Genetic Programming for Service Creation in Intelligent Networks

Peter Martin

Marconi Communications Limited, Poole, Dorset, UK
peter.martin@marconicomms.com

Abstract. Intelligent Networks are used by telephony systems to offer services to customers. The creation of these services has traditionally been performed by hand, and has required substantial effort, despite the advanced tools employed. An alternative to manual service creation using Genetic Programming is proposed that addresses some of the limitations of the manual process of service creation. The main benefit of using GP is that by focussing on what a service is required to do, as opposed to it's implementation, it is more likely that the generated programs will be available on time and to budget, when compared to traditional software engineering techniques. The problem of closure is tackled by presenting a new technique for ensuring correct program syntax that maintains genetic diversity.

1 Introduction to Intelligent Networks

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. A simplified view of this traditional system is shown in Figure 1.

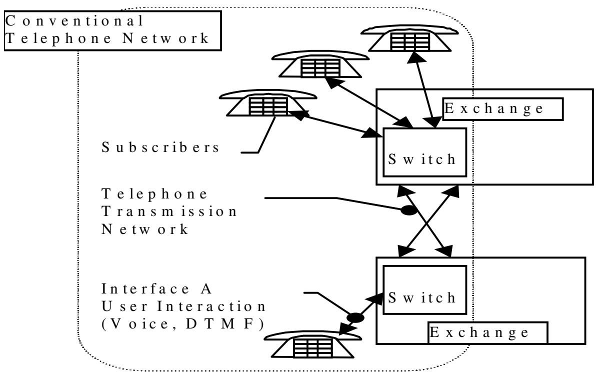

Figure 1 Traditional telephone system

The switches control the calls made by the subscribers. All the services such as ring back when free, are implemented in the switches. However, the time required to implement new services is in the order of 1 to 2 years (for example see [3]) because the services are tied closely to the switch development lifecycle.

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 alternative is to move the services off the switches onto standard computing platforms. This is the Intelligent Network (IN) solution.

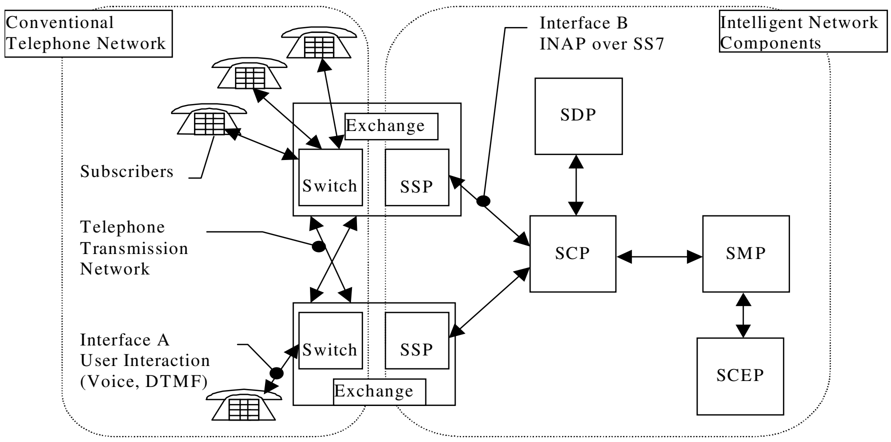

Figure 2 Basic elements of an Intelligent Network

The basic intelligent network is shown in Figure 2. Here it can be seen that the switches have been expanded to communicate with other network elements. The Service Switching Point (SSP) handles the interface between the telephone switch and the Intelligent Network. The interface uses a standardized Intelligent Network Application Part (INAP) which is carried over the standard Signaling System number 7 (SS7) network. In the Intelligent Network, the services are hosted on a Service Control Point (SCP). This is typically a number of high end UNIX servers. The SCP uses a Service Data Point (SDP) to store data, for example, numbers required to implement a FreePhone service. The system is managed by a Service management Point (SMP). Finally, to enable the rapid creation and deployment of services, a Service Creation Environment (SCE) is used to build, test and deploy services to the network.

The primary objective of Intelligent Networks (IN) then is to move the service computation from the embedded switches to readily available computers to gain the benefits of mainstream IT techniques.

A secondary aim of introducing IN is to reduce the time required to develop and deploy new services. As already mentioned, traditional switch based solutions typically require 2 years from the initial requirements being specified until the service is in operation. 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 SCE, or Service Creation Environment Function.

Experience has shown that the time required to capture the requirements and create an outline of the service is relatively short, but the time required to implement and test the low level details of complex services can be several months. A typical non-trivial service can require several hundred icons, and results in dozens of valid traversals of the graph. A means of reducing the duration of the detailed engineering phase is therefore of benefit to the network and service operators.

1.1 An Alternative approach to Service Creation

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 are still some defects found in the services themselves [2].

This work attempts to address the difficulties associated with the detailed engineering phase of service development, by means of automatically deriving an implementation from the requirements, or as Langdon [19] and others put it, by using Automatic Programming. This approach was hinted at by Boehm [5] Chap. 33 which mentions 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. 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 by Koza et al[15] and Sharman et al [25] 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 [2]. This study showed that for a complex service a team of engineers required 4.5 Man years of effort to analyse, design, code and test the service. A significant 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 according to Sommerville [26]; 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 [8] the earlier that 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 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.

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.

2 Applying GP to Service Creation

2.1 Functions and Terminals

Classical tree based GP [16] requires a set of functions which form the non-leaf nodes in the tree and terminals which form the leaf nodes of the tree. The set of functions and terminals must satisfy the closure and sufficiency properties. 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 summarizes these data types.

Table 1 Data types encountered in telephony services

Data Type	Comments
Telephone numbers	Strings of digits [0-9 # *] that can be dissected and concatenated. The string length may be up to 24 bytes.
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

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 messages, a mechanism for saving state information is required. The first approach to this requirement, Indexed Memory, was suggested by Teller [2] 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 [1] 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.

2.2 Achieving Closure

Several methods have been proposed to ensure that the closure property is maintained during initial creation and subsequent reproduction. This may be achieved in a number of ways. Firstly Koza [16] restricts the types of arguments and functions 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 [22] and Haynes et al [18] 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 [6] 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.

An alternative to the strongly typed approach 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 useful artifact of languages used by humans to help ease the burden on the programmer, by means of assisting machine interpretation. Perkis [2] 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 and biased , since the search space is constrained.

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 of different types $\boldsymbol{L}=\{\boldsymbol{d_1},\ldots\boldsymbol{d_n}\}$. The values of $\boldsymbol{L_n}.\boldsymbol{d_1}$, $\boldsymbol{L_n}.\boldsymbol{d_2}$ 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. An example of the use of this structure is when a program needs both integer and real values. Each location $\boldsymbol{L}$ would contain an integer data type and a floating point data type.

Memory M						
L_1			…	L_n		
d_1	d_2	d_3		d_1	d_2	D_3

Figure 3 Layout of Autonomous Polymorphic Addressable Memory

To support this memory architecture, the terminal set $\boldsymbol{T}$ consists of memory nodes $\boldsymbol{T}=\{\boldsymbol{TVAR_1,\ldots TVAR_n}\}$**.** Each node returns a reference to memory location $\boldsymbol{L_n}$**.** 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.

2.3 Choosing a level of abstraction

Sufficiency is a problem specific attribute. 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 $\boldsymbol{C}$ as described by Banzhaf et al [4]:

1. Icon level with attributes as terminals. This level is based on the set of functions offered to service creators using the GPT GAIN INventor ™ product [2]. 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.
2. Icon function level. This is the level used by the internal tools within GAIN INventor ™. Each icon typically makes use of between one and twenty functions. The total number of functions is around 200.
3. 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. 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.

The functions chosen for the initial experiments are shown in Table 2

Table 2. Functions for high level abstraction

FSTART	Takes two arguments. It accepts an IDP message from the SSP and the calledDN value is stored at the location returned as a result of evaluating it's first parameter.
FDBREAD	This reads a data base, using the value of the first argument as a key and placing the result in the location returned as a result of evaluating it's second parameter
FROUTE	Evaluates the first argument which is used to furnish the new routed number for the connection message.
FEND	Sends a pre-arranged end to the SSP.
STRSUB	Performs a simple string shift operation to simulate real-life number manipulations.

For the lower level abstraction, the functions were:

Table 3. Functions for low level abstraction

ReadMSG:	Accepts a message from the SSP or SDP and places the parameters into variables.
SendMSG	Takes a number of parameters and builds a message which is then sent to the SSP or SDP.
STRSUB	Operates as already described

The system supports five message types analogous to the real world Intelligent Network Application Part (INAP) and SDF operations. These are InitialDP which is generated by the SSP as a result of a trigger detection point being activated by a call, DB_REQUEST which issues a database request, DB_RESPONSE which accepts a data base response, Connect which is an instruction from the SCF to the SSP to connect party A to party B, and END which terminates a service dialogue.

2.4 The Fitness Function for Service Creation

The decision was made to measure the fitness of the GP at the external INAP interface since this is a standardized external interface as described in Q.1211 [12] and would allow the specification of services to be performed at the network level. This led to the use of Message Sequence Charts (MSCs) for deriving the fitness function. MSCs are commonly used in telecommunication system work for specifying system behaviour.

The Basic Call State Machine (BCSM) described in Q.1214 [13] is simplified, and called a Simple Call State Model (SCSM) in order to focus on the GP technique rather than being distracted by the complexities of the BCSM.

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

1. The number of correct state transitions made. Each correct transition is rewarded with a value of 100. Each incorrect transition is penalized with a value of -20. The reward and penalty values are summed. This value is called **s**.
2. 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 penalized with a value of -20. The reward and penalty values are summed. This value is called **p**. These values are then used to compute a normalized fitness value as follows:

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

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

A count is maintained of the number of correct and incorrect state transitions and correct and incorrect message parameter values to help with an analysis of the performance of GP.

2.5 Measuring performance and estimating effort

Additional information collected includes the total wall clock time taken for each run, the number of individuals processed, the number of unique individuals that were 100% fit, the number of 100% individuals at the final generation and details of the best individual of each run, including it's size.

3 Example of a Service - Complex Number Translation

We can now look at one of the experiments carried out as part of this research [21]. Number translation was chosen for this example since although it is one of the simpler services, it is also the most common service implemented using IN [9], and forms the basis of many more complex services such as Freephone, Premium Rate services and time based routing.

The experiment uses GP to evolve service logic for a number translation service where two data base lookups are required. This scenario occurs in the real world where a service requires 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.

For this experiment the population was set at 500, the number of generations was set at 200. These figures were arrived at after a number of trial runs using a wide range of values for the population size and number of generations. When this problem was run 50 times, GP created a 100% correct program 49 times. The probability of finding a correct solution at generation 200 was 72%.

Some interesting results were observed during this experiment. Firstly all 49 correct programs were different. The differences ranged from the selection of different variables to some counter intuitive program trees when compared to what a human programmer might have written. One of the less intuitive programs is shown in Figure 4. Note that the first function called is the route function, and the operation relies purely on the ordering of evaluation of function arguments, rather than a procedural sequencing more commonly found in human generated programs.

The number translation problem was performed twice, once using functions from the high level abstraction set as already described and once using functions from the lower level set. The use of the lower level abstraction system yielded solutions using fewer generations than the higher-level abstraction system. After 200 generations the probability of finding a 100% correct solution rose to 84%, compared to 74% for the high level functions. Weighed against this faster convergence is the fact for a given probability of finding a lower level abstraction, the low level functions required more time to evolve primarily due to the greater size of the program trees needed. An example of a 100% correct program is shown in Figure 5.

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) [17] for this level of functions would be beneficial since there are repeating patterns emerging. A possibility that was not explored is that the common subtrees come from a common ancestor formed early in the run.

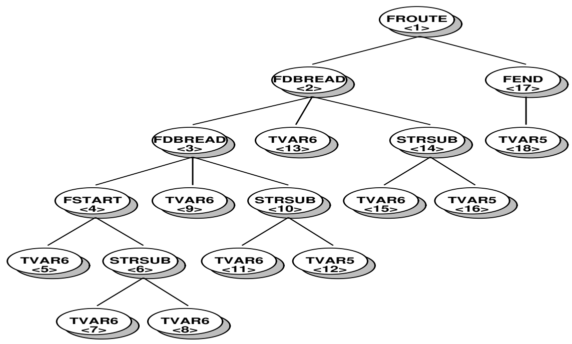

Figure 4 An example of a novel 100% correct program tree

3.1

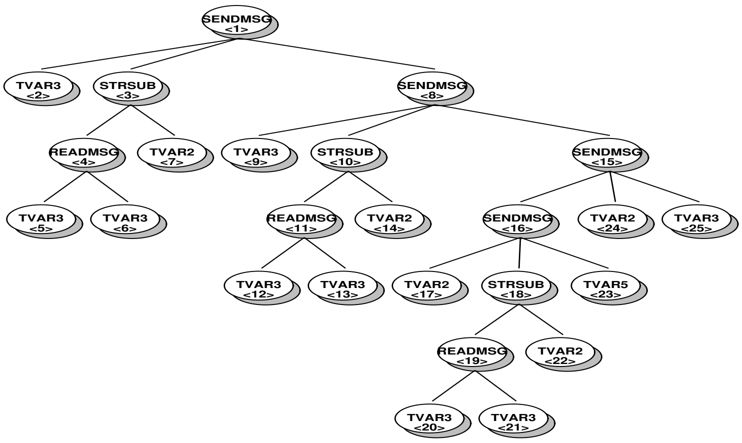

Figure 5 Example program tree using reduced complexity functions

3.2 Comparison of Performance Between High and Low Level Abstraction

Some additional measurements were taken to try to gauge the differences in the time required to find a solution and the resulting complexity of the 100% correct programs.

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. The wall clock time is significant when trying to compare automatic programming as an alternative to human programming.

Table 4 Summary of experiments and results

Average time per run (secs)	**Average Complexity of fittest**	**P(M,i) %**	**R(z)**	**ε**
44	19	72	4	2,000
71	28	82	3	1,500

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 simply the number of nodes in that individual.

4 Analysis and Discussion

During the early part of the work, considerable time was spent trying different combinations of the control parameters and the set arrived at for the experiments 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 [10] in his study on deme size, and the results presented as part of the GP kernel [28]. These and other questions raises the point made by Goldberg [11] 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 original choice of abstraction for the internal nodes gave satisfactory results, but as shown in the second experiment, 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 use of the Autonomous Polymorphic Addressable Memory System (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 [22] 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 its place in any tree. Doing this would preclude the example solution illustrated in Figure 4 that started with the ***FROUTE*** function.

4.1 Performance of GP Compared to a Human Programmer

In terms of raw performance, GP service creation compares well to a human performing the same task, with GP taking minutes to find a 100% correct individual, and a human taking around one hour to hand code a similar program and test it. However a comparison made purely on time to complete a task does not tell the whole story. In the case of GP, one of the important tasks of the fitness function is to rank how well an individual is able to solve a problem. The fitness function could also be the test case for the solution so a correct program could well be classified as 100%

tested, with the usual caveats concerning testing metrics. This is an appealing side effect from using GP.

4.2 Software Engineering Considerations

For many years automatic program generation has been a goal of software engineering. See for example [5]. GP at its current state of maturity goes some way to achieving this goal. This could be viewed as a threat to traditional software engineering, but such a narrow view misses the broader benefits of GP. GP offers an alternative to the traditional coding phase of software development, facilitating the creation of quality tested software. What remains of course are the essential activities of requirements capture and system specification.

A comment often raised when evolutionary techniques are discussed is how we can be sure that the evolved structures don't have any hidden surprises. For example, that a program may give erroneous results under a set of conditions that were not expected. Apart from the fact that programs written by humans are themselves often not free of hidden surprises, a strong argument against these objections is that every individual is tested against the fitness function. Since the fitness function in effect embodies the specification for the program, the fitness function can contain the test cases required to ensure that the evolved program performs as required.

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 [14] states '… a high level statement of the requirements …'.

This question concerning the opaqueness of programs generated using GP or other EC techniques has inspired some work to try to address the perceived deficiency. For instance Pringle [2] suggests an approach that tries to create programs that look like those produced by a human programmer, while Langdon [19] 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 [3] 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 revisiting 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' [7] (Chapter IV, 'Natural Selection').

Finally it is often remarked that understanding the evolved programs is often hard, since the programs do not always adhere to what a human programmer might consider good style. However, in a production environment it would be rare that an analytical understanding of the program is required. In any case, software engineering is currently happy to trust tools to generate code in other areas. A useful analogy here is to consider how CASE tools and high-level language optimizing compilers have replaced flow charts and hand crafted assembler code.

5 Conclusions

A new technique of achieving closure was developed that uses polymorphic data types. The principle advantage of using this approach is to facilitate a more complete search of the problem space by avoiding the selective search imposed by strongly typed techniques.

Choosing two different levels of abstraction for the function set indicated that selecting the optimum set of functions is not straightforward.

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.

GP still requires a significant amount of effort from the system designer in selecting a sufficient and appropriate set of functions and terminals, in selecting suitable run-time parameters and in fine-tuning the system during the development of useful programs.

This work has demonstrated that GP is a viable technique when applied to the problem of service creation. While there are some limitations with the present approach, ongoing research into GP is yielding better insights into the underlying theory of operation, and is delivering GP systems that can handle more complex tasks. Whether this application of GP is scalable to allow the creation of production quality services is still an open question.

6 Acknowledgements

I would like to thank Marconi Communications Limited for supporting this work and the encouragement and valuable comments from my colleagues Jeremy Bennett and Stuart Wray.

7 References

1 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

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

3 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.

4 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.

5 Boehm B. W. *Software Engineering Economics.* 1981 Prentice Hall

6 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.

7 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.

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

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

10 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

11 Goldberg, David E., and O'Reilly, Una-May. *Where Does the Good Stuff Go, and Why? How Contextual 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.

12 ITU-T Q.1211. *Introduction to Intelligent Networks CS-1* 1994.6

13 ITU-T Q.1214. *Distributed Functional Plane for Intelligent Networks CS-1* 1994.

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

15 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.

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

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

18 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.

19 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

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

21 Martin, Peter, N. *An investigation into the use of Genetic Programming Intelligent Network Service Creation.* MSc Dissertation, Bournemouth University, UK. http://www.martin21.freeserve.co.uk/gpproject.htm

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

23 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

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

25 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.

26 Sommerville I. *Software Engineering*. Fifth Ed. 1996. Addison Wesley Publishers Ltd.

27 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.

28 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