_path	The NodePath object that contains the addressing sequence to the *destination* DnaNode object.

Remarks **CopyNode** is used to make copies of genetic structures (DnaNode objects) which may be in any of the population manager's pools. DnaNode objects can be copied in the same genetic structure, or between two genetic structures, which can be located in the same, or different pools. The first argument is a NodePath object which specifies the *source* DnaNode object, the second argument specifies the addressing sequence of the *destination* DnaNode object.

Return Value No value is returned by this function.

MoveNode

Function Move DnaNode objects between two genetic structures.

Syntax

```
void MoveNode(NodePath& org_node_path,
              NodePath& dest_node_path);
```

org_node_path	The NodePath object that contains the addressing sequence of the DnaNode object's *original* location.
dst_node_path	The NodePath object that contains the addressing sequence specifying the *destination* location for the DnaNode object.

Remarks **MoveNode** is used to move genetic structures (DnaNode objects) from/to any other genetic structure on any of the population manager's pools. DnaNode objects can be moved between nodes of the same, or different, genetic structures. These can be located in the same or different pools as well. The first argument is a NodePath object which specifies the *original* addressing sequence of the DnaNode object to be moved, the second argument specifies its *destination* path.

Return Value No value is returned by this function.

DeleteNode

Function Remove a DnaNode object from a genetic structure.

Syntax

```
void DeleteNode(NodePath& node_path);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.

Remarks **DeleteNode** is used to remove DnaNode objects from any genetic structure maintained by the VM.

Return Value No value is returned by this function.

SwapNodes

Swap two DnaNodes.

```
void SwapNodes(NodePath& node_path1, NodePath& node_path2);
```

node_path1	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the first DnaNode object to be swapped.
node_path2	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the second DnaNode object to be swapped.

Remarks **SwapNodes** is used to exchange two DnaNode objects between two genetic structures. The genetic structures can be part of the same Individual object, for instance, or DnaNodes of different Individual objects. This function can be used to implement *crossover* operators by selecting and exchanging DnaNodes of two Individual objects in a pool.

Return Value No value is returned by this function.

InvertNode

Function Invert the sequence of DnaNode objects connected to a node of the genetic structure.

Syntax

```
void InvertNode(NodePath& node_path, [WORD num_nodes=ALL]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
num_nodes	The optional number of nodes, from the first node that will have their sequence inverted. If this argument is not given, the whole sequence is inverted into the node.

Remarks **InvertNode** is used to modify a genetic structure by inverting part or the whole sequence of DnaNode objects directly connected to another DnaNode object. The inversions always start at the first position of the node array. An optional argument can be given to specify the number of connections to invert, only if part of the sequence is to be inverted. This option is particularly useful for inverting a set of connections that does not start at the first position of the node array. In this case more, than one call to this function is necessary. This function can be used to implement *inversion* operators.

Return Value No value is returned by this function.

GetNumNodes

Function Request the partial or total number of DnaNode objects connected to another DnaNode object in the specified genetic structure.

Syntax

```
WORD GetNumNodes(NodePath& node_path,
                        [WORD level=ONE_LEVEL]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
level	Specifies whether only the number of DnaNode objects connected directly to the addressed node is to be counted, or all nodes of its branch, form its position downwards is to be counted.

Remarks **GetNumNodes** is used to obtain the current number of connections to DnaNode objects held by a particular DnaNode object in the genetic structure. The function's first argument specifies the addressing sequence of the inquired DnaNode object. The second, optional, argument is used to specify whether only the number of DnaNodes directly connected to the addressed node are to be counted (ONE_LEVEL - the default), or all the connections in the genetic structure, from the addressed node downwards are to be counted (ALL_LEVELS).

Return Value An integer value is returned with the current number of connections to the specified DnaNode object, according to the *level* argument.

GetMaxNodes

Function Request the maximum number of connections supported by a DnaNode object.

Syntax

```
WORD GetMaxNodes(NodePath& node_path);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.

Remarks **GetMaxNodes** is requests the maximum number of connections that a particular DnaNode object in the genetic structure can support.

Return Value The maximum number of connections that the specified DnaNode object supports is returned in a integer value.

DataUnit Manipulation Functions:

ReadData

Function Get a copy of a DataUnit object.

Syntax

```
DOUBLE ReadData(NodePath& node_path, [WORD index=0]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
index	An optional argument which specifies the position of the DataUnit object in the DnaNode's data array. If no value is given, the first position is assumed.

Remarks **ReadData** is used to get a copy of a DataUnit object from any genetic structure of a population manager's pool. Its first argument is a NodePath object that should contain the full addressing sequence that uniquely identifies the DnaNode object containing the required DataUnit object. The second argument, if given, specifies the position for the DataUnit object in the DnaNode's data array. This is an optional argument which, if omitted, indicates that the first DataUnit object is to be addressed. This function can be used to implement *mutation* operators. By requesting DataUnit objects from a genetic structure, the mutation operator can change their values and put them back to their original place in the genetic structure.

Return Value If no errors occurs, this function returns the value of the DataUnit object requested, otherwise NULL is returned.

WriteData

Function Connect a DataUnit object to a genetic structure.

Syntax

```
void WriteData(NodePath& node_path, DOUBLE value,
               [WORD index=0]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
value	The value to be written to the DataUnit of the addressed DnaNode object.

index	An optional argument which specifies the position of the DataUnit object in the DnaNode's data array. If no value is given, the first position is assumed.

Remarks **WriteData** is used to "write" a copy of a DataUnit object into any genetic structure of a population manager's pool. Its first argument is a NodePath object which should contain the full addressing sequence that identifies the DnaNode object that contains the DataUnit object. The third argument, if given, specifies the position for the DataUnit object in the DnaNode's data array. This is an optional argument which, if omitted, indicates that the first DataUnit object is to be addressed.

Return Value No value is returned by this function.

CopyData

Function Copy DataUnit objects between two DnaNode objects.

Syntax

```
void CopyData(NodePath& src_node_path,
              NodePath& dest_node_path,[WORD index=0]);
```

src_node_path	The NodePath object that contains the addressing sequence to the DnaNode object holding the source DataUnit.
dest_node_path	The NodePath object that contains the addressing sequence to the DnaNode object into which a copy of the source DataUnit will be placed.
index	An optional argument which specifies the position of the DataUnit object in both DnaNode's data arrays. If no value is given, the first position is assumed.

Remarks **CopyData** is used to make copies of DataUnit objects. They can be copied between DnaNode objects of the same genetic structure (Individual object), or between two different genetic structures; which can be located in the same or different pools. The first argument specifies the "address" of the DnaNode object which contains the DataUnit object to be copied. The second argument the "address" of the DataUnit object which will receive the new DataUnit object. If a DataUnit object is already connected at the destination DnaNode object, it is deleted to be replaced by the new one. The third argument specifies which of the DataUnit objects connected to the source DnaNode object is to be copied. The new object is connected to the same in the destination DnaNode. This is an optional argument, and if not specified, the first position of the data array is assumed.

Return Value No value is returned by this function.

DeleteData

Function Remove a DataUnit object from a genetic structure.

Syntax

```
void DeleteData(NodePath& node_path, [WORD index=0]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
index	An optional argument which specifies the position of the DataUnit object in the DnaNode's data array. If no value is given, the first position is assumed.

Remarks **DeleteData** is used to remove a DataUnit object from any genetic structure. Its first argument is a NodePath object that should contain the full addressing sequence that identifies the DnaNode object containing the required DataUnit object. The second argument, if given, specifies the position for the DataUnit object in the DnaNode's data array. This is an optional argument which, if omitted, indicates that the first DataUnit object is to be addressed.

Return Value No value is returned by this function.

SwapData

Function Swap two DataUnit objects between two DnaNode objects.

Syntax

```
void SwapData(NodePath node_path1, NodePath& node_path2,
              [WORD index=0]);
```

node_path1	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the first DnaNode object to have one of its DataUnit objects swapped.
node_path2	The handle to an NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the second DnaNode object to have one of its DataUnit objects swapped.
index	An optional argument which specifies the position of the DataUnit object in both DnaNode's data arrays. If no value is given, the first position is assumed.

Remarks **SwapData** is used to exchange two DataUnit objects between two DnaNode objects. They can be part of the same genetic structure (in the same or different pool), or DnaNodes from two different Individual objects. The first argument specifies the "address" of the DnaNode object which contains the first DataUnit object to swap. The second argument specifies the "address" of the second DataUnit object. The third argument specifies which of the DataUnit objects from the data array are to be swapped. This is an optional argument, and if not specified, the first position of the data array is assumed.

Return Value No value is returned by this function.

GetNumUnits

Function Get the current number of DataUnit objects connected to a DnaNode object.

Syntax `WORD GetNumUnits(NodePath& node_path);`

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that uniquely identify the particular DnaNode object.

Remarks **GetNumUnits** requests the current number of DataUnit objects connected to a DnaNode object. The "address" of the DnaNode object is given by a NodePath object in its single argument.

Return Value An integer value with the current number of connections to DataUnit objects is returned.

GetDataStatus

Function Get the validity status associated to a DataUnit object connected to a DnaNode object.

Syntax `BOOL GetDataStatus(NodePath& node_path, [WORD index=GLOBAL]);`

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
index	An optional argument which specifies the position of the DataUnit object in the DnaNode's data array.

Remarks **GetDataStatus** is used to check the validity status of a DataUnit object connected to a DnaNode object. Its first argument indicates the address of the DnaNode object. The second is optional and indicates the position of the DataUnit object's *status flag* in the `_valid_data` variable. If not given, the first DataUnit of the data array is assumed. This function can be used in conjunction with the `SetDataStatus` to implement polyploid genetic structures. I such cases, a number of DataUnit objects is stored into a DnaNode, but only one has its status flag set as valid - the *dominant allele.* All other DataUnit objects are then *recessive alleles*. The manipulation of the status flags is responsibility of the genetic operator that uses this feature.

Return Value A Boolean value is returned by this function indicating TRUE if the status flag associated to the DataUnit is set; otherwise FALSE is returned.

SetDataStatus

Function Get the validity status associated to a DataUnit object connected to a DnaNode object.

Syntax

```
void SetDataStatus(NodePath& node_path, BOOL flag,
                              [WORD index=0]);
```

node_path	The NodePath object. This object should contain a pool handle, and all the necessary addressing items that identify the particular DnaNode object.
flag	The condition to set the status flag associated to the DataUnit object.
index	An optional argument which specifies the position of the DataUnit object in the DnaNode's data array.

Remarks **SetDataStatus** is used to modify the validity status of a DataUnit object connected to a DnaNode object. Its first argument indicates the address of the DnaNode object. The second the new condition: TRUE or FALSE. And the third, which is optional, indicates the position of the DataUnit object's *status flag* in the `_valid_data` variable. If not given, the first DataUnit of the data array is assumed. This function can be used in conjunction with the **`GetDataStatus`** to implement polyploid genetic structures. In such cases, a number of DataUnit objects is stored into a DnaNode, but only one has its status flag set as valid - the *dominant allele*. All other DataUnit objects are then *recessive alleles*. The manipulation of the status flags is responsibility of the genetic operator that uses this feature.

Return Value No value is returned by this function.

Fitness Evaluator functions:

EvaluateFitness

Function Evaluate genetic structures according to the application-dependent objective function.

Syntax

```
void EvaluateFitness([NodePath& node_path =ALL]);
```

node_path	The NodePath object. This is an optional argument that should contain an Individual's pool handle and its index in that pool, in this order.

Remarks **EvaluateFitness** evaluates the fitness of the whole population or the fitness of a single Individual, as specified by the optional argument. Calling this function prior to GetFitness allows the virtual machine to better exploit parallelism of fitness evaluations via its parallel support module.

Return Value No value is returned by this function..

GetFitness

Function Request the fitness value of a genetic structure.

Syntax

```
DOUBLE GetFitness(NodePath& node_path, [WORD index=0]);
```

node_path	The NodePath object. This object should contain an Individual's pool handle and its index in that pool, in this order.
index	An optional argument which specifies which of fitness values (in multi-fitness applications) stored by an Individual object is required.

Remarks **GetFitness** requests fitness values to the virtual machine. The first argument specifies the "address" of the Individual object to be evaluated by the fitness evaluator. The second argument is optional, and can be used to implement multi-fitness applications. It specifies which of the DataUnit objects connected to an Individual data array, contains the requested fitness value. A mechanism that monitors any change in the genetic structure of Individual objects is used in conjunction with their ability to store DataUnit objects, to operate as a "cache memory" for fitness values. When an Individual object is evaluated by an application dependent fitness function, the value returned is stored into a DataUnit object, and connected to the Individual's data array in the position specified by *index* (if this is not given, the first position of data array is assumed by default). Hence, if no change occurs to its genetic structure before this function is called for a second time, the value stored in the DataUnit is returned. This mechanism is used to avoid re-evaluations when no change is made in the Individual's genetic structure.

Return Value This function returns the fitness value requested.

GetTotalFitness

Function Sum the fitness of a population.

Syntax

```
DOUBLE GetTotalFitness( [hPOOL pool_handle=0],
                        [WORD index=0]);
```

pool_handle	A handle which identifies a VM pool (optional).
index	An optional argument which specifies which of fitness values (in multi-fitness applications) stored by an Individual object is required is to be considered.

Remarks **GetTotalFitness** is called to compute a value which represents the sum of the fitness values of all Individuals in the population maintained by a virtual machine. Its first argument is optional and specifies a particular pool (if the VM contains more than one), and the second can be used by multi-fitness applications to specify the particular fitness values to be computed.

Return Value This function returns the total fitness value of population pools

GetAvgFitness

Computes the average fitness of a population.

```
DOUBLE GetAvgFitness([hPOOL pool_handle=0],[WORD index=0]);
```

pool_handle	A handle which uniquely identifies a VM pool (optional).
index	An optional argument which specifies which of fitness values (in multi-fitness applications) stored by an Individual object is required is to be considered.

Remarks **GetAvgFitness** is called to compute the average (total fitness / population) of the fitness values of all Individuals in the population maintained by a virtual machine. Its first argument is optional and specifies a particular pool (if the VM contains more than one), and the second can be used by multi-fitness applications to specify the particular fitness values to be considered.

Return Value This function returns the average fitness value of a population.

GetHighestFitness

Function Gets a population's highest fitness value and its "owner".

Syntax

```
BestIndividual* GetHighestFitness([hPOOL pool_handle=0],
                                  [WORD index=0]);
```

pool_handle	A handle which uniquely identifies a VM pool (optional).
index	An optional argument which specifies which of fitness values (in multi-fitness applications) stored by an Individual object is required is to be considered.

Remarks **GetHighestFitness** is called to request a value which represents the highest of the fitness values among the Individuals in the population maintained by a virtual machine. Its first argument is optional and specifies a particular pool (if the VM contains more than one), and the second can be used by multi-fitness applications to specify the particular fitness values to be considered.

Return Value If no errors occurs, this function returns a pointer to the `BestIndividual` structure described below, otherwise NULL is returned.

```
struc BestIndividual
{
    DOUBLE      best_value;
    NodePath    individual_address;
};
```

best_value	The highest fitness value of the population.
individual_value	A NodePath object containing the full addressing sequence that identifies the Individual of the population with the highest fitness value.

GetLowestFitness

Function Request a population's lowest fitness value and its "owner".

Syntax

```
BestIndividual* GetLowestFitness([hPOOL pool_handle=0],
                                  [WORD index=0]);
```

pool_handle	A handle which uniquely identifies a VM pool (optional).
index	An optional argument which specifies which of fitness values (in multi-fitness applications) stored by an Individual object is required is to be considered.

Remarks **GetLowestFitness** is used to obtain a value which represents the lowest of the fitness values among the Individuals in the population maintained by a virtual machine. Its first argument is optional and specifies a particular pool (if the VM contains more than one), and the second can be used by multi-fitness applications to specify the particular fitness values to be considered.

Return Value If no errors occurs, this function returns a pointer to the **BestIndividual** structure described below, otherwise NULL is returned.

```
struc BestIndividual
{
     DOUBLE      best_value;
     NodePath    individual_address;
};
```

best_value	The lowest fitness value of the population.
individual_value	A NodePath object containing the full addressing sequence that uniquely identifies the Individual of the population with the lowest fitness value.

Error Functions:

GetErrorStatus

Function	Get the status condition of the most recent VM-API function call.
Syntax	`MSGSTATUS GetErrorStatus(void);`
Remarks	**GetErrorStatus** is called to obtain the execution status of the latest call to VM-API functions. Every call to a VM-API function resets the internal status word. After an API command is executed by the virtual machine, it may return a message package to its sender with a command completion status. If the command execution is successful the virtual machine (by default) will not return the message package, to avoid undesirable overheads. However, on error conditions the message package will always be returned with the error status.
Return Value	The current value stored in the VM-API internal status word.

ClearErrorStatus

Function	Resets VM-API internal status word to a NO_STATUS condition.
Syntax	`void ClearErrorStatus(void);`
Remarks	**ClearErrorStatus** - the API internal status word keeps the condition of the latest function call until either a function is called or the status is cleared by the user, calling this function.
Return Value	This function does not return any value..

Appendix C

PEM-API

The functions of the PEM-API, listed below, are identified by a heading with their names. For each function a short description of its task, declaration syntax, arguments taken and returned result are provided.

Upper-Layer Functions:

StartComponent

Function Start a local or external components.

Syntax `HANDLE StartComponent (dComp component_descriptor);`

component_descriptor	A structure that contains the following:
component_name	Specifies whether the component to be started is LOCAL or EXTERNAL.
component_name	Specifies the name (including the path) of the component to be started.
port_name	specifies the name to be given to the component communication port. This argument is not necessary for a LOCAL component.
host_name	Optional. Specifies the name of the host where the component will be started.

Remarks **StartComponent** is called to load and start a component. The information required to start either a LOCAL or an EXTERNAL components are provided by the caller in the *component_descriptor*. If of the *port_name* field is not provided, it defaults to the component name used in MS-Windows. The UNIX implementations, however, require a numeric string representing a socket port number.

Return Value This function returns a handle that identifies an entry in the component data base. This handle is used by other PEM functions such as OpenConnection to get and update information related to this component in the data base. ERROR is returned if the component could not be started.

TerminateComponent

Function Terminate the execution of a local or external component.

Syntax

```
void TerminateComponent (HANDLE component_handle);
```

component_handle	Identifies the entry in the component data base that contains the relevant information about this component.

Remarks **TerminateComponent** may not terminate an external component, if it has other external components connected. In this case, only the connection (if there is any one active) is closed. Local components are always terminated. They force disconnection to all other components that are connected to them.

Return Value This function does not return any value.

OpenConnection

Function Create a bi-directional connection between two components.

Syntax

```
HANDLE OpenConnection (dComp& component_descriptor);
HANDLE OpenConnection (HANDLE component_handle);
```

component_descriptor	A structure that describes the PEM component to be connected .
component_handle	Identifies an entry in the component data base which has the description of the component to be connected.

Remarks **OpenConnection,** in its first format, is used when a component is already loaded and is not be known by the local data base. In this case a new entry is created with the information provided by the ComponentDescriptor structure and a connection is opened. The second format is used when a component has been already registered with the data base.

Return Value This function returns a handle that identifies an entry in the component data base. ERROR is returned if the connection could not be created or, in the second format, the handle is invalid.

CloseConnection

Function Terminate a connection between two components. This function may also be called to terminate all the connections maintained by a component.

Syntax

```
WORD CloseConnection (HANDLE component_handle);
```

component_handle	Identifies the entry in the component data base that contains the relevant information about this component.

Remarks **CloseConnection** updates, the with an `INACTIVE` connection status, the entries for both components in their respective component data bases, after disconnection. If later, a new connection to the same component is required, the entry can still be identified by its component handle.

Return Value This function returns the number of active connections of this component. ERROR is returned if the handle is invalid.

PostMail

Function Send message packages to other components.

Syntax

```
BOOL PostMail (MsgPackage& msg_package,
               HANDLE hComp,
               WORD [mode = NOREPLY],
               WORD [msg_id = 0]);
```

msg_package	The message package object that contains the message to be sent.
hComp	Identifies the entry in the component data base that contains information about the connection channels and status.
mode	Transmission mode. There are three modes: NOREPLY, REPLY and WAITREPLY. The first two specify asynchronous transmission and the third is synchronous. This argument defaults to NOREPLY.
msg_id	Optional. Stamps the message with an identifier specified by the user.

Remarks **PostMail** sends a message package object via the communication channel associated with the component specified by *hComp*. If *hComp* = 'ALL', the message package is broadcasted to all components. The delivery mode is specified in the third argument and defaults to NOREPLY, representing an asynchronous delivery, without reply. If REPLY is specified, an asynchronous message is sent but a reply is expected after the command is executed. WAITREPLY specifies a synchronous delivery mode. PostMail then waits for the other component to return the same message package. It is also possible to specify an identification tag for a message package by setting the *msg_id* argument.

Return Value This function returns TRUE if the message was successfully delivered, FALSE otherwise.

HasMail

Function Indicates the presence of message packages in the mailbox.

Syntax `WORD HasMail (WORD [msg_id=0]);`

msg_id	This optional argument specifies a particular identifier for the message to be checked in the queue. It defaults to '0' which means any message.

Remarks **HasMail** checks if there are messages in the local message queue. The argument of this function may be used to specify a particular message package.

Return Value This function returns the number of message packages to be processed found in the queue.

CollectMail

Function Retrieves a message package from the mailbox message queue.

Syntax `MessagePackage* CollectMail (WORD [msg_id=0]);`

msg_id	This optional argument specifies a particular identifier for the message to be checked in the queue. It defaults to '0' which means any message.

Remarks **CollectMail** takes a message package off the local message queue. The argument may be used to identify a particular message in the queue.

Return Value This function returns a pointer to the message package retrieved. If there is no message package in the queue it then returns 0;

ReplyMail

Function Return a message package to its sender.

Syntax `BOOL ReplyMail (MsgPackage& msg_package);`

msg_package	The message package object that contains the message to be posted back to the sender.

Remarks **ReplyMail** sends a message package object back to its sender. Only the messages sent with the REPLY mode specifier are sent back. This is a asynchronous transmission mode. A message package is normally replied with data and status resulting from a command execution.

Return Value It returns `TRUE` if the message was successfully delivered, `FALSE` otherwise.

WaitMail

Function This function is used for synchronisation. It implements a barrier mechanism adapted for event-driven processing

Syntax

```
void WaitMail (WORD msg_id,
               MsgPackage* notify_msg,
               WORD [num_msg = 1]);
```

msg_id	Specifies a particular identifier for the message to be checked in the queue.
notify_msg	The user defined message package object to be set to component (itself) when WaitMail exits.
num_msg	Specifies the number of messages to wait for.

Remarks **WaitMail** waits until the message package(s) specified by 'msg_id' arrives. A user defined message package object (*notify_msg*) must be provided. This message package is inserted into the local message queue this component has received the number of messages specified by *num_msg*.

Note: the user provided message package should have a command, defined by the user, to enable his application to proceed after all the messages containing the required msg_id arrives. This function implements a non-blocking synchronisation mechanism.

ProcessMail

Function This is a user defined function. It is called by the mailbox to have a message package processed.

Syntax

```
MsgPackage& ProcessMail (MsgPackage& msg);
```

msg	The message package to be processed.

Remarks **ProcessMail** processes all message packages retrieved from the local queue by the mailBox. This function is defined by the application developer. The implementation of this function must update the status of the message package received in its argument, before returning.

Return Value It returns the same message package. The programmer of this function should always update the status of the message package after is command is executed.

ProcessReply

Function This is a user defined function. It is called by the mailbox when a message package of type REPLY arrives.

Syntax

```
void ProcessReply (MsgPackage& msg);
```

msg	The message package to be processed.

Remarks **ProcessReply** processes all message packages sent by a component in REPLY mode, replied by other components. This function is defined by the application developer to handle asynchronous replies. Normally the status of the message would be examined in this function.

Return Value This function does not return any value.

Lower-Layer Functions:

StartLocalComponent

Function Start local components.

Syntax

```
HANDLE StartLocalComponent (char* comp_name, char* cmdline);
```

comp_name	Name and path of the component's executable file to be loaded.
cmdline	Command line argument to be passed on to the component being loaded/started.

Remarks **StartLocalComponent,** in general, uses dynamic linking mechanisms to load the required component (the child) into the caller (parent) address space (some implementations may create local threads instead). This function returns a unique handle identifying the newly loaded component.

Return Value

StartExtComponent

Function Start external components.

Syntax

```
HANDLE StartExtComponent (char* comp_name,
                          char* cmdline,
                          char* [comp_host =0]);
```

comp_name	Name and path of the component's executable file to be started.
cmdline	Command line argument to be passed on to the started component.
comp_host	Name of the host machine that will run the component (optional).

Remarks **StartExtComponent** starts a new independent process in the local or user specified host.

Return Value This function returns a unique handle identifying the newly started component

TerminateLocalComponent

Function Unloads a local component.

Syntax

```
BOOL TerminateLocalComponent (dComp& component_descriptor);
```

component_descriptor	The entry in the component data base describing the component to be unloaded.

Remarks **TerminateLocalComponent**. removes a local component from its parent addressing space. If the local component has other connections, besides its parent connection, it notifies that is being terminated.

Return Value This function returns TRUE if the component is successfully terminated, otherwise it returns FALSE.

TerminateExtComponent

Function Terminates an external component.

Syntax

```
BOOL TerminateExtComponent (dComp& component_descriptor);
```

component_descriptor	The entry in the component data base describing the component to be terminated.

Remarks **TerminateExtComponent** requests the operating system to 'kill' an external component process. If the external component has other connections, besides its parent connection, it does not terminate. In this case, it will terminate itself only after all the other components disconnect.

Return Value This function returns TRUE if the component is successfully terminated, otherwise it returns FALSE.

CreatePort

Function Assign an identifier (name) to the server object's communication port.

Syntax

```
BOOL CreatPort (char* port_name);
```

port_name	The alphanumeric string pointer for the string with the name of the component's communication port.

Remarks **CreatePort** is used to inform the low-level communication system of the identifier (or name) that other components will use to request connections to this component.

Return Value This function returns TRUE if the port is successfully created, otherwise it returns FALSE.

OpenConnection

Function Create a bi-directional communication channel between two components

Syntax

```
BOOL OpenConnection (dComp& component_descriptor);
```

component_descriptor	The entry in the component data base describing the component to be connected.

Remarks **OpenConnection** takes an entry in the local data base as its argument. It updates the hIpcConn field after opening the communication channel.

Return Value This function returns TRUE on success, otherwise FALSE is returned

CloseConnection

Function Terminates a bi-directional connection between two components.

Syntax

```
void CloseConnection (HANDLE hConn);
```

hConn	The handle that uniquely identifies the connection to be closed.

Remarks **CloseConnection** notifies the other component and disconnects.

Return Value This function does not return any value.

ConnectionStatus

Function Set or return the status of a connection between two components

Syntax

```
void    ConnectionStatus (HANDLE hIpcConn,
                          CONSTAT conn_status);
CONSTAT ConnectionStatus (HANDLE hIpcConn);
```

hIpcConn	The handle that uniquely identifies a connection.
conn_status	The new status for the connection.

Remarks **GetConnectionStatus** - a connection may be in three different states: INACTIVE, ACTIVATING or ACTIVE. The first form is called to set the connection state and the second to get its current state.

Return Value This function returns the current state of the connection associated to the specified component. `INACTIVE` is returned if no connection is found.

SendSync

Function Transmits a message package in synchronous mode.

Syntax **`MsgPackage& SendSync (MsgPackage&`** *`message`*`,` **`HANDLE`** *`hIpcConn`*`);`

message	The message package to be sent.
hIpcConn	The handle that uniquely identifies the connection between sender and receiver.

Remarks **SendSync** uses the low-level communication system to send a message package object to another component, according to the communication channel obtained from the *hIpcConn* argument. This is a blocking function that returns control to the caller only after the command taken by the message package object is executed, and the message package returned.

Return Value The updated message package is returned.

SendAsync

Function Transmits a message package in asynchronous mode.

Syntax **`BOOL SendAsync (MsgPackage&`** *`msg`*`,` **`HANDLE`** *`hIpcConn`*`);`

message msg	The message package to be sent.
hIpcConn	The handle that uniquely identifies the connection between sender and receiver.

Remarks **SendSync** uses the low-level communication system to send a message package object to another component, according to the communication channel obtained from the *hIpcConn* argument. This is a non-blocking function that returns control to the caller as soon as the message package is dispatched.

Return Value This function returns TRUE if a message package could be dispatched to the low-level communication system, otherwise FALSE is returned.

Reply

Function	Return a message package of type REPLY to its sender.
Syntax	`BOOL Reply (MsgPackage& message);`
	message msg — The message package replied.
Remarks	**Reply** uses the low-level communication system to return a message package object back to its original sender. This function performs is by definition an asynchronous communication. The handle identifying the sender connection is kept by the message package object.
Return Value	This function returns TRUE if the message package could be dispatched to the low-level communication system, otherwise FALSE is returned.

Appendix D

Class Hierarchy

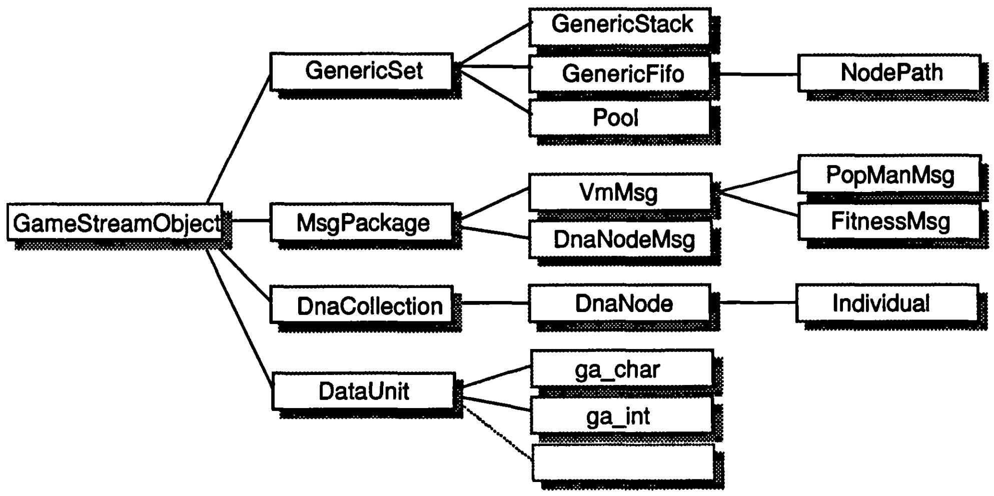
This appendix shows the diagrams of GAME's class hierarchies. The first diagram depicts all the classes rooted in the GameStreamObject class. The second shows the main classes derived from the pemComponent class, which is the basis for the GAME Component concept.

GameStreamObjects

The class hierarchy rooted in the GameStreamObject class comprise four principal branches. The GenericSet branch contains some general-purpose classes like GenericFifo, which are used for the implementation of more specialised classes such as NodePath. The second branch, headed by the MsgPackage class, contains the basic classes used in GAME's messaging system. The user should derive new classes from MsgPackage to support application-dependent commands to be exchanged between specific implementations of GAME Components.

The third and fourth branches contains the classes used to implement GAME's genetic-oriented representation abstractions. The user may derive new data types from the DataUnit class according to application's requirements.

Figure D.1 GAME Class Hierarchy



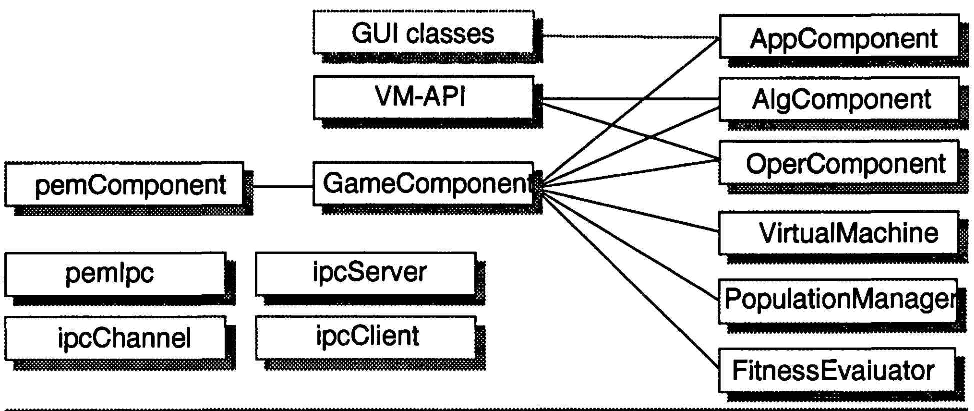
Objects instantiated from any class in this hierarchy are able to be serialised and passed across GAME Component objets.

GameComponents

The picture below shows the main class hierarchy that implements GAME Components and some auxiliary classes. The hierarchy rooted in the pemComponent class contains the GameComponent class, which is the basis for further specialisation leading to components such as the virtual machine and its modules. It is also the root of the framework that supports the implementation of user defined applications (from the AppComponent class), genetic algorithms (from the AlgComponent class) and genetic operators (from the OperComponent class). The last two classes use multiple inheritance to benefit from the VM-API class, which implements the functions of the Virtual Machine Application Program Interface. The AppComponent class may also use multiple inheritance to support platform-dependent graphic user interfaces.

The other four independent classes: pemIpc, ipcServer, ipcClient and ipcChannel are used in the implementation of PEM's lower-layer. They should be adapted by the user to port GAME to a variety of platforms.

Figure D.2 PEM Class Hirarchy



Genetic-Algorithm Programming Environments

José L. Ribeiro Filho and Philip C. Treleaven, University College London
Cesare Alippi, Politecnico di Milano

This review classifies genetic-algorithm environments into application-oriented systems, algorithm-oriented systems, and toolkits. It also presents detailed case studies of leading environments.

Evolution is a remarkable problem-solving machine. First proposed by John Holland in 1975,[1] genetic algorithms are an attractive class of computational models that mimic natural evolution to solve problems in a wide variety of domains. Holland also developed the concept of classifier systems, a machine learning technique using induction systems with a genetic component.[2] Holland's goal was twofold: to explain the adaptive process of natural systems and to design computing systems embodying their important mechanisms. Pioneering work by Holland,[1] Goldberg,[2] DeJong,[3] Grefenstette,[4] Davis,[5] Mühlenbein,[6] and others is fueling the spectacular growth of GAs.

GAs are particularly suitable for solving complex optimization problems and hence for applications that require adaptive problem-solving strategies. In addition, GAs are inherently parallel, since their search for the best solution is performed over genetic structures (building blocks) that can represent a number of possible solutions. Furthermore, GAs' computational models can be easily parallelized[7-9] to exploit the capabilities of massively parallel computers and distributed systems.

Classes of search techniques

Figure 1 groups search techniques into three broad classes.[2] *Calculus-based techniques* use a set of necessary and sufficient conditions to be satisfied by the solutions of an optimization problem. These techniques subdivide into indirect and direct methods. Indirect methods look for local extrema by solving the usually nonlinear set of equations resulting from setting the gradient of the objective function equal to zero The search for possible solutions (function peaks) starts by restricting itself to points with zero slope in all directions. Direct methods, such as those of Newton and Fibonacci, seek extrema by "hopping" around the search space and assessing the gradient of the new point, which guides the search. This is simply the notion of "hill-climbing," which finds the best local point by climbing the steepest permissible gradient These techniques can be used only on a restricted set of "well-behaved" problems.

Enumerative techniques search every point related to an objective function's domain space (finite or discretized), one point at a time. They are very simple to implemen but may require significant computation. The domain space of many applications i too large to search using these techniques. Dynamic programming is a good exampl of an enumerative technique.

Guided random search techniques are based on enumerative techniques but use additional information to guide the search. They are quite general in scope and can solve very complex problems. Two major subclasses are simulated annealing and evolutionary algorithms. Both are evolutionary processes, but simulated annealing uses a thermodynamic evolution process to search minimum energy states. Evolutionary algorithms, on the other hand, are based on natural-selection principles. This form of search evolves throughout generations, improving the features of potential solutions by means of biologically inspired operations. These techniques subdivide, in turn, into evolutionary strategies and genetic algorithms. Evolutionary strategies were proposed by Rechenberg[10] and Schwefel[11] in the early 1970s. They can adapt the process of "artificial evolution" to the requirements of the local response surface.[12] This means that unlike traditional GAs evolutionary strategies can adapt their major strategy parameters according to the local topology of the objective function.[13]

Following Holland's original genetic-algorithm proposal, many variations of the basic algorithm have been introduced. However, an important and distinctive feature of all GAs is the population-handling technique. The original GA adopted a *generational replacement policy*,[5] according to which the whole population is replaced in each generation. Conversely, the *steady-state policy*[5] used by many subsequent GAs selectively replaces the population. It is possible, for example, to keep one or more population members for several generations, while those individuals sustain a better fitness than the rest of the population.

After we introduce GA models and their programming, we present a survey of GA programming environments. We have grouped them into three major classes according to their objectives: Application-oriented systems hide the details of GAs and help users develop applications for specific domains, algorithm-oriented systems are based on specific GA models, and toolkits are flexible environments for programming a range of GAs and applications. We review the available environments and describe their common features and requirements. As case studies, we select some specific systems for more detailed examination. To conclude, we discuss likely future developments in GA programming environments.

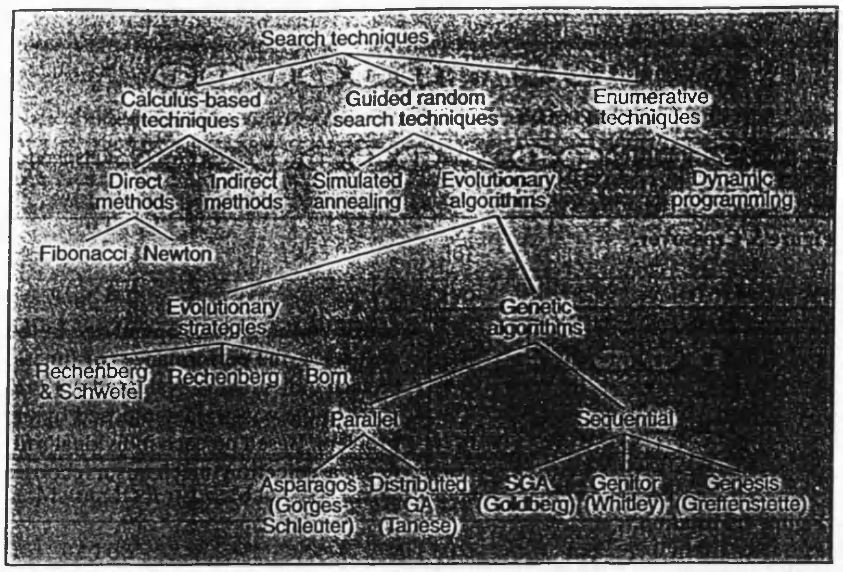


Figure 1. Classes of search techniques.

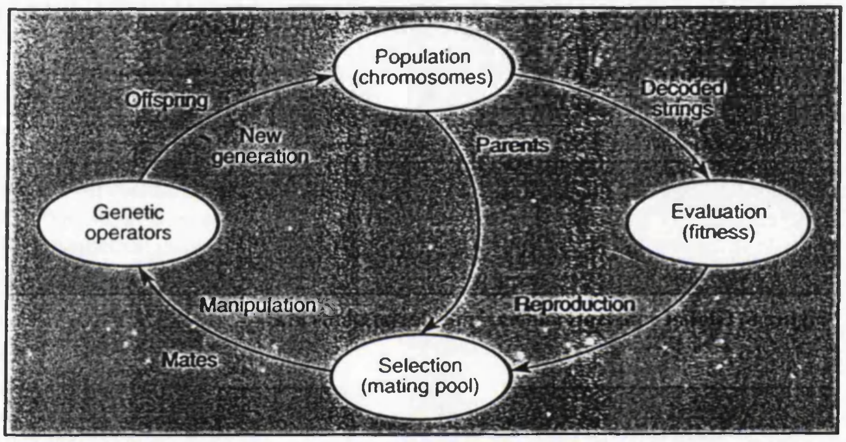


Figure 2. The GA cycle.

Genetic algorithms

A genetic algorithm emulates biological evolutionary theories to solve optimization problems. A GA comprises a set of individual elements (the population) and a set of biologically inspired operators defined over the population itself. According to evolutionary theories, only the most suited elements in a population are likely to survive and generate offspring, thus transmitting their biological heredity to new generations. In computing terms, a genetic algorithm maps a problem onto a set of (typically binary) strings, each string representing a potential solution. The GA then manipulates the most promising strings in its search for improved solutions. A GA operates through a simple cycle of stages:

(1) creation of a "population" of strings,
(2) evaluation of each string,
(3) selection of "best" strings, and
(4) genetic manipulation to create the new population of strings.

Figure 2 shows these four stages using the biologically inspired GA terminology. Each cycle produces a new generation of possible solutions for a given problem. At the first stage, an initial population of potential solutions is created as a starting point for the search. Each element of the population is encoded into a string (the chromosome) to be manip-

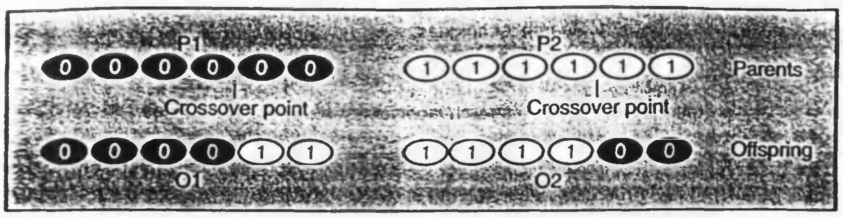


Figure 3. Crossover.

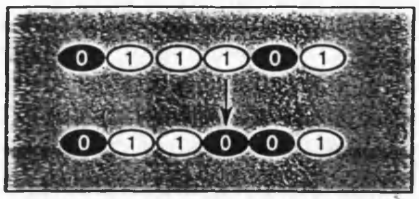


Figure 4. Mutation.

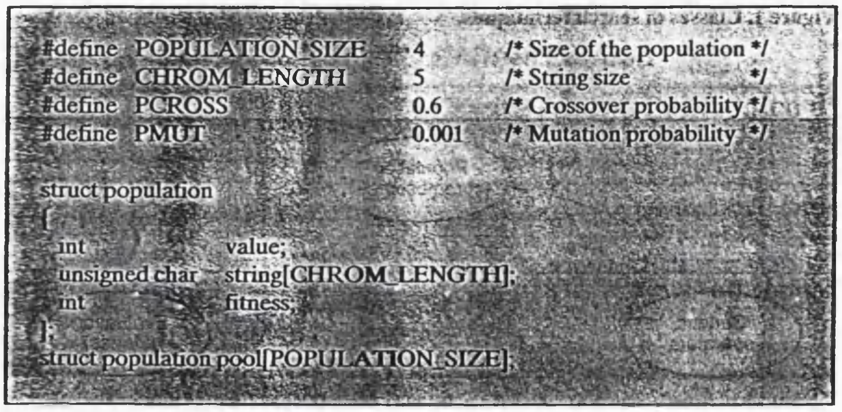
ulated by the genetic operators. In the next stage, the performance (or fitness) of each individual is evaluated with respect to the constraints imposed by the problem. Based on each individual's fitness, a selection mechanism chooses "mates" for the genetic manipulation process. The selection policy is ultimately responsible for assuring survival of the best fitted individuals. The combined evaluation and selection process is called reproduction.

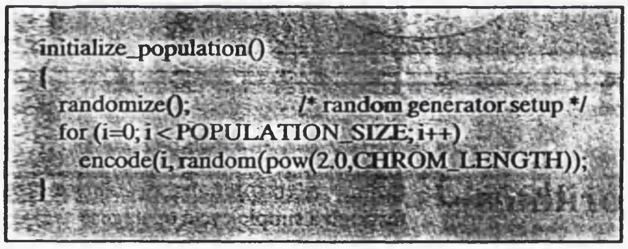
The manipulation process uses genetic operators to produce a new population of individuals (offspring) by manipulating the "genetic information," referred to as genes, possessed by members (parents) of the current population. It comprises two operations: crossover and mutation. *Crossover* recombines a population's genetic material. The selection process associated with recombination assures that special genetic structures, called building blocks, are retained for future generations. The building blocks then represent the most fitted genetic structures in a population.

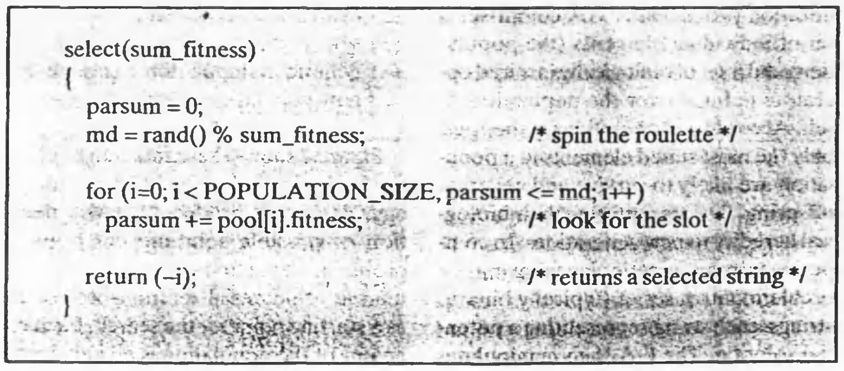
The recombination process alone cannot avoid the loss of promising building blocks in the presence of other genetic structures, which could lead to local minima. Also, it cannot explore search space sections not represented in the population's genetic structures. Here *mutation* comes into action. The mutation operator introduces new genetic structures in the population by randomly modifying some of its building blocks, helping the search algorithm escape from local minima's traps. Since the modification is not related to any previous genetic structure of the population, it creates different structures representing other sections of the search space.

The crossover operator takes two chromosomes and swaps part of their genetic information to produce new chromosomes. This operation is analogous to sexual reproduction in nature. As Figure 3 shows, after the crossover point has been randomly chosen, portions of the parent strings P1 and P2 are swapped to produce the new offspring strings O1 and O2. In Figure 3 the crossover operator is applied to the fifth and sixth elements of the string. Mutation is implemented by occasionally altering a random bit in a string. Figure 4 shows the mutation operator applied to the fourth element of the string.

A number of different genetic operators have been introduced since Holland proposed this basic model. They are, in general, versions of the recombination and genetic alteration processes adapted to the requirements of particular problems. Examples of other genetic operators are inversion, dominance, and genetic edge recombination.


```
#define  POPULATION_SIZE    4       /* Size of the population */
#define  CHROM_LENGTH       5       /* String size            */
#define  PCROSS             0.6     /* Crossover probability  */
#define  PMUT               0.001   /* Mutation probability   */

struct population
{
  int              value;
  unsigned char    string[CHROM_LENGTH];
  int              fitness;
};
struct population pool[POPULATION_SIZE];
```
Figure 5. Global constants and variable declarations in C.


```
initialize_population()
{
  randomize();              /* random generator setup */
  for (i=0; i<POPULATION_SIZE; i++)
    encode(i, random(pow(2.0,CHROM_LENGTH));
}
```
Figure 6. Initialization routine.


```
select(sum_fitness)
{
   parsum = 0;
   md = rand() % sum_fitness;                 /* spin the roulette */

   for (i=0; i< POPULATION_SIZE, parsum <= md; i++)
     parsum += pool[i].fitness;               /* look for the slot */

   return (–i);                               /* returns a selected string */
}
```
Figure 7. Selection function.

The offspring produced by the genetic manipulation process are the next population to be evaluated. Genetic algorithms can replace either a whole population (generational approach) or its less fitted members only (steady-state approach). The creation-evaluation-selection-manipulation cycle repeats until a satisfactory solution to the problem is found or some other termination criterion is met.

This description of the computational model reviews the steps needed to design a genetic algorithm. However, real implementations take into account a number of problem-dependent parameters such as the population size, crossover and mutation rates, and convergence criteria. GAs are very sensitive to these parameters (a discussion of the methods for setting them up is beyond the scope of this article).

Sequential GAs. To illustrate the implementation of a sequential genetic algorithm we use Goldberg's simple function optimization example[2] and examine its programming in C. The first step in optimizing the function $f(x) = x^2$ over the interval (parameter set) [0-31] is to encode the parameter set x, for example, as a five-digit binary string {00000-11111}. Next we generate the initial population of four potential solutions, shown in Table 1, using a random number generator.

To program this GA function optimization, we declare the population pool as an array with four elements, as shown in Figure 5, and then initialize the structure using a random generator, as shown in Figure 6. Our next step is reproduction. Reproduction evaluates and selects pairs of strings for mating according to their relative strengths (see Table 1 and the associated C code in Figure 7). One copy of string 01101, two copies of 11000, and one copy of 10011 are selected by using a roulette wheel method.[2]

Next we apply the crossover operator, as illustrated in Table 2. Crossover operates in two steps (see Figure 8). First it determines whether crossover is to occur on a pair of strings by using a flip function: tossing a biased coin (with probability *pcross*). If the result is heads (true), the strings are swapped; the *crossover_point* is determined by a random number generator. If tails (false), the strings are simply copied. In the example, crossover occurs at the fifth position for the first pair and the third position for the other.

After crossover, the mutation operator is applied to the new population, which may have a random bit in a given string modified. The mutation function in Figure 9 on the next page uses the biased coin toss (flip) with probability *pmut* to determine whether to change a bit.

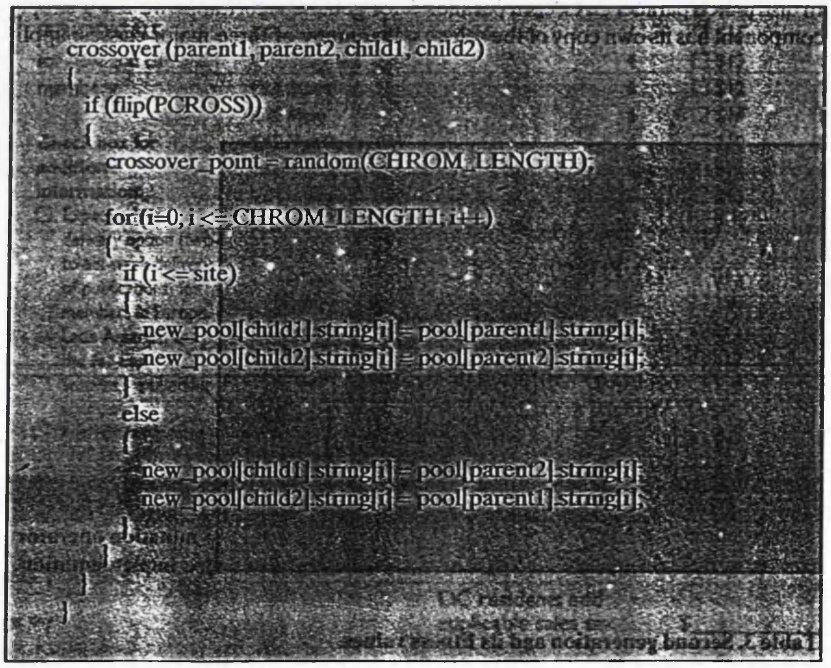
Table 3 shows the new population, to which the algorithm now applies a termination test. Termination criteria may include the simulation time being up, a specified number of generations exceeded, or a convergence criterion satisfied. In the example, we might set the number of generations to 50 and the con-

Table 1. Initial strings and fitness values.

Initial Population	x	$f(x)$ (fitness)	Strength (percent of total)
01101	13	169	14.4
11000	24	576	49.2
01000	8	64	5.5
10011	19	361	30.9
Sum_Fitness =		1,170	(100.0)

Table 2. Mating pool strings and crossover.

Mating Pool	Mates	Swapping	New Population
01101	1	0110[1]	01100
11000	2	1100[0]	11001
11000	2	11[000]	11011
10011	4	10[011]	10000



```
crossover (parent1, parent2, child1, child2)
 {
   if (flip(PCROSS))
    {
     crossover_point = random(CHROM_LENGTH);

     for (i=0; i <= CHROM_LENGTH; i++)
      {
       if (i <= site)
        {
         new_pool[child1].string[i] = pool[parent1].string[i];
         new_pool[child2].string[i] = pool[parent2].string[i];
        }
       else
        {
         new_pool[child1].string[i] = pool[parent2].string[i];
         new_pool[child2].string[i] = pool[parent1].string[i];
        }
      }
    }
 }
```

Figure 8. The crossover routine.

vergence as an average fitness improvement of less than 5 percent between generations. For the initial population, the average is 293, that is, (169 + 576 + 64 + 361) ÷ 4, while for the new population it has improved to 439, that is, 66 percent, (see the sidebar on Sequential GA C listing on page 34).

Parallel GAs. The GA paradigm offers intrinsic parallelism in searches for the best solution in a large search space, as demonstrated by Holland's schema theorem.[1] Besides the intrinsic parallelism, GA computational models can also exploit other levels of parallelism because of the natural independence of the genetic manipulation operations.

A parallel GA is generally formed by parallel components, each responsible for manipulating subpopulations. As was shown in Figure 1, there are two classes of parallel GAs: centralized and distributed. The first has a centralized selection mechanism: A single selection operator works synchronously on the global population (of subpopulations) at the selection stage. In distributed parallel GAs, each parallel component has its own copy of the selection operator, which works asynchronously. In addition, each component communicates its best strings to a subset of the other components. This process requires a migration operator and a migration frequency defining the communication interval.

The Asparagos algorithm[7] has a distributed mechanism. Figure 10 shows a skeleton C-like program, based on this algorithm, for the simple function optimization discussed for sequential algorithms. In this parallel program the statements for initialization, selection, crossover, and mutation remain almost the same as in the sequential program. For the main loop, parallel (PAR) subpopulations are set up for each component, as well as values for the new parameters. Each component then executes sequentially, apart from the parallel migration operator.

Taxonomy

To review programming environments for genetic algorithms, we use a simple taxonomy of three major classes: application-oriented systems, algorithm-oriented systems, and toolkits.

Application-oriented systems are essentially "black boxes" that hide the GA implementation details. Targeted at business professionals, some of these systems support a range of applications; others focus on a specific domain, such as finance.

Algorithm-oriented systems support specific genetic algorithms. They subdivide into

- algorithm-specific systems, which contain a single genetic algorithm, and
- algorithm libraries, which group together a variety of genetic algorithms and operators.

These systems are often supplied in source code and can be easily incorporated into user applications.

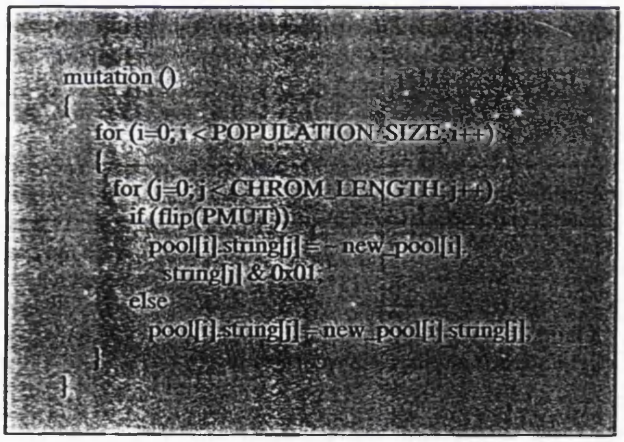
Toolkits provide many programming utilities, algorithms, and genetic operators for a wide range of application domains. These programming systems subdivide into

- educational systems that help novice users obtain a hands-on introduction to GA concepts, and
- general-purpose systems that provide a comprehensive set of tools for programming any GA and application.

Table 4 lists the GA programming environments examined in the next sections, according to their categories. For each category we present a generic system overview, then briefly review example systems, and finally examine one system in more detail, as a case study. The parallel environments GAUCSD, Pegasus, and GAME are also covered, but no commercial parallel environments are currently available. See the sidebar "Developers address list" on page 37 for a comprehensive list of programming environments and their developers.

Application-oriented systems

Many potential users of a novel computing technique are interested in applications rather than the details of the technique. Application-oriented systems are designed for business professionals who want to use genetic algorithms for spe-



```
mutation ()
{
  for (i=0; i < POPULATION_SIZE; i++)
  {
   for (j=0; j < CHROM_LENGTH; j++)
     if (flip(PMUT))
       pool[i].string[j] = ~ new_pool[i].
         string[j] & 0x01;
     else
       pool[i].string[j] = new_pool[i].string[j];
  }
}
```

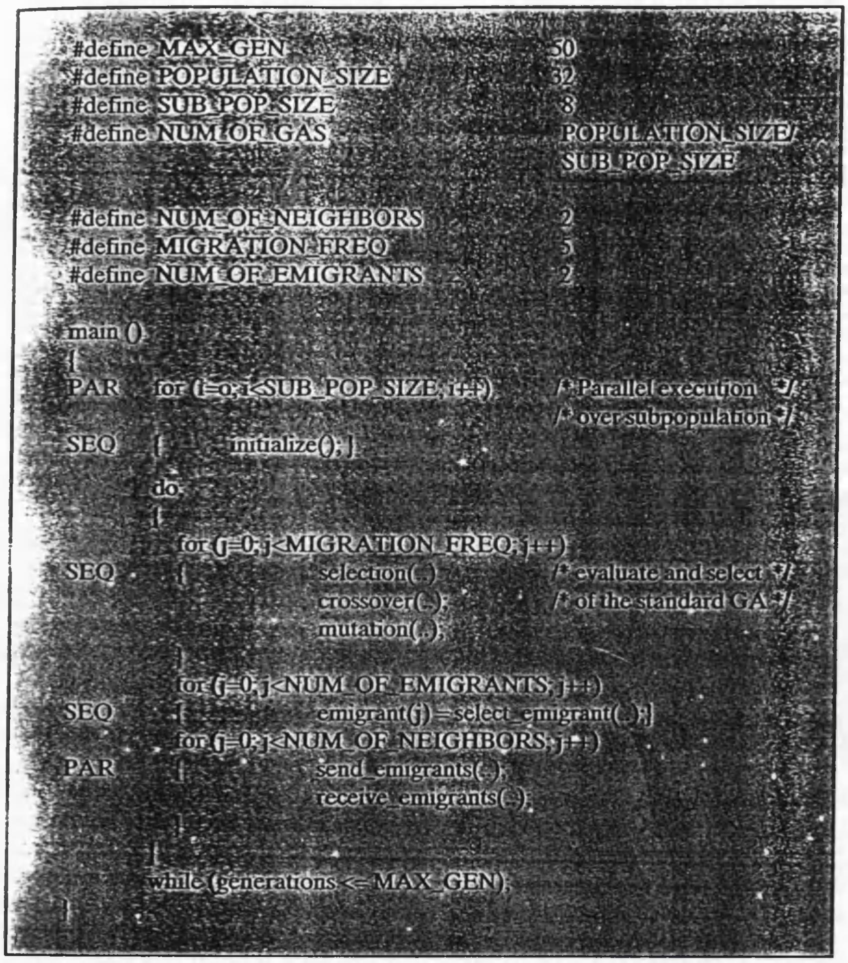
Figure 9. The mutation operator C implementation.

Table 3. Second generation and its fitness values.

Initial Population	x	$f(x)$ (fitness)	Strength (percent of total)
01100	12	144	8.2
11001	25	625	35.6
11011	27	729	41.5
10000	16	256	14.7
Sum_Fitness =		1,754	(100.0)

cific purposes without having to acquire detailed knowledge about them. For example, a manager in a trading company may need to optimize its delivery scheduling. By using an application-oriented programming environment, the manager can configure an application for schedule optimization based on the traveling-salesman problem without having to know the encoding technique or the genetic operators.

Overview. A typical application-oriented environment is analogous to a spreadsheet or word-processing utility. Its menu-driven interface (tailored to business users) gives access to parameterized modules (targeted at specific domains). The user interface provides menus to configure an application, monitor its execution, and, in certain cases, program an application. Help facilities are also provided.



```
#define MAX_GEN                          50
#define POPULATION_SIZE                  32
#define SUB_POP_SIZE                     8
#define NUM_OF_GAS                       POPULATION_SIZE/
                                         SUB_POP_SIZE

#define NUM_OF_NEIGHBORS                 2
#define MIGRATION_FREQ                   5
#define NUM_OF_EMIGRANTS                 2

main ()
{
PAR    for (i=o; i<SUB_POP_SIZE; i++)         /* Parallel execution  */
                                              /* over subpopulation */
SEQ    {       initialize(); }
       do
       {
         for (j=0; j<MIGRATION_FREQ; j++)
SEQ      {             selection(..)          /* evaluate and select */
                       crossover(..);         /* of the standard GA */
                       mutation(..);
         }
         for (j=0; j<NUM_OF_EMIGRANTS; j++)
SEQ      {             emigrant(j) = select_emigrant(..);}
         for (j=0; j<NUM_OF_NEIGHBORS; j++)
PAR      {             send_emigrants(..);
                       receive_emigrants(..);
         }
       }
       while (generations <= MAX_GEN);
}
```

Figure 10. Parallel GA with migration.

Survey. Application-oriented systems have many innovative strategies. Systems such as PC/Beagle and XpertRule GenAsys are expert systems that use GAs to generate new rules to expand their knowledge base of the application domain. Evolver is a companion utility for spreadsheets. Omega is targeted at financial applications.

Evolver. This add-on utility works within the Excel, Wingz, and Resolve spreadsheets on Macintosh and PC computers. Axcelis, its marketer, describes it as "an optimization program that extends mechanisms of natural evolution to the world of business and science applications." A user starts with a model of a system in the spreadsheet and calls the Evolver program from a menu. After the user fills a dialog box with the information required (the cell to minimize or maximize), the program starts working, evaluating thousands of scenarios automatically until it has found an optimal answer. The program runs in the background, freeing the user to work in the foreground.

When Evolver finds the best result, it notifies the user and places the values into the spreadsheet for analysis. This is an excellent design strategy, given the importance of spreadsheets in business. In an attempt to improve the system and ex-

Table 4. Programming environments and their categories.

	Algorithm-Oriented Systems		Toolkits	
Application-Oriented Systems	Algorithm-specific systems	Algorithm libraries	Educational systems	General-purpose systems
Evolver Omega PC/Beagle	Escapade GAGA GAUCSD	EM	GA Workbench	Engeneer GAME MicroGA Pegasus Splicer
XpertRule GenAsys	Genesis Genitor	OOGA		

pand its market, Axcelis introduced Evolver 2.0, which has many toolkit-like features. The new version can integrate with other applications in addition to spreadsheets. It also offers more flexibility in accessing the Evolver engine: This can be done from any Microsoft Windows application that can call a Dynamic Link Library.

Omega. The Omega Predictive Modelling System, marketed by KiQ, is a powerful approach to developing predictive models. It exploits advanced GA techniques to create a tool that is "flexible, powerful, informative and straightforward to use," according to its developers. Geared to the financial domain, Omega can be applied to direct marketing, insurance, investigations (case scoring), and credit management. The envi-

Sequential GA C Listing

```
    ****************************************************
    *        Simple Genetic Algorithm
    ****************************************************
    */
    #include <stdlib.h>
    #include <stdio.h>
    #include <sys/time.h>
    #include <math.h>

    #define RAND_MAX          0x7FFFFFFF
    #define random(num)     (rand()%(num))
    #define randomize()     srand((unsigned)time(NULL))

    #define POPULATION_SIZE   10
    #define CHROM_LENGTH      4
    #define PCROSS            0.6
    #define PMUT              0.050
    #define MAX_GEN           50

    struct population
    {
       int            value;
       unsigned char  string[CHROM_LENGTH];
       unsigned int   fitness;
    };

    struct population pool[POPULATION_SIZE];
    struct population new_pool[POPULATION_SIZE];

    int selected[POPULATION_SIZE];
    int generations;

    main()
    {
       int i;
       double sum_fitness, avg_fitness, old_avg_fitness;
       generations = 1;
       avg_fitness = 1;

       initialize_population();

       do
       {
          old_avg_fitness = avg_fitness;
          sum_fitness = 0;

          /* fitness evaluation */
          for (i=0; i<POPULATION_SIZE; i++)
          {
             pool[i].value = decode(i);
             pool[i].fitness = evaluate(pool[i].value);
             sum_fitness += pool[i].fitness;
          }

          avg_fitness = sum_fitness / POPULATION_SIZE;

          for (i=0; i<POPULATION_SIZE; i++)
             selected[i] = select(sum_fitness);

          for (i=0; i<POPULATION_SIZE; i=i+2)
             crossover(selected[i],selected[i+1],i,i+1);

          mutation();

          statistics();
        printf("\nImprovment: %f\n", avg_fitness/old_avg_fitness);
       }
      while ((++generations < MAX_GEN) &&
             ((avg_fitness/old_avg_fitness) > 1.005) ||
           ((avg_fitness/old_avg_fitness) < 1.0));
    }

    /*
    ****************************************************
    *       initialize_population
    *    Creates and initializes a population
    ****************************************************
    */
    initialize_population()
    {
      int i;

      randomize();
      for (i=0; i< POPULATION_SIZE; i++)
         encode(i, random(2^CHROM_LENGTH));
    }
    /*
    ****************************************************
    *       select
    *    Selects strings for reproduction
    ****************************************************
    */
    select(sum_fitness)
    double sum_fitness;
    {
      int i;
      double r, parsum;

      parsum = 0;

      r = (double)(rand() % (int)sum_fitness); /* spin the roulette */

      for (i=0; i< POPULATION_SIZE, parsum <= r; i++)
         parsum += pool[i].fitness;

      return (--i);    /* returns a selected string */
    }
    /*
    ****************************************************
    *       crossover
    *    Swaps 2 sub-strings
    ****************************************************
    */
    crossover (parent1, parent2, child1, child2)
    int parent1;
    int parent2;
    int child1;
    int child2;
    {
      int i, site;

      if (flip(PCROSS))
         site = random(CHROM_LENGTH);
```

ronment offers facilities for automatic handling of data; business, statistical, or custom measures of performance; simple and complex profit modeling; validation sample tests; advanced confidence tests; real-time graphics; and optional control over the internal genetic algorithm.

PC/Beagle. Produced by Pathway Research, this rule-finder program applies machine learning techniques to create a set of decision rules for classifying examples previously extracted from a database. It has a module that generates rules by natural selection. Further details are given in the case study section.

XpertRule GenAsys. XpertRule GenAsys is an expert system shell with embedded genetic algorithms. Marketed by Attar Software, this GA expert system solves scheduling and design problems.

```
    else
        site = CHROM_LENGTH-1;

    for (i=0; i< CHROM_LENGTH; i++)
    {
        if ((i<=site) || (site==0))
        {
            new_pool[child1].string[i] = pool[parent1].string[i];
            new_pool[child2].string[i] = pool[parent2].string[i];
        }else
        {
            new_pool[child1].string[i] = pool[parent2].string[i];
            new_pool[child2].string[i] = pool[parent1].string[i];
        }
    }
}
/*
*********************************************
*       mutation
*    Changes the values of string position
*********************************************
*/
mutation ()
{
    int i, j;

    for (i=0; i< POPULATION_SIZE; i++)
    {
        for (j=0; j < CHROM_LENGTH; j++)
            if (flip(PMUT))
                pool[i].string[j] = ~new_pool[i].string[j] & 0x01;
            else
                pool[i].string[j] = new_pool[i].string[j] & 0x01;
    }
}
/*
*********************************************
*       encode
*    Code a integer into binary string
*********************************************
*/
encode(index, value)
int index;
int value;
{
    int i;

    for (i=0; i< CHROM_LENGTH; i++)
        pool[index].string[CHROM_LENGTH-1-i] = (value >> i) &
0x01;
}
/*
*********************************************
*       decode
*    Decode a binary string into an integer
*********************************************
*/
decode(index)
int index;
{
    int i, value;

    value = 0;

    for (i=0; i < CHROM_LENGTH; i++)
        value += (int)pow(2.0,(double)i) *
pool[index].string[CHROM_LENGTH-1-i];

    return(value);
}
/*
*********************************************
*       evaluate
*    Objective function f(x)=x^2
*********************************************
*/
evaluate(value)
int value;
{
    return(pow((double)value,2.0));
}
/*
*********************************************
*       flip
*    Toss a biased coin
*********************************************
*/
flip(prob)
double prob;
{
    double i;

    i = ((double)rand())/RAND_MAX;

    if ((prob == 1.0) || (i < prob))
        return (1);
    else
        return (0);
}
/*
*********************************************
*       statistics
*    Print intermediary results
*********************************************
*/
statistics()
{
    int i, j;

    printf("\n;Generation: %d\n;Selected Strings\n;", generations);
    for (i=0; i< POPULATION_SIZE; i++)
        printf(" %d", selected[i]);

    printf("\n");

    printf("\n;X\tf(x)\t New_String\tX");

    for (i=0; i< POPULATION_SIZE; i++)
    {
        printf("\n %d\t%u\t", pool[i].value, pool[i].fitness);

        for (j=0; j<CHROM_LENGTH; j++)
            printf(" %d", pool[i].string[j]);

        printf("\t%d", decode(i));
    }
}
```

The system combines the power of genetic algorithms in evolving solutions with the power of rule-base programming in analyzing the effectiveness of solutions. Rule-base programming can also be used to generate the initial solutions for the genetic algorithm and for postoptimization planning. Problems this system can solve include optimization of design parameters in the electronics and avionics industries, route optimization in the distribution sector, and production scheduling in manufacturing.

Case study: PC/Beagle. PC/Beagle is a rule-finder program that examines a database of examples and uses machine learning techniques to create decision rules for classifying those examples, turning data into knowledge. The software analyzes an expression via a historical database and develops a series of rules to explain when the target expression is false or true. The system contains six main components generally run in sequence:

- SEED (selectively extracts example data) puts external data into a suitable format and may append leading or lagging data fields as well.
- ROOT (rule-oriented optimization tester) tests an initial batch of user-suggested rules.
- HERB (heuristic evolutionary rule breeder) generates decision rules by natural selection, using GA philosophy and ranking mechanisms.
- STEM (signature table evaluation module) makes a signature table from the rules produced by HERB.
- LEAF (logical evaluator and forecaster) uses STEM output to do forecasting or classification.
- PLUM (procedural language utility maker) can convert a Beagle rule file into a language such as Pascal or Fortran so other software can use the knowledge gained.

PC/Beagle accepts data in ASCII format, with items delimited by commas, spaces, or tabs. Rules are produced as logical expressions. The system is highly versatile, covering a wide range of applications. Insurance, weather forecasting, finance, and forensic science are some examples. PC/Beagle requires an IBM PC-compatible computer with at least 256 Kbytes of RAM and an MS-DOS or PC-DOS operating system, version 2.1 or later.

Algorithm-oriented systems

Our taxonomy divides algorithm-oriented systems into algorithm-specific systems that contain a single algorithm and algorithm libraries, which group together a variety of genetic algorithms and operators.

Algorithm-specific environments embody a single powerful genetic algorithm. These systems have typically two groups of users: system developers requiring a general-purpose GA for their applications and researchers interested in the development and testing of a specific algorithm and genetic operators.

Algorithm-specific environments embody a single powerful genetic algorithm.

Overview of algorithm-oriented systems. In general, these systems come in source code so expert users can make alterations for specific requirements. They have a modular structure for a high degree of modifiability. In addition, user interfaces are frequently rudimentary, often command-line driven. Typically the codes have been developed at universities and research centers, and are available free over worldwide computer research networks.

System survey. The most well known programming system in this category is the pioneering Genesis,[4] which has been used to implement and test a variety of new genetic operators. In Europe probably the earliest algorithm-specific system was GAGA. For scheduling problems, Genitor[14] is another influential and successful system. GAUCSD permits parallel execution: It distributes several copies of a Genesis-based algorithm to Unix machines in a network. Escapade[13] uses a somewhat different approach — an evolutionary strategy.

Escapade. Escapade (Evolutionary Strategies Capable of Adaptive Evolution) provides a sophisticated environment for a particular class of evolutionary algorithms, called evolutionary strategies. Escapade is based on Korr, Schwefel's implementation of a $(\mu, +\lambda)$-evolutionary strategy, where the μ best individuals of the λ offspring, added to their parents, survive and become the parents of the new generation. The system provides an elaborate set of monitoring tools to gather data from an optimization run of Korr. According to Escapade's author, it should be possible to incorporate a different implementation of an evolutionary strategy or even a GA into the system using its runtime support. The program is separated into several independent components that support the various tasks during a simulation run. The major modules are parameter setup, runtime control, Korr, generic data monitors, customized data monitors, and monitoring support.

During an optimization run, the monitoring modules are invoked by the main algorithm (Korr or some other evolutionary strategy or GA implementation) to log internal quantities. The system is not equipped with any kind of graphical interface. Users must pass all parameters for a simulation as command-line options. For output, each data monitor writes its data into separate log files.

GAGA. The Genetic Algorithms for General Application were originally programmed in Pascal by Hillary Adams at the University of York. The program was later modified by Ian Poole and translated into C by Jon Crowcroft at University College London. GAGA is a task-independent genetic algorithm. The user must supply the target function to be optimized (minimized or maximized) and some technical GA parameters, and wait for the output. The program is suitable for the minimization of many difficult cost functions.

GAUCSD. This software was developed by Nicol Schraudolph at the University of California, San Diego (hence UCSD).[15] The system is based on Genesis 4.5 and runs on Unix, MS-DOS, Cray operating system, and VMS platforms, but it presumes a Unix environment. GAUCSD comes with an *awk* script called "wrapper," which provides a higher level of abstraction for defining the evaluation function. By supplying the code for decoding and printing this function's parameters automatically, it allows the direct use of most C functions as evaluation functions, with few restrictions. The software also includes a dynamic parameter encoding technique developed by Schraudolph, which radically reduces the gene length while keeping the desired level of precision for the results. Users can run the system in the background at low priority using the *go* command.

The *go* command can also be used to execute GAUCSD on remote hosts. The results are then copied back to the user's local directory, and a report is produced if appropriate. If the host is not binary compatible, GAUCSD compiles the whole system on the remote host. Experiments can be queued in files, distributed to several hosts, and executed in parallel. The experiments are distributed according to a specified loading factor (how many programs will be sent to each host), along with the remote execution arguments to the *go* command. The *ex* command notifies the user via write or mail when all experiments are completed. GAUCSD is clearly a very powerful system.

Genesis. The Genetic Search Implementation System, or Genesis, was written by John Grefenstette[4] to promote the study of genetic algorithms for function optimization. It has been under development since 1981 and widely distributed to the research community since 1985. The package is a set of routines written in C. To build their own genetic algorithms, users provide only a routine with the fitness function and link it with the other routines. Users can also modify modules or add new ones (for example, genetic operators and data monitors) and create a different version of Genesis. In fact, Genesis has been used as a base for test and evaluation of a variety of genetic al-

Developers' address list

C Darwin II
Attar Software
Newlands Road
Leigh, Lancashire, UK
Telephone: +44 94 2608844
Fax: +44 94 2601991
E-mail: 100166.1547
@CompuServe.com

EM — Evolution Machine
H.M. Voigt and J. Born
Technical University of Berlin
Bionics and Evolution
Techniques Laboratory
Bio and Neuroinformatics
Research Group
Ackerstasse 71-76 (ACK1)
D-13355 Berlin, Germany
Telephone: +49 303 147 2677
E-mail: voigt@fb10.tu-berlin.de
born@fb10.tu-berlin.de

Escapade
Frank Hoffmeister
University of Dortmund
System Analysis Research Group, LSXI
D-44221 Dortmund, Germany
Telephone: +49 231 755 4678
Fax: +49 231 755 2450
E-mail: hoffmeister@ls11.
informatik.uni-dortmund.de

Engeneer
George Robbins
Systems Intelligence Division
Logica Cambridge Ltd.
Betjeman House
104 Hills Road
Cambridge CB2 1LQ, UK
Telephone: +44 71 6379111
Fax: +44 223 322315

Evolver
Axcelis Inc.
4668 Eastern Avenue North
Seattle, WA 98103
Telephone: (206) 632-0885
Fax: (206) 632-3681

GA Workbench
Mark Hughes
Cambridge Consultants Ltd.
Science Park, Milton Rd.
Cambridge CB4 4DW, UK
Telephone: +44 223 420024
Fax: +44 223 423373
E-mail: mrh@camcon.co.uk

GAGA
Jon Crowcroft
University College London
Gower St.
London WC1E 6BT, UK
Telephone: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: jon@cs.ucl.ac.uk

GAME
José L. Ribeiro Filho
Computer Science Department
University College London
Gower St.
London WC1E 6BT, UK
Telephone: +44 71 387 7050
Fax: +44 71 387 1398
E-mail: j.ribeirofilho@cs.ucl.ac.uk

GAUCSD
N.N. Schraudolph
Computer Science and Engineering
Department
University of California, San Diego
La Jolla, CA 92093-0114
Fax: (619) 534-7029
E-mail: nici@cs.ucsd.edu

Genesis
J.J. Grefenstette
The Software Partnership
PO Box 991
Melrose, MA 02176
Telephone: (617) 662-8991
E-mail: gref@aic.nrl.navy.mil

Genitor
Darrel Whitley
Computer Science Department
Colorado State University
Fort Collins, CO 80523
E-mail: whitley@cs.colostate.edu

MicroGA
Steve Wilson
Emergent Behavior
635 Wellsbury Way, Palo Alto, CA 94306
Telephone: (415) 494-6763
E-mail: emergent@aol.com

Omega
David Barrow
KiQ Ltd.
Easton Hall, Great Easton
Essex CM6 2HD, UK
Telephone: +44 371 870254

OOGA
Lawrence Davis
The Software Partnership
PO Box 991
Melrose, MA 02176

PC/Beagle
Richard Forsyth
Pathway Research Ltd.
59 Cronbrook Rd.
Bristol BS6 7BS, UK
Telephone: +44 272 428692

Pegasus
Dirk Schlierkamp-Voosen
German National Research Center
for Computer Science — GMD
Research Group for Adaptive Systems
PO Box 1316
D-53731 Sankt Augustin, Germany
Telephone: +49 224 114 2466
E-mail: dirk.schlierkamp-voosen@
gmd.de

Splicer
Cosmic
382 E. Broad St.
Athens, GA 30602
Telephone: (404) 542-3265
Fax: (706) 542-4807
E-mail: bayer@galileo.jsc.nasa.gov

XpertRule GenAsys
Attar Software
Newlands Road
Leigh, Lancashire, UK
Telephone: +44 94 2608844
Fax: +44 94 2601991
E-mail: 100166.1547@CompuServe.com

Xype
Ed Swartz
Virtual Image Inc.
75 Sandy Pond Road 11
Ayer, MA 01432
Telephone: (508) 772-0800

gorithms and operators. It was primarily developed to work in a scientific environment and is a suitable tool for research. Genesis is highly modifiable and provides a variety of statistical information on output.

Genitor. The modular GA package Genitor (Genetic Implementor) has examples for floating-point, integer, and binary representations. Its features include many sequencing operators, as well as subpopulation modeling. The software package is an implementation of the Genitor algorithm developed by Darrel Whitley.[14]

Genitor has two major differences from standard genetic algorithms. The first is its explicit use of ranking. Instead of using fitness-proportionate reproduction, Genitor allocates reproductive trials according to the rank of the individual in the population. The second difference is that Genitor abandons the generational approach (in which the whole population is replaced with each generation) and reproduces new genotypes on an individual basis. Using the steady-state approach, Genitor lets some parents and offspring coexist. A newly created offspring replaces the lowest ranking individual in the population rather than a parent. Because Genitor produces only one new genotype at a time, inserting a single new individual is relatively simple. Furthermore, the insertion automatically ranks the individual in relation to the existing pool — no further measure of the relative fitness is needed.

Case study: Genesis. Genesis[4] is the most well known software package for GA development and simulation. It runs on most machines with a C compiler. Version 5.0, now available from the Software Partnership, runs successfully on both Sun workstations and IBM PC-compatible computers, according to its author. The code is designed to be portable, but minor changes may be necessary for other systems.

Genesis provides the fundamental procedures for genetic selection, crossover, and mutation. The user is only required to provide the problem-dependent evaluation function.

Genesis has three levels of representation for the structures it evolves. The lowest level, packed representation, maximizes both space and time efficiency in manipulating structures. In general, this level of representation is transparent to the user. The next level, the string representation, represents structures as null-terminated arrays of characters, or "chars." This structure is for users who wish to provide an arbitrary interpretation of the genetic structures, for example, nonnumeric concepts. The third level, the floating-point representation, is appropriate for many numeric optimization problems. At this level the user views genetic structures as vectors or real numbers. For each parameter, or gene, the user specifies its range, number of values, and output format. The system then automatically lays out the string representation and translates between the user-level genes and lower representation levels.

Algorithm libraries provide a powerful collection of parameterized genetic algorithms and operators.

Genesis has five major modules:

- *Initialization.* The initialization procedure sets up the initial population. Users can "seed" the initial population with heuristically chosen structures, and the rest of the population is filled with random structures. Users can also initialize the population with real numbers.
- *Generation.* This module executes the selection, crossover, mutation, and evaluation procedures, and collects some data.
- *Selection.* The selection module chooses structures for the next generation from the structures in the current generation. The default selection procedure is stochastic, based on the roulette wheel algorithm, to guarantee that the number of offspring of any structure is bounded by the floor and ceiling of the (real-valued) expected number of offspring. Genesis can also perform selection using a ranking algorithm. Ranking helps forestall premature convergence by preventing "super" individuals from taking over the population within a few generations.
- *Mutation.* After Genesis selects the new population, it applies mutation to each structure. Each position is given a chance (according to the mutation rate) of undergoing mutation. If mutation is to occur, Genesis randomly chooses 0 or 1 for that position. If the mutated structure differs from the original one, it is marked for evaluation.
- *Crossover.* The crossover module exchanges alleles between adjacent pairs of the first *n* structures in the new population. The result of the crossover rate applied to the population size gives the number *n* of structures to operate on. Crossover can be implemented in a variety of ways. If, after crossover, the offspring are different from the parents, then the offspring replace the parents and are marked for evaluation.

These basic modules are added to the evaluation function supplied by the user to create the customized version of the system. The evaluation procedure takes one structure as input and returns a double-precision value.

To execute Genesis, three programs are necessary: *set-up*, *report*, and *ga*. The setup program prompts for a number of input parameters. All the information is stored in files for future use. Users can set the type of representation, number of genes, number of experiments, trials per experiment, population size, length of the structures in bits, crossover and mutation rates, generation gap, scaling window, and many other parameters. Each parameter has a default value.

The report program runs the genetic algorithm and produces a description of its performance. It summarizes the mean, variance, and range of several measurements, including on-line performance, off-line performance, average performance of the current population, and current best value.

Overview of algorithm libraries. Algorithm libraries provide a powerful collection of parameterized genetic algorithms and operators, generally coded in a common language, so users can easily incorporate them in applications. These libraries are modular, letting users select a variety of algorithms, operators, and parameters to solve particular problems. They allow parameterization so users can try different models and compare the results for the same problem. New algorithms coded in high-level languages like C or Lisp can be easily incorporated into the libraries. The user interface facilitates model configuration and manipulation, and presents the results in different shapes (tables, graphics, and so on).

Library survey. The two leading algorithm libraries are EM and OOGA. Both provide a comprehensive selection of genetic algorithms, and EM also supports evolutionary strategy simulation. OOGA can be easily tailored for specific problems. It runs in Common Lisp and CLOS (Common Lisp Object System), an object-oriented extension of Common Lisp.

EM. Developed by Hans-Michael Voigt, Joachim Born, and Jens Treptow[16] at the Institute for Informatics and Computing Techniques in Germany, EM (Evolution Machine) simulates natural evolution principles to obtain efficient optimization procedures for computer models. The authors chose different evolutionary methods to provide algorithms with different numerical characteristics. The programming environment supports the following algorithms:

- Rechenberg's evolutionary strategy,[10]
- Rechenberg and Schwefel's evolutionary strategy,[10,11]
- Born's evolutionary strategy,
- Goldberg's simple genetic algorithm,[2] and
- Voigt and Born's genetic algorithm.[16]

To run a simulation, the user provides the fitness function coded in C. The system calls the compiler and linker, which produce an executable file containing the selected algorithm and the user-supplied fitness function.

EM has extensive menus and default parameter settings. The program processes data for repeated runs, and its graphical presentation of results includes on-line displays of evolution progress and one-, two-, and three-dimensional graphs. The system runs on an IBM PC-compatible computer with the MS-DOS operating system and uses the Turbo C (or Turbo C++) compiler to generate the executable files.

OOGA. The Object-Oriented Genetic Algorithm is a simplified version of the Lisp-based software developed in 1980 by Lawrence Davis. He created it mainly to support his book,[5] but it can also be used to develop and test customized or new genetic algorithms and genetic operators.

Case study: OOGA. This algorithm is designed so each technique used by a GA is an object that can be modified, displayed, or replaced in an object-oriented fashion. It provides a highly modular architecture in which users incrementally write and modify components in Common Lisp to define and use a variety of GA techniques. The files in the OOGA system contain descriptions of several techniques used by GA researchers, but they are not exhaustive. OOGA contains three major modules:

Toolkits contain educational systems for novice users and general-purpose systems with a comprehensive set of tools.

- The *evaluation module* has the evaluation (or fitness) function that measures the worth of any chromosome for the problem to be solved.
- The *population module* contains a population of chromosomes and the techniques for creating and manipulating that population. There are a number of techniques for population encoding (binary, real number, and so on), initialization (random binary, random real, and normal distribution) and deletion (delete all and delete last).
- The *reproduction module* has a set of genetic operators for selecting and creating new chromosomes. This module allows GA configurations with more than one genetic operator. The system creates a list with user-selected operators and executes their parameter settings, before executing them in sequence. OOGA provides a number of genetic operators for selection (for example, roulette wheel), crossover (one- and two-point crossover, mutate-and-crossover), and mutation. The user can set all parameters, such as the bit-mutation and crossover rates.

The last two modules are, in fact, libraries of different techniques enabling the user to configure a particular genetic algorithm. When the genetic algorithm is run, the evaluation, population, and reproduction modules work together to evolve a population of chromosomes toward the best solution. The system also supports some normalization (for example, linear normalization) and parameterization techniques for altering the genetic operators' relative performance over the course of the run.

Toolkits

Toolkits subdivide into educational systems for novice users and general-purpose systems that provide a comprehensive set of programming tools.

Educational systems overview. Educational programming systems help novices gain a hands-on introduction to GA concepts. They typically provide a rudimentary graphical interface and a simple configuration menu. Educational systems are typically implemented on PCs for portability and low cost. For ease of use, they have a fully menu-driven graphical interface. GA Workbench[17] is one of the best examples of this class of programming environment.

Case study: GA Workbench. This environment was developed by Mark Hughes of Cambridge Consultants to run on MS-DOS/PC-DOS microcomputers. With this mouse-driven interactive program, users draw evaluation functions on the screen. The system produces runtime plots of GA population distribution, and peak and average fitness. It also displays many useful population statistics. Users can change a range of parameters, including the settings of the genetic operators, population size, and breeder selection.

GA Workbench's graphical interface uses a VGA or EGA adapter and divides the screen into seven fields consisting of menus or graphs. The *command menu* is a menu bar that lets the user enter the target function and make general commands to start or stop a GA execution. After selecting "Enter Targ" from the command menu, the user inputs the target function by drawing it on the *target* *function graph* using the mouse cursor.

The *algorithm control chapter* can contain two pages (hence "chapter"), but only one page is visible at a time. Clicking with the mouse on screen arrows lets the user flip pages forward or backward. The initial page, the "simple genetic algorithm page," shows a number of input variables used to control the algorithm's operation. The variable values can be numeric or text strings, and the user can alter any of these values by clicking the left mouse button on the up or down arrows to the left of each value. The "general program control variables page" contains variables related to general program operation rather than a specific algorithm. Here the user can select the source of data for plotting on the output plot graph, set the scale for the x or y axis, seed the random number generated, or determine the frequency with which the population distribution histogram is updated.

The *output variables box* contains the current values of variables relating to the current algorithm. For the simple genetic algorithm, a counter of generations is presented as well as the optimum fitness value, current best fitness, average fitness, optimum x, current best x, and average x. The *population distribution histogram* shows the genetic algorithm's distribution of organisms by value of x. The histogram is updated according to the frequency set in the general program control variables page. The *output graph* plots several output variables against time.

From any graph, the user can read the coordinate values of the point indicated by the mouse cursor. When the user moves the cursor over the plot area of a graph, it changes to a cross hair and the *axis value box* displays the coordinate values.

By drawing the target function, varying several numeric control parameters, and selecting different types of algorithms and genetic operators, the novice user can practice and see how quickly the algorithm can find the peak value, or indeed if it succeeds at all.

General-purpose programming systems overviw. General-purpose systems are the ultimate in flexible GA programming. Not only do they let users develop their own GA applications and algorithms; they also let users customize the system.

These programming systems provide a comprehensive toolkit, including

- a sophisticated graphical interface,
- a parameterized algorithm library,
- a high-level language for programming GAs, and
- an open architecture.

Users access system components via a menu-driven graphical interface. The algorithm library is normally "open," letting users modify or enhance any module. A high-level language — often object-oriented — may be provided for programming GA applications, algorithms, and operators through specialized data structures and functions. And because parallel GAs are becoming important, systems provide translators to parallel machines and distributed systems, such as networks of workstations.

General-purpose systems let programmers develop applications and algorithms and customize the system.

General-purpose survey. The number of general-purpose systems is increasing, stimulated by growing interest in GA applications in many domains. Systems in this category include Splicer, which presents interchangeable libraries for developing applications; MicroGA, which is an easy-to-use object-oriented environment for PCs and Macintoshes; and the parallel environments Engeneer, GAME, and Pegasus.

Engeneer. Logica Cambridge developed Engeneer[18] as an in-house environment to assist in GA application development in a wide range of domains. The C software runs on Unix systems as part of a consultancy and systems package. It supports both interactive (X Windows) and batch (command-line) operation. Also, it supports a certain degree of parallelism for the execution of application-dependent evaluation functions.

Engeneer provides flexible mechanisms that let the developer rapidly bring the power of GAs to bear on new problem domains. Starting with the Genetic Description Language, the developer can describe, at a high level, the structure of the "genetic material" used. The language supports discrete genes with user-defined cardinality and includes features such as multiple models of chromosomes, multiple species models, and nonevolvable parsing symbols, which can be used for decoding complex genetic material.

A descriptive high-level language, the Evolutionary Model Language, lets the user describe the GA type in terms of configurable options including population size, population structure and source, selection method, crossover type and probability, mutation type and probability, inversion, dispersal method, and number of offspring per generation.

An interactive interface (with on-line help) supports both high-level languages. Descriptions and models can be defined "on the fly" or loaded from audit files, which are automatically created during a GA run. Users can monitor GA progress with graphical tools and by defining intervals for automatic storage of results. Automatic storage lets the user restart Engeneer from any point in a run, by loading both the population at that time and the evolutionary model.

To connect Engeneer to different problem domains, a user specifies the name of the program to evaluate the problem-specific fitness function and constructs a simple parsing routine to interpret the genetic material. Engeneer provides a library of standard interpretation routines for commonly used representation schemes such as gray coding and permutations. The fitness evaluation can then be run as the GA's slave process or via standard handshaking routines. Better still, it can be run on the machine hosting Engeneer or on any sequential or parallel hardware capable of connecting to a Unix machine.

GAME. The Genetic Algorithm Manipulation Environment is being developed as part of the European Community (ESPRIT III) GA project called Papagena. It is an object-oriented environment for programming parallel GA applications and algorithms, and mapping them onto parallel machines. The environment has five principal modules.

The *virtual machine* (VM) is the module responsible for maintaining data structures that represent genetic information and providing facilities for their manipulation and evaluation. It isolates genetic operators and algorithms from dealing directly with data structures through a set of low-level commands implemented as a collection of functions called the VM Application Program Interface (VM-API). The VM also supports fine-grained parallelism and can execute several commands simultaneously. It comprises three modules: the production manager, the fitness evaluation module, and the parallel support module. The first executes genetic manipulation commands over the data structures residing in the VM population pools. The VM-API includes commands for swapping, inverting, duplicating, and modifying genetic structures. The fitness evaluation module performs the actual evaluation of genetic structures and such related calculations as total, average, highest, and lowest fitness values. The problem-dependent objective function is only "connected" to the fitness evaluation module at link time. Finally, the parallel support module schedules commands received by the VM among several copies of the population manager and fitness evaluation modules.

The *parallel execution module* (PEM) implements a hardware/operating system-independent interface that supports multiple, parallel computational models. It provides straightforward API-containing functions for process initiation, termination, synchronization, and communication. It is responsible for integrating application components (algorithms, operators, user interface, and virtual machine) defined as GAME components. The PEM is implemented in two layers. The upper layer defines the standard interface functions used by all GAME components of an application. The lower layer implements the functions that map the upper layer requests into the particular environment. PEM's design permits porting GAME applications to diverse sequential and parallel machines by simply linking with the PEM library implemented for the required machine/operating system.

A *graphical user interface* module containing simple graphic widgets for MS-Windows and X Windows environments is also provided. It enables applications to input and output data in a variety of formats. GAME's GUI contains standard dialog boxes, buttons, and charting windows that can be associated by the user with events reported by the monitoring control module.

The *monitoring control module* (MCM) collects and displays (through the GUI) events that occur during a simulation session. Each GAME component notifies the MCM about messages received or any modification of the data elements it maintains. Users can select the level of monitoring for each component. The MCM can also inform other GAME components about particular events through its "lists of interests" mechanism.

The *genetic algorithm libraries* comprise a collection of hierarchically organized modules containing predefined, parameterized applications; genetic algorithms; and genetic operators. New applications and algorithms can be created by simply combining the required components from the libraries and setting their parameters in a configuration file.

New applications and algorithms can be created by combining components from libraries and setting their parameters.

The environment is programmed in C++ and is available in source code for full user modification.

MicroGA. Marketed by Emergent Behavior, MicroGA is designed for a wide range of complex problems. It is small and easy to use, but expandable. Because the system is a framework of C++ objects, several pieces working together give the user some default behavior. In this, MicroGA is far from the library concept, in which a set of functions (or classes) is offered for incorporation in user applications. The framework is almost a ready-to-use application. MicroGA needs only a few user-defined parameters to start running. The package comprises a compiled library of C++ objects, three sample programs, a sample program with an Object Windows Library user interface (from Borland), and the Galapagos code-generation system. MicroGA runs on IBM PC-compatible systems with Microsoft Windows 3.0 (or later), using Turbo or Borland C++. It also runs on Macintosh computers.

The application developer can configure an application manually or by using Galapagos. This Windows-based code generator produces, from a set of custom templates and a little user-provided information, a complete stand-alone MicroGA application. It helps with the creation of a subclass derived from its "TIndividual" class, required by the environment to create the genetic data structure to be manipulated. Galapagos requests the number of genes for the prototype individual, as well as the range of possible values they can assume. The user can specify the evaluation function, but the Galapagos notation does not allow complex or nonmathematical fitness functions. Galapagos creates a class, derived from TIndividual, which contains the specific member functions as required by the user application.

Users can manually define applications requiring complex genetic data structures and fitness functions by having them inherit from the TIndividual class and writing the code for its member functions. After creating the application-dependent genetic data structure and fitness function, MicroGA compiles and links everything using the Borland or Turbo C++ compiler, and produces a file executable in Microsoft Windows.

MicroGA is very easy to use and lets users create GA applications quickly. However, for real applications the user must understand basic concepts of object-oriented programming and Windows interfacing.

Pegasus. The Programming Environment for Parallel Genetic Algorithms, or Pegasus, was developed at the German National Research Center for Computer Science. The toolkit can be used for programming a wide range of genetic algorithms, as well as for educational purposes. The environment is written in ANSI-C and is available for many different Unix-based machines. It runs on multiple instruction, multiple data parallel machines, such as transputers, and distributed systems of workstations. Pegasus is structured in four hierarchical levels:

- the user interface,
- the Pegasus kernel and library,
- compilers for several Unix-based machines, and
- the sequential and distributed or parallel hardware.

The user interface consists of three parts: the Pegasus script language, a graphical interface, and a user library. The user library has the same functionality as the Pegasus GA library. It lets the user define application-specific functions not provided by the system library, using the script language to specify the experiment. The user defines the application-dependent data structures, attaches the genetic operators to them, and specifies the I/O interface. The script language specifies the construction of subpopulations connected via the graphical interface.

The kernel includes base and frame functions. The *base functions* control the execution order of the genetic operators, manage communication among different processes, and provide I/O facilities. They build general frames for simulating GAs and can be considered as autonomous processes. They interpret the Pegasus script, create appropriate data structures, and describe the order of frame functions. Invoked by a base function, a *frame function* controls the execution of a single genetic operator. Frame functions prepare the data representing the genetic material and apply the genetic operators to it, according to the script specification. The library contains genetic operators, a collection of fitness functions, and I/O and control procedures. Hence, it gives the user validated modules for constructing applications.

Currently Pegasus can be compiled with the GNU C, RS/6000 C, ACE-C, and Alliant FX/2800 C compilers. It runs on Sun and IBM RS/6000 workstations, as well as on the Alliant FX/28 MIMD architecture.

Splicer. Created by the Software Technology Branch of the Information Systems Directorate at NASA Johnson Space Flight Center, with support from the Mitre Corporation,[19] Splicer is one of the most comprehensive environments available. We present it in the case study.

Case study: Splicer. The modular architecture includes three principal parts — the genetic-algorithm kernel, interchangeable representation libraries, and interchangeable fitness modules — and user interface libraries. It was originally developed in C on an Apple Macintosh and then ported to Unix workstations (Sun 3 and 4, IBM RS/6000) using X Windows. The three modules are completely portable.

The *genetic-algorithm kernel* comprises all functions necessary to manipulate populations. It operates independently from the problem representation (encoding), the fitness function, and the user interface. Some functions it supports are creation of populations and members, fitness scaling, parent selection and sampling, and generation of population statistics.

Interchangeable *representation libraries* store a variety of predefined problem-encoding schemes and functions, permitting the GA kernel to be used for any representation scheme. There are representation libraries for binary strings and permutations. These libraries contain functions for the definition, creation, and decoding of genetic strings, as well as multiple crossover and mutation operators. Furthermore, the Splicer tool defines interfaces to let the user create new representation libraries.

We expect the number and diversity of application-oriented systems to expand rapidly in the next few years.

Fitness modules are interchangeable and store fitness functions. They are the only component of the environment a user must create or alter to solve a particular problem. Users can create a fitness (scoring) function, set the initial values for various Splicer control parameters (for example, population size), and create a function that graphically displays the best solutions as they are found.

There are two user interface libraries: one for Macintoshes and one for X Windows. They are event-driven and provide graphical output in windows.

Stand-alone Splicer applications can be used to solve problems without any need for computer programming. However, to create a Splicer application for a particular problem, the user must create a fitness module using C. Splicer, Version 1.0, is currently available free to NASA and its contractors for use on government projects. In the future it will be possible to purchase Splicer for a nominal fee.

Future developments

As with any new technology, in the early stages of development the emphasis for tools is on ease of use. Application-oriented systems have a crucial role in bringing the technology to a growing set of domains, since they are targeted and tailored for specific users. Therefore, we expect the number and diversity of application-oriented systems to expand rapidly in the next few years. This development, coupled with the discovery of new algorithms and techniques, should bring an increase in algorithm-specific systems, possibly leading to general-purpose GAs. Algorithm libraries will provide access to efficient versions of these algorithms.

Interest in educational systems and demonstrators of GAs is rapidly growing. The contribution of such systems comes at the start of a new technology, but their usage traditionally diminishes as general-purpose systems mature. Thus we expect a decline in educational systems as sophisticated general-purpose systems become available and easier to use. General-purpose systems appeared very recently. With the introduction of Splicer, we expect commercial development systems in the near future. We should see programming environments for an expanding range of sequential and parallel computers, and more public-domain open-system programming environments from universities and research centers.

One high-growth area should be the association of genetic algorithms and other optimization algorithms in hybrid systems. Recently there has been considerable interest in creating hybrids of genetic algorithms and expert systems or neural networks. If a particularly complex problem requires optimization and either decision-support or pattern-recognition processes, then using a hybrid system makes sense. For example, neural networks and genetic algorithms have been used to train networks and have achieved performance levels exceedin that of the commonly used back-propa gation model. GAs have also been use to select the optimal configurations fo neural networks, such as learning rate and the number of hidden units and la ers. By the end of the century, hybrid G neural networks will have made signifi cant progress with some currently in tractable machine learning problems. Promising domains include autonomous vehicle control, signal processing, and intelligent process control.

Genetic algorithms are robust, adaptive search techniques that may be immediately tailored to real problems. The two major trends in future environments will be the exploitation of parallel GAs and the programming of hybrid applications linking GAs with neural networks, expert systems, and traditional utilities such as spreadsheets and databases. ■

Acknowledgments

We thank Lawrence Davis, Darrel Whitley, and Nicol Schraudolph for recommending GA programming environments for us to survey. We also thank Frank Hoffmeister, Hans-Michael Voigt, and Joachim Born for their advice. Finally, we acknowledge our colleagues Jason Kingdon and Suran Goonatilake for commenting on early drafts of this article.

References

1. J.H. Holland, *Adaptation in Natural and Artificial Systems*, Univ. of Michigan Press, Ann Arbor, Mich., 1975.

2. D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Mass., 1989.

3. K.A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, doctoral dissertation, Univ. of Michigan, Ann Arbor, Mich., 1975.

4. J.J. Grefenstette, "Genesis: A System for Using Genetic Search Procedures," *Proc. Conf. Intelligent Systems and Machines*, 1984, pp. 161-165.

5. L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

6. H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 416-421.

7. M. Gorges-Schleuter, "Asparagos: An Asynchronous Parallel Genetic Optimisation Strategy," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 422-427.

8. H. Mühlenbein, "Evolution in Time and Space — The Parallel Genetic Algorithm," in *Foundations of Genetic Algorithms*, G. Rawlins, ed., Morgan Kaufmann, San Mateo, Calif., 1991, pp. 316-337.

9. R. Tanese, "Distributed Genetic Algorithms," *Proc. Third Int'l Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, Calif., 1989, pp. 434-440.

10. I. Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution* [*Evolutionary Strategy: Optimization of Technical Systems According to the Principles of Biological Evolution*], Frommann-Holzboog Verlag, Stuttgart, Germany, 1973.

11. H.P. Schwefel, "Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie" [Numerical Optimization of Computer Models by Means of the Evolutionary Strategy], *Interdisciplinary Systems Research*, Vol. 26, Birkäuser, Basel, Switzerland, 1977.

12. F. Hoffmeister and T. Bäck, "Genetic Algorithms and Evolution Strategies: Similarities and Differences," Tech. Report "Grüne Reihe," No. 365, Dept. of Computer Science, Univ. of Dortmund, Germany, 1990.

13. F. Hoffmeister, "The User's Guide to Escapade 1.2: A Runtime Environment for Evolution Strategies," Dept. of Computer Science, Univ. of Dortmund, Germany, 1991.

14. D. Whitley and J. Kauth, "Genitor: A Different Genetic Algorithm," *Proc. Rocky Mountain Conf. Artificial Intelligence*, 1988, pp. 118-130.

15. N.N. Schraudolph and J.J. Grefenstette, "A User's Guide to GAUCSD 1.2," Computer Science and Eng. Dept., Univ. of California, San Diego, 1991.

16. H.M. Voigt, J. Born, and J. Treptow, "The Evolution Machine Manual — V 2.1," Inst. for Informatics and Computing Techniques, Berlin, 1991.

17. M. Hughes, "Genetic Algorithm Workbench Documentation," Cambridge Consultants, Cambridge, UK, 1989.

18. G. Robbins, "Engeneer — The Evolution of Solutions," *Proc. Fifth Ann. Seminar Neural Networks and Genetic Algorithms*, IBC Technical Services Ltd., London, 1992, pp. 218-232.

19. NASA Johnson Space Flight Center, "Splicer — A Genetic Tool for Search and Optimization," *Genetic Algorithm Digest*, Vol. 5, Issue 17, 1991, p. 4.

José L. Ribeiro Filho is a research staff member in the Núcleo de Computação Eletrônica at the Universidade Federal do Rio de Janeiro, Brazil. His research interests include computer architectures, parallel processing, communication systems, and optimization techniques such as genetic algorithms.

Ribeiro Filho received an MS in computer science in 1989 from the Federal University of Rio de Janeiro and is now working on a PhD at University College London.

Philip C. Treleaven is a professor of computer science at University College London. His research interests are in neural computing, computing applications in finance, and fifth-generation computers for artificial intelligence. He has consulted for IBM, DEC, GEC, Fujitsu, Mitsubishi, Philips, Siemens, and Thomson, and acted as adviser to government ministers in Japan, Germany, France, Korea, and other countries. Among the European collaborative research projects he is involved with is the Galatea neural computing project.

Cesare Alippi is working on a PhD in artificial intelligence at Politecnico di Milano, where he is analyzing the sensitivity of neural networks to neural value quantization. His other research interests include genetic algorithms and fault tolerance. Previously he was a researcher in the Department of Computer Science at University College London. Alippi received a BS degree in electronic engineering from Politecnico di Milano in 1990.

Readers can contact Ribeiro Filho at the Department of Computer Science, University College London, Gower St., London WC1E 6BT, UK; e-mail j.ribeirofilho@cs.ucl.ac.uk.