AN INVESTIGATION INTO THE USE OF GENETIC PROGRAMMING FOR INTELLIGENT NETWORK SERVICE CREATION

BY

PETER MARTIN

A dissertation submitted in partial fulfilment of the requirements for the degree of

MSc Advanced Computing

Bournemouth University

1998

'… from so simple a beginning endless forms most beautiful and most wonderful have been and are being evolved'

Charles Robert Darwin (1809-1882), On the Origin of Species by Means of Natural Selection. 1st Edition.

ABSTRACT

Service creation is crucial to the success of Intelligent Networks (IN). However, the time required to develop complex services is increasing. By reducing the elapsed time needed to generate the service logic and by reducing the opportunity for implementation errors to appear in the service logic, a higher quality IN service can be delivered.

This project explores an alternative method to the existing manual service creation, by exploiting the properties of Genetic Programming (GP). Genetic Programming is a powerful method for evolving computer programs via the process of natural selection. [Koz92]. The use of Genetic Programming to produce service logic programs for IN is analysed and a number of key features identified. Principally for GP to be of benefit to IN it must be able to reduce the time to create a service and reduce the number of implementation errors in the resultant program.

Experimental evidence is presented that shows that using Genetic Programming is a viable method for service creation in Intelligent Networks, and can reduce the time to create a program by several orders of magnitude compared to a human. The case is also argued that since GP needs a fitness function to be developed, the initial specification should be of a higher quality than one produced for a human programmer, thereby reducing the number of errors in the final program.

To implement the experimental prototype, existing methods of evolving complex systems using GP were researched. A new method of ensuring the property of closure is presented that does not constrain the development of novel service logic implementations, in contrast to existing methods commonly employed in GP.

Further work is identified at the end to improve upon the performance and to explore more complex services.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

AUTHOR DECLARATIONS

1. During the period of registered study in which this dissertation was prepared the author has not been registered for any other academic award or qualification.

2. The material included in this dissertation has not been submitted wholly or in part for any academic award or qualification other that that which is now submitted.

This dissertation consists of 117 pages.

ACKNOWLEDGMENTS

Firstly I would like to thank my project supervisor Mike Jones of Bournemouth University for his help and encouragement. I am also grateful to my colleague Dr. Jeremy Bennett for reading a late draft of this paper and his helpful comments and encouragement.

Finally I would like to thank Marconi Communications Limited, formerly GPT Limited for sponsoring me to do this MSc.

CHAPTER 1. INTRODUCTION

This chapter gives an outline of the project and describes the approach used.

1.1. Project Outline

As telecommunication systems become more complex and the effort needed to create services in a timely manner becomes greater, so the need for alternative means of realising systems becomes more urgent.

The purpose of this work is to explore how a method from the field of evolutionary computing can be of assistance in the field of Intelligent Networks (IN) in helping to create new telecommunications services.

The premise used in this work is that using a branch of evolutionary computing, a system can translate a specification into an implementation without the direct assistance of a human programmer. The benefits to be gained are faster system realisation and a more reliable implementation by focusing the effort on the requirements of a system rather than it's implementation.

1.2. Project Approach

Starting from the idea that some form of automatic programming was a feasible method to use, a detailed analysis of one method – Genetic Programming (GP) – was made. From this analysis a number of questions were raised concerning the basic feasibility, performance and scalability.

To explore the issues raised a number of experiments were then devised. Finally the experimental results were analysed and further questions uncovered.

The paper is organised into the following sections:

Chapter 2	presents the basics of IN and makes the case for considering the use of GP.
Chapter 3	gives an outline of GP in a problem independent context.
Chapter 4	discusses the problem specific details of Genetic Programming
Chapter 5	presents the experiments devised to establish the suitability of GP for IN
Chapter 6	discusses the results in the context of IN
Chapter 7	gives some outline of further work
Chapter 8	presents the general conclusions.

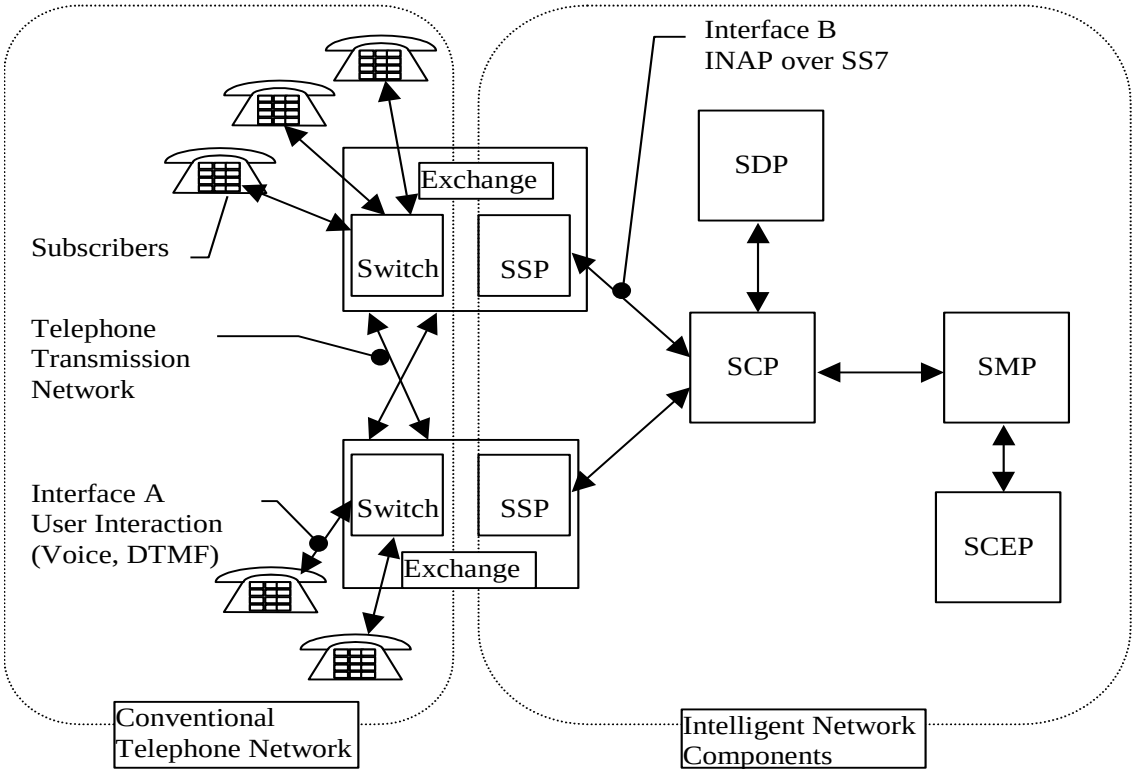
CHAPTER 2. INTELLIGENT NETWORKS AND SERVICE CREATION

This chapter introduces the idea of Intelligent Networks in telecommunications networks and describes the role of service creation. Some of the drawbacks of current methods of service creation are discussed and the rationale for alternative methods given.

Traditional telephony in the past 20 years has concentrated on delivering telephony services to customers by means of stored program switches. Customers have, until recently, been restricted to relatively crude terminal equipment that supports voice and Dual Tone Multi Frequency (DTMF) user controls.

As the number of services offered has grown and the sophistication of telephone equipment has risen, it has become clear that offering services via the traditional embedded switch technology does not scale well, and that other platforms for providing the services are required.

The primary objective of Intelligent Networks is to move the service computation to readily available computers. The basic intelligent network is shown in Figure 1.



Key	Name	Function
BCSM	Basic Call State Machine	The description of the internal operation of the IN portion on an SSP
SCP	Service Control Point.	The computing platform that executes the service logic
SCEP	Service Creation Environment Point.	Used to create the services that execute on the SCP
SDP	Service Data Point.	Supplies database functionality.
SMP	Service Management Point.	Used to manage the network and subscriber data
SSP	Service Switching Point.	Performs normal telephony and associated service triggering

Figure 1 Basic elements of an Intelligent Network

A secondary aim of introducing IN was to reduce the time required to develop and deploy new services. Traditional switch based solutions typically require 2 years

from the initial requirements being specified until the service is in operation [BJ97]. In a highly competitive environment this is too long, and the market window will have disappeared by the time the services come into operation. IN aims to reduce this to around 6 months by exploiting mainstream IT techniques.

In order to achieve such a startling reduction in timescales, new methods of creating service applications were required. From this followed the introduction of the SCEP, or Service Creation Environment Point.

One such system has been developed by Marconi Communications Limited, formerly GPT Limited [Mar96] and is marketed as GAIN INventor (TM) . This employs a service lifecycle shown in Figure 2. This shows a simplified waterfall model where the stages 1-4 as a whole map to subgoals 2 to 7 described by Boehm [Boe81] Chap. 4. Page 37. An implicit assumption is that the feasibility of a service has already been established. Maintenance and phaseout are part of the service creation process but are not considered for development purposes.

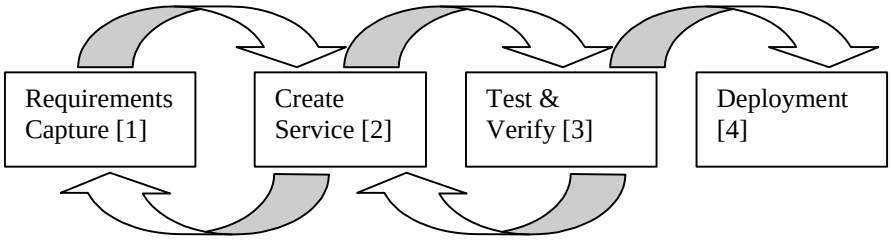


Figure 2 Service Creation Lifecycle

The first two phases occur when either a customer specifies their service requirements directly to the provider of the network, or as a result of collaboration between user and provider. They are carried out using the tools and techniques provided by the IN equipment vendor. During the requirements capture phase, it is quite likely that the same tools will be used in order to produce rapid prototypes so

that the customer can verify the essential requirements early on in the development of the service.

In the GAIN INventor (TM) system the user selects sets of iconic images from a palette and joins them together to create a directed graph. Each node (icon) in the graph has a set of attributes that the service creator can change to determine the eventual behaviour of the service being constructed.

A compiler is used to translate the abstract service representation to C code that conforms to the requirements of the runtime environment.

Experience has shown that the time required to complete the first phase is relatively short, but the time required to implement complex services in phases 2 and 3 can be several months. A typical non-trivial service can require several thousand icons, and results in dozens of valid traversals of the graph. A means of reducing the duration of these phases is therefore of benefit to the network and service operators.

2.1. An Alternative approach

The major problems encountered in the existing system are associated with software engineering management issues namely, productivity and quality control. Despite the promises of the early IN systems and the advanced tools available, complex services still take a considerable amount of time to develop using traditional software engineering techniques and there is still a level of defects found in the services themselves [BJ97].

This work attempts to address the difficulties with the first two phases, by means of automatically deriving an implementation from the requirements, or as Teller [TA97], Langdon [Lan98] and others put it, by using Automatic Programming. This approach was hinted at by Boehm [Boe81] Chap. 33 where a mention is made of automatic programming. In 1981 the idea was considered interesting but 'somewhat beyond the current frontier of the state of the art'. This paper demonstrates that

automatic programming by using Genetic Programming (GP) is now a viable alternative in the domain of IN.

To be able to judge whether an alternative approach to manual programming is worthwhile a number of questions need to be answered with regards to the alternative:

1. Has the alternative approach demonstrated that it can generate programs that perform as well as or better than a human?

2. In the domain being considered what are the observable and measurable attributes of the process of generating programs?

3. What are the observable and measurable attributes of the generated programs ?

4. Does the alternative have the ability to create IN applications.

5. Can it handle the range of program complexity that a human can; i.e.; is it scalable?

Firstly, GP has demonstrated that it can produce results that are at least as good as a human programmer and in some cases provide solutions to problems that a human has not been able to achieve as in the case of discovering an electronic circuit to yield a cube root function [KBF96], and to create a rule for cellular automata that performs better than any rule written by a human [ABK96]. Sharman et al [SEL95] has also shown that programs for Digital Signal Processors (DSPs) evolved using GP can outperform existing programs. Clearly then GP has the potential to generate programs that humans find hard.

Secondly, we can consider an existing service creation case study [BJ97]. This study showed that for a complex service a human required 4.5 Man years of effort to analyse, design, code and test the service. The principle measurable attribute is therefore the elapsed time required to implement the service and this attribute will

be quantified for GP by experimental data presented later. Other attributes are cost of equipment and the degree of human intervention but are not considered further in this work.

Thirdly, a key measurable attribute of the program is the level of defects. Broadly defects fall into one of two categories [Som96]; errors due to incorrect requirements analysis and errors due to implementation deficiencies either by errors in programming or design. The first type is common to whatever method of programming is adopted. As summarised by Davis [Dav93] the earlier requirement related errors are found, the lower the cost to remedy the error. As will be seen later using GP forces the designer to consider requirements in more detail initially (for fitness evaluation) so the implication is that using GP will result in fewer errors introduced by faults in the requirements. Again the study by Boulton et al [BJ97] shows that even using advanced tools such as INventor, there were 15 failures associated with the service. Anecdotal evidence suggests that these were all implementation errors.

Fourthly, there is no existing information on using GP for IN services creation. Experimental work will be required to ascertain whether GP can be used for service creation.

Lastly, the question of whether GP can scale can only be answered in full by analysing experimental data, but initial indications show that GP can create programs to solve complex problems in other domains.

It is worth noting that other alternatives such as artificial neural networks, hill climbing, decision trees, reinforcement learning, combinatorial search or knowledge based systems have not been explored in the context of this problem, but Koza [Koz96] makes a powerful argument why such a comparison would not be beneficial anyway. The main point of his argument is that most machine learning

paradigms are highly specialised and any attempt to do a cross paradigm comparison will 'gravitate to utterly trivial problems'.

Notwithstanding the above, one area that promises to offer a viable alternative to GP is Inductive Logic Programming [BG95], and a useful comparison has been made between Inductive Logic Programming and GP by Tang [TCM98] albeit for a fairly simple problem. Furthermore some limited experimental results have been presented between traditional Genetic Algorithms (GA) and GP in the domain of telecommunications applications by Sinclair [SS97] and Aiyarak [ASS97]. None of these comparisons offers any convincing arguments in favour of any particular method, indeed, the comparisons between GA and GP give contradictory results and appear to be heavily influenced by the type of problem being solved.

CHAPTER 3. GENETIC PROGRAMMING PRINCIPLES

This chapter describes Genetic Programming as a general method for solving problems.

Genetic Programming (GP) is an extension of Genetic Algorithms (GA) first proposed by Holland [Hol92] where the individuals that make up a population are not fixed length, limited alphabet strings, but rather structures that represent programs. The structures are typically trees that describe the program [Koz92], but may take on other forms such as a binary string [Ban93]. The purpose therefore is to evolve programs that can solve the problem presented to the system

GP uses four steps to solve a problem:

1. A set of individuals (programs) is randomly created. This is the initial population.

2. These are then evaluated (executed or interpreted) for fitness, and a fitness value is assigned to each individual.

3. These individuals are then used to form the next population by means of probabilistically selecting one of:

   - asexual reproduction
   - sexual reproduction or crossover
   - mutation.

   This new population is then re-evaluated.

4. This cycle is repeated until either a pre-determined number of generations have been processed or an individual meets a predetermined level of fitness.

This is illustrated as a flow chart in Figure 3

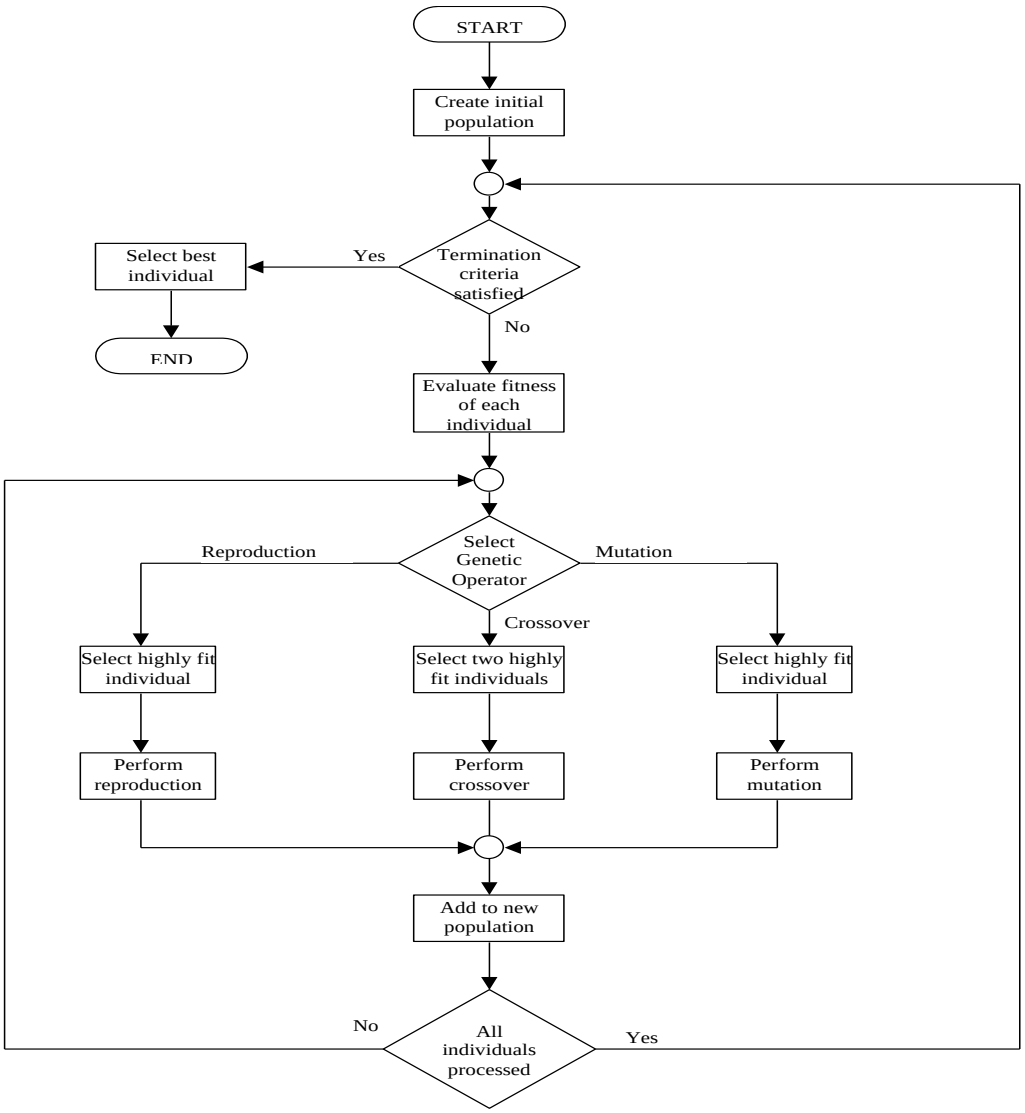


Figure 3 Flowchart of Genetic Programming

3.1. Function and terminal sets

In classic tree based GP each genetic program consists of one or more nodes, chosen from one of two sets. The non-leaf nodes are known as the function set $\boldsymbol{F}=\{f_1,\ldots, f_n\}$.

All nodes in $\boldsymbol{F}$ have arity (that is can take a number of arguments) one or greater. The leaf nodes are the terminal set $\boldsymbol{T}=\{t_1,\ldots t_n\}$. Nodes in $\boldsymbol{T}$ have arity of zero.

If the members of $\boldsymbol{T}$ are considered as functions with arity zero, then the total set of nodes is:

$$\boldsymbol{C} = \boldsymbol{F} \cup \boldsymbol{T}$$

The search space is the set of all possible compositions of the members of $\boldsymbol{C}$. This set must exhibit two properties [Koz92]: closure and sufficiency.

Closure requires each member of $\boldsymbol{C}$ to accept as its arguments any other member in $\boldsymbol{C}$. This property is required in order to guarantee that programs can operate without run time errors being generated. The common example cited is that of protecting the division operator to prevent division by zero errors, but also extends to data types used when calling functions and accessing terminal types.

This may be achieved in a number of ways. Firstly Koza [Koz92] restricts the types of arguments and function return types to compatible types. For instance, all floating point types as in the symbolic regression examples or logical in the Boolean examples. For simple problems with single data types this is sufficient.

Secondly, in strongly typed approaches such as those described by Montana [Mon95] and Haynes et al [HWSS95]constraints are placed on the creation of individuals to satisfy the type rules. The advantage here is reducing the size of the search space by eliminating individuals that would fail due to syntax errors. Clack [CY97] extended this work to show that expression based parse trees can yield more correct programs, and introduced the idea of polymorphism into the data types. Later an alternative is presented to strongly typed approaches that removes some of the deficiencies.

The sufficiency property requires that the set of functions in $\boldsymbol{C}$ is sufficient to express a program capable of solving the problem under consideration [Koz92]. This is a problem specific property and must be determined before any GP can be evolved. This together with determining a suitable fitness test requires the most effort by a user of GP.

3.2. Creation of initial population

To create the initial population a number of randomly selected nodes from the function set ***F*** are used to build trees according to the arity of the function. Leaf nodes from ***T*** are inserted according to certain criteria. Two main methods are described by Koza [Koz92]; the full, and the grow methods.

In the full method, members of ***F*** are selected until the tree reaches a pre-determined depth, then from ***T***. This results in trees with uniform depth.

The grow method differs in that a node is selected from ***C*** if the depth is less than a pre-determined maximum, else it selects from ***T***.

A third method combining the full and grow is called 'ramped half and half '. Ramped half and half operates by creating an equal number of trees with a depth between 2 and a pre-determined maximum. That is if the maximum depth is 10, then 1/9 will have depth 2, 1/9 depth 3 and so on up to depth 10. Then for each depth, 50% of the trees are created using the full method and 50% using the grow method. This is claimed by Koza [Koz92] to offer a wider variety of shapes and size in the initial population. The difference in performance between the three methods is documented in [Koz92] and [Ban93], with ramped half-and-half clearly yielding higher probabilities of success on a number of problems. Therefore this is the method used in all cases in the work described in this paper.

During the operation of GP, one of three methods of producing the next generation are used, reproduction, crossover and mutation.

Reproduction is the straightforward copying of an individual to the next generation, otherwise known as Darwinian or asexual reproduction.

Crossover, or sexual recombination, consists of taking two individuals **A** and **B**, and randomly selecting a crossover point in each. The two individuals are then split at

these points creating four subtrees $\mathbf{A_1}$ $\mathbf{A_2}$ $\mathbf{B_1}$ $\mathbf{B_2}$, and two new individuals created **C** and **D** by combining $\mathbf{A_1}$ $\mathbf{B_2}$ and $\mathbf{B_1}$ $\mathbf{A_2}$. This is shown in Figure 4

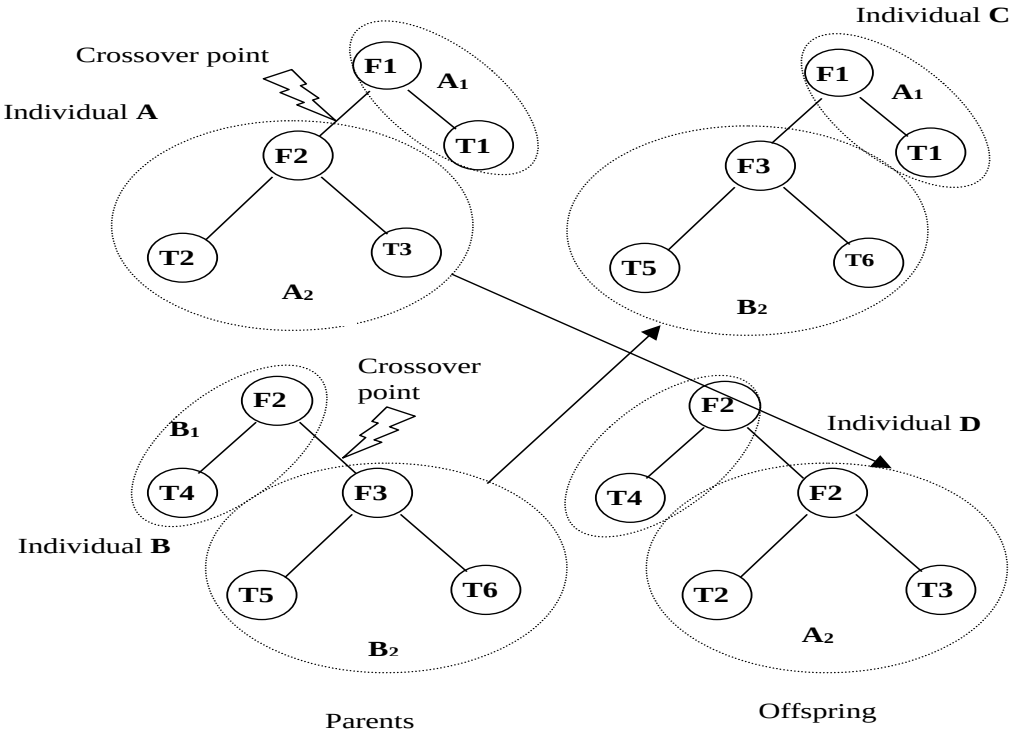


Figure 4 Operation of crossover in Genetic Programming

Mutation consists of randomly selecting a mutation point in a parse tree and substituting a new randomly generated sub tree at that point.

There is still much debate over whether crossover and mutation are useful operators [GPMAIL]. Koza [Koz92]claims that mutation does not play a large part in finding fit individuals and consequently does not use it in most of his experiments. In contrast studies by Banzhaf et al [BFN96] and Luke and Spector [LS97] show that mutation can be useful in some cases, however they have not discovered any robust heuristics that allow the selection of optimal settings. Finally,

Angeline [Ang97a] puts forward some evidence that crossover may be a form of macromutation and not play any real role in propagating so called building blocks.

3.3. Selection methods

The two main methods of selecting individuals from a generation are fitness proportionate and tournament. When using fitness proportionate, all individuals are ranked according to their absolute fitness values and the best selected. A refinement on this is rank selection [GD91] which reduces the influence of single highly fit individuals.

In tournament selection, *n* individuals are selected and the best one in the selection is propagated to the next generation. The value of *n* can be any number greater than one. The winning individual can be left in the donor population, resulting in so called over selection, where it stands a chance of being reselected as a result of further tournaments.

The choice of selection method was based on the work by Banzhaf [Ban93] where tournament selection with over selection performed better in most cases, therefore tournament selection with over selection was used in the work described later.

In order to identify good individuals, a fitness function is required that can provide a measure of how good (or bad) an individual is. Some problems use the result of the program directly as the fitness measure, for example symbolic regression. Other problems use side effects, such as the Ant problem [JCC92] that is commonly used as a benchmark of GP systems. The Ant problem uses a two dimensional toroidal grid containing a trail of food. A simulated ant is placed in this grid. The objective is to discover a controlling program that allows the ant to collect the maximum amount of food in a given time. The ant is able to move forward, turn left or right and sense food in the cell adjacent to the direction it is facing.

CHAPTER 4. GENETIC PROGRAMMING APPLIED TO SERVICE CREATION

This chapter explores the domain specific details required to use Genetic Programming in creating services for an Intelligent Network.

Section 3.1 explained that the set of functions must satisfy the sufficiency property. That is they must be rich enough to allow an evolving program to be able to satisfy the functional requirements. For instance, a requirement for a program to generate messages would require one or more functions to support this. The functions selected however also depend on the level of abstraction selected. This is dealt with in section 4.1.

Terminals may be side affecting or yield data. For this work, the functions were chosen to perform all external operations, while the terminals were chosen to yield data. In order to arrive at a sufficient set of data types, it is useful to consider what types of data are commonly encountered in telephony services. Table 1 summarises these data types.

Data Type	**Comments**
telephone numbers	Strings of digits [0-9 # *] that can be dissected and concatenated. The string length may be up to 24.
constant integral values	Used for counters and message parameter values
boolean values	Flags and status values
message types	An enumerated set used to distinguish messages

Table 1 Data types encountered in telephony services

From this it is clear that restricting functions and terminals to use a single data type in order to satisfy the closure property is not feasible.

In addition, since most IN services require some state information to be stored between message transfer points, a mechanisms for saving state information was required. The first approach to this requirement, Indexed Memory, was suggested by Teller [Tel94] where he argues that in order for GP to be able to evolve any conceivable algorithm, GP needs to be Turing Complete and that addressable memory enables this. A useful side effect of this is that memory also allows state information to be explicitly saved and retrieved.

Of course other approaches to saving state information are possible as for example in the work by Angeline [Ang97b] that uses Multiple Interacting Programs (MIPS). However for the purposes of this work Indexed Memory was chosen since it was thought that it would be easier to analyse the operation of the evolving programs.

As already noted in 3.1, several methods have been proposed to ensure that the closure property is maintained during initial creation and subsequent reproduction. An alternative is proposed in this work, based on polymorphic data types with independent values for each type supported.

This approach was devised as an alternative to the strongly typed methods by making the observation that it is possible that the criteria used to decide what is a correct program has more to do with correctness as seen by a human programmer rather than any inherent property of GP. In other words, strong typing is a necessary artefact of languages used by humans to help ease the burden on the programmer, by means of assisting machine interpretation. Perkis [Per94] has shown that an apparently haphazard mechanism in the form of a stack can yield useful results. Another objection to using a strongly based type system was that the potential number of solutions could be greatly diminished.

The work presented here uses a new data type termed Autonomous Polymorphic Addressable Memory (APAM). This consists of a set of memory locations $\boldsymbol{M} = \{\boldsymbol{L_1} \ldots \boldsymbol{L_n}\}$ which can be addressed randomly or by name. Each location is a set of

data items $L = \{d1, \ldots d_n\}$. The values of $L_n.d_1$, $L_n.d_2$ etc are independent of each other. Selection of the correct type and therefore value is performed by any function that is passed a memory reference as an argument.

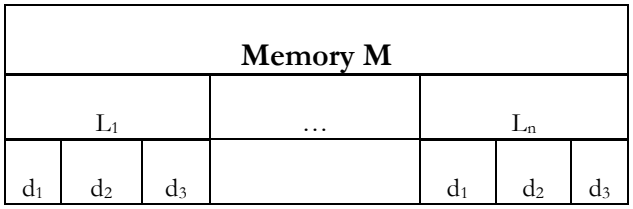


Figure 5 Layout of Autonomous Polymorphic Addressable Memory

To support this memory architecture, the terminal set T consists of memory nodes $T = \{TVAR1. . TVARn\}$. Each node returns a reference to memory address $Ln.$ and can be passed as arguments to any function.

It should be noted that this is not the same as using a generic data type where a data item is coerced into the correct type at run time. A difficulty with coercion is that many automatic conversions are meaningless. For example, in the context of telephony it would be hard to imagine what the coercion of a Boolean value into a telephone number would mean.

4.1. Choosing a level of abstraction

The number of functions in C and their arity can be used to estimate the size of the search space as described by Iba [Iba96] and Langdon [Lan97]. It is clear that a large function set would result in a large search space, and therefore reduce the probability of achieving good performance. Therefore, a level of abstraction that uses a smaller number of functions is desirable.

As an example, consider several sizes for F, assuming each member of F has arity of two, and that there are ten members of T. The population size is calculated using Langdon's method [Lan97] and the results summarised in Table 2

Size of F	Number of possible trees of depth 10
5	8.0×10^{9}
10	1.3×10^{11}
15	6.0×10^{11}
20	2.0×10^{12}
25	5.0×10^{12}
100	1.2×10^{15}

Table 2 Potential size of population for different size F

In the domain of IN, there are three main levels of abstraction that can be considered. This list does not include low-level functions, for instance the UNIX API, or raw machine language, though the latter is clearly feasible as demonstrated by the use of Java byte code as the working set for C as described by Banzhaf et al [BNO97]:

- ICON level with attributes as terminals. This level is based on the set of functions offered to service creators using the GPT GAIN INventor (TM) product [Mar96]. Other service creation systems have similar or even higher level of abstraction. A subset of around twenty icons is sufficient to construct the majority of services encountered in existing networks.

- Icon function level. This is the level used by the internal tools within GAIN INventor (TM). Each ICON typically makes use of between one and twenty functions. The total number of functions is around 200.

- API level. This is the lowest practical level. This is the set of API functions offered by the target platform. In the case of the GPT GAIN INventor (TM) product, this is a set of over one hundred and fifty function calls designed to allow services and other applications to be constructed.

For these experiments, the level was initially pitched at the ICON level since this level allows humans to create production quality services, giving a potential size of $\boldsymbol{F}$ of around twenty. In this work only a small subset of this potential set was chosen. An attempt was made to see if this level of abstraction was optimal by carrying out additional experiments using a level closer to the API. Initial results indicate that using the ICON level may not be the most effective.

4.2. Method of measuring fitness

The decision was made to measure the fitness of the GP at Interface B (Figure 1) since this is a standardised external interface [Itu94a] and would allow the specification of services to be performed at the network level.

The Basic Call State Machine (BCSM) of the standards [Itu94a] is simplified, and called a Simple Call State Model (SCSM) in order to focus on the GP methodology rather than being distracted by the complexities of the BCSM.

By treating the GP as a black box it should be possible to have a high degree of confidence that individuals operate as expected. This would operate by means of sending messages to each individual and waiting for an appropriate response. At the conceptual level this is exactly what is done, but at the practical level things are not so simple.

The initial attempt used this approach, setting a timeout against each response expected, but this resulted in excessive time required to test poor individuals since many timeouts were encountered for highly unfit individuals. It was also very hard to debug such a system.

In order to simplify the system, the execution of the system was driven by the service logic so that when evaluating fitness, the service logic is executed directly. It then makes requests to the SCSM as required. This inversion of roles removes the problems of detecting non-responsive service logic programs, and simplifies the initial debugging and verification.

When running a fitness test, there are two problem specific related measures used to determine how fit an individual is, as well as non-problem specific measures such as parsimony:

- The number of correct state transitions made. Each correct transition is rewarded with a value of 100. Each incorrect transition is penalised with a value of -20. The reward and penalty values are summed. Call this value **s**.

- The number of correct parameter values passed back to the SCSM. A correct parameter value is rewarded with a value of 100, and each incorrect value is penalised with a value of -20. The reward and penalty values are summed. Call this value **p**.

Raw fitness **r** is given by $\mathbf{r = s + p}$

Normalised fitness **n** is given by $\mathbf{n} = \boldsymbol{k} - \mathbf{r}$ where $\boldsymbol{k}$ is a constant that is dependent on the number of state transtions and message parameters in the problem being considered, such that for a 100% fit individual $\mathbf{n} = 0$.

A count is maintained of the number of correct and incorrect state transitions and correct and incorrect message parameter values.

4.3. Measuring performance and estimating effort

Koza [Koz92] p.191 describes a method of measuring the performance of a GP system that consists of running a large number of trials noting for each run, whether the run yielded a correct individual, and the generation number that the run produced such an individual.

For a population size *M*, the cumulative probability of success *P (M, i)* for any generation *i* is calculated. This is a measure of the success of the particular set of configuration settings. From this it is possible to estimate the effort required to find a satisfactory outcome. The cumulative probability *P* (*M*, *i*), is the total number of

runs that produced a successful outcome up to and including generation *i*, divided by the number of runs conducted.

From this, an estimate can be made of the number of independent runs required to reach a satisfactory result with probability z for generation *i*, using equation 1 ([Koz92] p.194):

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil \quad (1)$$

In all cases described in this work, z=99%

The quantities *P* (*M, i*) and *R* (z) are plotted on a graph.

The effort ε required to find a solution by generation *i* for is given by equation 2:

$$\varepsilon = M.R(z) \quad (2)$$

Additional information collected includes the total time taken for each run (τ), the number of individuals processed, the number of unique individuals that were 100% fit (Ψ), the number of 100% individuals at the final generation (ϑ) and details of the best individual of each run.

4.4. Implementation details

Given that the purpose of the project was to investigate the usefulness of the GP method in a practical application, there was little to be gained by implementing a GP kernel since several implementations were already available. An existing study into GP tools on the Web by Deakin and Yates [DY97] only addressed basic operational issues and considered whether the package compiled. It also only considered five implementations.

The choice of an implementation for the experimental work was based on a larger number of criteria than Deakin's [DY97]; implementation language, portability, performance, availability, flexibility, extensibility, level of support from the author and popularity. In addition, this work considered 12 implementations.

The language issue was looked at first. Without considering GP in particular, an early search revealed several languages in use in the field of Evolutionary Computing (EC). These are summarised in Table 3 with some of their important characteristics.

Language	Comments
C++	Common language. Highly portable. OO features may help in producing problem specific solutions. Good performance characteristics
C	Common language. Very portable. Good performance characteristics.
LISP	Interpreted language. Finds favour with AI community. Initial work on GP by Koza [Koz92] was done in LISP. Performance dependent on machine and environment. Fairly portable, but requires platform specific changes to optimise some functionality.
Smalltalk	Usually interpreted environment. Another favourite language with the AI community. Requires specialised run-time environment. Strongly Object Oriented.
Java	New language. Safer to use than C++. Good support for OO. It is an interpreted language and therefore its performance is poor, but forthcoming compilers should improve the situation. Claims to be highly portable.

Table 3 Languages used for implementing common EC systems

Because C and C++ exhibit good performance characteristics, portability, ready availability on the platforms to be used for the work, and maturity, these were the preferred languages when selecting an implementation. Performance was considered particularly important since the computational effort of using GP can be high, and using commodity computers (see APPENDIX A) meant that limited computational power was available. The need to use commodity computers is related to the fact that a GP approach would not be acceptable in a commercial environment if exotic and therefore expensive computers were required.

Secondly, support for a range of computing platforms was desirable to allow work to be carried out at various locations on Windows95/NT, Linux, Solaris, and other UNIX platforms.

Thirdly, in order to be considered an implementation should be able to demonstrate the ability to implement some of the commonly encountered example tasks. These include symbolic regression and the artificial Ant problem [JCC92], and while they cannot be considered as standard, these examples demonstrate the ability to perform basic GP tasks having been described in the early works [Koz92].

Fourthly, the popularity of each package was assessed by trying to determine which implementation (if any) has been discussed or used in a sample of over 50 papers studied, the GP mailing list archive [GPMAIL] and an informal straw poll conducted on the GP mailing list. Whilst not a rigorous investigation, it indicated whether the GP community regards the packages with confidence. Popularity was rated as high (H) if the package appeared in 5 or more papers or mail threads, medium (M) if found more than twice, otherwise low (L).

Finally, easy access to the tuning options of GP without re-compilation was important to allow semi-automatic tests to be performed. This was especially important given the number of runs that have to be made.

The initial search revealed a number of implementations. These are summarised in Table 4. Implementations that were not found originally have also be included in this list for completeness.

Name	Language	Platforms Supported	Popularity	Passed Tests	Comments and reference
LISP Kernel and problems	LISP	UNIX, Linux, Win95	H	Y	As described in [Koz92] and available from [Koz97] This is subject to U.S. Patents #4,935,877, 5,136,686 (symbolic regression), 5,148,513 (co-evolution), 5,343,554 (automatically defined functions) and 5,390,282. Other patents pending, but is free for academic applications.
Lilgp	C	UNIX	H	Y	A C implementation of the work described in [Koz92], but with many additions. Available from [Lil98]. Further additions made by Sean Luke [Luk97].
Gpc++	C++	UNIX, Linux, Win95	H	Y	A strongly Object Oriented based implementation. Originally by Adam Fraser, but now maintained by Thomas Weinbrenner. [Wei97].
GP-COM	C++, Tcl, Tk	UNIX	L	N	Component based system with GUI. [HB96. Not evaluated.
Gpquick	C++	DOS, Win95, UNIX	M	N	Simple GP system written in C++ by Andy Singleton. And can be found at [Sin94]
GPDATA	C++	UNIX, Linux	M	Y	[Lan97]
Geppetto	?	?	L	N	Written by David Glowacki. Available from [Gep98]
Gpsys	Java	All	L	N	Available from [Qur98].
Gpjpp	Java	All	L	N	A Java implementation of the Gpc++ kernel ([Wei97]). Available from Kok98
SGPC	C	All	H	Y	Simple Genetic Programming in C. Based on Koza's LISP code. Written by Walter Tacket and Avi Cormi. Can be found at [Sgp1998]

Vienna	C++	All	L	N	Not located in time for the initial stud, but included here for completeness. Available from [Vie98]
Gpeist4	Small talk	?	L	N	Not evaluated due to language restriction. Available from [Gpe98].

Table 4 Available implementations of GP systems

Each implementation was obtained and an attempt was made to exercise the two common tests. Some failed because of lack of a suitable platform.

By considering platforms supported, the desired language (C++ followed by C), probable performance and popularity, the Gpc++ package was finally selected for the implementation of the experimental work.

CHAPTER 5. DETAILS OF PROTOTYPE AND OUTCOME OF THE INVESTIGATION

This chapter describes the experimental work carried out in order to address some the issues previously raised.

The main body of the experimental work uses a set of increasingly complex service scenarios based on a simple but complete number translation service. Number translation, also popularly known as 'freephone' or 'premium rate' is the most common service offered by IN platforms [Ebe98]. Further scenarios look at how error conditions and decisions can be handled within the evolving service.

A number of simplifying assumptions were made at the start of the work:

- the number of functions present in the function set was limited in order to concentrate on the essential requirements of services.

- the external world is implemented such that the SDP is integrated with the SCSM to ease the job of the fitness evaluation function. In the real world the two would be separate functions.

- the number of parameters passed in the messages was limited in order to ease the implementation of the fitness function.

These simplifications do not detract from the basic goals of the project and can be eliminated in future work.

The system supports five message types analogous to the real world Intelligent Network Application Part (INAP) and SDF operations. These are summarised in Table 5

Message type	Equivalent INAP/SDF message	Parameters	Comments
IDP	InitialDP	CalledDN and Flag	The message is generated by the SSP as a result of a trigger detection point being activated by a call.
DBREQ	DB_REQUEST[1]	Key	A database request.
DBRESP	DB_RESPONSE[1]	Result and status of request	The response from the database
CONNECT	Connect	Translated Address	An instruction from the SCF to the SSP to connect party A to party B.
END	Pre-arranged end	None	A message issued by either end of a SS7 link to terminate an active session.

Table 5 Messages supported in prototype

5.1. Function and terminal set

The function set ***F*** selected for the initial set of experiments consists of five functions: ***FSTART***, ***FROUTE***, ***FDBREAD***, ***FEND*** and ***STRSUB***.

FSTART takes two arguments. It accepts a message of type IDP from the SCSM. The first argument is then evaluated and the value from the message is stored at the location returned. The second argument is then evaluated and returned.

FDBREAD takes 3 arguments. The first argument is evaluated, and the parameter value passed in a DBREQ message sent to the SCSM. It then accepts a DBRESP message and the second argument is evaluated, and the parameter value from the message is stored at the location returned. Finally, the third argument is evaluated and returned.

[1] GPT proprietary message used for SCF to SDF communication.

FROUTE takes two arguments. The first argument is evaluated and the parameter value used in the Connect message sent to the SCSM. The second argument is then evaluated and returned.

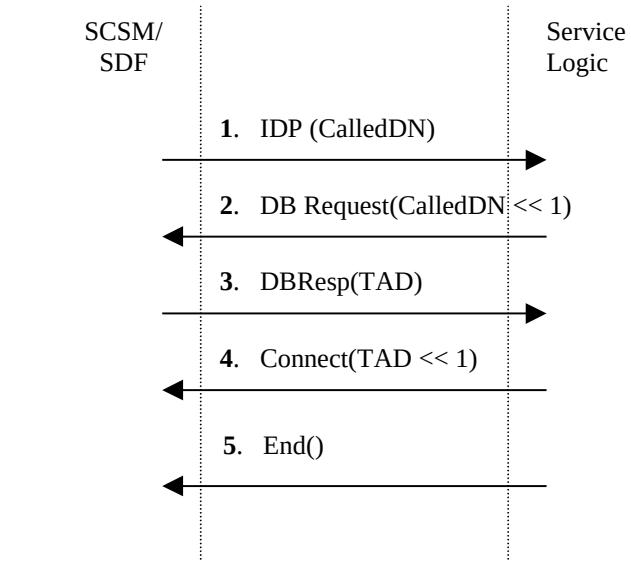
FEND takes one argument, which is evaluated and returned. An END message is sent to the SCSM.

STRSUB was included because real life services require digit string manipulations. It takes two arguments, which are both evaluated. The string value from the first argument is shifted left by one character and stored at the location returned by the second argument. The result of the second argument is returned.

5.2. Experiment 1. Simple Number Translation

This simple initial experiment was devised to discover how well GP could solve a simple problem and to act as a testbed while debugging the system. It was during the development of this experiment that the ideas for APAMS were formed and the implementation put in place.

A simple number translation service is required to translate in incoming number to a new number. Typically this is done by means of using the incoming number (CalledDN) as a key to read the new number from a database, and to issue a connection command to the SSP with the translated address (TAD). A message sequence chart for this service is shown in Figure 6.



NOTE: CalledDN << 1 denotes that the CalledDN string is shifted left by one character, discarding the 1st character. This is used to simulate the real world operational requirement to modify the digit string in some way, for instance to strip any leading zeros from a national number.

Figure 6 Message sequence chart for a simple number translation service

The external operation of this service is as follows:

1. The SSP sends an initial trigger message called the Initial Detection Point (IDP) containing the number (CalledDN) dialled by the user (the A party).

2. The service logic makes a request to the SDF to get the real number to route the call to.

3. The SDF returns the number (TAD) to the service logic.

4. The service logic sends a Connect message to the SSP, causing it to route the caller (A party) to the correct number (B party).

5. The SCP sends an END message to the SSP indicating that the service logic has relinquished control of the call and is no longer concerned with any events generated by the SSP as a result of the call progressing.

From the message sequence chart an external state model can be derived as shown in Figure 7.

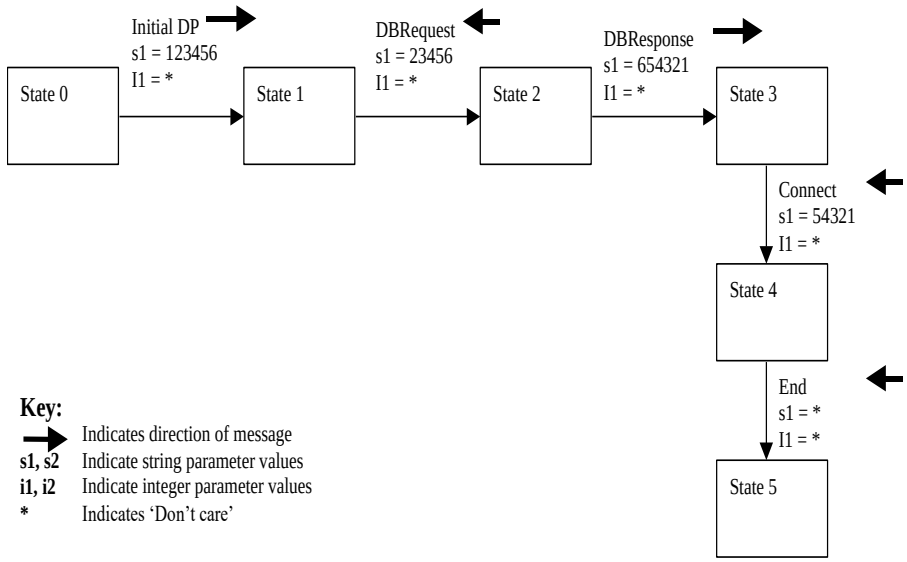


Figure 7 State diagram for simple number translation service

This experiment was evaluated using a range of populations between fifty and five hundred to determine the behaviour for different population sizes. Fifty independent runs were made for each population, and the total number of successful outcomes for each generation was recorded.

Each run was allowed to complete to generation 200, irrespective of whether it had found a 100% fit individual.

The raw performance of this experiment is shown in Table 6. A good run is where a 100% fit individual was produced or formally where $P(M,200) > 0$.

Population M	Number of good runs	Time τ (secs)
50	12	105
100	22	326
150	35	476
200	42	783
250	42	971
300	49	1159
350	48	1382
400	50	1749
450	48	1937
500	50	2140

Table 6 Successful outcomes vs population size for simple number translation

Two points can be deduced from these results. Firstly, even for small population sizes, a significant number of successful and therefore useful programs were generated. Secondly, the running time is roughly proportional to the population size.

The raw data was then processed to show the probabilities of success and the number of independent runs required. This is shown graphically in Figure 8 for a population size $\boldsymbol{M}$ of 500. Note that $R(z)$ is proportional to the effort required from equation (2). The performance summary is shown in Table 7.

Since for a population of 500, there was an 80% probability of success, this population size was used for all subsequent experiments.

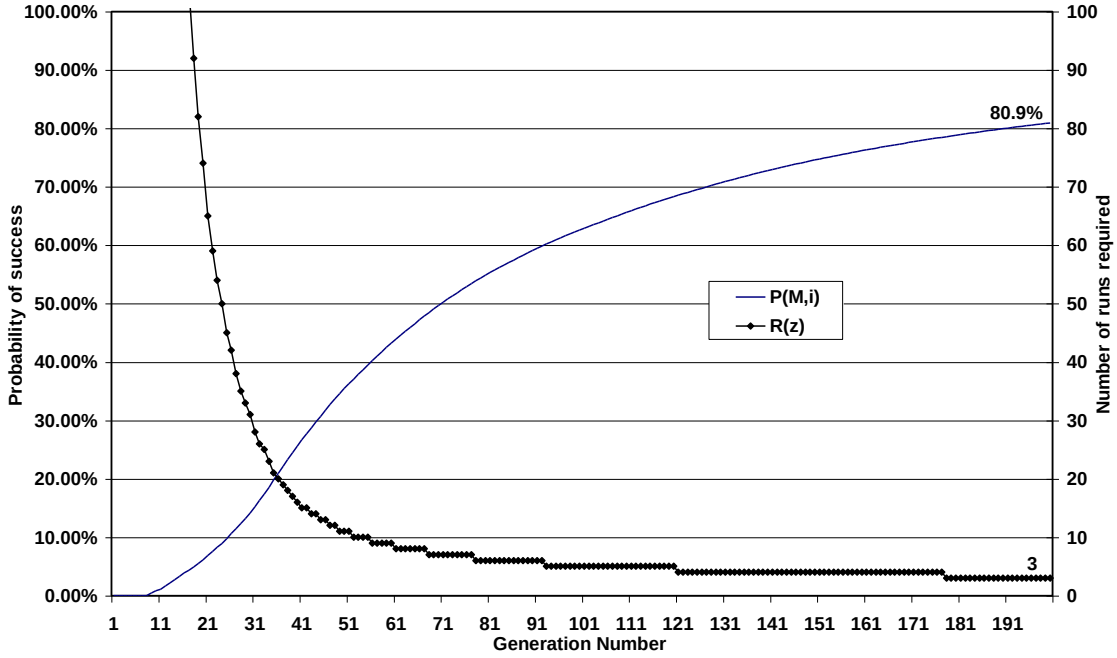


Figure 8 Performance of simple number translation for **M**=500

Effort ε	**Number successful at generation 200 ϑ**	**Number of different 100% fit programs ψ**
1,500	50	50

Table 7 Summary of performance for simple number translation

5.3. Experiment 2. Complex Number Translation

The purpose of this experiment was to observe how more complex external behaviour affected the GP process in terms of processing required to solve the problem. Some additional results are also presented to illustrate the behaviour of the GP system and to explore some of the operational issues of GP.

This is an extension of the simple case with an additional database request, and additional variable manipulation requirements. This scenario occurs in the real world where a service requries two items of data in order to route a call. For example, a service may need to route to one number during working hours and

another number during out of work hours. The first database request in this example represents the query that determines a time based key for the subsequent request. Again a message sequence chart and state diagram are shown in Figure 9 and Figure 10

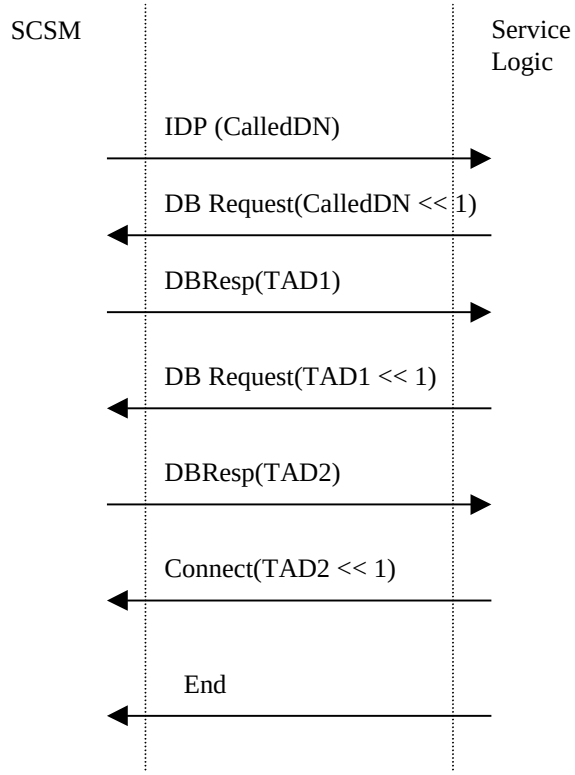


Figure 9 Message sequence chart for an extended number translation service

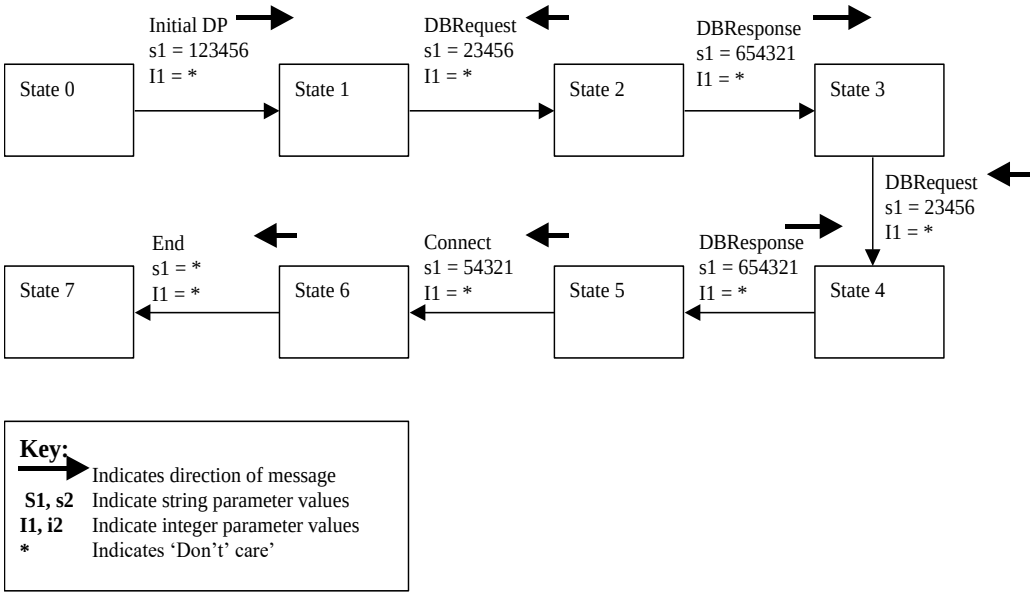


Figure 10 State diagram for extended number translation

The performance of this experiment is shown in Figure 11 and the performance summary in Table 8.

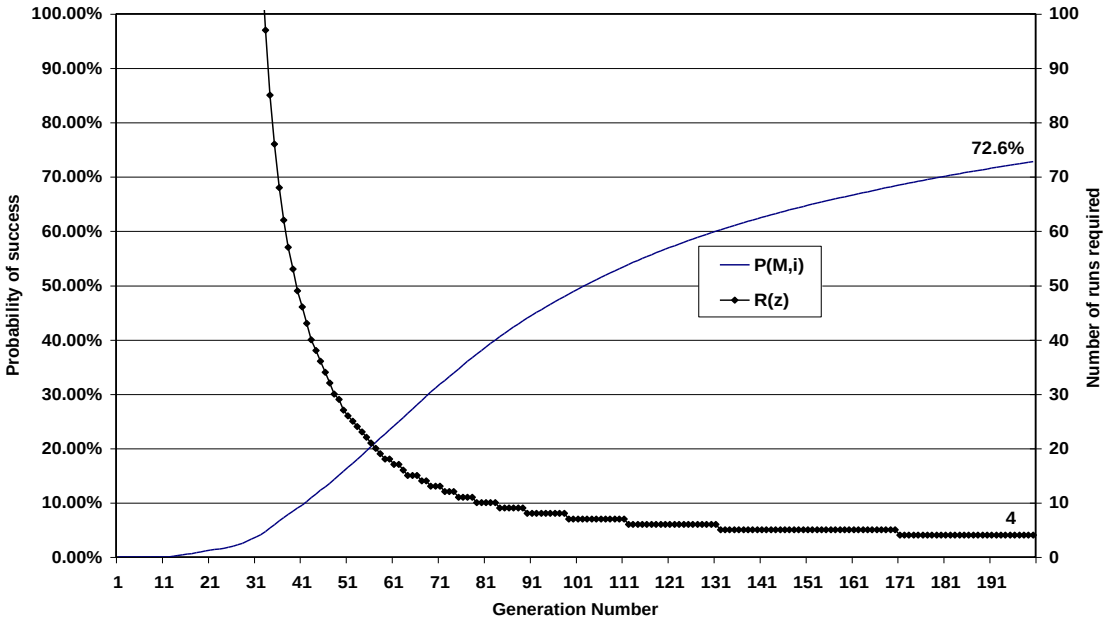


Figure 11 Performance of complex number translation for **M**=500

Effort ε	Number successful at generation 200 ϑ	Number of different 100% fit programs ψ
2,000	49	49

Table 8 Summary of performance for complex number translation

The main conclusion that can be drawn from the above is that in comparison with the simple case, more processing effort was required.

Some sample 100% correct programs from this experiment are shown below. These were taken from the run when ***M***=500, from runs 1, 9 and 13.

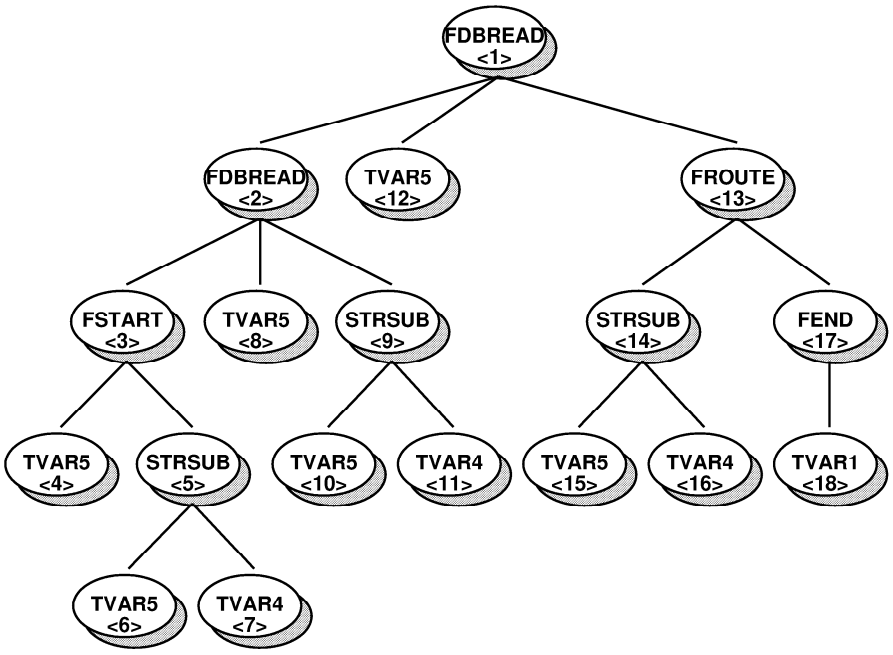


Figure 12 Example program tree for complex number translation – 1

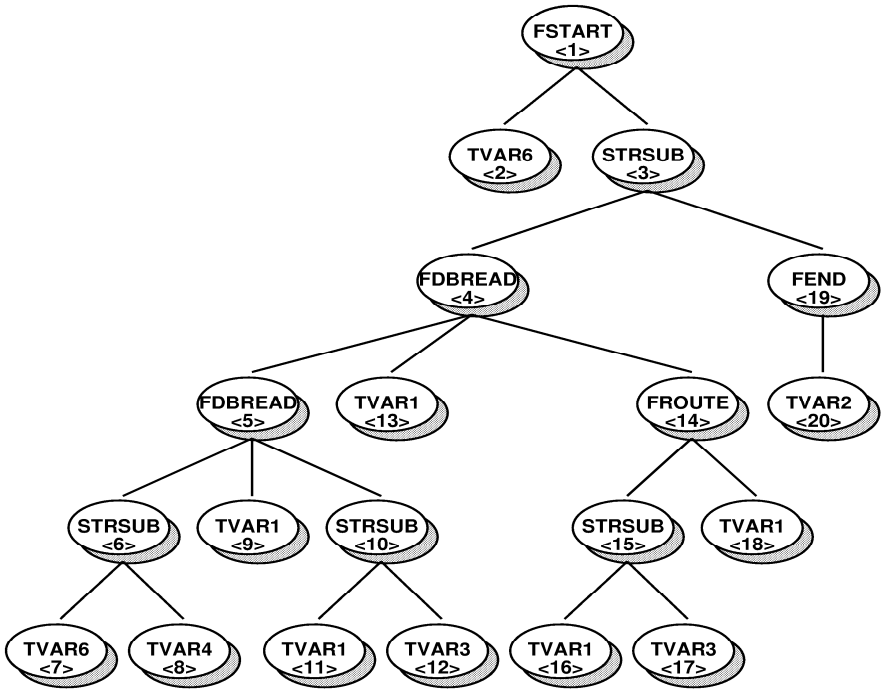


Figure 13 Example program tree for complex number translation – 2

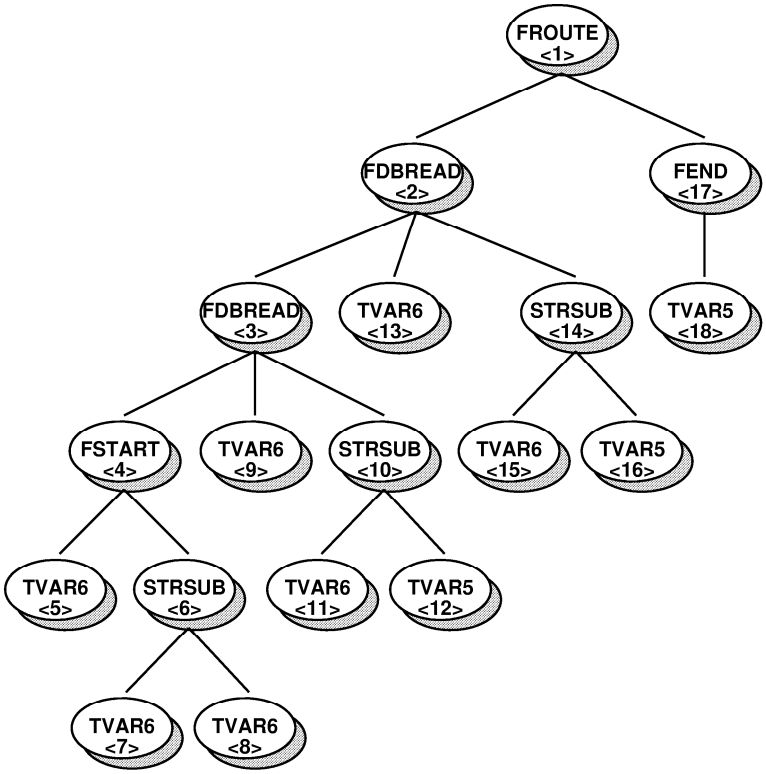


Figure 14 Example program tree for complex number translation – 3

From these three examples, the potential variety of solutions that appear can be seen. It is possible that a human programmer would have come up with that shown in Figure 13, since this begins with the ***FSTART*** operation, followed in sequence by the other operations, although using the side effect of node 5 is not intuitive. It is unlikely that a human would have started with the ***FROUTE*** operation. The variety can also be seen from the fact that the 49 successful runs produced 49 different solutions.

It is also interesting to note the presence of unproductive nodes. For instance, in Figure 14, node <10> performs an ***STRSUB*** between ***TVAR6*** and ***TVAR5.*** ***TVAR5*** is not used again, until it is overwritten with the result of the string manipulation at node 14, therefore node <10> is a redundant non result affecting node.

The presence of these unproductive nodes – called introns – is part of the evolutionary process. It has been argued by Nordin [NFB96] that the presence of introns can help the evolutionary process by isolating productive sequences from other sequences, and thus preserving them for future generations. The presence of these introns can be seen by observing the change in population size of a totally fit individual while running the GP system.

The presence of introns during evolution is shown in Figure 15 for M=500, run number 24. The plot shows the normalised fitness (§4.2) and tree size of the fittest individual for each generation. It is important to note that this does not show any one particular individual throughout the run.

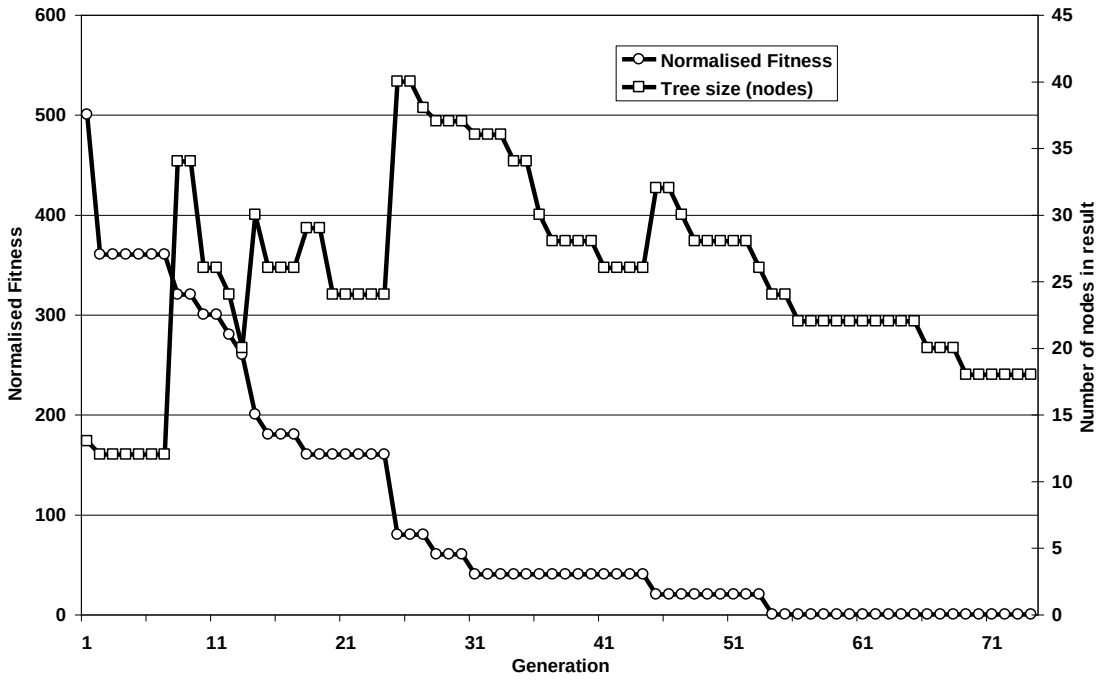


Figure 15 Size of fittest individual with generation

As the fitness improved, the size of the fittest individual tended to increase initially presumably due to the effects of crossover introducing new material to the fittest individual, or due to other fitter individuals being produced. It can also be seen that the size increased at the same point that an improvement was made to the fittest individual. However additional pressure to evolve parsimonious individuals resulted

in each stable case decreasing in size. This is true even for the 100% fit individual that appeared at generation 53.

The problem presented here has two distinct measures for fitness:

1. Whether states are handled correctly

2. Whether the message values returned from the program are correct.

It is interesting to observe the rates at which the system can find totally fit solutions. To do this, the number of successful and unsuccessful state transitions and messages was recorded by the SCSM for each generation.

Figure 16 shows the progression of the states for M=500, run number 19, and Figure 17 shows the message fitness progression.

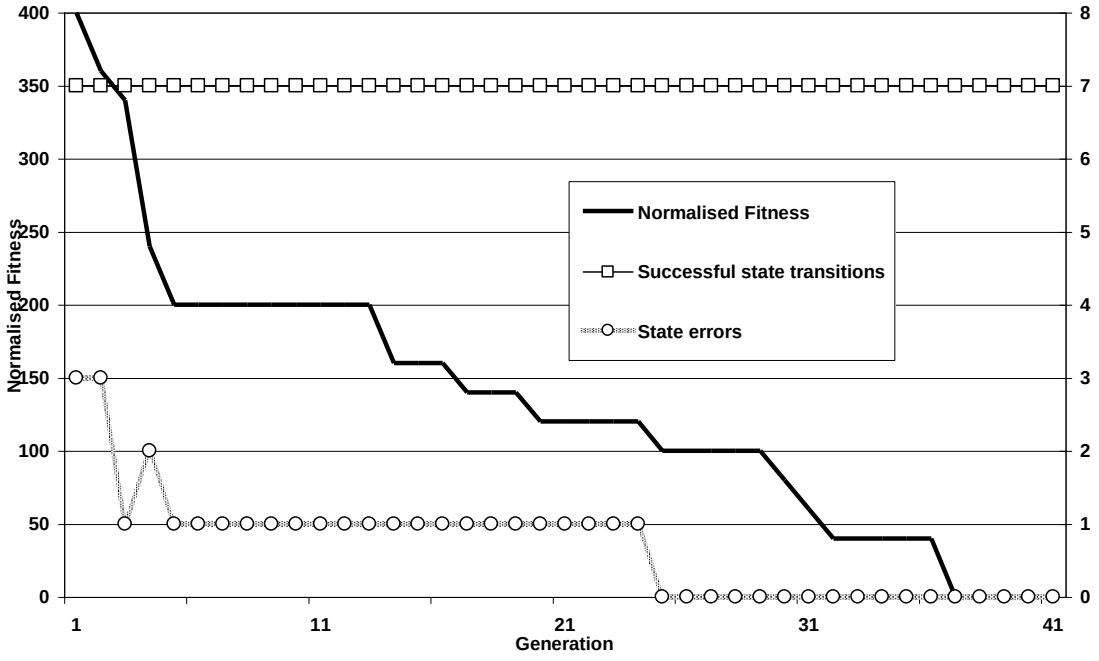


Figure 16 Progression of state fitness evolution

This shows that the state handling is quickly evolved, with 100% of the states correctly handled by the first generation. However, there are still a number of extra incorrect states present until generation 25. This can be attributed to the fact that bad states incur a penalty of 20, while good states are rewarded with a value of 100.

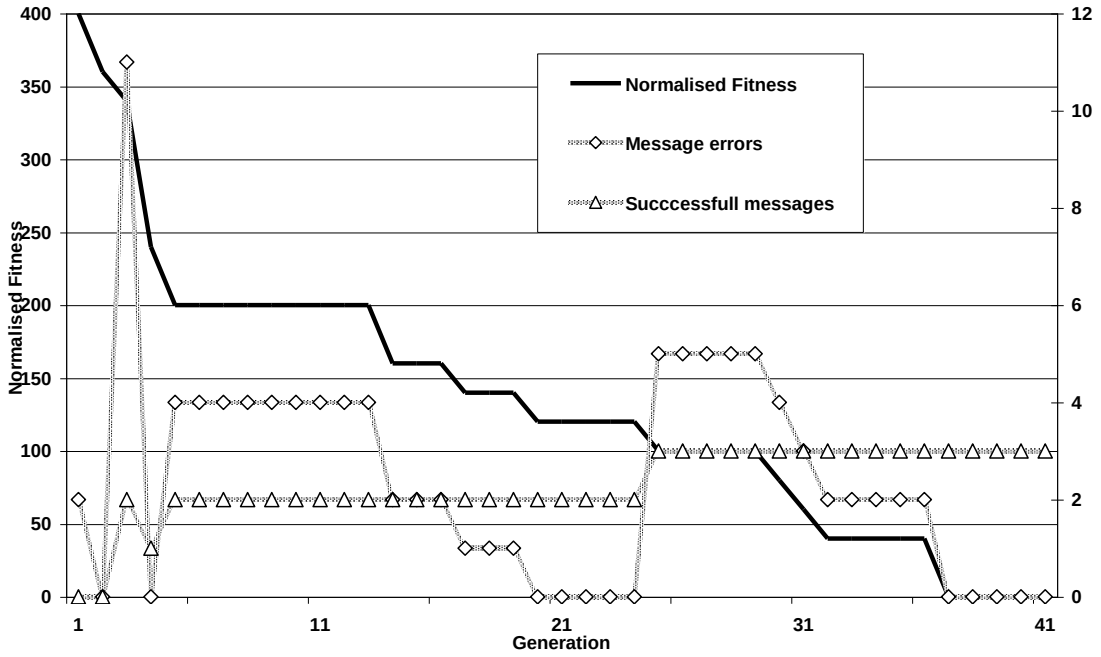


Figure 17 Progression of message fitness evolution

By comparing Figure 16 and Figure 17 it can be seen that message fitness lags behind the state fitness. This is because a correct message cannot be delivered until the correct state handler is in place

The implication of this behaviour is that if there are fitness measures (as in message parameter values) that are completely dependent on other fitness measures (such as the message ordering) , the effort required to evolve solutions increases. . There is therefore effectively a hierachy of fitness and there is probably a practical limit to the depth of such a hierachy.

5.4. Experiment 3. Run-time decision making – simple case

The cases studied so far require a linear sequence of message exchanges and the correct data passed with those messages, but in real life systems exceptions occur which must be handled. Additionally, services often make decisions based on the current state, user inputs, database values or environmental factors such as the time or date.

This experiment was designed to discover if logic could evolve to handle these cases.

The simple case requires the service logic to return one of two numbers depending on the value of a flag that is passed into the service from the SSP. Such a flag may indicate that a particular caller is denied access to parts of a service.

The message sequence is shown in Figure 18 and the state diagram in Figure 19

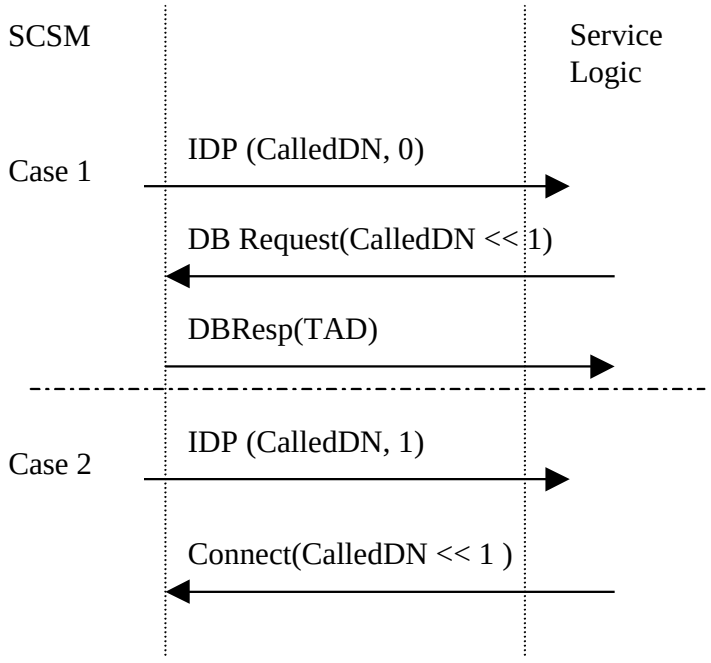


Figure 18 Message sequence chart for early run time decision making

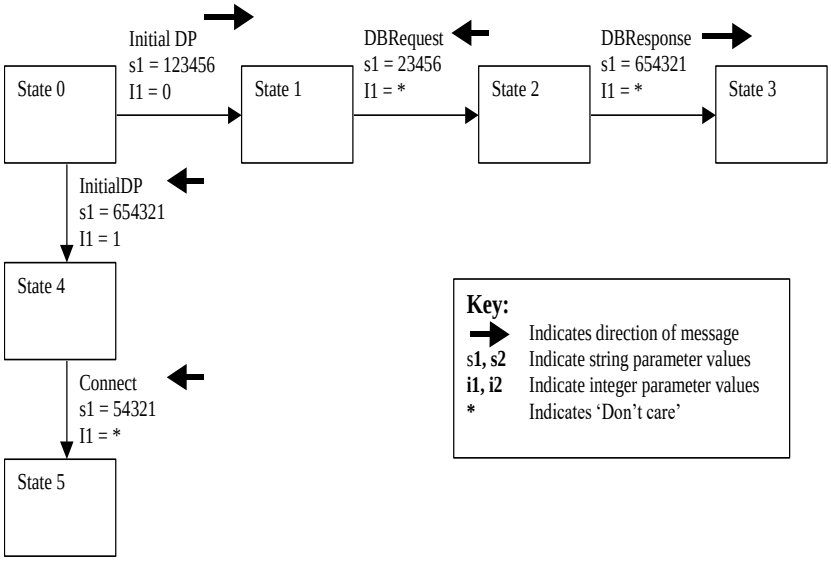


Figure 19 State diagram of early run time decision making

To handle run time decisions, two changes are required to the system previously described:

1. The fitness function must expose the GP to all the cases.

2. An additional test function needs to be added to the existing function set.

Two methods of modeling this behaviour suggest themselves. Firstly, a probabilistic or sampling model that only subjects each individual to a subset of possible sequences. Secondly, a deterministic approach that models the complete set of behaviour required by any correct solution. These two alternatives are examined from a theoretical point of view, and then some experimental results are presented.

The probabilistic method requires only a single fitness run for each individual, the fitness case being selected on a weighted random basis from all possible fitness cases. The obvious attraction is that the number of fitness tests could be less that of the deterministic method. However, there are several difficulties with this method. Firstly there is a danger of losing useful genetic material and finding a poor local

minimum if an individual should score poorly for the normal case, but in fact contain a good solution for the error case. Secondly, if the error cases are only evaluated on average according to their probability of occurring, then the error cases will not have as much exposure to the normal evolutionary effects as the normal case. If this were to happen, then the normal cases would likely evolve at a rate similar to the single case, while the error cases would require additional computational effort to yield a highly fit individual.

The third difficulty is more fundamental in that to identify 100% fit individuals, all paths must be traversed at some point. If this is done for each generation then this degenerates into the deterministic method described next, or it must be done at prescribed points, for instance after a given number of generations. The question then arises as to what the real fitness criteria is. Is it the partial result from probabilistically selecting a subset of all paths, or is or is it the complete set of path traversals?

Because of this fundamental problem, this approach was not pursued any further. An alternative is suggested by Gathercole [GR94] and [GR97] in adaptively modifying how many fitness cases should be used with a technique called Dynamic Subset Selection (DSS). This may help in future work in this area.

When considering the alternative deterministic method we are concerned with achieving a full coverage of all possible sequences of events and messages. This would ensure that all cases have an equal chance of evolving correctly. The major disadvantage, at least for large problems, is the time required to fully evaluate each case, since each individual must be subjected to all possible cases.

The number of fitness tests is proportional to the number of paths in the problem ***N***. If all possible paths are the same length, then the net result on the time required to find a solution would be at least ***N*** times the time required for the case where a single thread of control existed. The real problem is that the number of paths is likely to increase exponentially in the number of nodes that must handle error cases.

For the purposes of simplifying the prototype work the number of paths was restricted to two. More work would be required to allow arbitrary numbers of paths to be evaluated.

The new function added to the function set ***F*** is the equality test ***FEQ***. This takes three arguments and operates as follows:

The first argument is evaluated and if the integer portion of the result is equal to 1, then the second argument is evaluated, else the third argument is evaluated. The result of the final argument evaluated is returned.

The performance of the experiment is shown in Figure 20 and the performance summary in Table 9.

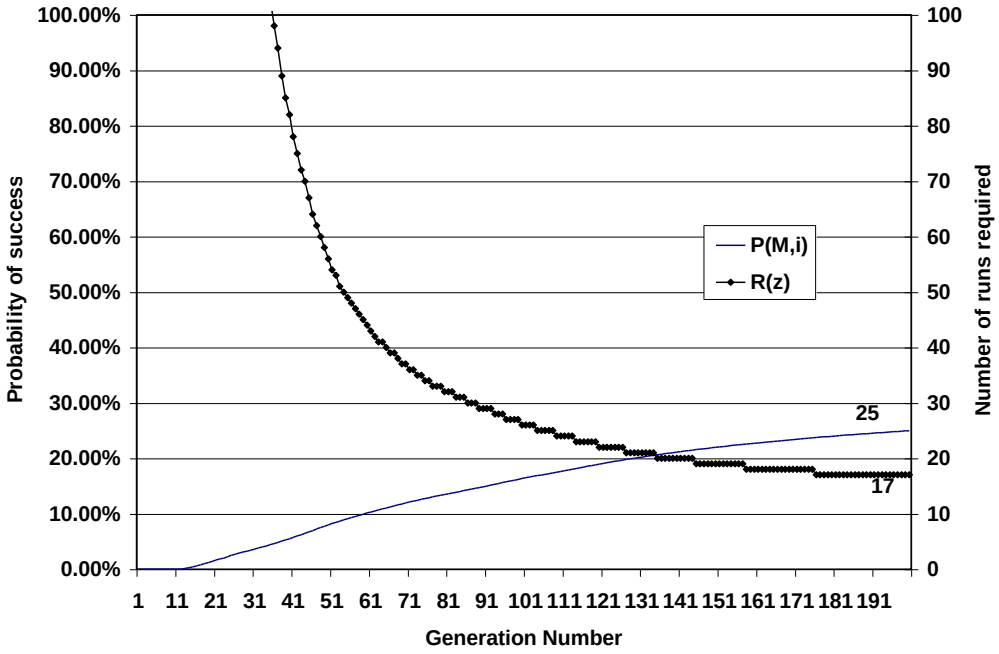


Figure 20 Performance of early decision making for **M**=500

Effort ε	Number successful at generation 200 ϑ	Number of different 100% fit programs ψ
8,500	17	17

Table 9 Summary of performance for early decision making

5.5. Experiment 4. Run-time decision making – more complex case

The previous case involved the service in making a decision early in its execution tree. This case involves a making a decision later in the tree where, for instance, the database cannot find a record corresponding to the key. Other failures such as an internal database error or a communications failure are also covered by this example. The decision point was moved to see what effect on the evolutionary process was of delaying the point at which the branch was required.

The response from the database contains an additional parameter, in this case used to indicate success or failure. It is an integer value that takes the value 0 indicating normal operation, or 1 indicating an error condition. The error condition results in the value returned by the error response being sent as the translated address.

This case uses the same function set as the previous experiment, and uses the full coverage (deterministic) method of fitness evaluation.

The message sequence chart is shown in Figure 21 and the associated state diagram in Figure 22.

The performance of this experiment is shown in Figure 23 and the summary in Table 10.

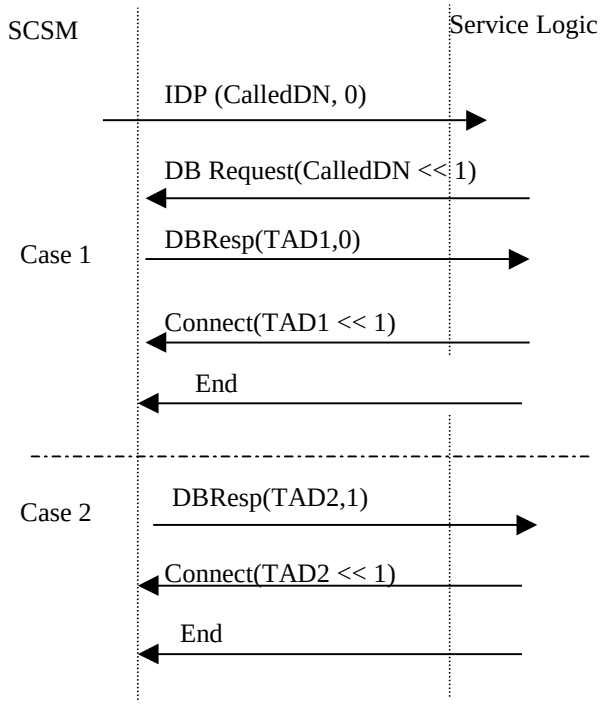


Figure 21 Message sequence chart for late decision experiments

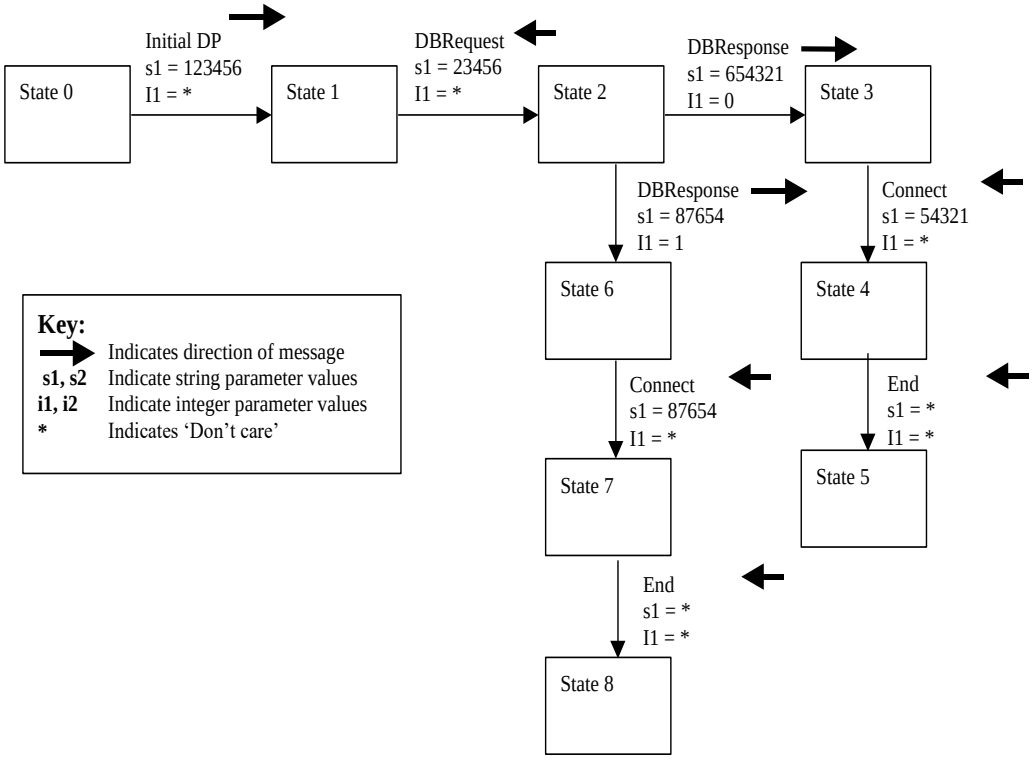


Figure 22 State diagram for late decision experiment

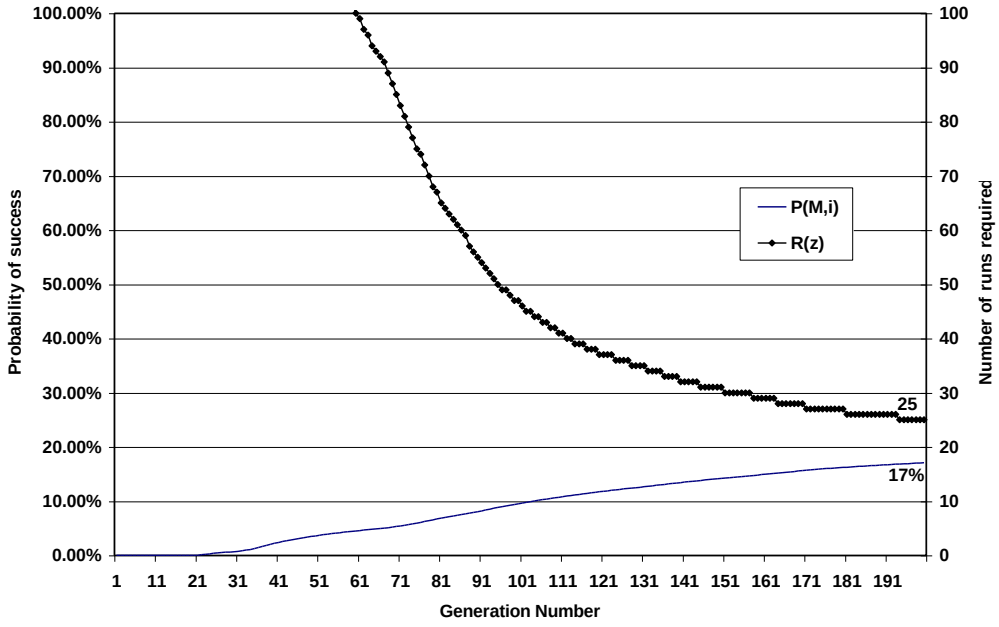


Figure 23 Performance of late decision making for **M**=500

Effort ε	Number successful at generation 200 ϑ	Number of different 100% fit programs ψ
12,500	11	11

Table 10 Summary of performance for late decision making

This graph clearly shows the inferior performance of this experiment using the standard value of M=500, the effort being much greater than 2 * ε for experiment 1. Obviously other factors come into play when trying to evolve such a program.

5.6. Experiment 5. Reduced Complexity Function Set

Earlier the choice of function set was discussed (sect. 4.1). This experiment was devised to give an indication of whether the original level of abstraction was reasonable or whether by using a lower level of abstraction in the function set better performance could be achieved. A literature search failed to find any detailed discussion of this aspect of GP. Most problems discussed in the literature deal in small problems whose function and terminal set are fairly obvious.

The input for this experiment is identical to the complex number translation service described in section 5.3. This case was chosen since it was the first experiment that had a value of ϑ of < 50 and was therefore seen as not trivial.

The high level functions ***FSTART, FDBREAD, FROUTE*** and ***FEND*** were removed and two new functions ***ReadMSG*** and ***SendMSG*** were added.

- ***ReadMSG*** accepts an incoming message and places the parameters into memory locations as provided by the arguments to this function. In this case, only one parameter is accepted.

- ***SendMSG*** constructs a message containing a message type and a single parameter value.

In §4.1 it was indicated that the lower the level of abstraction the more functions would be needed. The fact that there are two fewer functions for this experiment compared to §5.3 is explained by the fact that the higher level funcitons FSTART etc are synthesised from these lower level functions.

Lastly, the memory cells were extended to contain a message type, enumerated over the range of message types required.

The overall performance of the system when using lower level functions is shown in Figure 24 with the performance summary in Table 11.

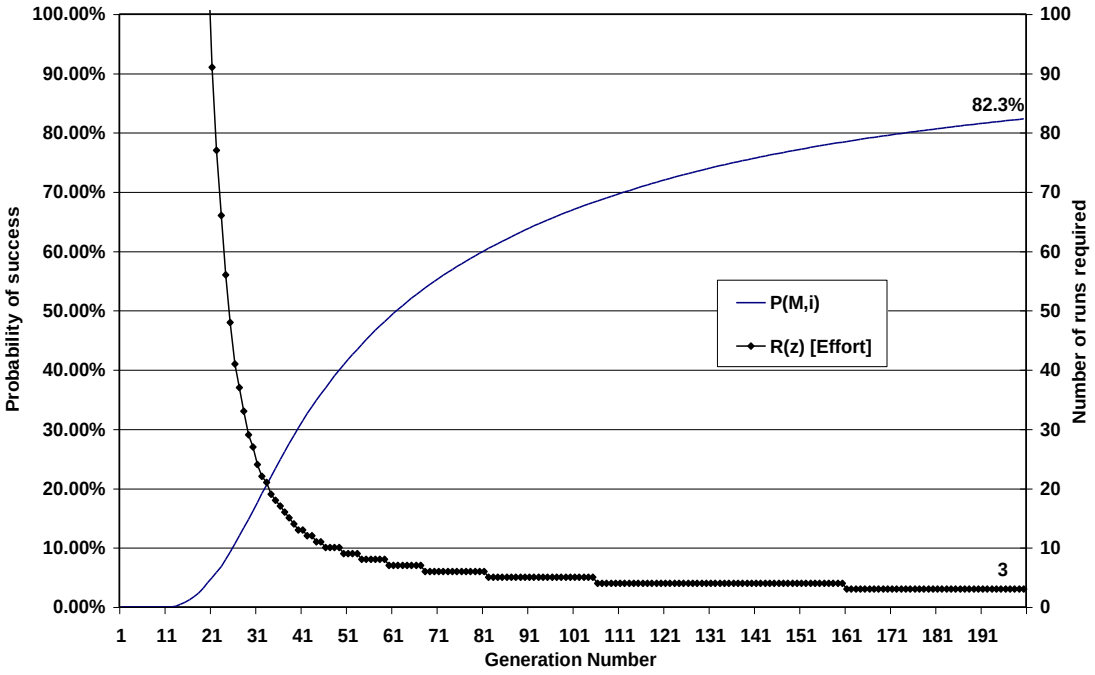


Figure 24 Performance using reduced complexity functions for **M**=500

Effort ε	**Number successful at generation 200 ϑ**	**Number of different 100% fit programs ψ**
1,500	49	49

Table 11 Summary of performance using reduced complexity functions

It is interesting to note the performance in comparison to that using higher level functions in Figure 11. The effort curve *R(z)* reaches the value 10 at an earlier point and the probability curve is also steeper. The disadvantage of this method though is the more CPU time required to process each run. This is shown in Table 12 and may be an important factor when considering the scalability of GP.

As an example of the difference in output, an example from this experiment, run 1 is shown in Figure 25.

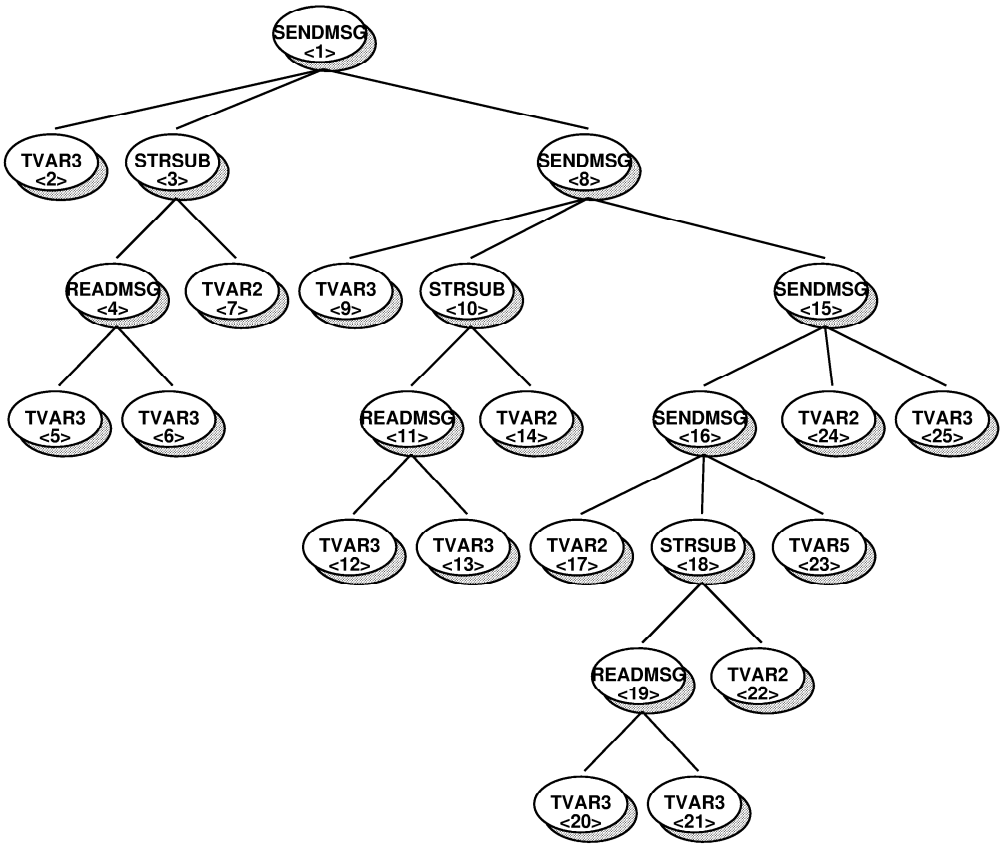


Figure 25 Example program tree using reduced complexity functions

An interesting feature of this particular example is the regularity with which the pattern at nodes 3, 4, 5, 6 and 7 occur. This pattern is repeated at the subtrees rooted at nodes 10 and 18. It is likely that using Automatically Defined Functions

(ADFs) [Koz94] for this level of functions would be beneficial since there are repeating patters emerging.

5.7. Summary of experiment results

In order to be able to estimate the difficulty of any problem, some means of expressing the problem in its constituent parts is required. This section summarises the experiments in terms of the problem complexity, and the corresponding results.

In all cases, the results are for a population M=500, and the number of generations **I**=200.

The average time for a run to complete is taken as the total wall clock time of the experiment divided by the number of runs, which was 50 in each case.

Input requirements				Output results				
Experiment Number	**Input States**	**Parameters**	**Paths**	**Average time per run (secs)**	**Average Complexity of fittest**	**P(M,i) %**	**R(z)**	**ε**
1	6	2	1	42	13	81	3	1,500
2	8	3	1	44	19	72	4	2,000
3	6	2	2	117	21	25	17	8,500
4	9	3	2	124	32	17	25	12,500
5	8	3	1	71	28	82	3	1,500

Table 12 Summary of experiments and results

The average complexity is the sum of the complexity values of the fittest 100% correct individuals in each run, divided by the number of runs that produced a 100% correct individual. The complexity of an individual is the number of nodes in that individual.

The data presented is not the only data pertaining to the experiments but is an initial attempt to try to identify any useful patterns that may exist in the system.

CHAPTER 6. ANALYSIS

This chapter analyses the results from chapter 5 and attempts to answer some of the questions raised previously.

The results presented in experiment 2 show that even for small problems, there is more than one program that satisfies the problem statement. This can be argued as true program induction taking place, since any mechanical translation or mapping of the input specification would result in the same program every time. Surprisingly, every experiment resulted in 100% diversity in the solutions found giving some indication of the size of the solution space, even when using a small number of nodes. This result is important since it confirms the idea that for any given problem statement there is a very large number of possible programs that satisfy the problem statement.

It is clear from the experiments, that as the problems get 'harder' the longer a solution will take to be found, and the greater the population size or the greater the number of attempts required to find a solution. Unfortunately, there is no standard measure of difficulty in the current GP literature. This is a problem when trying to determine the settings to use in systems to get the best results.

In the context of the work described in this paper, the difficulty of any problem is related to a number of factors:

1. The number of states in the input requirements

2. The number of messages it has to handle

3. The number and position of decision points required

Handling decision points required much more computational effort to find a correct program. In a real service, there would be many such decision points, and it is not clear how well this approach can scale to accommodate this requirement.

Much of the GP literature is concerned with solving problems that have no 100% correct solution. In the problem domain of IN however it is not sensible to consider anything but a 100% correct program. Barring operating system and hardware faults, it is expected that a service will operate correctly for all customers all the time.

This has two effects on using GP:

- The evaluation of fitness is simpler since there is no doubt as to whether a program is successful or not

- It requires that the fitness cases cover 100% of the possibilities. In the problems considered here, this is not an issue, but when there are many decision points, the number of fitness cases increases greatly. This obviously has an effect on the time required to complete a run.

During the early part of the work, considerable time was spent trying different combinations of the control parameters. The set arrived at for the experiments (APPENDIX C) is probably not optimal.

Two questions arise from this:

1. Is there an envelope of operation that gives good results?

2. Is it possible to determine all environment control values by some method?

It should also be noted, that although studies into different control parameter values has some measurable effect on particular problems the scale of effect is often small, and the universality of the effect is often limited, as for instance reported by Goldberg [GKH95] in his study on deme size, and the results presented as part of the GP kernel [Wei97]. These and other questions raises the point made by Goldberg [GO98] that unlike GA there is no good theoretical basis for GP, and that until one is developed we are reliant on empirical methods for determining the operational parameters for GP.

The use of APAMS was very powerful. It meant that evolving programs were not constrained in the shape they took. The memory locations were used for several different purposes in the experiments – targets for storing message parameters, both string and integer, and a source for function arguments, and as a constant value as when used by some examples using the ***FEQ*** function. In the last experiment they also contained message types. Extension of APAMS would prove beneficial in future developments such as using it to hold partial or complete messages.

APAMS also contributed to the great variety seen in the 100% correct solutions by avoiding the need to restrict the semantic structure as in [HWSS95], [Mon95], [CY97] and others. To examine this claim, a simple hypothetical case can be considered, such as the ***FSTART*** function. A strict typing of this by a human programmer during the early stages of building a GP system could define this function returning a status, or particular parameter to a calling function and having arguments of type DialledNumber for the first and some other type for the second. Immediately it can be seen that by adding these constraints, a human programmer imposes their own perceived structure on the function and therefore it's place in any tree. Doing this would preclude two out the three solutions illustrated in section 5.3

The work originally by Koza [Koz92] used LISP as the implementation language. This was attractive in one sense in that the programs that were evolved were LISP s-expressions, and could be executed by the run-time environment without any external translation. Using C++ means that the structures being evolved cannot be used directly as programs and an additional stage needs to be added if the output is to be used in any practical application. This does have one big advantage however, in that the structures can be viewed as high level languages and are therefore amenable to mechanical translation and optimisation, for example using the techniques described in [Ben96] and [Hlb90].

The original choice of abstraction for the internal nodes gave satisfactory results, but as shown in experiment 5, a lower level of abstraction gives a better overall performance (higher probability of yielding a 100% correct program) using the same basic system architecture, but required approximately 40% more processing effort. Interestingly the average complexity of the reduced complexity experiment was also approximately 40% greater than the standard experiment. This suggests there may be a direct link between the two measures. Additionally, it is suggested that using ADFs could well be useful in this case. Clearly more work is required in order to arrive at an optimal level of abstraction.

The initial choice of the function set and terminal set had some interesting properties. Firstly, most of the external behaviour was determined by the state affecting functions ***FSTART***, ***FDBREAD, FROUTE*** and ***FEND***. The only non state affecting function ***STRSUB*** and in later experiments ***FEQ*** had the ability to appear in a program in one of two modes:

1. Result affecting

2. Non result affecting.

The latter mode introduced introns into the program, allowing it to evolve via more than one route. Clearly, the state affecting functions could not operate as introns. In any subsequent work, it would be useful to observe the effect of adding more functions that can operate in both modes. The use of Explicitly Defined Introns (EDI) as described by Nordin [NFB96] has the potential to improve the performance of GP, but it has also been reported by Blickle [Bli96] and Andre [AT96] that EDI can degrade the performance in some applications. The utility of EDI appears to be problem specific.

A question that arises when this type of system is discussed is the degree of confidence with which the result can be trusted. There is a perception that a program created by human is somehow more trustworthy than one created by a machine. This perception is not helped when looking at large, apparently

unstructured programs generated by GP systems. The flaw in this perception is that for all programs whether created by a human or by some mechanical means the final arbiter of correctness is the behaviour of the program, or more properly does the program behave as the specification requires it to? Consider a common example of a simple C program such as shown in Figure 26:

```
main ()
{
   int a= 10;
   int b= 20;
   printf("Sum = %d\n", a + b);
}
```

Figure 26 Simple C program

While the program may be obvious to a human reader, the output from an optimising compiler would be hard to follow, and the execution ordering of the machine instructions in a modern RISC chip could be understood by only the most knowledgeable of engineers, yet our experience give us confidence that the program will work correctly.

The opaqueness of machine generated programs can of course be considered to be a positive attribute in that it forces the systems engineer to look more closely at the specification and the associated system testing. A consequence of this is that the systems engineer must specify *exactly* what the system should do, not as the introduction to Koza's third book [KABK98] states '… a high level statement of the requirements …'.

This question concerning the opaqueness of programs generated using GP or other EC technique has inspired some work to try to address the perceived deficiency. For instance Pringle [Pri95] suggests an approach that tries to create programs that look like those produced by a human programmer, while Langdon [Lan98] has dedicated a whole book to automatic programming adopting techniques used by human programmers as building blocks. A potential flaw in this approach is that practices such as modularity, data hiding, object oriented disciplines, data structures and other 'good engineering practices' have been developed to aid human

programmers in writing fault free and maintainable software. They are not of themselves required for a program to be correct and while the aforementioned work has delivered some useful techniques and insights it does not address any of the essential features of GP. A counter argument has been made by Blickle [Bli96] pointing out that a clear structured program can give valuable insights into the problem being solved. For example when trying to find an analytical expression to difficult integral equations, a clear analytic expression would allow further investigation of the problem. However it is worth revisting the original inspiration for this work and noting that Darwin observed 'nature cares nothing for appearances, except so far as they may be useful to any being' [Dar1859] (Chapter IV, 'Natural Selection').

Lastly the question of whether GP can perform as well as or better than a human programmer needs to be considered. In section 2.1 it was claimed that GP would only be worthwhile if it could generate an implementation in a shorter time and with fewer defects that a human. The problems considered in this paper have been trivial compared to those encountered in existing IN systems, and comparing these results directly with a human is not a reliable indication of scalability. However an indication that GP is at least as good as a human for simple services can be seen when ad-hoc experiments with a few engineers show that a simple number translation service requires less than an hour of effort to complete using INventor. This compares well with the results in Table 12. If the first correct program was chosen, then the time required by GP can be measured in minutes. Clearly further work is required to explore this issue for complex services.

CHAPTER 7. AREAS FOR FURTHER WORK

This chapter describes some further work that will extend the utility of the work presented so far and will provide answers to some of the harder questions raised.

Some areas for further work have already been pointed out. The main area for study is how to model more complex services, with an arbitrary number of paths. From the experimental results it is apparent that a monolithic approach would fail or at least be very time consuming for fairly small numbers of paths. Partitioning the problem would be a useful technique to consider when extending the system. Alternatively, it may turn out that ADFs would give useful benefits for complex problems. This is a project requiring several engineer months of effort.

Another area for future work involves developing robust interfaces to the system to enable it to operate in a commercial environment. This involves adding a requirements specification interface at the front end and completing the back end program generation to yield usable programs. This would complete the feasibility of using this approach. Use of the techniques described by Bennett [BM98] may be of use. This is project requiring perhaps an engineer year.

Considering the theoretical aspects of using GP, a means of selecting the best set of parameters, possibly based on GA techniques would remove this burden from the user. Certainly, this is needed if GP is to have any utility outside of a research context. Some work has already been done to look at adaptive mechanisms by Angeline [AK96], Gathercole [GR97], Teller [TA97] and others but these tend to focus on the internal operation of GP. This is a project requring several engineer months.

Whether GP is highly scalable and therefore whether it can help in realising complex new services is still an open question. In order to answer this, a more detailed understanding of the computational effort required to create such services is needed, and what, if any, limitations there are on the complexity of the service

requirements. The experiments presented in this paper show that as the complexity of the requirements increases, so does the computational effort required to solve the problem. With the further work described above it should be feasible to explore large services and therefore gain a better insight into this question.

CHAPTER 8. CONCLUSIONS

This work has extended the application of Genetic Programming by demonstrating that it can be used to generate service logic for an Intelligent Network application, namely Number Translation.

Once the system had been set-up, the elapsed time required to create the service logic program was several orders of magnitude less than using the existing manual toolkits available for simple services, thereby potentially reducing the time required to create IN services.

The level of defects in the generated application due to implementation errors is zero due to the fitness evaluation applied to the application. The level of defects due to errors in requirements should be reduced since more attention is needed at the specification stage.

The scalability of GP is still not well characterised and work is required to address this area further. The work presented uses a new and novel memory architecture, which removes the need for strict typing. In contrast to strict typing, there is no such thing as a syntactically incorrect program, which leaves the GP system free to evolve a semantically correct solution.

APPENDIX A. HARDWARE AND SOFTWARE CONFIGURATION

A.1. Hardware

The hardware used for generating the results presented and performing the experiments consisted of:

- PC computer with:
  - AMD K6 CPU running at 200MHz.
  - 512Kbytes of cache
  - 32Mbytes of main memory.

A.2. Software

Operating system

The software environment consisted of the Linux operating system, kernel version 2.0.0 from the Slackware version 3.0 distribution.

Compiler

The GNU C++ compiler gcc 2.7.2 was used to compile the software. All the runs were run performed using optimisation level 3.

Other tools

Gpc++ version 0.5.2 [Wei97]was used to construct the GP system.

Tcl version 7.6, Tk version 4.2 [Ous94] and the Tree package [Bri97] version 4.2 were used to create the parse tree diagrams.

PERL version 5.003 was used to write the scripts used to drive the experiments.

GSView version 2.4 was used to prepare the parse tree diagrams for inclusion into this paper.

Microsoft Excel 97 was used to prepare the performance graphs.

APPENDIX B. GLOSSARY

This appendix explains the abbreviations specific to telecommunications and Intelligent Networks.

DTMF *Dual Tone Multi Frequency*. A dual audio tone signalling method used by telephone instruments to indicate to the switch the digits 0-9 and the * and # symbols.

ICON A graphical abstraction of a building block used in the GPT GAIN INventor™ product.

IN. *Intelligent Networks*. The use of standard computing platforms to extend the functionality of traditional telephone networks.

INAP *Intelligent Network Application Part*. The standard protocol defined to allow the SSP and SCP to communicate with each other.

SCP *Service Control Point*. The Intelligent network node that contains the service logic programs.

SDP *Service Data Point*. This function provides traditional database support for the service logic.

SS7 *Signalling System number 7*. An internationally agreed standard for carrying signalling information between nodes in a telephony network.

SSP *Service Switching Point*. The traditional Stored Program switch that contains the IN trigger and message functions.

SMP *Service Management Point*. A network and customer management system.

TCAP *Transaction Capability Part*. The transaction layer in the SS7 stack.

APPENDIX C. RUN TIME PARAMETER VALUES

Within the GP Kernel there are a number of tuneable parameters. In order to gain an insight into the performance of GP for different problems without introducing other variables, the variable factors were kept the same throughout all the experiments and are listed here for reference.

Parameter	**Value**	**Comments**
PopulationSize	10	
NumberOfGenerations	200	
CreationType	2	2 = ramped half and half
CrossoverProbability	100	Crossover operations will always be used
CreationProbability	0	No replacement of a subtree with random subtree during crossover
MaximumDepthForCreation	10	
MaximumDepthForCrossover	17	
SelectionType	1	Use tournament selection
TournamentSize	10	Size of the tournament
DemeticGrouping	1	ON
DemeSize	10	
DemeticMigProbability	10	
SwapMutationProbability	10	
ShrinkMutationProbability	10	
SteadyState	0	Not steady state
AddBestToNewPopulation	1	Always reproduce best of generation

Table 13 Run-time parameter values for GP Kernel

APPENDIX D. SOURCE CODE LISTINGS

This appendix contains listings for the code developed specifically for this project. It does not contain any code from the GPC++ GP kernel.

The following files are listed:

Gpsc.cc	The main GP program. Contains the driver for the GP system.
Gpsc.h	The class definitions and general file for the GP system
Scsm.cc	The simple call state machine and APAMs class methods
Scsm.h	The class definitions and general header info for the simple call state machine
Problem.h	There are five problem specific files, each one specifying the required behaviour of the system.
Nodeset.h	There are three nodeset files that define the function and terminal set for the various experiments.
Showgptree	A TCL program used to display a graphical representation of the program trees.

Supporting scripts, written mainly in PERL and used to analyse results and generate statistical data, are not listed here.

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
//   ####   #####    ####    ####           ####    ####
//  #    #  #    #  #       #    #         #    #  #    #
//  #       #    #   ####   #              #       #
//  #  ###  #####        #  #        ###   #       #
//  #    #  #       #    #  #    #   ###   #    #  #    #
//   ####   #        ####    ####    ###    ####    ####
//
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// gpsc.cc
//
// This is the main GP program
//
// Notes:
// ======
// This file is common to all experiments.
// The problem specific part is contained in nodeset.h which defines the
// nodeset for each problem
//
////////////////////////////////////////////////////////////////////////////
// HISTORY
// =======
// 27-Sept-97   Initial Version
// 04-Nov-97    Added parameter passing in messages
// 12 Nov 97    Changed filenames from gpsc to gpsc and scsm to scsm
// 19 Dec 97    Added generation report specialisation
// 20 Dec 97    Added support for multi-path support
// 29 Dec 97    Added support for reduced complexity functions
// 11 Apr 98    Removed unused functions and general tidy up
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// System header files
////////////////////////////////////////////////////////////////////////////
#include <stdlib.h>
#include <unistd.h>
#include <new.h>
#include <fstream.h>
#include <strstream.h>

////////////////////////////////////////////////////////////////////////////
// Application specific header files
////////////////////////////////////////////////////////////////////////////
#include "gp.h"
#include "gpconfig.h"
#include <gpsc.h>
#include <scsm.h>

////////////////////////////////////////////////////////////////////////////
// Define some global flags that control debug behaviour
////////////////////////////////////////////////////////////////////////////
int debug       = 0;        // If 1 then emit run-time debug information
int thegen      = 0;
int check_child = 1;
int quiet       = 1;
int optim       = 0;

////////////////////////////////////////////////////////////////////////////
// Externals used for multi fitness case evaluation
////////////////////////////////////////////////////////////////////////////
extern const int  ntrials;    // The number of passes required by evaluate
extern int        trial;      // Current fitness case

////////////////////////////////////////////////////////////////////////////
// Define configuration parameters and the neccessary array to
// read/write the configuration to a file.  If you need more
// variables, just add them below and insert an entry in the
// configArray.
////////////////////////////////////////////////////////////////////////////
GPVariables cfg;
struct GPConfigVarInformation configArray[]=
```

```
{
  {"PopulationSize", DATAINT, &cfg.PopulationSize},
  {"NumberOfGenerations", DATAINT, &cfg.NumberOfGenerations},
  {"CreationType", DATAINT, &cfg.CreationType},
  {"CrossoverProbability", DATADOUBLE, &cfg.CrossoverProbability},
  {"CreationProbability", DATADOUBLE, &cfg.CreationProbability},
  {"MaximumDepthForCreation", DATAINT, &cfg.MaximumDepthForCreation},
  {"MaximumDepthForCrossover", DATAINT, &cfg.MaximumDepthForCrossover},
  {"SelectionType", DATAINT, &cfg.SelectionType},
  {"TournamentSize", DATAINT, &cfg.TournamentSize},
  {"DemeticGrouping", DATAINT, &cfg.DemeticGrouping},
  {"DemeSize", DATAINT, &cfg.DemeSize},
  {"DemeticMigProbability", DATADOUBLE, &cfg.DemeticMigProbability},
  {"SwapMutationProbability", DATADOUBLE, &cfg.SwapMutationProbability},
  {"ShrinkMutationProbability", DATADOUBLE, &cfg.ShrinkMutationProbability},
  {"AddBestToNewPopulation", DATAINT, &cfg.AddBestToNewPopulation},
  {"SteadyState", DATAINT, &cfg.SteadyState},
  {"Penalty", DATAINT, &cfg.Penalty},
  {"Reward",  DATAINT, &cfg.Reward},
  {"", DATAINT, NULL}
};

////////////////////////////////////////////////////////////////////////
// Define class identifiers
////////////////////////////////////////////////////////////////////////
const int MyGeneID=GPUserID;
const int MyGPID=GPUserID+1;
const int MyPopulationID=GPUserID+2;

////////////////////////////////////////////////////////////////////////
// Define a memory object
// This is the Automous Polymorphic Addressable Memory object that
// holds a number of VarVal objects. It is pre-defined with a number of
// cells
////////////////////////////////////////////////////////////////////////
Memory memory(10);

////////////////////////////////////////////////////////////////////////
// Define a BCSM object
// This defines the call related functionality. It contains the details
// of the required call bahavour.
////////////////////////////////////////////////////////////////////////
Scsm  myScsm;

////////////////////////////////////////////////////////////////////////
//
// Name:       usage
//
// Parameters: cmd. A string containing the command name
//
// Returns:    void function
//
// Purpose:    Usage function. Prints the usage to stdout
//
////////////////////////////////////////////////////////////////////////
void usage(const char * const cmd)
{
  cout << "Usage: " << cmd <<
    "[-debug] [-v]" << endl;
  cout << "\t-debug turns on all run-time debugging" << endl;
  cout << "\t-v turns on a summary run-time status report" << endl;
  exit(1);
}

////////////////////////////////////////////////////////////////////////
//
// Name:       evaluate
//
// Member of:  MyGene
//
// Parameters: scsm    a reference to a simple call state machine
//                     that defines a service
//             gp      a reference to a GP object defining the current
//                     individual
//
// Returns:    A reference to a VarVal object that is the result of
//             this or a subtree.
```

```
//
// Purpose:    This function evaluates the fitness of a genetic tree.
//             Each gene may either interact with the BCSM and change
//             it's state, or it may alter the values of the current
//             GP variable members
//
////////////////////////////////////////////////////////////////////////
VarVal & MyGene::evaluate (Scsm & scsm, MyGP & gp)
{
  Msg    *inmsg;

  if(debug)
    {
      cout << "Evaluating gene <" << node->value() << ">" ;
      printOn(cout);
      cout << endl;
    }
  Msg msg;

  switch (node->value ())
    {
    case  FSTART:
      ////////////////////////////////////////////////////////////////////////
      //
      // This node take two parameters and accepts an input message
      // The message contains the CallersDN and a flag parameter.
      // The calledDN and flag are placed into the location returned by the
      // first parameter.
      // The result of the second parameter is returned from this function
      //
      ////////////////////////////////////////////////////////////////////////
      {
        VarVal child0;

        inmsg = scsm.emitMsg();

        child0=NthMyChild(0)->evaluate(scsm, gp);
        if(debug) {
          cout << " Child 0 = " << child0 << endl;
          cout << "Got a message from the BCSM. Type = " << inmsg->_type <<
            inmsg->p1() << endl;
        }
        if(debug)
          cout << "ABC Assigning " << inmsg->p1() <<
            " to child " << child0 << endl;
        memory.write(child0.index(), inmsg->p1());
        if(debug) cout << "Setting child 0 to " << child0 << endl;
        memory.print();
        // Go and evaluate the rest of the tree returning the value
        return NthMyChild(1)->evaluate(scsm, gp);
        break;
      }
    case  FDBREAD:
      ////////////////////////////////////////////////////////////////////////
      //
      // the first parameter is an input containing a key value
      // the second is the output containing the result.
      //
      ////////////////////////////////////////////////////////////////////////
      {
        VarVal child0;
        VarVal child1;


        child0=NthMyChild(0)->evaluate(scsm, gp);

        msg._type = Dbreq ;
        if(debug) cout << "XYZ:Assigning value " << child0 <<
                    " to message p1 " <<endl;
        msg.p1() = child0;
        if(debug){
          cout << "XYZ:Assigned value to message. Actual msg p1 = " <<
            msg.p1() << endl;
        }

        scsm.acceptMsg(msg);
        inmsg=scsm.emitMsg();
```

```
    child1=NthMyChild(1)->evaluate(scsm, gp);

    memory.write(child1.index(), inmsg->p1());
    return NthMyChild(2)->evaluate(scsm, gp);
  }
break;
case  FROUTE:
  ////////////////////////////////////////////////////////////////////////
  //
  // The first parameter is the number to route to so evaluate it first
  //
  ////////////////////////////////////////////////////////////////////////
  {
    VarVal child0;

    child0=NthMyChild(0)->evaluate(scsm, gp);

    // Plug the value into the message and send it
    msg.p1() = child0;
    msg._type = Connect;
    scsm.acceptMsg(msg);

    // Evaluate the rest of the tree
    return NthMyChild(1)->evaluate(scsm, gp);
    break;
  }
case  STRSUB:
  {
  ////////////////////////////////////////////////////////////////////////
  // For now we will support a simple shift left 1 char operation
  // Child 0 is shifted left 1 character and copied to child1
  // returns child1
  ////////////////////////////////////////////////////////////////////////

    VarVal child0, child1;
    char tmp[1024];

    child0 = NthMyChild(0)->evaluate(scsm, gp);
    child1 = NthMyChild(1)->evaluate(scsm, gp);

    if(strlen(child0.strval) > 1)
      {
        strcpy(tmp, &child0.strval[1]);
        strcpy(child1.strval, tmp);
        memory.write(child1.index(), child1);
      }
    // return the memory cell modified
    return memory[child1.index()];
    break;
  }
case  FEND:
  ////////////////////////////////////////////////////////////////////////
  // Evaluates the child and sends a message to the  SCSM.
  // The result of evaluating child 0 is returned.
  ////////////////////////////////////////////////////////////////////////
  msg._type = End;
  scsm.acceptMsg(msg);
  return NthMyChild(0)->evaluate(scsm, gp);
  break;
case  FEQ:
  ////////////////////////////////////////////////////////////////////////
  //
  // Child 0 is evaluated. If the integer part of the child is equal to 1
  //    then child1 is evaluated and returned
  // else
  //    child 2 is evaluated and returned.
  //
  ////////////////////////////////////////////////////////////////////////
  {
    if(debug)
      {
        cout << "Evaluating an FEQ node" << endl;
      }
    VarVal child0;

    child0 = NthMyChild(0)->evaluate(scsm, gp);
```

```
    if(child0.intval == 1)
      {
        if(debug) cout << "Got a TRUE in FEQ" << endl;
        return NthMyChild(1)->evaluate(scsm,gp);
      }
    else
      {
        if(debug) cout << "Got a FALSE in FEQ" << endl;
        return NthMyChild(2)->evaluate(scsm,gp);
      }
    break;
  }
case  READMSG:
  ////////////////////////////////////////////////////////////////////////
  //
  // The message is accepted from the scsm.
  // Child 0 is evaluated, and the parameters from the message
  //   placed into the resulting location.
  // Child 1 is then returned
  //
  ////////////////////////////////////////////////////////////////////////
  {
    VarVal child0;

    inmsg = scsm.emitMsg();

    child0=NthMyChild(0)->evaluate(scsm, gp);
    if(debug) {
      cout << " Child 0 = " << child0 << endl;
      cout << "Got a message from the BCSM. Type = " << inmsg->_type <<
        inmsg->p1() << endl;
    }
    if(debug)
      cout << "ABC Assigning " << inmsg->p1() << " to child " << child0 << endl;
    memory.write(child0.index(), inmsg->p1());
    if(debug) cout << "Setting child 0 to " << child0 << endl;
    memory.print();
    // Go and evaluate the rest of the tree returning the value
    return NthMyChild(1)->evaluate(scsm, gp);
    break;
  }
case  SENDMSG:
  ////////////////////////////////////////////////////////////////////////
  //
  // Child 0 is evaluated to get the message type
  // Child 1 is evaluated to get the data value
  // Child 2 is evaluated to get the next tree
  //
  ////////////////////////////////////////////////////////////////////////
  {
  VarVal child0;
  VarVal child1;


  child0=NthMyChild(0)->evaluate(scsm, gp);
  child1=NthMyChild(1)->evaluate(scsm, gp);

  msg._type =  child0.msgType;
  msg.p1()  =  child1;

  scsm.acceptMsg(msg);

  return NthMyChild(2)->evaluate(scsm, gp);
  }
  break;
case  TVAR1:
  ////////////////////////////////////////////////////////////////////////
  // Each TVARx simply returns a reference to a memory location.
  // These are all terminal nodes
  ////////////////////////////////////////////////////////////////////////
  return memory[0];
  break;
case  TVAR2:
  return memory[1];
  break;
case  TVAR3:
```

```
      return memory[2];
      break;
    case  TVAR4:
      return memory[3];
      break;
    case  TVAR5:
      return memory[4];
      break;
    case  TVAR6:
      return memory[5];
      break;
    case  TVAR7:
      return memory[6];
      break;
    case  TVAR8:
      return memory[7];
      break;
    case  TVAR9:
      return memory[8];
      break;
    case  TVAR10:
      return memory[9];
      break;
    default:
      ////////////////////////////////////////////////////////////////////////
      // A node value is unrecognised. Possibly a fault in the nodeset
      ////////////////////////////////////////////////////////////////////////
      GPExitSystem ("MyGene::evaluate", "Undefined node value");
    }
  GPExitSystem ("MyGene::evaluate", "Invalid node evaluation");
  exit(1);
}

//
// Load the problem specific node set
//
#include "nodeset.h"

////////////////////////////////////////////////////////////////////////////
//
// Name:         evaluate
//
// Member of:    MyGP
//
// Parameters:   None
//
// Returns:      void function
//
// Purpose:      Evaluate the fitness of a GP and save it into the
//               class variable fitness.
//
// Notes:
//               The external variable ntrials controls how many
//               individual fitness trials are performed.
//               The external variable trial contains the current
//               fitness case being evaluated against.
//
////////////////////////////////////////////////////////////////////////////
void MyGP::evaluate ()
{
  double  tempfitness = 0.0;
  double  fitness;

  // Evaluate main tree
  if(debug)
    {
      cout << "===============================================\n";
      cout << "Evaluating a GP\n";
    }

  goodx=badx=goodm=badm=0;
  for(trial=0;trial < ntrials; trial++)
    {
      memory.reset();
      myScsm.reset();
      if(debug) cout << "Doing trial " << trial << endl;
      NthMyGene (0)->evaluate (myScsm, *this);
```

```
      fitness = myScsm.finalStateFitness();
      tempfitness += fitness;
      goodx += myScsm.good;
      badx  += myScsm.bad;
      goodm += myScsm.goodparm;
      badm  += myScsm.badparm;
    }

  fitness = tempfitness;



  if(debug)
    cout <<" STAT1 @ gen " << thegen << ' '
         << '[' << goodx << ' '
         << badx << ' '
         << goodm << ' '
         << badm << ']' << endl;
  stdFitness = fitness;
  if(optim && fitness == 0)
    {
      cout << "Got totally fit individual on generation " << thegen << endl;
      printOn(cout);
      exit(0);
    }
}

////////////////////////////////////////////////////////////////////////////
//
// Name:        checkForValidCreation
//
// Member of:   MyPopulation
//
// Paremeters:  MyGP & a ref to a created GP object
//
// Returns:     1 if indivudual is ok, 0 otherwise
//
// Purpose:     To perform per individual checks. In this application
//              this is a null function, since all individuals are deemed
//              to be OK by virtue of APAM which ensures closure.
//
////////////////////////////////////////////////////////////////////////////
int MyPopulation::checkForValidCreation(MyGP & )
{
  return 1;
}

////////////////////////////////////////////////////////////////////////////
//
// Name          newHandler
//
// Purpose:      To handle an out of memory situation. In this application
//               we just terminate the entire run since there is little
//               that can be done.
//
////////////////////////////////////////////////////////////////////////////
void newHandler ()
{
  cerr << "\nFatal error: Out of memory." << endl;
  exit (1);
}

////////////////////////////////////////////////////////////////////////////
//
// Name:         main
//
// Parameters:   int argc  a count of the number of arguments
//               char argv[] an array of pointers to strings. Each string
//               contains a command line argument.
//
// Purpose:      Standard C++ main function. This drives the entire
//               program
//
////////////////////////////////////////////////////////////////////////////
int main (int argc, char *argv[])
{
```

```
char * filename = NULL;

if(argc > 1)
  {
    for(int i=1; i<argc; i++)
      {
        if(argv[i][0] == '-')
          {
            if(strcmp(argv[i], "-debug") == 0)
              {
                cout << "Setting debug = 1\n";
                debug=1;
              }
            else if(strcmp(argv[i], "-v") == 0)
              quiet = 0;
            else
              usage(argv[0]);
          }
        else
          filename = argv[i];
      }
  }

// Set up a new-handler, because we might need a lot of memory, and
// we don't know it's there.
set_new_handler (newHandler);

// Init GP system.
GPInit (0, -1);

myScsm.init(filename);

// Read configuration file.
GPConfiguration config (cout, "gpsc.ini", configArray);

// Print the configuration
if(!quiet)
  cout << cfg << endl;

// Print state table
if(!quiet)
  myScsm.printStates();

// Create the adf function/terminal set and print it out.
GPAdfNodeSet adfNs;
createNodeSet (adfNs);
if(!quiet)
  cout << adfNs << endl;

// Open the main output file for the data and statistics file.
// First set up names for data file.  Remember we should delete the
// string from the stream, well just a few bytes
ostrstream strOutFile, strStatFile;
strOutFile  << "data.dat" << ends;
strStatFile << "data.stc" << ends;
ofstream fout (strOutFile.str());
ofstream bout (strStatFile.str());

// Create a population with this configuration
if(!quiet)
  cout << "Creating initial population ..." << endl;
MyPopulation* pop=new MyPopulation (cfg, adfNs);
pop->create ();
pop->createGenerationReport (1, 0, fout, bout);

// This next for statement is the actual genetic programming system
// which is in essence just repeated reproduction and crossover loop
// through all the generations ...
MyPopulation* newPop=NULL;
for (int gen=1; gen<=cfg.NumberOfGenerations; gen++)
  {
    thegen=gen;
    // Create a new generation from the old one by applying the
    // genetic operators
    if (!cfg.SteadyState)
      newPop=new MyPopulation (cfg, adfNs);
    pop->generate (*newPop);
```

```
      // Delete the old generation and make the new the old one
      if (!cfg.SteadyState)
        {
          delete pop;
          pop=newPop;
        }

      // Create a report of this generation and how well it is doing
      pop->createGenerationReport (0, gen, fout, bout);
    }
  return 0;
}


////////////////////////////////////////////////////////////////////////
//
// Name:               printOn
//
// Member of:          MyGene
//
// Parameters:         ostream & a reference to an output stream on which
//                     to write the information
//
// Purpose:            Print function to display and record the details of
//                     an individual
//
////////////////////////////////////////////////////////////////////////
void MyGene::printOn (ostream& os)
{
  if (node->isFunction ())
    os << "(";
  os << *node;

  // Print all children, if any
  for (int n=0; n<containerSize(); n++)
    {
      GPGene* current=NthChild (n);

      os << ' ';
      if (current)
        os << *current;
      else
        os << "(NULL)";
    }

  if (node->isFunction ())
    os << ")";
}


////////////////////////////////////////////////////////////////////////
//
// Name:               createGenerationReport
//
// Member of:          MyPopulation
//
// Parameters:         printLegend. A flag to indicate whether to print
//                                  a legend
//                     generation   The generation number
//                     fout         a stream reference to a file
//                     bout         a stream reference to the console
//
// Purpose:            Output all the data found in a generation....
//
////////////////////////////////////////////////////////////////////////
void MyPopulation::createGenerationReport (int printLegend, int generation,
                                           ostream& fout, ostream& bout)
{
  if (printLegend)
    {
      if(!quiet)
        cout << "Gen|       Fitness     |      Length       |   Depth\n"
             << "   |  Best|Avg.|Worst  |  Best|Avg.|Worst  |  Best|Avg.|Worst\n";
      bout << "#Gen|       Fitness     |      Length       |   Depth\n"
           << "#   |  Best|Avg.|Worst  |  Best|Avg.|Worst  |  Best|Avg.|Worst\n";
    }
```

```
  bout << generation
       << ' ' << NthMyGP(bestOfPopulation)->getFitness()
       << ' ' << avgFitness
       << ' ' << NthMyGP(worstOfPopulation)->getFitness()
       << "    "
       << ' ' << NthMyGP(bestOfPopulation)->length ()
       << ' ' << avgLength
       << ' ' << NthMyGP(worstOfPopulation)->length ()
       << "    "
       << ' ' << NthMyGP(bestOfPopulation)->depth ()
       << ' ' << avgDepth
       << ' ' << NthMyGP(worstOfPopulation)->depth ();
  if(!quiet)
    bout << "\t[ " << NthMyGP(bestOfPopulation)->goodx
         << ' ' << NthMyGP(bestOfPopulation)->badx
         << ' ' << NthMyGP(bestOfPopulation)->goodm << ' '
         << NthMyGP(bestOfPopulation)->badm << " ]"
         << endl;
  else
    bout << "\n";

  if(debug)
    cout << "Best of population = " << bestOfPopulation << endl;
  if(!quiet)
    cout << generation
         << ' ' << NthMyGP(bestOfPopulation)->getFitness ()
         << ' ' << avgFitness
         << ' ' << NthMyGP(worstOfPopulation)->getFitness ()
         << "    "
         << ' ' << NthMyGP(bestOfPopulation)->length ()
         << ' ' << avgLength
         << ' ' << NthMyGP(worstOfPopulation)->length ()
         << "    "
         << ' ' << NthMyGP(bestOfPopulation)->depth ()
         << ' ' << avgDepth
         << ' ' << NthMyGP(worstOfPopulation)->depth ()
      ;
  if(!quiet)
    cout << "\t[ " << NthMyGP(bestOfPopulation)->goodx
         << ' ' << NthMyGP(bestOfPopulation)->badx
         << ' ' << NthMyGP(bestOfPopulation)->goodm << ' '
         << NthMyGP(bestOfPopulation)->badm << " ]"
         << endl;

  // Place the best of generation in output files
  fout << "Best of generation " << generation
       << " (Fitness = " << NthMyGP (bestOfPopulation)->getFitness ()
       << ", Structural Complexity = " << NthMyGP (bestOfPopulation)->length ()
       << ")" << endl
       << NthMyGP(bestOfPopulation)->goodx
       << ' ' << NthMyGP(bestOfPopulation)->badx
       << ' ' << NthMyGP(bestOfPopulation)->goodm << ' '
       << NthMyGP(bestOfPopulation)->badm << ' '
       << endl
       << *NthMyGP (bestOfPopulation)
       << endl;
}
```

```
////////////////////////////////////////////////////////////////////////
//
//   ####   #####    ####    ####           #    #
//  #    #  #    #  #       #    #          #    #
//  #       #    #   ####   #               ######
//  #  ###  #####         #  #        ###    #    #
//  #    #  #       #    #  #    #   ###    #    #
//   ####   #        ####    ####    ###    #    #
//
////////////////////////////////////////////////////////////////////////
// gpsc.h
// Class definitions for the GP service creation system
//
// Revision history
// 30  Sept 1997   Initial version using standard gp kernel
// 04  Nov  97     Added parameter passing in messages
// 12  Nov  97           Changed name to gpsc and scsm
// 11  Apr  98     Removed unused functions and general tidy up
////////////////////////////////////////////////////////////////////////

#ifndef GPSC_H
#define GPSC_H

#include "gp.h"
#include "gpconfig.h"
#include <scsm.h>

// Define function and terminal identifiers
enum  FTids
{
  FSTART  = 0,
  FDBREAD,  FROUTE,
  STRSUB,   FEND,
  TVAR1,    TVAR2,
  TVAR3,    TVAR4,  TVAR5,
  TVAR6,    TVAR7,  TVAR8,
  TVAR9,    TVAR10,

  FEQ,                          // For multi path experiments

  READMSG, SENDMSG,             // For reduced complexity functions

  LastID
};

#define MAXMEM 100

////////////////////////////////////////////////////////////////////////
// A memory object. This is used as indexed memory for the GP
// It stores an array of VarVal objects
////////////////////////////////////////////////////////////////////////
class Memory
{
private:
  int        size;

public:
  VarVal     varVal[MAXMEM]; // A global array of variables

//
// Constructor with default size
//
  Memory()
  {
    size = MAXMEM;
    reset();
  }

//
// Constructor with required size
//
  Memory(int wanted)
  {
    if(wanted > MAXMEM)
      {
        cout << "Max memory size is " << MAXMEM << " elements." << endl;
        exit(1);
```

```
      }
    else
      {
        size=wanted;
        reset();
      }
  }

//
// Reset the memory to empty
//
  void reset()
  {
    if(debug) cout << "Resetting memory \n";
    for (int cnt=0;cnt<MAXMEM;cnt++)
      {
        varVal[cnt].strval[0] = '\0';
        varVal[cnt].intval    = cnt;
        varVal[cnt].setIndex(cnt);
        varVal[cnt].msgType   = (MsgType)cnt;
      }
  }

//
// Overload the [] operator to allow indexing
//
  VarVal & operator[](int index)
  {
    if(index < 0 || index >= size)
      {
        cout << "Illegal index into memory array <" << index << ">" << endl;
        exit(1);
      }
    if(debug)  cout << "Returning memory index " << index << " value = " <<
                 varVal[index] << endl;
    return varVal[index];
  }

//
// print function. Displays the memory object contents, primarily for
// debugging purposes
//
  void print(void)
  {
    if(debug) {
      for(int i=0;i<size;i++)
        {
          cout << "Memory location " << i <<  " => "
               << varVal[i] << endl;
        }
    }
  }
//
// Write function. This is used when we dont want to
// destroy the index value of the memory cell
//
  void write(int i, VarVal &v)
  {
    varVal[i].intval=v.intval;
    strcpy(varVal[i].strval, v.strval);
  }
};

class MyGP;   // Forward declaration
////////////////////////////////////////////////////////////////////////
// Inherit the three GP classes GPGene, GP and GPPopulation
// These classes define the problem specific details of GP
////////////////////////////////////////////////////////////////////////
//
// Class:         MyGene
//
// Derived From:  GPGene
//
// Purpose:       Defines the individual genes
//
////////////////////////////////////////////////////////////////////////
class MyGene : public GPGene
```

```
{
public:
// Duplication (mandatory)
  MyGene (const MyGene& gpo) : GPGene (gpo) { }
  virtual GPObject& duplicate () { return *(new MyGene(*this)); }

// Creation of own class objects (mandatory)
  MyGene (GPNode& gpo) : GPGene (gpo) {}
  virtual GPGene* createChild (GPNode& gpo) {
    return new MyGene (gpo); }

// Tree evaluation
  VarVal & evaluate (Scsm & scsm, MyGP & gp);

// Load and save
  MyGene () {}

  virtual GPObject* createObject() { return new MyGene; }

// Print
  virtual void printOn (ostream& os);

// Access children
  MyGene* NthMyChild (int n) {
    return (MyGene*) GPContainer::Nth (n); }
};

//////////////////////////////////////////////////////////////////////////
//
// Class:         MyGP
//
// Derived From:  GP
//
// Purpose:       Defines an individual program
//
//////////////////////////////////////////////////////////////////////////
class MyGP : public GP
{
public:
// Counters for good and bad transtitions and message parameters
  int   goodx, badx, goodm, badm;

// Duplication (mandatory)
  MyGP (MyGP& gpo) : GP (gpo)
  {
    goodx=gpo.goodx;
    badx=gpo.badx;
    goodm=gpo.goodm;
    badm=gpo.badm;
  }
  virtual GPObject& duplicate () { return *(new MyGP(*this)); }

// Creation of own class objects (mandatory)
  MyGP (int genes) : GP (genes) {}
  virtual GPGene* createGene (GPNode& gpo) {
    return new MyGene (gpo); }

// Tree evaluation (mandatory)
  virtual void evaluate ();

// Load and save (not mandatory)
  MyGP () {}
  virtual GPObject* createObject() { return new MyGP; }

// Access trees (not mandatory)
  MyGene* NthMyGene (int n)
  {
    return (MyGene*) GPContainer::Nth (n);
  }
};


//////////////////////////////////////////////////////////////////////////
//
// Class:         MyPopulation
//
// Derived From:  GPPopulation
```

```
//
// Purpose:       Defines a complete population of programs
//
////////////////////////////////////////////////////////////////////////
class MyPopulation : public GPPopulation
{
public:
// Constructor (mandatory)
  MyPopulation (GPVariables& GPVar_, GPAdfNodeSet& adfNs_) :
    GPPopulation (GPVar_, adfNs_) {}

// Duplication (mandatory)
  MyPopulation (MyPopulation& gpo) : GPPopulation(gpo) {}
  virtual GPObject& duplicate () { return *(new MyPopulation(*this)); }

// Creation of own class objects (mandatory)
  virtual GP* createGP (int numOfTrees) { return new MyGP (numOfTrees); }

// Load and save (not mandatory)
  MyPopulation () {}
  virtual GPObject* createObject() { return new MyPopulation; }

// Access genetic programs (not mandatory)
  MyGP* NthMyGP (int n) {
    return (MyGP*) GPContainer::Nth (n); }
  virtual void createGenerationReport (int printLegend, int generation,
                                       ostream& fout, ostream& bout);

// Check for valid trees
  virtual int  checkForValidCreation(MyGP &gpo);
};

#endif // GPSC_H
```

```
////////////////////////////////////////////////////////////////////////
//
//   ####    ####    ####   #    #           ####    ####
//  #       #    #  #       ##  ##          #    #  #    #
//   ####   #       ####   # ## #          #       #
//       #  #           #  #    #   ###    #       #
//  #    #  #    #  #    #  #    #   ###    #    #  #    #
//   ####    ####    ####   #    #   ###     ####    ####
//
////////////////////////////////////////////////////////////////////////
//
// Simple SCSM to simulate an external switch interface
// Pete MArtin
// Revision History
//
// Sept 21 1997               Initial version
// Nov  12 1997         Changed name to Simple Call State Model
// Dec  19 1997         Add support for multi fitness cases with multiple
//                      paths thro state machine.
////////////////////////////////////////////////////////////////////////


#include <stream.h>
#include <scsm.h>
#include <stdio.h>

////////////////////////////////////////////////////////////////////////
// Decalartions
////////////////////////////////////////////////////////////////////////

extern int debug;
extern GPVariables cfg;
int    optimistic = 0; // If set then state transitions need not be strict

////////////////////////////////////////////////////////////////////////
//
// define the message text to type translation table
//
////////////////////////////////////////////////////////////////////////
MsgTable table[MAXMSGS] =
{
  {None, "None"},
  {Any, "Any"},
  {Idp, "Idp"},
  {Connect, "Connect"},
  {Dbreq, "Dbreq"},
  {Dbresp, "Dbresp"},
  {Dberr, "Dberr"},
  {End, "End"},
  {Timeout, "Timeout"},
  {(MsgType)-1, ""}
};

////////////////////////////////////////////////////////////////////////
// Include the problem description file
////////////////////////////////////////////////////////////////////////
#include <problem.h>

//
// Globals used for controlling multi fitness case experiments
//
int        ntrials = NTRIALS;   // Defined in the problem file
int        trial;               // Current fitness case

////////////////////////////////////////////////////////////////////////
//
// Name:         decode
//
// Member of:    n/a
//
// Parameters:   A string containing a message type
//
// Purpose:      Converts a text string to a message number
//               Used when initialising the state tables
//
////////////////////////////////////////////////////////////////////////
```

```
MsgType decode(char *s)
{
  MsgType  result = None;

  for(int i=0;i<MAXMSGS; i++)
    {
      if(table[i].enumtype == -1)
        break;
      if(strcmp(s, table[i].strtype) == 0)
        result = table[i].enumtype;
    }
  if(debug)
    cout << "@@@@@ Decode of type " << s << " to value " << result << endl;
  return result;
}


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//
// Name:               Scsm
//
// Member of:          Scsm
//
// Parameters:         None
//
// Purpose:            Constructor for an SCSM object
//
////////////////////////////////////////////////////////////////////////
Scsm::Scsm()
{
  if(debug)
    cout << "Constructing a Scsm " << endl;
  ok = 500;
  fitness = 3.0;
}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
//
// Name:               ~Scsm
//
// Member of:          Scsm
//
// Parameters:         None
//
// Purpose:            Destructor for an Scsm object
//
////////////////////////////////////////////////////////////////////////
Scsm::~Scsm()
{
  if(debug)
    cout << "Destructor for SCSM\n";
}

////////////////////////////////////////////////////////////////////////
//
// Name:               init
//
// Member of:          Scsm
//
// Parameters:         None used
//
// Purpose:            Initilise on a per run basis the Scsm.
//
////////////////////////////////////////////////////////////////////////
void  Scsm::init(char * )
{
  for(int i=0;i<MAXSTATES;i++)
    {                                          // Initialise the state table
      if(st[i].c == -1)
        break;  // All done
      stateTable.insertState(st[i].c,
                             Msg(st[i].o, st[i].op1, st[i].op2),
                             Msg(st[i].e, st[i].ip1, st[i].ip2),
```

```
                            st[i].n,
                            st[i].f);
    }
}

////////////////////////////////////////////////////////////////////////
//
// Name:               print
//
// Member of:          StateTable
//
// Parameters:         None
//
// Purpose:            Prints the state table for information purposes
//                     to standard out
//
////////////////////////////////////////////////////////////////////////
void  StateTable::print(void)
{
  cout << "Index\tCur\tOutMsg\tp1\tp2\tInMsg\tp1\tp2\tNext\n";
  for(int i=0;i<next; i++ )
    {
      cout << i << '\t' << table[i].cur << '\t'
           << table[i].outmsg._type  << '\t'
           << table[i].outmsg.p1().strval    << '\t'
           << table[i].outmsg.p2().strval    << '\t'
           << table[i].event._type   << '\t'
           << table[i].event.p1().strval    << '\t'
           << table[i].event.p2().strval    << '\t'
           << table[i].next
           << table[i].fitness << endl;
    }
}

////////////////////////////////////////////////////////////////////////
//
// Name:               emitMsg
//
// Member of:          Scsm
//
// Parameters:         None
//
// Purpose:            called as part of the evaluation of fitness.
//                     It locates an entry in the state table for the
//                     current scsm state, and emits a message to the
//                     evolving program
//
////////////////////////////////////////////////////////////////////////
Msg  * Scsm::emitMsg()
{
  int    status;
  static Msg  thisMsg;
  State  next;


  status = stateTable.state(state, next);
  next.print();
  if(status == 0)
    {
      // still in the same state, so no message to output
      // return a timeout message to the service
      thisMsg._type = Timeout;
      if(debug)
        cout << "Did not get good transition\n";
      penalise();
      bad++;
    }
  else
    {
      thisMsg = next.outmsg;
      thisMsg.p1() = next.outmsg.p1();
      if(debug)
        cout << "Setting p1 to " << thisMsg.p1().strval << endl;
      thisMsg.p2() = next.outmsg.p2();
      if(debug)
        cout << "<<<Going from state " << state << " To " << next.next << endl;
      state = next.next;
```

```
      reward();  // We had a correct transition here
      if(debug)
        cout << "Got good transtion in accept\n";
      good++;
    }

  return &thisMsg;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               emitMsg
//
// Member of:          Scsm
//
// Parameters:         A message object reference
//
// Purpose:            called as part of the evaluation of fitness.
//                     It locates an entry in the state table for the current
//                     scsm state, and matches the recieved message.
//                     If Optimistic is set, then a transition is made even if the
//                     the current state is wrong. This is to try to maintain
//                     diversity in the population during early generations
//
////////////////////////////////////////////////////////////////////////
void Scsm::acceptMsg (Msg & msg)
{
  State next;
  int status;
  int optimistic = 0;

  status = stateTable.state(state, msg, next, optimistic);
  if (status == 0)
    {
      // Did not get a transition out of the current state
      // do nothing
      if(debug)
        cout << ">>>Did not accept message type " << msg._type <<
          " in state " << state << endl;
      penalise();
      bad++;
      return;
    }
  if(debug)
    cout << "Message parameter value = " << msg.p1() << endl;
  if(next.event.p1().strval[0] != '*')
    {
      if(debug)
        cout << "$$$$Expecting a parameter value of "
             << next.event.p1().strval << endl;
      if(strcmp(next.event.p1().strval, msg.p1().strval) == 0)
        {
          if(debug)
            cout << "Got good parameter " <<  msg.p1() << endl;
          reward();
          goodparm++;
        }
      else
        {
          if(debug)
            cout << "Bad parameter. Got " << msg.p1().strval << endl;
                penalise();
          badparm++;
        }
    }

  if(debug)
    cout <<">>>Going from state " << state << " To " << next.next << endl;
  state = next.next;
  // Got a good transition
  reward();
  good++;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               insertState
```

```
//
// Member of:          StateTable
//
// Parameters:         c = state number
//                     o = output message type
//                     e = input message type
//                     n = next state
//                     f = weighting factor (not used)
//
// Purpose:            Insert a state entry into the table
//
////////////////////////////////////////////////////////////////////////
void StateTable::insertState(int c, const Msg & o,
                             const Msg & e,
                             int n, double f)
{
  if(next > MAXSTATES) {
    cout << "Too many states\n";
    exit(1);
  }
  table[next].cur    = c;
  table[next].outmsg = o;
  table[next].event  = e;
  table[next].next   = n;
  table[next].fitness=f;
  next++;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               state
//
// Member of:          StateTable
//
// Parameters:         cur = current state
//                     msg = message to trigger transtition
//                     ret = ref to a return variable for next state
//                     optimistic. See below.
//
// Purpose:            Given a message and state, locate a state entry
//                     Returns the table entry if a match found, or current state if
//                     not found
////////////////////////////////////////////////////////////////////////
int StateTable::state(int cur, Msg & msg,  State &ret, int optimistic)
{

  for(int i=0;i<next;i++)
    {
      if(table[i].event._type == msg._type)
        {
          switch (optimistic)
            {
            case 0:
              if(table[i].cur == cur)
                {
                  if(debug)
                    cout << "+++ACCEPT+++Got match in state "<< cur <<
                      " to goto state " << table[i].next << "Index = " << i
                        << "cur = " << cur << endl;
                  current = i;
                  ret = table[i];
                  return 1;
                }
              break;
            case 1:
              if(debug)
                cout << "+++ACCEPT-OPTIMIST+++Got match in state "<< cur <<
                  " to goto state " << table[i].next << "Index = " << i
                    << "cur = " << cur << endl;
              current = i;
              ret = table[i];
              return 0;
              break;
            }
        }
    }
  // Failed to find a valid event in this state,
```

```
  ret = table[current];
  return 0;
}

////////////////////////////////////////////////////////////////////////////
//

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
// Name:               state
//
// Member of:          StateTable
//
// Parameters:         cur = current state
//                     ret = return variable which will hold next state
//
// Purpose:            To loacte a state if an output message is required
//
// Given a state, locate a state entry that has an outmsg
// Returns the table entry if a match found, or current state if
// not found
//
// If there are more than one possible states, then this function decides
// which one to select.
// To do this, the table is searched for the number of matching states
// If there are zero, then returns the same state
// If there is more than one say N , then for now we select 1 from N
// using the trial number. Contrast to random selection
//
////////////////////////////////////////////////////////////////////////////
int StateTable::state(int cur,  State &ret )
{
  extern int trial;  // Holds the run number : 0 or 1
  int matches=0;

  // Find number of matching states
  for(int i=0;i<next;i++)
    {
      if(table[i].cur == cur && table[i].outmsg._type != None)
        {
          matches++;
        }
    }

  if(matches == 0)
    {
      // Failed to find a valid event in this state,
      if(debug) cout << "+++ACCEPT(emit)+++Failed to find valid state\n";
      ret = table[current];
      return 0;
    }
  else if(matches == 1)
    {
      // Go back and find the match
      for(int i=0;i<next;i++)
        {
          if(table[i].cur == cur && table[i].outmsg._type != None)
            {
              if(debug)
                cout << "+++ACCEPT(Emit)+++Got match in state "<< cur <<
                  " to goto state " << table[i].next << "Index = " << i
                     << "cur = " << cur << endl;
              current = i;
              ret = table[i];
              return 1;
            }
        }
    }
  else
    {
      // there were more than one match, so work out a probability that it was
      // the first or second (assuming here that there are only two states)

      long randval;
      int  count=0;
```

```
      randval = GPrand() % matches;
      if(debug)cout << "randval = " << randval << endl;
      for(int i=0;i<next;i++)
        {
          if(table[i].cur == cur && table[i].outmsg._type != None)
            {
              // Found a valid entry. If the random value == count then select
              // this entry else select second
              if(trial == count)
                {
                  if(debug) cout << "Selecting the entry number " <<
                                 count << endl;
                  current = i;
                  ret = table[i];
                  return 1;
                }
              else
                {
                  if(debug)cout << "Incrementing the counter\n";
                  count ++;
                }
            }
        }
    }
  cout << "ERROR> DID NOT FIND VALID ENTRY BUT SHOULD HAVE\n";
  return 0;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               operator =
//
// Member of:          VarVal
//
// Parameters:         ref to a varval
//
// Purpose:            Assignment operator
//
////////////////////////////////////////////////////////////////////////
VarVal & VarVal::operator=(const VarVal & v)
{
  if(debug) cout << "Ref assignment to VarVal with value " << v  << endl;


  strcpy (strval, v.strval);
  intval = v.intval;
  if(_index == -1)
    // Dont change if it will overide an esiting address
    _index  = v._index;

  return *this;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               operator =
//
// Member of:          VarVal
//
// Parameters:         s = a string
//
// Purpose:            Assignment for string values to a varval
//
////////////////////////////////////////////////////////////////////////
VarVal & VarVal::operator=(const char * s)
{
  if(debug) cout << "String assignment to VarVal with value " << s  << endl;

  strcpy(strval, s);

  return *this;
}

////////////////////////////////////////////////////////////////////////
//
// Name:               operator <<
//
```

```
// Member of:           VarVal
//
// Parameters:          stream and ref to varval
//
// Purpose:             To provide output for debugging
//
////////////////////////////////////////////////////////////////////////
ostream & operator << (ostream & op, VarVal v)
{
  return op << "[String = " << v.strval << " Int = " << v.intval << " Index = " << v._index << "]" ;
}
```

```
////////////////////////////////////////////////////////////////////////
//
//   ####    ####    ####   #    #          #    #
//  #       #    #  #       ##  ##          #    #
//   ####   #       ####   # ## #          ######
//       #  #           #  #    #   ###    #    #
//  #    #  #    #  #    #  #    #   ###    #    #
//   ####    ####    ####   #    #   ###    #    #
//
////////////////////////////////////////////////////////////////////////
// SCSM simulates a simple an external switch interface
// Pete Martin
// Revision History
// Sept 21 1997   Initial version
// 04-Nov-97    Added parameter passing in messages
// 19 Nov 97    Added message type member to VarVal
// 21 Apr 98    Tidy up
////////////////////////////////////////////////////////////////////////

#ifndef _SCSM_H_
#define _SCSM_H_

#include <stream.h>
#include <gp.h>
//
// Some manifest constants
//
#define MAXMSGS    100
#define MAXSTATES   50

//
// General externals
//
extern int         debug;
extern GPVariables cfg;

////////////////////////////////////////////////////////////////////////
// Msg Types.
// Not all these get used!
////////////////////////////////////////////////////////////////////////
enum   MsgType
{
  None,    Any,      Idp,     Connect, Alarm,
  Dbreq,   Dbresp,   Dberr,   End,     Timeout
};

////////////////////////////////////////////////////////////////////////
// Name:  VarVal
//
// Purpose:  This class is the type returned from each SLP function or terminal
//           It can represent a string value or an integral value
////////////////////////////////////////////////////////////////////////
typedef   char   StrVal[24];    // Declare string type. 24 is ITU value

class VarVal
{
private:
  int    _index;
public:
  VarVal()
  {
    strval[0]='\0'; intval = 0;
  }
  VarVal(const char *v)
  {
    strcpy(strval,v);
  }
  VarVal(const int v)
  {
    intval=v;
  }

  VarVal(const VarVal & v)
  {
    intval=v.intval;
    strcpy(strval, v.strval);
```

```
  }
  VarVal & operator=(const VarVal *);
  VarVal & operator=(const char * s);
  StrVal             strval;
  int                intval;
  MsgType            msgType;
  int                index() { return _index;}
  void               setIndex(int i) { _index = i; }
  friend ostream & operator << (ostream &, VarVal);
};


////////////////////////////////////////////////////////////////////////
// MsgTable.
// A member class that holds the name and value of a message for
// translation from one to the other
////////////////////////////////////////////////////////////////////////
class  MsgTable
{
public:
  MsgType    enumtype;
  char      *strtype;
};

////////////////////////////////////////////////////////////////////////
// Msg class.
/// Defines the structure of a message
////////////////////////////////////////////////////////////////////////
class Msg
{
private:
  VarVal        _p1;
  VarVal        _p2;

public:
  Msg()
  {
    _type = None;
  }
  Msg(MsgType t)
  {
    _type = t;
  }
  Msg(MsgType t, const char *p1, const char *p2)
  {
    _type = t;
    _p1=p1;
    _p2=p2;
  }

  Msg(MsgType t, const char *p1, const int p2)
  {
    _type = t;
    _p1=p1;
    _p1.intval=p2;
  }

  MsgType _type;
  VarVal       &p1(void) {return _p1;}
  VarVal       &p2(void) {return _p2;}
  void          print()
  {
    cout << "Message Type " << _type << " p1 strval = "
   << _p1.strval << " p1 intval = " << _p1.intval
   << " ps = " << _p2.strval << endl;
  }
};

////////////////////////////////////////////////////////////////////////
// Class to hold the definitions of state information.
////////////////////////////////////////////////////////////////////////
class State
{
public:
  State()
  {
    cur=-1; outmsg._type = None; event._type = None;
```

```
    next = -1; fitness = 0.0;
  }
  void print()
  {
    if(debug)
      cout << "State: cur = " << cur << " next = " << next
    << endl;
  }

  int   cur;           // This state. -1 if entry not used
  Msg   outmsg;        // Any message to output
  Msg   event;         // The event that gets us here
  int   next;          // The state to go to
  double fitness;      // How fit is this state?
};

////////////////////////////////////////////////////////////////////////
// StateTable class
// This class defines the entire set of states for a run.
////////////////////////////////////////////////////////////////////////
class StateTable
{
public:
  StateTable() {next = 0; current = 0; reset(); }
  void    insertState(int,
         const Msg &,
         const Msg &,
         int,
         double);
  void    print(void);
  int     state(int state, Msg & msg, State &, int);
  int     state(int state, State &);
  int     nState() {return next;}

private:
  int      next;
  int      current;
  State    table[MAXSTATES];
  void reset() {
    for (int i=0; i< MAXSTATES;i++) {
      table[i].cur = -1;
    }
  }
};

////////////////////////////////////////////////////////////////////////
//
// Name:  St
//
// Purpose: State table entry object. Contains the description of a state
//
////////////////////////////////////////////////////////////////////////
struct St                              // The state table
{
  int     c;
  MsgType o;
  char *  op1;
  int     op2;
  MsgType e;
  char *  ip1;
  int     ip2;
  int     n;
  double  f;
};

////////////////////////////////////////////////////////////////////////
//
// Name:  Scsm
//
// Purpose: A simple call state machine.
//          Represents the internal call processing states of the SSP
//
////////////////////////////////////////////////////////////////////////
class Scsm
{
private:
  int          state;                // Current state
```

```
  double       fitness;
  StateTable   stateTable;
  int          ok;

  void read_state_table(char *);

public:
  int          good;
  int          bad;
  int          goodparm;
  int          badparm;


  Scsm();
  ~Scsm();
  void  init(char *);
  void  reset(void) {
    fitness = 1300.0;   // Initial fitness value. This goes down as fitness goes up
    state = ok = good = bad = goodparm = badparm= 0;
  }
  void  printStates(void) {stateTable.print();}
  Msg   * emitMsg();
  void  acceptMsg(Msg &);
  int   curState();
  double finalStateFitness(void) {
    double f;
    f = fitness;
    f = f - (ok);
    if(debug)
      cout << "Scsm fitness = " << f << endl;
    return f;
  }
  void penalise(void)   { ok -= cfg.Penalty; }
  void reward(void)     { ok += cfg.Reward;}
  void summary()        { cout << "Good = " << good << " Bad = " << bad <<
        "Goodparms = " << goodparm << " badparms = "
           << badparm << endl;}
};

#endif // _SCSM_H_
```

D.1. Problem 1 description

```
////////////////////////////////////////////////////////////////////////////
// This problem is for experiment 1. A simple number translation service
// is to be created.
// The Initial DP carries the calledDN. The resultant DB request key is
// the CalledDN stripped of the first number.
// The connect is carries the result of the DBresponse stripped of the
// first digit.
////////////////////////////////////////////////////////////////////////////

static struct St st[MAXSTATES] = {
        {0, Idp,   "123456", 0, None,    "*",      0, 1, 10.0},
        {1, None,  "*",      0, Dbreq,   "23456",  0, 2, 5.0},
        {2, Dbresp,"654321", 0, None,    "*",      0, 3, 1.0},
        {3, None,  "*",      0, Connect, "54321",  0, 4, 1.0},
        {4, None,  "*",      0, End,     "*",      0,-1, 1.0},
        {-1, None, "*",      0, None,    "*",      0,-1, 0.0}};

#define NTRIALS  1
```

D.2. Problem 2 description

```
////////////////////////////////////////////////////////////////////////////
// This problem is for experiment 2. A complex number translation service
// is to be created.
// The Initial DP carries the calledDN. The resultant DB request key is
// the CalledDN stripped of the first number. A second Db request is made
// The connect is carries the result of the DBresponse stripped of the
// first digit.
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

static struct St st[MAXSTATES] = {
        {0, Idp,   "123456", 0, None,    "*",      0,  1, 10.0},
        {1, None,  "*",      0, Dbreq,   "23456",  0,  2, 5.0},
        {2, Dbresp,"654321", 0, None,    "*",      0,  3, 1.0},
        {3, None,  "*",      0, Dbreq,   "54321",  0,  4, 5.0},
        {4, Dbresp,"987654", 0, None,    "*",      0,  5, 1.0},
        {5, None,  "*",      0, Connect, "87654",  0,  6, 1.0},
        {6, None,  "*",      0, End,     "*",      0, -1, 1.0},
        {-1, None, "*",      0, None,    "*",      0, -1, 0.0}};

#define NTRIALS  1
```

D.3. Problem 3 description

```
////////////////////////////////////////////////////////////////////////////
// This is for experiment 3.
// A simple multi-path input file
////////////////////////////////////////////////////////////////////////////

static struct St st[MAXSTATES] = {
        {0, Idp,   "123456", 0, None,    "*",      0,  1, 0.5},
        {0, Idp,   "654321", 1, None,    "*",      0,  2, 0.5},
        {1, None,  "*",      0, Dbreq,   "23456",  0,  3, 1.0},
        {3, Dbresp,"999999", 0, None,    "*",      0, -1, 1.0},
        {2, None,  "*",      0, Connect, "54321",  0, -1, 1.0},
        {-1, None, "*",      0, None,    "*",      0, -1, 0.0}};

#define NTRIALS 2
```

D.4. Problem 4 description

```
////////////////////////////////////////////////////////////////////////////
// This is for experiment 4.
// A complex multi-path input file
////////////////////////////////////////////////////////////////////////////
static struct St st[MAXSTATES] = {
        {0, Idp,   "123456", 0, None,    "*",      0, 1, 1.0},
        {1, None,  "*",      0, Dbreq,   "23456",  0, 2, 1.0},
        {2, Dbresp,"654321", 0, None,    "*",      0, 3, 0.5}, // The ok case
        {3, None,  "*",      0, Connect, "54321",  0, 4, 1.0},
```

```
        {4, None,  "*",      0, End,     "*",      0,-1, 1.0},
        {2, Dbresp,"000",    1, None,    "*",      0, 6, 0.5}, // The error case
        {6, None,  "*",      0, Connect,  "000",   0, 7, 1.0},
        {7, None,  "*",      0, End,     "*",      0,-1, 1.0},
        {-1, None, "*",      0, None,    "*",      0,-1, 0.0}};

#define NTRIALS 2
```

D.5. Problem 5 description

```
////////////////////////////////////////////////////////////////////////////
// This is for experiment 5.
// A complex Number translation service, but using reduced complexity
// nodes
////////////////////////////////////////////////////////////////////////////

static struct St st[MAXSTATES] = {
        {0, Idp,   "123456", 0, None,    "*",      0,  1, 10.0},
        {1, None,  "*",      0, Dbreq,   "23456",  0,  2, 5.0},
        {2, Dbresp,"654321", 0, None,    "*",      0,  3, 1.0},
        {3, None,  "*",      0, Dbreq,   "54321",  0,  4, 5.0},
        {4, Dbresp,"987654", 0, None,    "*",      0,  5, 1.0},
        {5, None,  "*",      0, Connect, "87654",  0,  6, 1.0},
        {6, None,  "*",      0, End,     "*",      0, -1, 1.0},
        {-1, None, "*",      0, None,    "*",      0, -1, 0.0}};

#define NTRIALS  1
```

D.6. Nodeset 1 description

```
////////////////////////////////////////////////////////////////////////
// nodeset.h
//
// This is the definition of the terminal set.
// for experiments 1 & 2
//
////////////////////////////////////////////////////////////////////////
void createNodeSet (GPAdfNodeSet& adfNs)
{
  // Reserve space for the node sets
  adfNs.reserveSpace (1);

  // Now define the function and terminal set for each ADF and place
  // function/terminal sets into overall ADF container
  GPNodeSet& ns1=*new GPNodeSet (11);

  adfNs.put (0, ns1);

  // Define functions/terminals and place them into the appropriate
  // sets.  Terminals take two arguments, functions three (the third
  // parameter is the number of arguments the function has)

  ns1.putNode (*new GPNode(  FSTART,    "FSTART",   2));
  ns1.putNode (*new GPNode(  FROUTE,    "FROUTE",   2));
  ns1.putNode (*new GPNode(  FDBREAD,   "FDBREAD",  3));
  ns1.putNode (*new GPNode(  STRSUB,    "STRSUB", 2));
  ns1.putNode (*new GPNode(  FEND,      "FEND", 1));
  ns1.putNode (*new GPNode(  TVAR1,     "TVAR1"));
  ns1.putNode (*new GPNode(  TVAR2,     "TVAR2"));
  ns1.putNode (*new GPNode(  TVAR3,     "TVAR3"));
  ns1.putNode (*new GPNode(  TVAR4,     "TVAR4"));
  ns1.putNode (*new GPNode(  TVAR5,     "TVAR5"));
  ns1.putNode (*new GPNode(  TVAR6,     "TVAR6"));
}
```

D.7. Nodeset 2 description

```
////////////////////////////////////////////////////////////////////////
// Create function and terminal set
// For experiments 3 and 4
////////////////////////////////////////////////////////////////////////
void createNodeSet (GPAdfNodeSet& adfNs)
{
  // Reserve space for the node sets
  adfNs.reserveSpace (1);

  // Now define the function and terminal set for each ADF and place
  // function/terminal sets into overall ADF container
  GPNodeSet& ns1=*new GPNodeSet (12);
  adfNs.put (0, ns1);

  // Define functions/terminals and place them into the appropriate
  // sets.  Terminals take two arguments, functions three (the third
  // parameter is the number of arguments the function has)

  ns1.putNode (*new GPNode(  FSTART,    "FSTART",   2));
  ns1.putNode (*new GPNode(  FROUTE,    "FROUTE",   2));
  ns1.putNode (*new GPNode(  FDBREAD,   "FDBREAD",  3));
  ns1.putNode (*new GPNode(  FEQ,       "FEQ",      3));
  ns1.putNode (*new GPNode(  STRSUB,    "STRSUB", 2));
  ns1.putNode (*new GPNode(  FEND,      "FEND", 1));
  ns1.putNode (*new GPNode(  TVAR1,     "TVAR1"));
  ns1.putNode (*new GPNode(  TVAR2,     "TVAR2"));
  ns1.putNode (*new GPNode(  TVAR3,     "TVAR3"));
  ns1.putNode (*new GPNode(  TVAR4,     "TVAR4"));
  ns1.putNode (*new GPNode(  TVAR5,     "TVAR5"));
  ns1.putNode (*new GPNode(  TVAR6,     "TVAR6"));
}
```

D.8. Nodeset 3 description

```
////////////////////////////////////////////////////////////////////////
// Create function and terminal set
// for experiment 5 using reduced complexity nodes
////////////////////////////////////////////////////////////////////////
void createNodeSet (GPAdfNodeSet& adfNs)
{
  // Reserve space for the node sets
  adfNs.reserveSpace (1);

  // Now define the function and terminal set for each ADF and place
  // function/terminal sets into overall ADF container
  GPNodeSet& ns1=*new GPNodeSet (9);
  adfNs.put (0, ns1);

  // Define functions/terminals and place them into the appropriate
  // sets.  Terminals take two arguments, functions three (the third
  // parameter is the number of arguments the function has)

  ns1.putNode (*new GPNode(  READMSG,    "READMSG",   2));
  ns1.putNode (*new GPNode(  SENDMSG,    "SENDMSG",   3));
  ns1.putNode (*new GPNode(  STRSUB,     "STRSUB",    2));
  ns1.putNode (*new GPNode(  TVAR1,      "TVAR1"));
  ns1.putNode (*new GPNode(  TVAR2,      "TVAR2"));
  ns1.putNode (*new GPNode(  TVAR3,      "TVAR3"));
  ns1.putNode (*new GPNode(  TVAR4,      "TVAR4"));
  ns1.putNode (*new GPNode(  TVAR5,      "TVAR5"));
  ns1.putNode (*new GPNode(  TVAR6,      "TVAR6"));
}
```

```
#!/usr/local/bin/tree_wish -f
# -*-Tcl-*-
################################################################################
#
#   ####   #    #   ####   #    #   ####   #####    #####  #####   ######  #####
#  #       #    #  #    #  #    #  #    #  #    #     #    #    #  #       #
#   ####   ######  #    #  #    #  #       #    #     #    #    #  #####   #####
#       #  #    #  #    #  # ## #  #  ###  #####      #    #####   #       #
#  #    #  #    #  #    #  ##  ##  #    #  #          #    #   #   #       #
#   ####   #    #   ####   #    #   ####   #          #    #    #  ######  #####
################################################################################
# This script parses a GP data file and produces an X tree
# Based on the dirtree demo from Alan Brightons tree package
#
################################################################################

option add *highlightThickness 0

################################################################################
# create a canvas with horizontal and verical scrollbars in the
# given frame with the given name
################################################################################
proc MakeCanvas {frame canvas} {
    set vscroll [scrollbar $frame.vscroll -command "$canvas yview"]
    set hscroll [scrollbar $frame.hscroll -orient horiz -command "$canvas xview"]
    set canvas [canvas $canvas \
                    -xscrollcommand "$hscroll set" \
                    -yscrollcommand "$vscroll set"]
    pack $vscroll -side right -fill y
    pack $hscroll -side bottom -fill x
    pack $canvas -fill both -expand 1
    bind $canvas <ButtonPress-2> "$canvas scan mark %x %y"
    bind $canvas <B2-Motion> "$canvas scan dragto %x %y"

    return $canvas
}



################################################################################
# layout the components of the given node depending on whether
# the tree is vertical or horizontal
################################################################################
proc LayoutNode {canvas tree dir} {
    set text $dir:text
    set bitmap $dir:bitmap

    if {[$tree cget -layout] == "horizontal"} {
        scan [$canvas bbox $text] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $bitmap -anchor se
        $canvas coords $bitmap $x1 $y2
    } else {
        scan [$canvas bbox $bitmap] "%d %d %d %d" x1 y1 x2 y2
        $canvas itemconfig $text -anchor n
        $canvas coords $text [expr "$x1+($x2-$x1)/2"] $y2
    }
}

set  uniq 0
set  SPC ""
set  LB   "<"
set  RB   ">"

################################################################################
# add the given node to the tree
#
# Args:
#  canvas  - tree's canvas
#  tree    - the tree
#  parent  - name of parent node
#  dir     - name of new node being added
#  text    - text for tree node label (last component of name)
################################################################################
proc AddNode {canvas tree parent dir text} {
    global dirtree
    global uniq
```

```
    global SPC
    global LB
    global RB

    set temp [string trimleft $text 0123456789]
    set text $temp
    set font $dirtree(font)
    set font $dirtree(boldfont)
    set cnt  "$LB$uniq$RB"

    $canvas create oval -20 -10 30 20  -tags $dir -fill grey
    $canvas create oval -25 -15 25 15  -tags $dir -fill white
    $canvas create text 0 10 -font $font -text $cnt -tags $dir
    $canvas create text 0 0 -font $font -text $text -tags $dir
    set line [$canvas create line 0 0 0 0 -tag "line"]
    $tree addlink $parent $dir $line -border 2

    set x1 [$canvas coords $dir]

}

set  loc 0

##########################################################################
# GetToken performs a simple (inneffeicient) lexical analysis of the expression
# return either a ( ) or a string of alpha numeric chars
##########################################################################
proc GetToken {} {
global loc
global expr
global uniq

set result " "

# Get a char and see if it is a parenthesis.
    while {$result == " "} {
        set result [string index $expr $loc]
        incr loc
    }
    if {$result == "("} {
        return [string trim $result]
    }
    if {$result == ")"} {
        return $result
    }
# Not a parenthesis, so get as many chars as we can and make a string
# token, not forgetting to 'put back' and non alphanumeric characters we find

    set tok $result

    while {$loc < [string length $expr] } {
        set result [string index $expr $loc]

        if {$result == ")"} {
            set temp "$uniq$tok"
            incr uniq
            set tok $temp
            return $tok
        }
        if {$result == "("} {
            set temp "$uniq$tok"
            incr uniq
            set tok $temp
            return $tok
        }
        if {$result == " "} {
            incr loc
            if {$tok != "" } {
                set temp "$uniq$tok"
                set tok $temp
                incr uniq
                return [string trim $tok]
            }
        }
        incr loc
        set temp "$tok$result"
        set tok $temp
```

```
    }
    set temp "$uniq$tok"
    set tok $temp
    incr uniq
    return [string trim $tok]
}

################################################################################
# Main procedure to tie it all together
################################################################################
proc ListGP {canvas tree root} {
    global stack
    set t "xx"
    while {$t != ""} {
        set t [GetToken]
        if {$t == "("} {
            set t [GetToken]
            AddNode $canvas $tree $root $t $t
            ListGP $canvas $tree $t
        } elseif {$t == ")"} {
            return
        } else {
            AddNode $canvas $tree $root $t $t
        }
    }
}

################################################################################
# Define the graphic scafolding
################################################################################
wm geometry . 400x275

set canvas [MakeCanvas . .c]
set tree [tree $canvas.t -layout vertical]

set dirtree(font) -Adobe-Helvetica-Medium-R-Normal--*-100-*
set dirtree(boldfont) -Adobe-Helvetica-Bold-R-Normal--*-100-*

button .print -text Print
pack .print

################################################################################
# The print method outputs a postscript rendition of the tree
################################################################################
bind .print <Button-1> {
.c postscript -pagewidth 6.0i -file tmp.ps
}

.c configure -background white

################################################################################
# Read the input stream ready for processing
################################################################################
gets stdin expr

#Get the first two tokens as the root of the tree
set t [GetToken]
set t [GetToken]
set root $t
AddNode $canvas $tree {} $t $t

ListGP $canvas $tree $root

################################################################################
# Run the main procedure to generate the tree and display it in a window
################################################################################
update idletasks
```

BIBLIOGRAPHY

[ABK96] Andre D, Bennett III and Koza J R. *Discovery by Genetic Programming of a Cellular Automata Rule that is Better than any Known Rule for the Majority Calssification Problem.* In Koza J R, Goldberg D E, Fogel D, and Riolo R. (Eds). Genetic Programming 1996: Proceedings of the First Annual Conference, pp. 3-11, July 28-31, 1996, Stanford University. Cambridge, MA. MIT Press.

[AT96] Andre, David., and Teller, Astro. *A Study in Response and the Negative Effects of Introns in Genetic Programming.* In Proceedings of the First Annual Conference : Genetic Programming 1996, pp. 12-30, MIT Press, 28-31 July 1996

[AK96] Angeline, P. and Kinnear, K. (Eds). (1996*). Advances in Genetic Programming.* Volume II. Cambridge, Massachusetts: The MIT Press.

[Ang97a] Angeline P. *Subtree Crossover: Building Block Engine or Macromutation?* pp 9-17 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Ang97b] Angeline P. *An Alternative to Indexed Memory for Evolving Programs with Explicit State Representations.* pp 423-430 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA

[ASS97] Aiyarak P, Saket, A.S and Sinclair, M.C., *Genetic Programming Approaches for Minimum Cost Toplogoy Optimisation of Optical Telecommunications Networks*, Proc. IEE/IEEE Intl. Conf. On Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA '97), University of Strathclyde, Glasgow, September 1997, pp.415-420.

[Ban93] Banzhaf W. *Genetic programming for pedestrians.* In Forrest S. Ed., proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93. Page 628, University of Illinois at Urbana-Champaign, 17-21 July, 1993. Morgan Kaufmann.

[BFN96] Banzhaf W., Francone F., and Nordin P. *The Effect of Extensive Use of the Mutation Operator on Generalization in Genetic Programming Using Sparse Data Sets.* In Goos G., Hartmanis J., and van Leewen J. (Eds.) Parallel Problem Solving from Nature: Proceedings International Conference on Evolutionary Computation, The 4th Conference on Parallel Problem Solving from Nature, Berlin, Germany, September 22-26, 1996, Springer-Verlag.

[BNO97] Banzhaf W, Nordin P, and Olmer M. *Generating Adaptive Behaviour for a Real Robot using Function Regression within Genetic Programming.* pp 35-43 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Ben96] Bennet, Jeremy, P., *Introduction to Compiling Techniques.* Second Edition, 1996. McGraw-Hill Publishing Company, England.

[BM98] Bennett J. Owen H., and Martin P. *A Programming Language to assist service creation.* Proceedings of the 7th IEEE Intelligent Network Workshop. Bordeaux, May 11th 1998. IEEE Press

[BG95] Bergadano F, and Gunetti D. 1995. *Inductive Logic Programming: From Machine Learning to Software Engineering.* MIT Press, 1995.

[Bli96] Blickle T. *Evolving Compact Solutions in Genetic Programming: A Case Study.* In Voight H., Ebeling W., Recenberg I., Schwefel H., (Eds.): Parallel Problem Solving from Nature IV. Proceedings of the International Conference on Evolutionary, Berlin, September 1996. LNCS 1141, pp. 564-573, Heidelberg. Springer-Verlag.

[Boe81] Boehm B. W. *Software Engineering Economics.* 1981 Prentice Hall

[BJ97] Boulton C., Johnson W., and Prince M. 1997*A Persoanl Number Service for British Telecom. (BT OneNumber).*. Unpublished paper by GPT Limited

[Bri97] Brighton, Allan. *Tk Tree Widget. Version 4.2*
ftp://mirror.neosoft.com/pub/tcl/alcatel/extensions/
Visited December 26 1997

[CY97] Clack T, and Yu T. *Performance Enhanced Genetic Programming.* In Angeline P., Reynolds R., McDonald J., and Eberhart R., Eds., Proceedings of the sixth conference on Evloutionary Programming, Volume 1213 of Lecture Notes in Computer Science, Indianapolis, Indiana, USA, 1997, Springer-Verlag.

[Dar1859] Darwin, Charles. *On the origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* 1st Edition. 1859.

[Dav93] Davis A M. *Software Requirements; Objects, Functions and States*. 1993. Prentice Hall

[DY97] Deakin Alan and Yates Derek F. *GP Tools Available on the Web: A first Encounter* Pp 420 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Ebe98] Eberhagen S. *Considerations for a successful introduction of Intelligent Networks from a marketing perspective*. In the Proceedings of the 5th Intenational Conference on Intelligence in Networks, Bordeaux, France. 13/15 May 1998. Adera, France.

[GR94] Gathercole C and Ross P. *Dynamic Training Subset Selection for Supervised Learning in Genetic Programming.* Davidor Y, Schwefel H and Manner R (Eds) Parallel Problem Solving from Nature III, Jerusalem, 9-14 October 1994. Springer-Verlag.

[GR97] Gathercole C and Ross P. *Small Populations over Many Generations can beat Large Populations over Few Generations in Genetic Programming.* Pp 111-118 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Gep98] Geppetto GP system. http://ferarri.snu.ac.kr/~madduk/genetic/impl/gp-systems/geppetto/V20 Last visited 21st Aug. 1998.

[GD91] Goldberg D. E., and Deb K. 1991. *A comparative analysis of selection schemes used in genetic algorithms*. In Rawlins G. (Ed), Foundations of Genetic Algorithms. Morgan Kaufmann.

[GKH95] Goldberg, David E, Kargupta Hillol, Horn Jeffrey and Cantu-Paz Erik. *Critical Deme Size for Serial and Parallel Genetic Algorithms.* IlliGAL Report No. 95002. January 1995

[GO98] Goldberg, David E., and O'Reilly, Una-May. *Where Does the Good Stuff Go, and Why? How Contexual semantics influences program structure in simple genetic programming*, in Banzhaf W., Poli R., Schoenauer M., and Fogarty T.C., (Eds.): First European Workshop, EuroGP'98, Paris, France, April 1998 Proceedings. LNCS 1391, Springer-Verlag.

[Gpe98] Gpeist GP system.
http://corvo.cpgei.cefetpr.br/EC/GP/src/GPEIST4.tar.gz
Lst visited 21st Aug. 1998.

[GPMAIL] Archive online at:
http://adept.cs.twsu.edu/~thomas/gpmail.html
Visited May 6th 1998

[HB96] Harris C. and Buxton B. *GP-COM: A distributed, Component-Based Genetic Programming System in C++*. Research Note RN/96/2, UCL, Gower Street, London, WC1E 6BT, UK, January 1996.

[Hol92] Holland, John, H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975. Revised 2nd edition 1992 from the MIT press.

[Hlb90] Holub, Alan, I., *Compiler Design in C*. 1st edition. Prentice Hall.

[HWSS95] Haynes T., Wainwright R., Sen S., and Schoenefeld D., *Strongly Typed Genetic Programming in Evolving Cooperation Strategies*. In Eshelman L., (Ed) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pages 271-278, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Iba96] Iba, Hitoshi. *Random Tree Generation for Genetic Programming*. In Goos G., Hartmanis J., and van Leewen J. (Eds.) Parallel Problem Solving from Nature: Proceedings International Conference on Evolutionary Computation, The 4th Conference on Parallel Problem Solving from Nature, Berlin, Germany, September 22-26, 1996, Springer-Verlag.

[Itu94a] ITU-T Q.1211. *Introduction to Intelligent Networks CS-1* 1994.6

[Itu94b] ITU-T Q1.1214. *Distributed Functional Plane for Intelligent Network CS-1* 1994.

[JCC92] Jefferson D, Collins R, Copper C, Dyer M, Flowers M Karf R, Taylor C and Wang *A. Evolution as a theme in Artificial Life: The Genesys/Tracker System*. In Langton C et al (Eds), Artificial Life II. 1992. Addison-Wesley Publishing Company Inc.

[Kok98] Kokkonen, Kim. *Genetic Programming in Java*
http://www.pcisys.net/~kimk/gpjpp.htm
Visited 6th Feb. 1998

[Koz92] Koza, John, R. Genetic Programming, *On the Programming of Computers by Means of Natural Selection*. 1st Ed. MIT Press 1992.

[Koz94] Koza John R. *Genetic Programming II. Automatic Discovery of Reusable Programs.* 1st Ed. MIT Press, 1994

[Koz96] Koza John R. *Comments on Cross Paradigm Comparisons of Genetic Programming with existing machine learning paradigms.* http://www-cs-faculty.stanford.edu/~koza/Cross-Para-8-17-95.html Visited 19th November 1997

[KBF96] Koza J R, Bennett III, Forrest H, Andre D, Keane M. *Automated WYWIWYG design of both the topology and component values of analogue electrical circuits using genetic programming.* In Koza J R, Goldberg D E, Fogel D, and Riolo R. (Eds). Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University. Cambridge, MA. MIT Press.

[Koz97] Koza John R. *Home page.* http://www-cs-faculty.stanford.edu/~koza Visited 14th June 1997

[KABK98] Koza John R. Andre David, Bennett Forret H and Keane, Martin. *Genetic Programming III* Unpublished draft version, available on the GP MAILING LIST

[Lan97] Langdon W B. *ntrees.cc – A program to calculate size of GP random trees.* ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code Visited 14th June 1997

[Lan98] Langdon W B. *Genetic Programming and Data structures: Genetic Programming and Data structures = Automatic Programming!* 1st Edition. The Kluwer International Series in Engineering and Computer Science. Vol. 438. Kluwer Academic Publishers, Boston. 1998

[Lil98] *Lil-gp Genetic Programming System.* http://GARAGe.cps.msu.edu/software/lil-gp/lilgp-index.html Last visited 21 Aug 1998

[Luk97] Luke Sean, *Patched lil-gp Kernel.* http://www.cs.umd.edu/users/seanl/patched-gp Visited 12th September 1997.

[LS97] Luke S, and Spector L. *A comparison of crossover and Mutation in Genetic Programming.* In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference, pp. 240-248, July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Mar96] Martin, Peter, N. *Service Creation for Intelligent Networks: Delivering the Promise.* Proceedings of the 4th International Conference on Intelligence in Networks, Bordeaux. 1996. ADERA.

[Mon95] Montana, David, J. *Strongly Typed Genetic Programming.* Evolutionary Computation, Volume 3, Issue 2, pp. 199-230. Summer 1995. MIT Press

[NFB96] Nordin P., Francone F., and Banzhaf W. *Explicitly Defined Introns and Destructive Crossover in Genetic Programming.* In Angeline P., and Kinnear K., Eds., Advances in Genetic Programming 2, Chapter 6, pp. 111-134. MIT Press, Cambridge, MA, USA, 1996.

[Ous94] Ousterhout, John K. *Tcl and the Tk Toolkit.* 1st Ed. Addison-Wesley Publishing Company Inc. 1994

[Per94] Perkis, Timothy. *Stack Based Genetic Programming.* In the proceedings of the 1994 IEEE World Congress on Computational Intelligence 1994. Volume 1, pages 148-153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press

[Pri95] Pringle W. ESP: *Evolutionary Structured Programming.* Technical Report, Penn State University, Greate Valley Campus, PA, USA, 1995.

[Qur98] Qureshi, Adil. *Gpsys*
http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys.html
Visited 5th Jan. 1998

[SEL95] Sharman K., Anna I. Esparcia A., and Yun Li.
Evolving signal processing algorithms by genetic programming. In A. M. S. Zalzala, editor, First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, (GALESIA), volume 414, pages 473--480, Sheffield, UK, 12-14 September 1995. IEE.

[Sgp1998] *Simple Genetic Programming in C.*
ftp://www.aic.nrl.navy.mil/pub/galist/src/sgpc.1.0.1.tar.Z
Last visited 21st August 1993..

[SS97] Sinclair M., and Shami S. *Evolving Simple Software Agents: Comparing Genetic Algorithm and Genetic Programming Performance.* Proceedings of the second IEE/IEEE Intl. Conf. On Genetic Algorithms 'n Engineering Systems: Innovations and Applications (GALESIA '97), University of Strathclyde, Glasgow, September 1997, pp. 421-426.

[Sin94] Singleton Andy. *GPQUICK A simple Genetic Programming system in C++ Version 2*, released 2/12/94
http://corvo.cpgei.cefetpr.br/EC/GP/src/gpquick-2.1.tar.gz
Last visited 21st August 1998.

[Som96] Sommerville I. *Software Engineering*. Fifth Ed. 1996. Addison Wesley Publishers Ltd.

[TCM98] Tang L, Califf M, Mooney R. *An Experimental Comparison of Genetic Programming and Inductive Logic Programming on Learning Recursive List Functions*. Technical Report, University of Texas, Austin. Number A198-27. March 1998.

[Tel94] Teller, Astro. *Turing Completeness in the Language of Genetic Programming with Indexed Memory*. Proceedings of the 1994 IEEE World Congress on Computational Intelligence., volume 1, Orlando, Florida, USA. June 1994. IEEE Press.

[TA97] Teller, Astro and Andre David. *Automatically choosing the Number of fitness Cases: The rational Allocation of Trials*. pp 321-328 In Koza J, Deb K, Dorigo M, Fogel D.B, Garzon M, Iba H, and Riolo R (Eds.) Genetic Programming 1997. Proceedings of the Second Annual Conference July 13-16, 1997 Stanford University. Morgan Kaufmann Publishers, San Francisco, CA.

[Vie98] Vienna University of Economics *Genetic Programming Kernel*.
http://aif.wu-wien.ac.at/%7Egeyers/archive/gpk/vuegpk.html
Last vistited 21 Aug 1998.

[Wei97] Weinbrenner, Thomas, *The genetic Programming Kernel,* Version 0.5.2:
http://www..emk.e-tecknik.th-darmstadt.de/~thomasw/gp.html
Visited 12th Sept 1997