Visualisation of Genetic Algorithms

Graham Clark

EPCC–SS94–07

Visualisation of Genetic Algorithms

Graham Clark

EPCC–SS94–07

September 1994

Abstract

In the last few years, the use of genetic algorithms has increased greatly, finding application in a diverse field of search and optimisation problems. Due to their success, several tools have been developed to facilitate their implementation. One such tool, called RPL2, has been developed here at EPCC.

Unfortunately, due to the nature of genetic algorithms, locating a good solution to a reasonably sized problem will generally take some time. Also, since there are several parameters which can influence the behaviour of a genetic algorithm, both intuition, and trial and error play a part in finding the best combination.

This project attempts to address this problem, by connecting EPCC's genetic algorithm tool RPL2 with a visualisation program. The main aim is to provide an interface, allowing RPL2 to process requests from the visualiser, and facilitating the efficient communication of data back from RPL2. Many of the projects within EPCC involve GAs. Each of these could potentially benefit from having some means of visually determining the behaviour of the system.

This report details the main stages of providing this interface. This includes problems encountered and issues raised, work completed on the design, and details of the implementation, including a description of the communications which used the Message Passing Interface, MPI.

1 Introduction

In the last few years, the use of genetic algorithms has increased greatly, finding application in a diverse field of search and optimisation problems. These schemes, as the name suggests, are derived from the selection processes used by Nature itself. Due to their success, several tools have been developed to facilitate their implementation, with different levels of generality. One such tool, called RPL2 [Radcliffe and Surry, 1994a], has been developed here at EPCC. It is flexible, fast (since it supports parallel architectures), and has been used extensively on a number of applications.

Unfortunately, due to the nature of genetic algorithms, locating a good solution to a reasonably sized problem will generally take some time. Also, since there are several parameters which can influence the behaviour of a genetic algorithm, both intuition, and trial and error play a part in finding the best combination. This project attempts to address this problem, by helping construct a method of visualising genetic algorithm data.

Many visualisers exist at the moment (e.g. Explorer, AVS), which provide a high degree of flexibility in their control of data. Such a capability would be of great benefit to an RPL2 user implementing a genetic algorithm. With skilful use, the user should be able to gain new insight into the evolution of the solution space maintained by RPL2, and therefore be able to tune the algorithm in subsequent attempts to attain an even better result. Providing a graphical portrayal of a solution space will give the user intuitive and digestible feedback. An extension to this visualisation model would allow feedback from the visualiser to RPL2. In this way, the user could interactively alter parameters and therefore directly influence the evolution of the genetic algorithm.

This project focuses on connecting EPCC's genetic algorithm tool RPL2 with such a visualiser. The main aim is to provide an interface, allowing RPL2 to process requests from the visualiser, and facilitating the efficient communication of data back from RPL2. It is envisaged that this will use the new implementation of MPI to handle all message passing requirements. In a separate EPCC project, a visualiser is being constructed, which by meeting the interface specification should attain the overall goal of genetic algorithm visualisation.

This report details the main stages of this project, the problems and issues raised, and work completed on the implementation. Some background information is given first, in Section 2. Next some visualisation problems are discussed in Section 3. This leads on to the design of the interface in Section 4, and implementational details in Section 5. Some examples of the use of the interface in simple visualisation problems are given in Section 6, and finally further work is suggested in Section 7.

2 Background

In this section we examine some concepts that are necessary in understanding this project. Also, some technical detail is given about the software used, and the ideas used in its construction. Section 2.1 gives a short introduction to genetic algorithms, the concept underpinning all of the work described here. This leads on to Section 2.2 which describes some relevant aspects of RPL2, and Section 2.3 which explains the architecture of RPL2 with respect to the execution of a GA.

2.1 Genetic Algorithms

Since this project is involved in the visualisation of genetic algorithms, it would be instructive to first describe these methods. As mentioned, this novel approach is analogous with the process of natural selection observed in Nature itself. In Nature, the fittest individuals generally survive. This approach is mapped onto the general task of optimising a given problem. A typical approach might be as follows :

1. *Seed the Initial Population :* a genetic algorithm implementation will maintain a collection of solutions, or more generally *genomes*, for the problem at hand. The first thing that must be done is

to generate the collection of genomes, perhaps choosing at random, or perhaps using a heuristic. This means that each genome provides a solution to the given problem, although some may be "better" than others - this leads to what is called the "fitness" of a genome.

2. *Evaluate the Genomes :* this is the first step in the evolution of the population. Each solution, however complex its representation, must be capable of evaluation to yield its fitness. For example, if the very simple problem was to generate as large a number as possible, a solution might be 1000, and it might have a direct evaluation to yield a fitness of 1000. Therefore, a genome evaluated to 2000 will be deemed fitter than the first. This evaluation may be very complex and is dependent on the representation (for a complex example, see [Radcliffe and Surry, 1994a, Section 6.1]). The genetic algorithm package will then use the relative fitnesses of these genomes to implement "the survival of the fittest" strategy.

3. *Recombination :* the next step is to select a pair of genomes and "mate" them to produce a progeny. The probability that a genome is selected is usually based on its fitness — the higher the fitness, the better the chance. "Mating" two abstract genomes is maybe an alien concept — this simply means that a part of each genome is selected, and the two parts joined. For example, if the genome was a binary string, recombination might involve taking the different halves of each string, and concatenating them to produce the child.

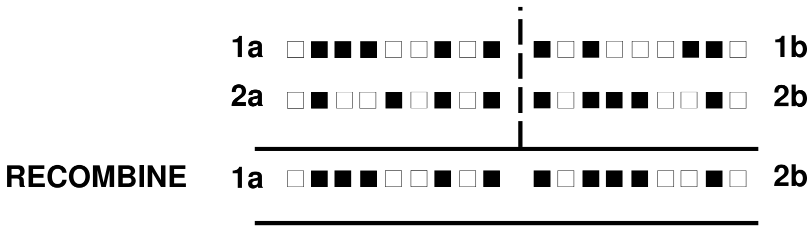


Figure 1: Example of Recombination

This process is crucial to the genetic algorithm and helps to ensure that "desirable" features combine in a child to produce a genome fitter than either of its parents.

4. *Mutation :* the next step is also analogous to a natural phenomenon. The child genome produced by the process above may undergo a minor alteration, according to some usually small probability. Using the example above of binary strings, this may mean taking one of the binary digits and flipping its value. This ensures that the population does not converge to a particular solution prematurely. It may be the case that without mutation, a desirable feature of a solution does not exist in the entire population. In this instance, the feature will never appear — recombination will never produce it. Mutation can help alleviate this problem, by sporadically introducing a random element.

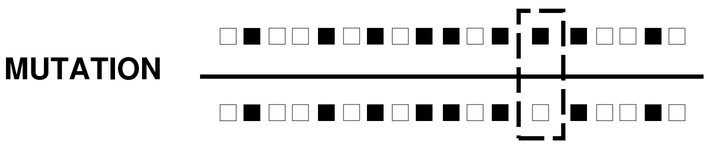


Figure 2: Example of Mutation

5. *Insertion :* the final step in the loop is to insert the progeny into the existing solution space; the loop will then restart from Evaluation above.

Many variations on this strategy exist. These include positively beneficial ones such as structured populations (See Section 2.2), and highly contentious ones such as incest prevention.

2.2 The Software - Reproductive Plan Language

Under the auspices of the EPCC Summer Scholarship Programme, and the Department of Computer Science's M.Sc. programme, the Reproductive Plan Language (RPL [Russo, 1991]) was developed, as a programming tool to allow the implementation of genetic algorithms. This language supported only limited population models, but made use of a parallel architecture and provided a very general framework for designing and implementing GAs.

In conjunction with British Gas, this has now been superseded by RPL2 [Radcliffe and Surry, 1994b]. While maintaining the generality of RPL, RPL2 now supports a richer syntax, provides a wider range of structured population models, and can make more varied use of parallelism on an appropriate architecture. This general tool is now widely used in EPCC itself.

Of particular note is RPL2's support for structured population models. There is empirical evidence that these give better performance (in terms of a generally fitter best solution) than a traditional unstructured or *panmictic* population, for an equal number of iterations of the genetic algorithm[1]. Whereas with panmictic populations there is no concept of a genome's location, this is introduced for structured populations, with attendant consequences e.g. two solutions may only be combined if they are in the same locale. This ties in even more closely with Nature.

This very general population framework raises issues of pictorial representation of the solution space that must be addressed by a visualisation tool. Two examples of structured populations are given below. Figure 3 also shows an example of what is called a *Fine Grained* population model. In this model each solution has a notion of its location in the solution space, and also a notion of its neighbours. Typically, recombination will take place between 2 solutions in a deme, a neighbourhood of the solution space (marked by the dotted line). Providing data from RPL2 with respect to this model was addressed in this project. Figure 3 shows a more complicated model in which *islands* of populations evolve largely independently of each other. Support is provided in RPL2 for migration of solutions between islands — this prevents "stagnation" of the genome pool.

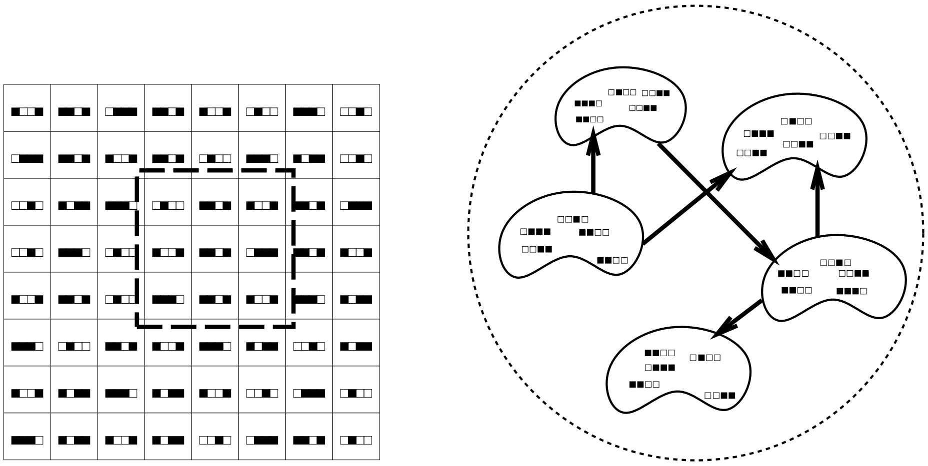


Figure 3: A Fine-Grained Population and the Island Model of Population

RPL2 is programmed in a language very similar in syntax to Pascal. These programs are named "reproductive plans". They describe the genetic algorithm the user wishes to implement, typically using standard GA operators. Because of the generality of the language, these plans can be written in a very flexible way. There is no need to adhere to a typical GA cycle as in Section 2.1.

RPL2 supports a number of datatypes in these plans. These are

- `genome` — a datatype representing a member of the GA population. Typically a set of genomes will be evolved over a number of generations.

[1] In the parlance of genetic algorithm programming, the number of iterations is known as the *generation*

- `gstack` — a variable of this type is notionally a stack of genomes. This is useful for collecting genomes together into populations e.g. for selection operators
- `int, real, bool, string` — these are defined as one would expect

2.3 Architecture

Since this project required changes to the existing RPL2 framework, the current architecture is described here. This application was built to take advantage of parallelism; there is a very natural synergy between parallelism and genetic search. RPL2 now supports two forms of work decomposition, those being the "task farm" and "regular domain decomposition"[2]. This section will explain how these methods are employed, while detailing the steps used by RPL2 in implementing a genetic algorithm.

When an RPL2 plan is submitted for execution, it is compiled into an abstract machine language by one process. Therefore, RPL2 interprets this abstract code, rather than compiling a plan into machine language and executing it directly.

This may at first sight seem like an obvious inefficiency, but the speed loss due to interpretation is not usually significant. This is because in the genetic algorithm cycle (see Section 2.1), the bottleneck in real world applications is generally the Evaluation step. Now RPL2 is flexible in that it can work with any programmable genome representation, but that is not to say there are direct language facilities to create these. In order to use new representations, the user will have to provide some C functions of his/her own, which can be accessed by RPL2 as binary code, in library form. It is clear that an evaluation function, for a user-defined representation, will have to be provided by the user, so any inefficiency here is the user's own. This is a side-effect of using a complex representation for the real-world problem. Therefore, during the evaluation step, RPL2 will simply pass control over to this user-written evaluation function ; unfortunately this code, by its nature, is usually time-consuming.

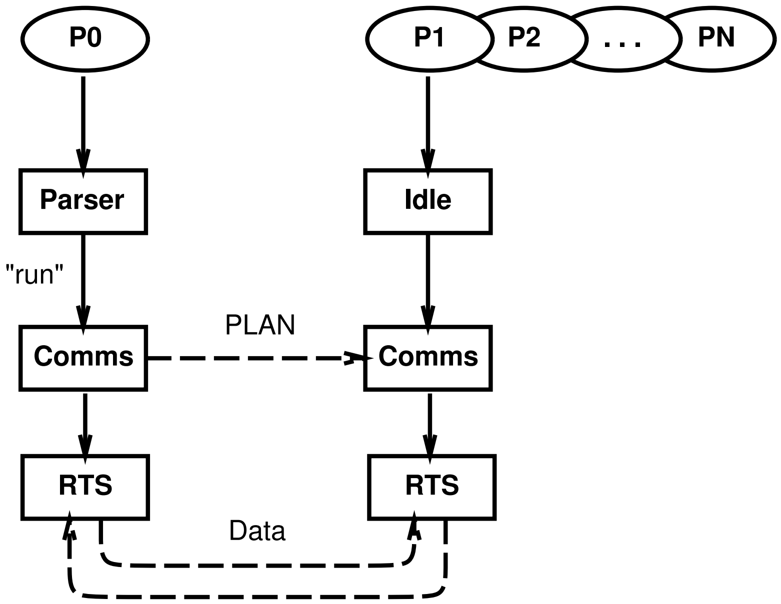


Figure 4: Structure of RPL2

After "compilation", a distinction in operation must be made, depending on whether the task farm or regular domain decomposition parallel model has been chosen by the user. The model most relevant to this project is regular domain decomposition.

[2] Note that both these facilities are implemented using PUL libraries, respectively PUL-TF and PUL-RD

On a parallel architecture, the compiled plan, in abstract machine- language form, is transmitted from the compiling process to every other process involved in computation (see Figure 4). In this parallel mode of operation, each process is responsible for its own set of genomes or solutions i.e. one of the first jobs done is to spread the solution space as equally as possible over the processors. This model is excellently matched to the operation of RPL2, especially using structured populations. Consider each processor owning an "island" of the solution space as in Figure 3. Evolution can occur independently on each processor with limited communication overhead — this only occurs when a genome must migrate to another island. Synchronisation may not occur at all. Communication is portrayed in Figure 4 as arrows running from RTS to RTS (run time system).

Another model, less relevant to this project, is that of the Task Farm. This is useful in applying an operation to a collection of genomes, for instance evaluation. In this system, the plan is executed by one processor only, who will "farm" jobs out to worker's as they request, i.e. one processor will be given the job of evaluating a genome, will complete it, and will communicate the data to the sink (in this case, the farmer process). The farming continues until all genomes have been operated on. This model is not too important to the project, but is taken into account in work described later.

2.4 Visualisation

At present, no satisfactory visualiser exists that is capable, or set- up for interpreting data produced by RPL2. Many visualisers exist, and it may be the case that one of these can be configured to provide the flexibility required. The process of building a visualiser is the focus of a current project within EPCC, and has received external funding. This project. in contrast, aims only to provide a standard interface between RPL2 and such a visualiser.

For the purposes of this project, it was possible to use one of the existing visualisers in a "dirty" fashion to view preliminary data produced and for testing purposes. This was unsatisfactory though, not least because of the inconvenience of using a system not tailored precisely to the project's needs. The example results shown in this document were instead produced in a different way ; this involved using a popular image display program, as described in Section 6.

3 Visualisation Problems

In order to eventually provide a useable and flexible visualisation tool for RPL2, there are problems to be avoided and areas on which to improve. Two concerns are identified here, these being

- Avoiding the generation of extremely large data sets
- Improving upon RPL2's existing monitoring and statistics facilities

A solution to the first has been devised but not implemented. The second problem is addressed by the provision of a general data interface to RPL2, the aim of this project.

3.1 Large Data Sets

When executing a genetic algorithm, it is common to use an extremely large pool of genomes in order to provide the greatest wealth of good "genetic" traits. This helps to provide the best solution to the problem since it is more likely that any specific good trait will exist in the collection of genomes. As a consequence, when using a structured population model (see Figure 3), any graphical method of visualising the space can not be quickly interpreted by a human. This is illustrated in Figure 5 — this a screenshot taken from a visualisation program called `Prism`[3].

[3] Unfortunately `Prism` can only visualise data generated by code written for the Connection Machine

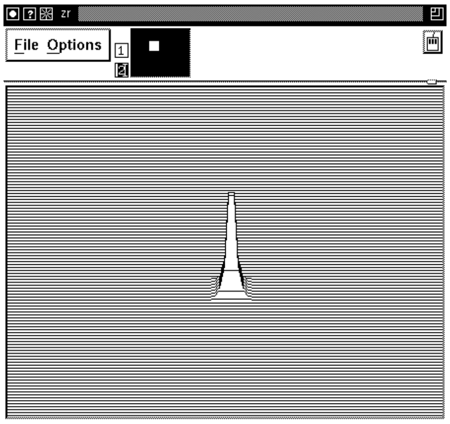


Figure 5: Visualisation using `Prism`

It shows a large 2-dimensional array of integers, visualised as a "surface". The area to note here is the black rectangle at the top of the picture — this represents the entire 2-dimensional space, whereas the smaller enclosed white rectangle represents the extent of the surface image shown. The area of interest, for argument's sake, is the peak shown. It can be seen that the co-ordinates of the peak occupy only a very small area of the entire space; the problem is locating the peaks.

When studying the evolution of a genetic algorithm, it would prove useful to focus on only certain areas of the genome space. This would allow observation of small niches of the population, and, for example, the study of the development of a particular genetic property. Thus the ability to restrict the user's view to a particular subset of the genome space will :

- Facilitate interpretation of the evolutionary processes
- Allow fine studying of subsets of the population
- Reduce communications overhead, thus improving the performance of RPL2 under visualisation

This data "restriction" is described later.

3.2 Improving Data Generation

At present, RPL2 is capable of a limited form of monitoring, although it does not support visualisation. The way this is done is through statistics functions, which gather particular details about the population, and output the data to the screen. An example of this is shown in Figure 6 — here RPL2 is executing a genetic algorithm plan called `ga1-1.rpl`.

It is clear that this form of data output will not provide a great deal of assistance to a user trying to understand patterns of evolution. In the example given, the fields represent the generation, the fittest genome in the population and the elapsed time. Unfortunately this says nothing about how evolutionary behaviour is changing or differing over the population. The only way to observe this is by using graphical methods, and allowing the user the flexibility to view data how he/she wishes. This project is a step on the ladder to flexible visualisation, by providing an interface that can handle general user requests for data.

4 Project Design

This section of the report details the steps taken in the design process for this project. It includes a loose specification, and goes on to describe a flexible interface that avoids the problems mentioned above.

```
gcla@ssp$ rpl2 ga1-1.rpl
0       0.5969449817    0
1       0.6019692567    0.2
2       0.6412556147    0.416667
3       0.7361730357    0.616667
4       0.8127060626    0.833333
5       0.7467729774    1.03333
6       0.7264560056    1.23333
7       0.5872495327    1.43333
8       0.5872495327    1.61667
9       0.7231052651    1.81667
10      0.813079136     2.01667
```

Figure 6: RPL2 Statistics

4.1 Informal Specification

This project was not formally specified, in that no document exists that precisely defines the required system. This gave a lot of scope for arriving at a suitable design for the interface. Despite this, there are still some requirements that had to be satisfied. These are given below.

4.1.1 Conceptual View

It was mentioned earlier that the aim of this project was to provide an interface between the genetic algorithm tool RPL2 and a visualiser. This interface should be flexible in that requests for any reasonable data held by RPL2 can be met, and the data communicated to the visualiser in a useful form. It is useful to see how the interface fits in with what exists already — this is shown pictorially in Figure 7.

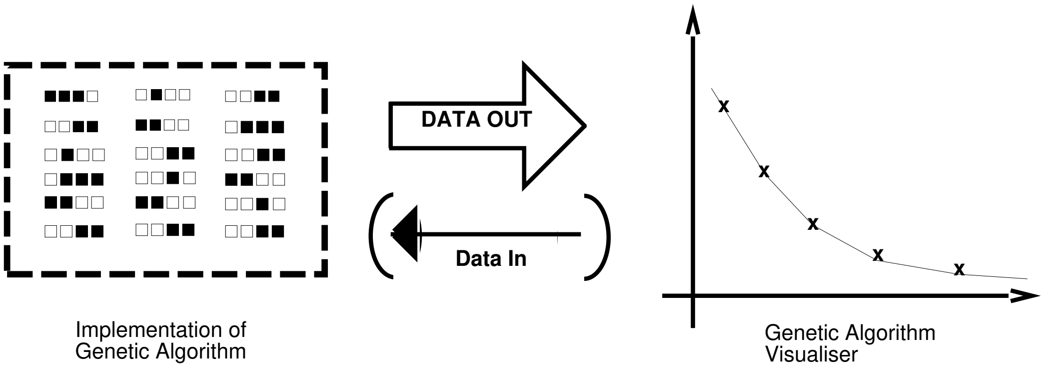


Figure 7: The Interface

The arrows represent the flow of data, generally from RPL2 to the visualiser. An extension to this model of one-way data flow is to augment the visualiser with more general capabilities, such as those found in a general purpose debugging package e.g. `dbx`. Some examples of these features would be the interactive changing of data held by RPL2, and the ability to place *checkpoints* in a genetic algorithm. This ability is shown in Figure 7 by a bracketed arrow. Since this generalisation was deemed important, the interface has been designed with it in mind.

4.1.2 Requirements of the Interface

These were specified as follows :

Functional : the interface should support

- communication of RPL2 data types, importantly `genomes`
- tracing and debugging features — breakpoints and single-stepping[4] were definitely required.
- multiple "threads" — this would allow the user to be visualising more than one set of data at a time, e.g. watching a set of genomes evolve, while keeping an eye on the generation counter.

Performance : obviously, the interface should impact as little as possible on RPL2's performance. To this end, it was deemed necessary to be able to completely switch off any visualisation mechanisms if the user desired. Therefore, in this mode, the performance of RPL2 would not be affected in any way. Such a facility was devised.

4.1.3 User Interaction

Since at the beginning of this project no suitable visualisation tool existed, a method of requesting data from RPL2 had to be specified. It was decided that a text-based system would be used, similar in style to a command line interface[5]. The user would type commands into this shell using a defined grammar, and in this way unambiguously request data from RPL2. By using different commands of the grammar the user could then instruct RPL2 in other ways e.g. to place a breakpoint at a designated point in its genetic algorithm plan. The disposal of the data sent back to the interface by RPL2 was unspecified at this point. In fact, to provide some degree of testing, appropriate received data is written out as a sequence of files, in a simple graphics format. These can then be viewed using an image display program. The results of this method are shown in Section 6.

4.2 The Grammar Defined

In this section, I have listed the commands available to the user through the visualiser interface. These are named usually by a single lowercase word. Following the token may be a `parameter`. The grammar was designed according the requirements above, and the tokens fall into four categories :

- "Commands" : types of request the user can make
- "Triggers" : when RPL2 responds to the command i.e. when the command is triggered
- "Modifiers" : these further alter when the command is triggered, or what data RPL2 sends back through the interface
- Miscellaneous

For a full Backus-Naur definition of the grammar, see Appendix A.

4.2.1 Commands

This category can also be split into two logical sets —

- "Data" related commands : these interactively access RPL2 data (e.g. to request data to visualise, the crux of the project)
- "Plan" related commands : these allow the user control over the execution of the current genetic algorithm plan.

[4] In effect, placing a temporary breakpoint after each instruction, allowing the user to "walk" through a genetic algorithm plan
[5] The style of input used in UNIX shells

Data related :

1. *send* `variable` :

This command will request to RPL2 that it send back the data associated with `variable`. Typically this data will then be visualised. This command also takes an optional trigger, defining at what point RPL2 should begin transmission.

2. *change* `variable` *to* `expr` * :

This command instructs RPL2 to change the value of `variable` to the result of RPL2's evaluation of `expr`. This is only defined for scalar variables at the moment, since `expr` will evaluate to a scalar value. It is not well defined to assign a scalar value to a vector, such as a multi-dimensional genome. This command takes an optional trigger.

Plan related :

3. *synchronise* :

This gives the user the ability to keep each parallel process of RPL2 in synchronisation. Due to the nature of certain communications in RPL2, it is often the case that processes are synchronised without any user-intervention. Consider though an island model of population (e.g. Figure 3). It is possible that each RPL2 process could receive an "island" of genomes. Synchronisation is not necessary here, and in fact it may be positively beneficial to allow different speeds of evolutionary development. If synchronisation does not occur, but is desired, the user can enforce it with this command. This command also takes an optional trigger.

4. *break* :

Using this command, the user can set a breakpoint in the plan currently being executed. It takes an optional trigger. When a breakpoint is activated, the RPL2 process to which it is attached will halt. To remove the breakpoint, use *kill* (see Token 13.)

5. *continue* :

This will restart RPL2 if it is halted by a breakpoint.

6. *step* :

This is to be used when RPL2 has been halted by a breakpoint. When RPL2 receives such a command, it will act as though it has received a `continue`, for the duration of the current plan instruction. When the next instruction is reached, RPL2 will act as though it has reached a breakpoint. This allows the user to see the effects of single instructions, and execute the plan at his/her own pace.

4.2.2 Triggers

These tokens are not commands in their own right. They must be given in conjunction with one of the commands above. The grammar has been designed to be reasonably lucid and readable; to this end, triggers follow commands. For example, a valid request through the interface would be "*send* `my_genome` *online* `50`". The first half is the command as given above, the second half is the trigger, telling RPL2 when to send the appropriate data. The valid triggers are given below :

7. *online* *integer* :

The request sent to RPL2 will be triggered when RPL2 reaches line *integer* of its current plan (the instruction on line *integer* is not executed). RPL2 may undergo further work in the future allowing multiple files to be used together to construct the genetic algorithm plan. The interface does not support this, but extending it to do so would be relatively trivial.

* Not Implemented

8. *every* *integer*:

The request is triggered every *integer* seconds. This provides a reliable mechanism for watching a data set update at regular intervals.

9. *after* *integer*:

The data request is triggered once only, after *integer* seconds. After it has been triggered once, the request is discarded by RPL2.

10. *onchange* `variable` * :

The request is triggered when RPL2 detects that `variable` has changed in value. So for example, if the trigger is "*onchange* `counter`", and while executing the plan, counter is incremented, this request will be triggered.

11. *when* *Boolean expression* * :

This is a very general trigger. The request for data is processed when RPL2 detects that *Boolean expression* is true. This expression is only defined in terms of scalar values at the moment.

In most commands it is not necessary to give a trigger. In this case, the request is serviced by RPL2 as soon as it is received. This corresponds to the default trigger, notionally *now*.

4.2.3 Modifiers

These tokens are used to alter the interpretation of requests given to RPL2. They are typically placed at the end of the request for data, giving a general request form of "*command trigger modifier*". Those considered are as follows :

12. *stride* *integer* * :

By adding this phrase to a data request, another layer of triggering is added. Notionally, a counter for this request will be set to 0. If the command without the modifier would have triggered, the counter is incremented. If counter = *integer*, the data request is triggered. This allows the user a greater degree of control over the data sent back by RPL2. If a request is triggering too often, perhaps slowing the performance of RPL2, it can be altered by adding this *stride* modifier, causing only the specified fraction of the original triggers instead.

13. *mask* *expression* * :

This concept attacks the problem described in Section 3.1. It introduces the idea of a parallel mask — this restricts the data sent back by RPL2. By using this modifier, the user can focus his/her interest on a particular subset of data. It may be possible to describe the mask on the command line, but if the population is large, or the structure complex, it will be easier to allow this to be read from a file. In this way it can be defined once, and used easily.

The syntax for the mask is a tricky problem and has not been defined, although it may be wise to follow these guidelines :

1. A syntax is already defined by RPL2 to allow operations to be executed on subsets of the population (see [Surry, 1993]). As an example, a plan may declare a 2-dimensional $n \times n$ population of genomes. To introduce a variable representing a member of the population, the following declaration can be given :

   ```
   genome [*,*] joe_public
   ```

   Thus a `joe_public` genome will exist for each location in the population[6]. But by giving the following declaration :

   ```
   genome [*,-] boss
   ```

*Not Implemented

[6] Strictly one will exist corresponding to each element $\in \{1...n\} \times \{1...n\}$

a `boss` genome will exist for each "column" of the population[7]. A similar approach could be taken with masks but a problem here would be the requirement for more refinement in which elements are selected. This could be solved by replacing the general "*" with a Boolean array, in this example of length n e.g.

`[10101010,11111111], where` $n = 8$

This would select $4 \times 8 = 32$ genomes for visualisation, since data is requested from every second "column" of genomes, and from each location in that "column".

This would still not allow arbitrary patterns such as "circular areas" to be requested — perhaps a simple $n \times n$ length Boolean list is the simplest method of achieving this.

Another method suggested is to define a parallel Boolean variable in the RPL2 plan. This can be initialised at run-time by assigning it an appropriate vector early in the GA plan. It can then be used to determine which sub-set of the parallel variable (genome, parallel integer) should be sent back by RPL2. This would be a fast and relatively simple method of restricting data.

2. The design of the interface was based somewhat on the feel of RPL2 itself. Using masks in this way would increase the logical correspondence that exists, and not force the user into using different grammars when writing plans and using the visualiser.

4.2.4 Miscellaneous

These tokens don't fit into the above categories; the closest appropriate name would be administration.

14. *kill* *integer*:

This is used to stop a current visualisation request. As mentioned earlier, it is possible to have several threads of data being communicated back to the interface at once. Therefore, if appropriate, once a request has started, it will continue to yield data each time it is triggered. Therefore, a mechanism must exist to terminate a request. *integer* corresponds to a unique "ID" associated with each request. This method is the only way to terminate an outstanding request — it should be used to halt requests for data, and for removing breakpoints.

15. *outstanding* :

This simple request displays to the user the ID of each request still outstanding, and the request as typed by the user. This can be used to gain the ID for an appropriate request, which can then be terminated with *kill*.

4.3 Communication Specifics - the EVENT structure

It can be seen from the grammar above that the user can input quite a wide range of commands. In most of these cases, this command must be communicated to RPL2, and therefore in an unambiguous way. The request RPL2 receives should be fairly simple if possible, and allow for quick interpretation. The user input could be packaged up into a string of bytes and sent directly to RPL2, but forcing the parsing job on RPL2 is unnecessarily wasteful of time in which it could be doing useful GA work. Therefore the approach taken is to tokenize and parse the user input at the visualiser's side. As this parsing is done, a data structure is automatically built up, containing all the relevant data for RPL2. This data structure is known as an EVENT.

The EVENT was designed to be small, and also to encompass every request the user could make to RPL2. Whether the request is for data, or for placing breakpoint in a plan, the command can be unambiguously packed into an EVENT.

Figure 8 gives the data structure in "pseudo" C :

Each of these "fields" is now explained individually

[7] Strictly one will exist corresponding to each element $\in \{1...n\}$

```
struct EVENT
{
  unsigned int  id;
  int           order;
  int           when;
  int           {var_to_send, id_to_kill};
  int           {line_no, time_interval};
  int           stride;
};
```

Figure 8: The EVENT Data Structure

`id` : every EVENT sent to RPL2 has attached to it a unique ID number, simply an integer. This is necessary in maintaining multiple threads of data. At any point, there may be several distinct data requests capable of service from RPL2. When the interface receives data from RPL2, it must have a method of knowing to which request the data pertains. This is done by making RPL2 attach the original EVENT ID number to the data sent back.

`order` : this is the field that RPL2 uses to identify the type of request it has received. The symbolic values this field can take correspond to the command and miscellaneous tokens described above (token no.s 1, 2, 3, 4, 5, 6 and 14). Using this field, RPL2 decides if the request is plan or data related, or simply an order to terminate an outstanding request.

`when` : this is another field that is assigned symbolic values that correspond to tokens described above. It is used to determine the type of trigger for the request. RPL2 can then determine when to send the data requested (; the relevant token values are 7, 8, 9, 10 and 11).

`var_to_send` : this field contains an integer that corresponds to the variable the user wishes to visualise. The correspondence between the integer value and the variable name is explained in Section 5.2. Therefore, this field is used in conjunction with the `order` field.

`id_to_kill` : this field is symbolically defined as `var_to_send`, and therefore refers to the same integer in the EVENT structure. This action is deliberate, and helps make the interface source files easier to understand. There is no ambiguity here — the symbolic value `id_to_kill` is used in the implementation when it is known the `order` field contains the value representing the *kill* token (token no. 14). This abstraction should assist a maintainer reading the interface code.

`line_no` : this is used in conjunction with the `when` field, and again has a symbolic equivalent (below). Its name is suggestive of its purpose; it holds an integer that tells RPL2 the condition for triggering the request. Specifically, if the `when` field holds a value representing an *online* trigger, the integer in this field will determine on which line of the plan the request is triggered.

`time_interval` : the symbolic equivalent of the above, therefore this field serves the same purpose as `line_no`. It is used with this name in parts of the implementation when it is known that the trigger type is *every* or *after*. It then represents the number of seconds until the request is triggered.

`stride` : again this field holds an integer value. It is used to determine the number of times a request must be triggered before the EVENT is processed.

This completes the description of the fields of an EVENT, but it is instructive to see examples of how these fields work together. Two are given in Figures 9 and 10.

Care must be taken with the stride field. If a stride is not required by the user, a transmitted EVENT will still have a value in this field, even if it is uninitialised. To prevent RPL2 interpreting the uninitialised stride value incorrectly, the visualiser sets the default value to 1. This means that the data is sent each time the request is triggered. This is consistent with a user request that does not define a stride value.

Note that this provides no scope for the *change* command, since no field exists for a replacement value or

ID	ORDER	WHEN	VAR_TO_SEND	TIME_INTERVAL	STRIDE
(UNIQUE)	SEND	EVERY	id(child_genome)	10	

Figure 9: *send* `child_genome` *every* 10

ID	ORDER	WHEN	VAR_TO_SEND	LINE_NO	STRIDE
(UNIQUE)	BREAK	ONLINE		50	5

Figure 10: *break online* 50 *stride* 5

expression in the EVENT structure. This is deliberate since *change* has not yet been implemented. The structure could easily be extended to encompass this need.

This then describes the EVENT, the unit of requests communicated to RPL2. Another structure is described below, the DATA structure. This is not completely defined, for two reasons. Firstly, the disposal of data transmitted back by RPL2 is not defined in this project. Secondly, the representation of some RPL2 variables as data has not been designed and implemented. Despite this, an "interim" structure exists, and suffices for sending back simple information[8].

4.4 The DATA structure

Again a "pseudo" C structure is given for the design of the data structure. This will be used for all communications from RPL2 back to the interface and hence ultimately to the visualiser. Some fields are used at the moment, some fields will be required, but are not implemented.

```
struct _struct_DATA
{
  unsigned int id;
  unsigned int event_id;
  int          type;

  union data_item
  {
     double fitness;
     int    the_int;
     BOOL   the_bool;
     real   the_real;
     genome the_genome;
  }
};
```

Figure 11: The DATA Structure

Again, each field is explained in turn

[8] This includes information such as genome fitness and scalar values

`id` : every DATA message transmitted back from RPL2 is assigned a unique ID. This may be useful in the future.

`event_id` : this field is set to the ID of the request for which this data was generated. This allows the interface to determine to which visualisation thread the data belongs.

`type` : this field takes a symbolic constant which represents the type of the data sent back e.g. `int`, `genome` etc. This is necessary, the interface must be able to correctly interpret the data it receives.

`data_item` : the actual data to be visualised. This can be any of several datatypes supported by RPL2. Note that the size of this field can vary enormously, from a handful of bytes to represent an integer, to kilobytes for a genome with a complex representation [9]. A non-existent type is included here called fitness — this field would then contain the value of a particular genome's fitness rather than the entire genome itself.

One field not designed is absolutely required for sending genomes. Since a genome is a parallel variable, the visualiser will need to know at which location in the population it resides. Since the population may be multi-dimensional and structured, a co-ordinate field is required. This could take the form of a simple list of integers, of length the dimension of the population structure. Each integer would give the location of the genome in a specific dimension. So, using the example and conventions of Section 4.2.3, a "co-ordinate" could be

```
[4,4,8,2]
```

giving a precise location in a 4-dimensional population structure.

This method would have to be used to visualise ordinary parallel data such as parallel integers etc.

An issue arises here about how much data needs to be transmitted from RPL2 to the interface. One might expect the value of a scalar variable to be constant over all RPL2 worker processes. For this reason, it should not be necessary to send the same scalar value from each RPL2 process. This is not the case — as mentioned earlier, it is possible for genomes to evolve at different speeds if located on different processors, particularly using island population models. Synchronisation may not occur, and therefore, scalars could have different values on different processors, if processor speeds differ.

5 Implementation

In this section, many of the implementational details are discussed. The layout attempts to follow a typical usage of the interface (ultimately by a visualiser) from startup with RPL2, through making requests for data. This begins with an overview of the MPI methods used in Section 5.1. Next the startup procedure of the interface is described in Section 5.2, and Section 5.3 details the interface in operation. This starts at parsing a user request, and follows the visualisation process through to the processing of the request by RPL2.

5.1 MPI Implementation of Requests

In this section, some details of how different commands are processed are given in terms of MPI. First it is necessary to define the new groups (and communicators) that are used. These are shown in Figure 12.

IO represents the distinguished IO node (it compiles and communicates the plan), VIS represents the new interface process, and the others are workers. In the implementation, only MPI intra-communicators were used. Therefore, communication between processes took place within the confines of one communicator. Some implementation methods for requests are given below :

`send` : a worker process will execute a non-blocking MPI send call to the VIS process, with the appropriate data. This is done over `MPI_COMM_VIS`, the communicator created from `MPI_GROUP_VIS`.

[9] No method has been designed for RPL2 `strings` and `gstacks`

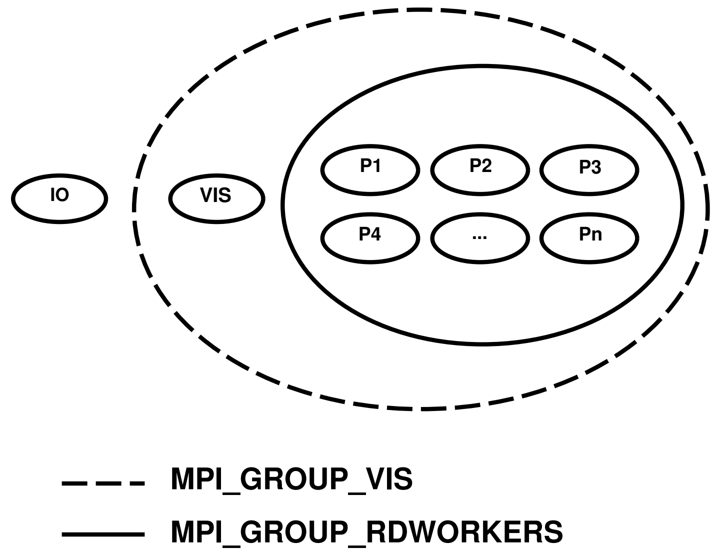


Figure 12: Defined MPI Groups

`sync` : once a worker finds a *sync* EVENT that has triggered, it makes a call to `MPI_BARRIER` over the communicator `MPI_COMM_RDWORKERS`, created from `MPI_GROUP_RDWORKERS`. Therefore it will be blocked until every process has reached this point and called `MPI_BARRIER` too. This causes synchronisation.

`break` : when this EVENT triggers, the RPL2 worker first sends a small message to the interface process to the effect of "I have hit a breakpoint". This uses the simple DATA structure. Next the worker calls `MPI_BARRIER`, but this time over `MPI_COMM_VIS`. Therefore each worker AND the interface process must call this function before any workers can proceed. The interface process does not call `MPI_BARRIER` until the user wishes to proceed past the breakpoint. Thus each worker is stopped.

`continue` : this must allow workers to proceed past a breakpoint. This is simply achieved by the interface process making the final required call to `MPI_BARRIER` over `MPI_COMM_VIS`. Each worker is then released to continue.

`step` : this is very similar to the above. Each worker is allowed to continue by the same mechanism as for *continue*. The difference is that the RPL2 framework notes that this is only a *step* and not a *continue*, and than acts as though a breakpoint has been hit on the next line of the plan. Thus it then behaves as *break*.

`kill` : this is a simple message exchange. Once an RPL2 worker receives the *kill* command, it removes the appropriate EVENT from its list, and then sends a simple acknowledgement back over `MPI_COMM_VIS` to the interface process. This is necessary because the *kill* command can be given a trigger, and thus may not be activated for a period of time. A user issuing the *outstanding* command may be misled into thinking an EVENT has terminated when it has not.

These operations should work successfully with the other parallel mode of operation supported by RPL2, that of the task farm, with some alterations. Since support for this mode has not been implemented, not much detail is given here. One change that is necessary is to alter the definition of the new MPI groups. This is because when RPL2 is using a task farm, only one process executes the compiled plan, and the others wait until tasks can be given to them, such as the evaluation of a genome. Therefore, *send* requests need only go to this one process. Also, while RPL2 is actively issuing tasks and receiving data from farm workers, it would not make sense to issue a *sync* request. Only one process, the farmer, would react to it, therefore rendering it redundant.

5.2 Starting the Interface

As shown in Figure 4, RPL2 consists of several processes. In the parallel mode most relevant to this project, regular domain decomposition, all but one processes will be workers, with a distinguished IO process taking care of parsing and compiling the genetic algorithm plan etc. The interface is simply an addition to these processes (it will now also be referred to as the VIS process).

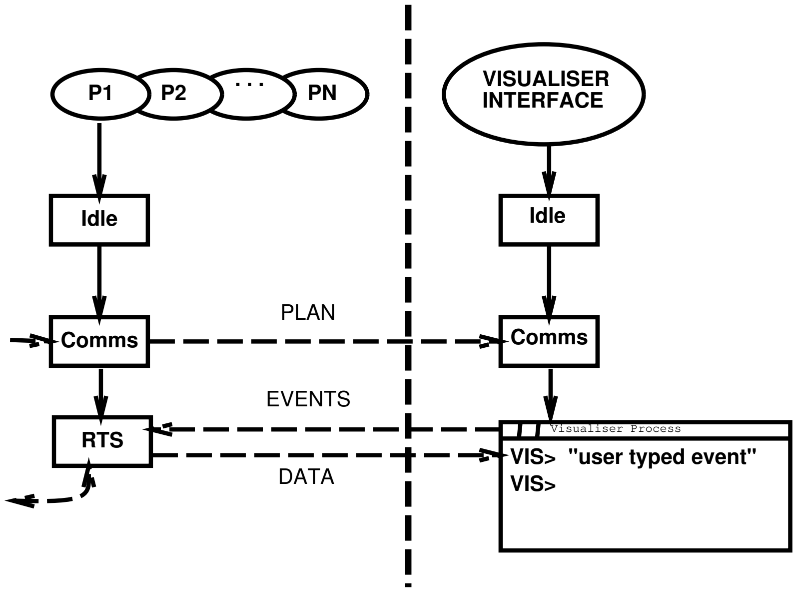


Figure 13: The Interface Process

In fact, the startup procedure of the interface is conceptually very similar to the startup procedures of the workers. Figure 13 is an extension to Figure 4; the VIS process is shown on the right with the workers on the left. Until a plan is read by RPL2, the interface is inactive, as are the workers. Once a plan is submitted to RPL2, it is compiled into an abstract machine language, as described in Section 2.3. But now this compiled plan, as well as being communicated to the workers, is transmitted to the VIS process as well.

This idea helps to shift more work onto the visualiser process. Consider a user's request to visualise a variable. This is typed into the interface command line and parsed, but the variable name, as a string, is not transmitted in the EVENT structure. Instead an integer is used, and the relationship between this number and the variable name is contained in the communicated plan[10]. Therefore, the visualiser makes this conversion — it would be pointless sending the variable name as a string, and wasting the time of worker processes in each recognising it for itself. This method also allows the visualiser to perform a degree of error checking. If a variable is requested that is not defined in the current plan, an error can be generated immediately without any communication.

5.3 The Interface in Operation

Following the communication of the plan, the interface can be accessed by the user. As mentioned this interface takes the form of a shell, with requests typed in by the user. At this point, the interface enters a loop, that can be expressed by the following much simplified pseudo-code :

```
Interface :
   while (RPL2 is running)
```

[10] The plan contains information about the variables used within it, and each variable has a unique ID number

```
      Read input from user
      if complete EVENT input
         Transmit the EVENT

      Check for Received Data
      if Data received
         Visualise the Data
   end
```

5.3.1 Creating an EVENT

A method must be used to translate the user input at the command line into an EVENT structure which can be communicated. This has been done using two UNIX tools called `lex`[11] and `yacc`[12]. These are often used fruitfully together to produce an input parser. parser, and were very useful in this project.

Each program serves its own purpose — a `lex` produced program scans an input file, and on recognising user-defined strings, will output user-defined tokens. A `yacc` produced program will scan an input file of tokens, and parse it according to some user-defined grammar. Actions can be performed on the successful recognition of constructions.

Therefore these were used together — the user's input was analysed by `lex`, and a sequence of tokens were generated. `yacc` then parsed the sequence of tokens according to the grammar given above (see Section 4.2) and in Appendix A. So the user may type :

```
VIS> synchronise online 25
```

This would be tokenised by `lex` using code of a similar style to :

```
%%...
"synchronise"          { return(SYNC); }
...%%
```

Therefore, the first token returned here would be `SYNC`. The sequence of tokens would be parsed by a piece of `yacc` in a similar style to that given below :

```
%token  SEND SYNC...

event
        : SEND  { DoSend(); } var  delay
        | SYNC  { DoSync(); }      delay
```

So if the sequence of tokens returned by `lex` matched the second `event` grammar rule, the function `DoSync()` would be executed. These functions simple fill in fields in the appropriate EVENT structure, e.g. `DoSync()` may place the symbolic value *sync* in the `order` field of the EVENT.

In this way, as soon as the user input has been completely parsed, the EVENT has been completely and unambiguously initialised for the correct request to be made to RPL2.

5.3.2 Sending the EVENT

This is dealt with in more detail in Section 5.1. Briefly, the interface uses a non-blocking "broadcast"[13] to send the EVENT to each RPL2 worker process. For details on the MPI groups, see Section 5.1.

[11] Strictly, a functionally similar program called `flex` was used
[12] This was replaced with a package called `bison`
[13] Or simulates one, since this feature does not exist in MPI

5.3.3 Storing Received EVENTs

EVENTs received by each RPL2 worker process must be stored so that they can be serviced when triggered. This is done using a simple data structure, as shown in Figure 14. This is a simple-linked list; no special techniques are applied to e.g. sort or traverse this structure.

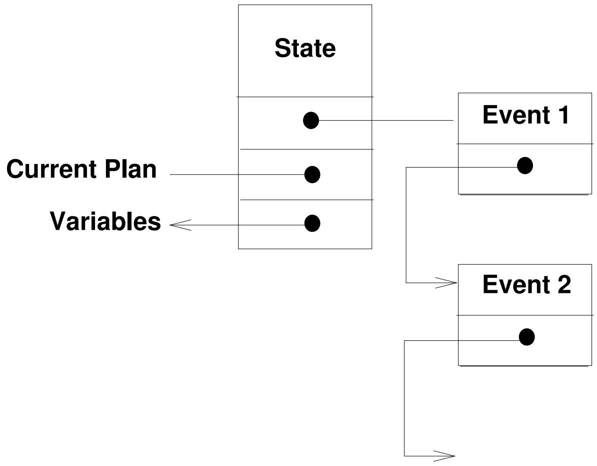


Figure 14: Storage of EVENTS by RPL2 Workers

This list is attached to a structured variable called the `state`. Each RPL2 process has a `state` variable. It contains information relevant to the current GA such as pointers to the compiled plan, the instruction pointer (program counter) etc. Therefore, a list of EVENTs for the current plan would logically be stored here.

The only subtlety used in storing EVENTS is illustrated in the difference between those that are data related and those that are plan related. When a data related EVENT is received e.g. a request for data to be sent, this is stored at the head of the linked list. When a plan related request is received, this is stored at the tail of the linked list. This is because when RPL2 checks the EVENT list to see if any EVENTs should be triggered, it (arbitrarily) starts at the head. Therefore, this mechanism ensures that all data related requests are satisfied before a potential breakpoint (a plan related EVENT) is activated.

5.3.4 Triggering an EVENT

Before a user request is satisfied, the EVENT must be triggered i.e. the condition for RPL2 to take action on it must be satisfied. Therefore, as well as working on the GA, RPL2 worker processes must take some time to work through the EVENT list, checking each EVENT's trigger condition.

This is provided by a simple plan instrumentation method. A brief clip of an RPL2 plan is shown below.

```
population_size := 100;
num_generations := 10;

for i := 1 to population_size
   child := RandomGenome();
...
```

Figure 15: Clip of RPL2 Plan

As stated in Section 2.3, this plan is compiled into an abstract machine language that is interpreted by RPL2. When visualisation is required, the plan is compiled, but also instrumented with `OP_LINE` instructions. These instructions perform no function in terms of the GA, but instead give RPL2 an opportunity to check the EVENT list. Given in Figure 16 is the compiled instrumented version of Figure 15.

```
Main instructions
        4       OP_LINE simple.rpl:19
        5       OP_ASSIGN       storeid 41      storeid 58
        6       OP_LINE simple.rpl:20
        7       OP_ASSIGN       storeid 42      storeid 61
...
        37      OP_PUSHMASK
        33      OP_FORMASK      storeid 43      storeid 41      storeid 37
        38      OP_BRANCH       opid 34
        24      OP_LINE simple.rpl:40
        25      OP_INST RandomGenome     storeid 48
```

Figure 16: Clip of Compiled Instrumented RPL2 Plan

The correspondence between the written and compiled instructions is reasonably clear. With this in mind, it can be seen that each written instruction, when compiled, is preceded by an `OP_LINE` instruction. Therefore, before executing each "written" instruction, RPL2 encounters an `OP_LINE` and checks the EVENT list. This procedure is necessary at least once for every written instruction, in order to ensure any *online* triggers are checked.

This mechanism is also very useful in that it satisfies the performance requirement stated in Section 4.1.2. This is because if no visualisation is required, the instrumentation mechanism can be turned off. Therefore, RPL2 will never receive any EVENTs, or get an opportunity to service them. This means there will be no reduction in its operational speed.

It is now relatively simple to check an EVENT trigger. For each EVENT in the list, the *when* field is checked. This determines the trigger type. For example, if the type is *every*, RPL2 checks how long it has been since the last trigger. If sufficiently long, the time of triggering is stored, and the request processed.

5.3.5 Sending Data back to the Interface

This has been implemented in a very restricted form, so not much implementational detail can be given. In short, if an appropriate data related EVENT is triggered, RPL2 will search its current plan's variables. As mentioned earlier these have a unique ID, and this is checked against the ID given in the EVENT structure. When the match is found, appropriate action is taken to copy the data into the DATA structure. This is then sent back to the interface.

6 Premature Results

In this section, two genetic algorithm plans are "visualised" using the facilities provided by the interface. As the disposal of the data received is undefined for the purposes of this project, I have chosen to write it out, as it is received, in a sequence of simple files. These have been created especially to conform to the `ppm`[14] graphics format. Then a program called `xv` was used to view these data sets. These images are shown here.

Both results here are in no way novel or interesting in the field of genetic algorithms, they merely serve to illustrate the capabilities of the interface at this point.

Both plans utilised a fine-grained population model (see Figure 3) of size 8×8. The first plan uses recombination and mutation operators in an effort to maximise the number of TRUTH values in a length 32 Boolean list. The population is initially completely random.

[14] Portable Pixmap

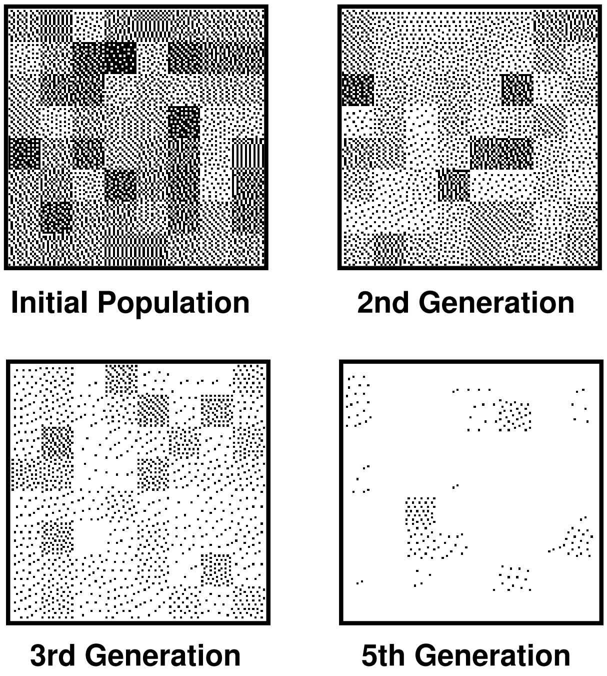


Figure 17: Genetic Algorithm with Recombination and Mutation

For each square, the lighter the shade of grey, the fitter the genome. It can be seen in Figure 17 that even after a very small number of generations, the population has become very fit, and indeed would probably not become much fitter if many more generations were visualised.

Figure 18 is almost the same genetic algorithm, except that the recombination operator has been removed. With only mutation to "evolve" the population, one would not expect much improvement over the generations. Clearly, even after a large number of generations, there is no improvement in the general population fitness.

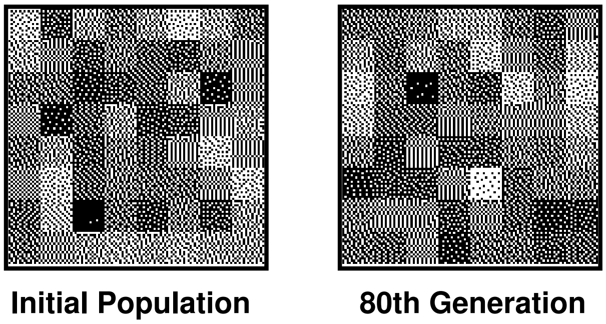


Figure 18: Genetic Algorithm with Mutation only

This method of visualising is very simple — a 2D grid of squares seems entirely intuitive. A harder problem for the visualiser designers will be how to visualise arbitrarily complex population structures.

7 Further Work

There are many ideas for further work in this area, indeed this project has barely scraped the surface in terms of what will be required in a final visualisation system. Some of these ideas are :

- Implement the data restriction, perhaps using the idea of masks
- Allow entire genomes to be sent to the visualiser, and eventually support for arbitrary population models
- Provide support for the Task Farm parallel mode
- Build the *visualiser* itself!

8 Administration

There are several files that make up the interface, C files which should be linked against the main RPL2 framework, and a Makefile, written to be as compatible with the existing framework as possible.

These files can all be found in :

`/home/gcla/rpl2/ss`

A brief explanation of the purpose of these files is given below

`ss_interface.c, ss_interface.h` : these contain code that deals with interface matters e.g. sending EVENTs, receiving data etc.

`testbed.c, testbed.h` : these contain code that attaches to the RPL2 framework, and deals with receiving EVENTs, checking triggers and sending data

`vis.l` : this is `flex` source to generate the tokenizer for the interface grammar. It compiles into `lex.visyy.c`

`vis.y` : this is `bison` source to generate the parser for the interface grammar. It compiles into `vis.tab.c` and generates a header file `vis.tab.h` of tokens, which are used universally throughout the code.

`Makefile` : for the serial framework only.

The directory also contains a `README` file.

The interface was generally implemented in accordance with the ISG C Coding Standard (see [ISG, 1993]). In order to maintain control over older revisions of source files, the UNIX tool `rcs` was used, as it was in the main RPL2 framework.

9 Acknowledgements

First of all I would like to thank my supervisors, Pat Surry and Gordon Cameron, for being so patient over the course of the project, despite busy work schedules. Their technical help was absolutely vital, especially with RPL2 matters. Thanks also to Colin Brough (programme co-ordinator) for constructive criticism and prudent help. Finally, thanks to the EPCC support staff for maintaining a high-level of feedback and civility, in spite of my occasionally reckless questions.

A The Interface Grammar

```
%{

static char *sRCS_VIS_Y = "$Id: vis.y,v 1.6 1994/09/11 19:52:30 gcla Exp gcla $";

/* ******************************************************************** *
 *      Filename:       gcla-grammar.txt
 *
 *      Author:         Graham Clark
 *
 *      Purpose:        Parsing for the Vis. interpreter
 *
 *      Contents:       Grammar rules for Vis. commands
 *
 *      Use:            Definition of the Vis. language
 *                      This grammar has been changed a great deal to that
 *                      which exists in vis.y. All instrumentation has
 *                      been removed, with only a skeleton left.
 *
 * ******************************************************************** */

%union
{
        char    *n;             /* Declare a union type to allow the INTEGER */
        char    *s;             /* and VARIABLE tokens to pass back strings  */
}

/* These tokens correspond to the commands that can be issued by the user */

%token          SEND    SYNC    BREAK   CONTIN  OUTSTND KILL    STEP


/* These tokens correspond to the triggers that can be used to determine
   when a request is processed */

%token          WHEN    ONCHANGE        ALWAYS  AFTER   EVERY   ONLINE
%token          EXPRS   NOW


/* This is a modifier token - note that the MASK device has not been
   implemented */

%token          STRIDE

/* These tokens return a (char *) to the calling process, allowing the
   textual value of the variable or integer to be returned directly */

%token  <s>     INTEGER VARIABLE


/* Simple token, to allow the interface to acknowledge data transmissions
   from the RPL2 workers */
```

```
%token          ACK


/* These grammatical non-terminals ultimately return a (char *) as
   their result */

%type   <s>     const   var     expr    boolexpr


/* The user must always enter an EVENT, so always try to match the
   event non-terminal  */

%start          event

%%

/* ********************************************************************* *
 *
 * Rules section - BNF Grammar for the Interface
 *
 * ********************************************************************* */

/* These are the commands that can be issued */

event
        : SEND  var delay stride        /* e.g. SEND x ONLINE 50 STRIDE 5 */
        | SYNC  delay                   /* e.g. SYNC EVERY 100            */
        | BREAK delay                   /* e.g. BREAK ONLINE 50           */
        | CONTIN                        /* only matches CONTIN            */
        | STEP                          /* only matches STEP              */
        | KILL  INTEGER                 /* e.g. KILL 5                    */
        | OUTSTND                       /* only matches OUTSTND           */
        ;


/* These are the triggers that may be used with a command */

delay
        : /* Perhaps the user wishes no delay */
        | ONCHANGE var                  /* When a variable's value changes*/
        | WHEN boolexpr                 /* When an expression holds       */
        | AFTER time                    /* After certain time, but only once */
        | EVERY time                    /* Every time seconds             */
        | ONLINE lineno                 /* On given plan line no.         */
        ;


/* The stride, adds another layer of triggering */

stride
        : /* Perhaps the user wishes no stride */
        | STRIDE INTEGER                /* e.g. STRIDE 5, every 5 triggers*/
        ;

time
```

```
        : INTEGER                       /* time is simply an integer, the */
        ;                               /* no. of seconds desired         */

lineno
        : INTEGER                       /* lineno is simply an integer    */
        ;

boolexpr
        : expr '=' expr                 /* NOT IMPLEMENTED                */
        ;

expr                                    /* Standard forms of an expr      */
        : const /* NOT IMPLEMENTED                */
        | var
        | '(' expr ')'
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        ;

const
        : INTEGER { $$ = $1; }          /* Return the value as a string   */
        ;

var
        : VARIABLE { $$ = $1; }         /* Return the value as a string   */
        ;
```

References

[Filho *et al.*, 1994] José L. Ribeiro Filho, Philip C. Treleaven, and Cesare Alippi. Genetic-algorithm programming environments. 1994.

[ISG, 1993] ISG. *ISG C Coding Standard*, 1993.

[MPI Forum, 1994] MPI Forum. *MPI : A Message Passing Interface Standard*, 1994.

[Radcliffe and Surry, 1994a] Nicholas J. Radcliffe and Patrick D. Surry. The reproductive plan language rpl2 : Motivation, architecture and applications. 1994.

[Radcliffe and Surry, 1994b] Nicholas J. Radcliffe and Patrick D. Surry. The reproductive plan language rpl2. 1994.

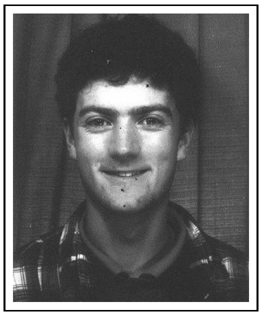
[Russo, 1991] Claudio V. Russo. A general framework for implementing genetic algorithms. Technical report, Edinburgh Parallel Computing Centre, 1991.

[SGI, 1988] SGI. *IRIS-4D Programmer's Guide Update*, 1988.

[Surry, 1993] Patrick D. Surry. *RPL2 Functional Specification*, 1993.

[Surry, 1994a] Patrick D. Surry. *RPL2 Design Document*, 1994.

[Surry, 1994b] Patrick D. Surry. *RPL2 User Guide*, 1994.



Graham Clark is a student of Computer Science at the University of Edinburgh. After graduation in 1995 he hopes to continue his studies on a postgraduate course.
This project was supervised by Patrick Surry (EPCC) and Gordon Cameron (EPCC)
Contact:

Patrick Surry
Edinburgh Parallel Computing Centre
James Clerk Maxwell Building
King's Buildings
University of Edinburgh
West Mains Road
Edinburgh

E-Mail: pds@epcc.ed.ac.uk

Gordon Cameron
Edinburgh Parallel Computing Centre
James Clerk Maxwell Building
King's Buildings
University of Edinburgh
West Mains Road
Edinburgh

E-Mail: gordonc@epcc.ed.ac.uk