Appears in "Parallel Problem Solving from Nature III", Eds: Y. Davidor, H-P. Schwefel, R. Manner, Springer-Verlag LNCS 866, pp628-637, 1994.

RPL2: A Language and Parallel Framework for Evolutionary Computing

Patrick D. Surry, Nicholas J. Radcliffe

pds@epcc.ed.ac.uk, njr@epcc.ed.ac.uk

Edinburgh Parallel Computing Centre
King's Buildings, University of Edinburgh
Scotland, EH9 3JZ

Abstract

The Reproductive Plan Language 2 (RPL2) is an extensible interpreted language for writing and using evolutionary computing programs. It supports arbitrary genetic representations, all structured population models described in the literature together with further hybrids, and runs on parallel or serial hardware while hiding parallelism from the user. This paper surveys structured population models, explains and motivates the benefits of generic systems such as RPL2 and describes the suite of applications that have used it to date.

1 Motivation

As evolutionary computing techniques acquire greater popularity and are shown to have ever wider application a number of trends have emerged. The emphasis of early work in genetic algorithms on low cardinality representations is diminishing as problem complexities increase and people find it more convenient and effective to use more natural data structures. There is now extensive evidence, both empirical (Davis, 1991; Michalewicz, 1993) and theoretical (Mason, 1993; Radcliffe, 1994), that the arguments for the superiority of binary representations (Holland, 1975; Goldberg, 1989, 1990) were at least overstated, and as the fields of evolution strategies (Baeck *et al.,* 1991), genetic programming (Koza, 1992), evolutionary programming (Fogel, 1993) come together with genetic algorithms an ever increasing range of representation types are becoming commonplace.

During the last decade, as interest in evolutionary algorithms has increased, there has been the simultaneous development and wide-spread adoption of parallel and distributed computing. The inherent scope for parallelism evident in evolutionary computation has been widely noted and exploited, most commonly through the use of *structured* population models in which mating probabilities depend not only on fitness but also on *location.* Thus in these structured population models each member of the population (variously referred to as a chromosome, genome, individual or solution) has a site—most commonly either a unique coordinate (e.g. Gorges-Schleuter, 1990) or a shared

island number (e.g. Norman, 1988)—and matings are more common between members that are close (share an island or have similar coordinates) than between those that are more distant. Such structured population models, which are described in more detail in section 2, have proved not only highly amenable to parallel implementation, but also in many cases computationally superior to more traditional *panmictic* (unstructured) models (in the sense of requiring fewer evaluations to solve a given problem, Gordon & Whitley, 1993). Despite this, so close has been the association between parallelism and structured population models that the term *parallel genetic algorithm* has tended to be used for both. It is a minor objective of this paper to encourage the adoption of the term *structured population model* when it is this aspect that is referred to.

The authors of this paper both work for Edinburgh Parallel Computing Centre, which makes extensive use of evolutionary computing techniques (in particular, genetic algorithms) for both industrial and academic problem solving, and wished to develop a system to simplify the writing of and experimentation with evolutionary algorithms. The primary motivations were to support arbitrary representations and genetic operators along with all population models in the literature and their hybrids, to reduce the amount of work and coding required to develop each new application of evolutionary computing, and to provide a system that allowed the efficient exploitation of a wide range of parallel, distributed and serial systems in a manner largely hidden from the user. RPL2, the second implementation of the *Reproductive Plan Language* and framework, was produced in partnership with British Gas plc to satisfy these aims. This paper outlines the main benefits of exploiting such a system, focusing in particular on the population models supported by RPL2 (section 2), its support for arbitrary representations (section 3), the modes of parallelism it supports (section 4) and the applications for which it has been used (section 5).

RPL2 takes the form of an interpreted language with some specialised data structures and functions designed to simplify drastically the task of implementing and experimenting with evolutionary algorithms. Example *reproductive plans* are given in the appendix. Both parallel and serial implementations of the run-time system exist and will execute the same plans without modification.

2 Population models

Structured populations fall into two main groups—fine-grained and coarse-grained. In the fine-grained models, also known variously as *diffusion* (Muehlenbein *et al.*, 1991) or *cellular* models (Gordon & Whitley, 1993), it is usual for every individual to have a unique coordinate in some space, typically a grid of some dimensionality either with fixed or cyclic boundary conditions. In one dimension lines or rings are most common, in two dimensions regular lattices or tori and so forth, but more complex topologies in higher dimensions are certainly possible. Individuals then mate only within a neighbourhood called a *deme* and these neighbourhoods overlap by an amount that depend on their size and shape. Replacement is also local. This model is the natural one to use on so-called Single-Instruction Multiple-Data (SIMD) parallel computers (also called array processors or (loosely) data parallel machines) in which a (typically) large number of (typically) simple processors all execute a single instruction stream synchronously on different data items, usually configured in a grid (Hillis, 1991). Despite this, one of the earlier implementations was by Gorges-Schleuter (1989), who used a transputer array. It need hardly be said that the model is of general applicability

on serial or general parallel hardware.

The characteristic behaviour of such fine-grained models is that the in-breeding within demes tends to cause speciation as clusters of related solutions develop, leading to natural niching behaviour (Davidor, 1991). Over time, strong characteristics developed in one neighbourhood of the population gradually spread across the grid because of the overlapping nature of demes, hence the term *diffusion* model. As in real diffusive systems, over time there is of course a tendency for the population to become homogeneous, but less quickly than in panmictic models. This population model tends to help in avoiding premature convergence to local optima. Moreover, if the search is stopped at a suitable stage the niching behaviour allows a larger degree of coverage of local optima to be obtained than is typically possible with unstructured populations. Both this and the alternative coarse-grained models are illustrated in figure 1. Other papers describing the variants of the diffusion model include Manderick & Spiessens (1989), Muehlenbein (1989), Spiessens & Manderick (1991), Davidor (1991), Baluja (1993), Maruyama *et al.* (1993) and Davidor *et al.* (1993).

RPL2 supports fine-grained population models by allowing populations to be declared as arbitrary-dimensional structures with fixed or cyclic boundaries and provides the `structfor` loop construct, which allows (any part of) a reproductive plan to be executed over such a structured population in an unspecified order, allowing the system to exploit parallelism if it is available. In the fine-grained model a deme structure must also be specified. Demes are specified using a special class of user-definable operator (of which several standard instances are provided), and indicate a pattern of neighbours for each location in the population structure.

The other principal structured population model is the *coarse-grained* model, probably better known as the *island* model. In this several panmictic populations are allowed to develop in parallel, occasionally exchanging genomes in migration steps that might occur after some number of generations. In some cases the island to which a genome migrates is chosen stochastically and asynchronously (e.g. Norman, 1988), in others deterministically in rotation (e.g. Whitley *et al.*, 1989) while in still others the islands themselves have a structure such as a ring and migrations only occur between neighbouring islands (e.g. Cohoon *et al.*, 1987); this last case is sometimes known as the *stepping stone* model. The largely independent course of evolution on each island again encourages niching (or *speciation*) while ultimately allowing genetic information to migrate anywhere in the (structured) population, and again this helps to avoid premature convergence and to encourage covering if the algorithm is run with suitably low migration rates.

Coarse grained models are typically only loosely synchronous, and work well even on distributed systems with very limited communications bandwidths. They are supported also in RPL2 by special declarations and use of the `structfor` loop construct and migration operators. There is sufficient flexibility included to allow arbitrary hybrid models also, for example, an array of islands each with fine-grained populations or a fine-grained model in which each site has an island (which could be viewed as a generalisation of the stepping stone model). Other papers describing variants of the island model include Petty & Leuze (1989) Cohoon *et al.* (1990) and Tanese (1989).

Unstructured (panmictic) populations are also, of course, available using simple variable-length arrays which may be indexed directly or treated as last-in-first-out stacks.

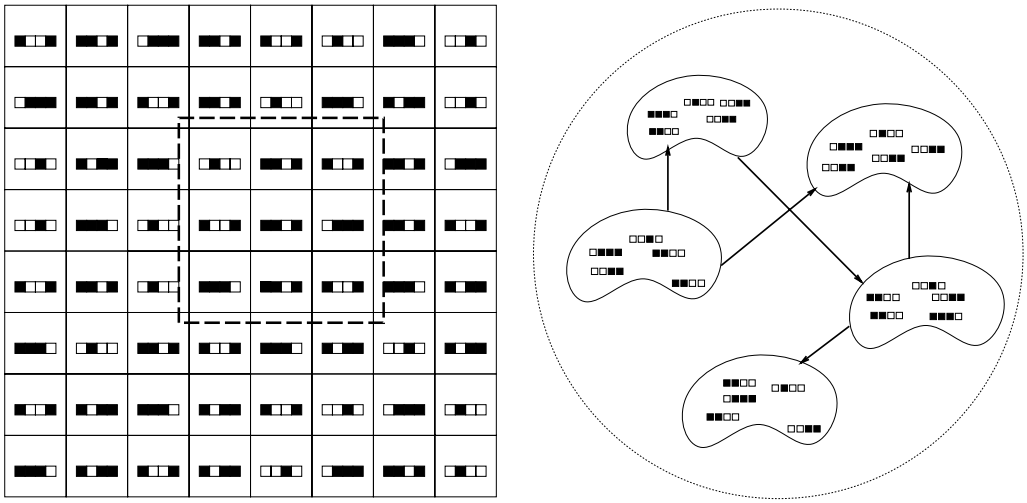

Figure 1: The picture on the left illustrates a so-called *fine-grained* (*diffusion* or *cellular*) population structure. Each solution has a spatial location and interacts only within some neighbourhood, termed a deme. Clusters of solutions tend to form around different optima, which is both inherently useful and helps to avoid premature convergence. Information slowly diffuses across the grid.
The picture on the right shows the coarse-grained *island* model, in which isolated subpopulations exist on different processors, each evolving largely independently. Genetic information is exchanged with low frequency through migration of solutions between subpopulations. Again this helps track multiple optima and reduces the incidence of premature convergence.

3 Representation Issues

It was stressed in section 1 earlier that a major design consideration for RPL2 was that it impose no constraints on the data structures used to represent genomes. This is achieved by including a generic pointer in the `genome` data structure that references a user-definable data structure together with fields that are (potentially) relevant to all problems. A distinction is then made between *representation-independent* operators, whose action depends only on the standard fields of a genome (such as fitness measures), and *representation-dependent* operators, which may manipulate the problem-specific part. Examples of representation-independent operators include selection mechanisms, replacement strategies, migration and deme collection. All "genetic" operators (most commonly recombination, mutation and inversion) are representation-dependent, as are evaluation functions, local optimisers and generators of random solutions This strongly promotes code re-use as domain-independent operators can form generic libraries. Even representation-dependent operators may have fairly wide applicability since many different problems may share operators at the genetic level: it is only evaluation functions that invariably have to be developed freshly for new problem domains.

Some evolutionary algorithms have been developed that employ more than one representation at a time. A notable example of this is the work of Hillis (1991), evolving sorting networks with a parasite model, where the evaluation function evolved to change the test set as the sorting networks improved. Similarly, Husbands & Mill (1991) have used co-evolution models with different populations optimising different parts of a pro-

cess plan which are then brought together for arbitration necessitating the use of multiple representations. There are also cases in which control algorithms are employed to vary the (often large number of) parameters as an evolutionary algorithm progresses, such as the work of Davis (1989) adapting operator probabilities on the basis of their observed performance. RPL2 caters for the simultaneous use of multiple representations in a single reproductive plan, which greatly simplifies the implementation of such schemes.

4 Parallelism

Evolutionary algorithms that use populations are inherently parallel in the sense that—depending on the exact reproductive plan used—each chromosome update is largely independent of the others. There are a number of options for implementation on parallel computers, several of which have been proposed in the literature and implemented. As has been emphasised, population structure has tended to be tied closely to the architecture of a particular target machine to date, but there is no reason, in general, why this need be so.

Parallelism is supported in RPL2 at a variety of levels. Data decomposition of structured populations can be achieved transparently, with different regions of the population evolving on different processors, possibly partially synchronised by inter-process communication. Distribution of fine-grained models tends to require more interprocess communication and synchronisation so their efficiency is more sensitive to the computation-to-communications ratio for the target platform.

Task farming of compute intensive tasks, such as genome evaluation (e.g. Verhoeven *et al.,* 1992, Starkweather *et al.,* 1990), is also provided via the `forall` loop construct, which indicates a set of operations to be performed on all members of a population stack in no fixed order. This is particularly relevant to real-world optimisation tasks for which it is almost invariably the case that the bulk of the time is spent on fitness evaluation. (For examples of this see section 5, which discusses applications.) User operators may themselves include parallel code or run on parallel hardware independently of the framework, giving yet more scope for parallelism.

RPL2 will run the same reproductive plan on serial, distributed or parallel hardware without modification. It uses the Marsaglia pseudo-random number generator (Marsaglia *et al.,* 1990), which as well as producing numbers with much better statistical distributions should allow identical results to be produced on different processors provided that they use the same floating point representation.

5 Applications

Applications that have currently been tackled using RPL2 include optimising a gas pipeline network with British Gas using a variable-cardinality integer representation (Boyd *et al.,* 1994), the Travelling Sales-rep Problem using a permutation representation with Random Assorting Recombination (Radcliffe, 1994) and data mining using a hierarchical genetic algorithm to evolve interesting sets of rules concerning data (Radcliffe & Surry, 1994). Trivial test problems using binary representations have also been implemented. Applications that used the prototype RPL system include optimising a retail network for Ford cars using an evaluation function that was a spatial interaction model running on a Connection Machine with a set representation (George, 1994), stock

market tracking with a hybrid quadratic programming scheme and a set-based representation (Shapcott, 1992) and neural network topology optimisation (Dewdney, 1992).

Several libraries of operators and representations are provided with the RPL2 framework. Constructing new operators and representations is a simple matter of writing standard ANSI C functions with return values and arguments corresponding to RPL2 data types. A preprocessor generates appropriate wrapper code to allow the operators to be called dynamically at plan run-time. New libraries of operators may optionally include initialisation and exit routines, start-up parameters, and check-pointing facilities. A library of representation-independent operators need include nothing further, while a new representation must define several routines that can be used by the framework to pack, unpack and free the user-defined component of a genome. The packing and unpacking routines are necessary to allow genomes to be passed between processors cleanly but may be null for serial applications.

Customised versions of the framework are built by linking together whatever combination of operator libraries and representations are desired, allowing locally developed operators to be tested in the context of existing libraries, maintained in some central location. RPL2 will be distributed freely in object form by the time of publication. Inquiries are welcomed by electronic mail at `rpl2-support@epcc.ed.ac.uk`. A wide range of serial and parallel hardware platforms are supported.

Acknowledgements

The serial prototype of RPL2 was implemented by Claudio Russo (1991), who developed many of the ideas with Radcliffe. The parallel prototype was developed by Graham Jones (1992). Mark Green and Ian Boyd from British Gas worked together with Patrick Surry on the design and implementation of RPL2.

References

Bäck *et al.*, 1991. Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1991.

Baluja, 1993. Shumeet Baluja. Structure and performance of fine-grain parallelism in genetic search. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1993.

Boyd *et al.*, 1994. Ian D. Boyd, Patrick D. Surry, and Nicholas J. Radcliffe. Constrained gas network pipe sizing with genetic algorithms. Technical Report EPCC-TR94-11, Edinburgh Parallel Computing Centre, 1994.

Cohoon *et al.*, 1987. J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates (Hillsdale, New Jersey), 1987.

Cohoon *et al.*, 1990. J. P. Cohoon, W. N. Martin, and D. S. Richards. Genetic algorithms and punctuated equilibria. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving From Nature*, pages 134–144. Springer-Verlag, October 1990.

Davidor *et al.*, 1993. Yuval Davidor, Takeshi Yamada, and Ryohei Nakano. The ECOlogical framework II: Improving ga performance at virtually zero cost. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1993.

Davidor, 1991. Yuval Davidor. A naturally occurring niche and species phenomenon: The model and first results. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 257–263. Morgan Kaufmann (San Mateo), 1991.

Davis, 1989. Lawrence Davis. Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1989.

Davis, 1991. Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold (New York), 1991.

Dewdney, 1992. Nigel Dewdney. Genetic algorithms for neural network optimisation. Master's thesis, University of Edinburgh, 1992.

Fogel, 1993. David B. Fogel. Evolving behaviours in the iterated prisoner's dilemma. *Evolutionary Computing*, 1(1), 1993.

George, 1994. Felicity A. W. George. Using genetic algorithms to optimise the configuration of networks of car dealerships. Technical Report EPCC-TR94-05, Edinburgh Parallel Computing Centre, 1994.

Goldberg, 1989. David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley (Reading, Mass), 1989.

Goldberg, 1990. David E. Goldberg. Real-coded genetic algorithms, virtual alphabets, and blocking. Technical Report IlliGAL Report No. 90001, Department of General Engineering, University of Illinois at Urbana-Champaign, 1990.

Gordon and Whitley, 1993. V. Scott Gordon and Darrell Whitley. Serial and parallel genetic algorithms as function optimisers. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1993.

Gorges-Schleuter, 1989. Martina Gorges-Schleuter. ASPARAGOS: an asynchronous parallel genetic optimization strategy. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 422–427. Morgan Kaufmann (San Mateo), 1989.

Gorges-Schleuter, 1990. Martina Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In H. P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, pages 150–159. Springer Verlag (Berlin), 1990.

Hillis, 1991. W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Stephanie Forrest, editor, *Emergent Computation*. MIT Press (Cambridge, MA), 1991.

Holland, 1975. John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press (Ann Arbor), 1975.

Husbands and Mill, 1991. Philip Husbands and Frank Mill. Simulated co-evolution as the mechanism for emergent planning and scheduling. In *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1991.

Jones, 1992. Graham P. Jones. Parallel genetic algorithms for large travelling salesrep problems. Master's thesis, University of Edinburgh, 1992.

Koza, 1992. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Bradford Books, MIT Press (Cambridge, Mass), 1992.

Manderick and Spiessens, 1989. B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433, San Mateo, 1989. Morgan Kaufmann Publishers.

Marsaglia *et al.*, 1990. G. Marsaglia, A. Zaman, and W. W. Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, 1990.

Maruyama *et al.*, 1993. Tsutomo Maruyama, Tetsuya Hirose, and Akihiko Konagaya. A fine-grained parallel genetic algorithm for distributed parallel systems. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1993.

Mason, 1993. Andrew J. Mason. Crossover non-linearity ratios and the genetic algorithm: Escaping the blinkers of schema processing and intrinsic parallelism. Technical Report Report No. 535b, School of Engineering, University of Auckland, 1993.

Michalewicz, 1993. Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag (Berlin), 1993.

Mühlenbein *et al.*, 1991. Heinz Mühlenbein, M. Schomisch, and J. Born. The parallel genetic algorithm as function optimiser. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann (San Mateo), 1991.

Mühlenbein, 1989. H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 416–421. Morgan Kaufmann (San Mateo), 1989.

Norman, 1988. Michael Norman. A genetic approach to topology optimisation for multiprocessor architectures. Technical report, University of Edinburgh, 1988.

Petty and Leuze, 1989. Chrisila C. Petty and Michael R. Leuze. A theoretical investigation of a parallel genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1989.

Radcliffe and Surry, 1994. Nicholas J. Radcliffe and Patrick D. Surry. Co-operation through hierarchical competition in genetic data mining. Technical Report EPCC-TR94-09, Edinburgh Parallel Computing Centre, 1994.

Radcliffe, 1994. Nicholas J. Radcliffe. The algebra of genetic algorithms. *To appear in Annals of Maths and Artificial Intelligence*, 1994.

Russo, 1991. Claudio V. Russo. A general framework for implementing genetic algorithms. Technical Report EPCC–SS91–17, Edinburgh Parallel Computing Centre, University of Edinburgh, 1991.

Shapcott, 1992. Jonathan Shapcott. Genetic algorithms for investment portfolio selection. Technical Report EPCC–SS92–24, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

Spiessens and Manderick, 1991. Piet Spiessens and Bernard Manderick. A massively parallel genetic algorithm: Implementation and first analysis. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 279–286. Morgan Kaufmann (San Mateo), 1991.

Starkweather *et al.*, 1990. T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving From Nature*, pages 176–185. Springer-Verlag, October 1990.

Tanese, 1989. Reiko Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann (San Mateo), 1989.

Verhoeven *et al.*, 1992. M. G. A. Verhoeven, E. H. L. Aarts, E. van de Sluis, and R. J. M. Vaessens. Parallel local search and the travelling salesman problem. In R. Männer and B. Manderick, editors, *Parallel Problem Solving From Nature, 2*, pages 543–552. Elsevier Science Publishers/North Holland (Amsterdam), 1992.

Whitley *et al.*, 1989. Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. Technical Report CS-89-117, Colorado State University, 1989.

Appendix: Example RPL2 plans

Panmictic example

```
% This is a simple panmictic (unstructured) example that illustrates how
% parallelism can be applied to such problems (the forall construct)
% The plan is based on a population/cache model illustrating generational update.

plan(PanmicticExample)

%% Preliminary section - declare the libraries and variables that we use

use     StdInst, Binary(128);           % the Binary repn is parameterised by string length
string  sFile;
bool    bMaxIsBest;

int     iCounter, nGenerations, i, nPopsize;

genome  gNew,gChild,gParentA,gParentB;
gstack  gsPop, gsCache, gsParents;      % population, a "shadow", and space for parents

% Initialisation section

sFile := "stdout";                      % direct output to the terminal
nPopsize := 200;                        % population size
nGenerations := 100;                    % number of generations
bMaxIsBest := TRUE;                     % maximise the evaluation function

Randomize(0);                           % pseudo-random initialisation

for iCounter := 1 to nPopsize           % create an initial population of nPopsize genomes
        gNew := RandomGenome();
        Push(gNew,gsPop);
endfor

forall gChild in gsPop                  % evaluate (in parallel) the initial population
        EvalOneCount(gNew);             % trivial evaluation equal to number of 1s in string
endforall

for i := 1 to nGenerations              % generational update scheme

        Empty(gsCache);                 % empty the stack where we'll build the next gen
        Empty(gsParents);               % empty the array of parents

        ScaleRanked(gsPop, bMaxIsBest, 0.0, 1.0);       % scale population on ranking
        SelectScaledSUS(gsPop, 2 * nPopsize, gsParents); % choose all parents at once

        for iCounter := 1 to nPopsize   % create new generation of genomes
                gParentA := Pop(gsParents);
                gParentB := Pop(gsParents);
                gChild := CrossNpt(gParentA, gParentB, 3, 0.8); % 3pt cross, 20% clone only
                Push(gChild,gsCache);
        endfor

        forall gChild in gsCache        % mutate and evaluate the new generation
                Mutate(gChild, 0.01);   % bit-wise mutation rate of 1%
                EvalOneCount(gChild);
        endforall

        Swap(gsPop, gsCache);           % swap the new generation for the old

        StatsPrint(i,10,gsPop,sFile);   % collect population statistics every 10 generations
endfor
endplan
```

Hybrid population structure example

```
% This plan shows a more advanced use of RPL2.  Here, the population is a set of islands,
% each of which contain a fine-grained set of genomes. A generational update is performed
% by crossing the best and a random genome at each point to get a new population.

plan(CombinationExample)

%% Preliminary section - declare libraries and the population structure
% declare the population template and the distance metric which defines the deme
% Note '@' means cyclic (toroidal) boundaries for that axis and ':' is a no-wrap boundary

use StdInst, Set(200);                  % use a 200 element set representation
structure [5:island, 5:island, 10@fine, 10@fine] deme Taxicab(2);

%% Declarations section - declare all variables that we use

int     iGeneration, nGenerations;
bool    bMaxIsBest;

% These declarations correspond to the entire population structure (5x5x10x10 elements)
gstack  [*,*,*,*]       gsTmp;
genome  [*,*,*,*]       gPop, gMate, gSelf;

% These arrays correspond only to the island axes of the population (5x5 elements)
gstack  [*,*,-,-]       gsImmigrants;
genome  [*,*,-,-]       gBestImmigrant, gEmigrant;

%% Instructions section - the executable code for the plan

nGenerations := 100;
bMaxIsBest := FALSE;
Randomize(0);                           % pseudo-random initialisation

structfor [*,*,*,*]                     % loop over everything to create the population
        gPop := RandomGenome();         % generate a random initial population
        Squash(gPop);                   % simple evaluation function
endstructfor

for iGeneration := 1 to nGenerations    % loop for a fixed number of generations

        structfor [*,*,-,-]             % loop over the island axes

                structfor [*,*]         % loop over remaining (fine grained) axes

                        DemeCollect(gsTmp,gPop);        % collect each genome's neighbours
                        gSelf := SelectRawTournament(gsTmp, bMaxIsBest, 2, 0.7, FALSE);
                        gMate := SelectRandom(gsTmp);
                        gPop := rarw(gSelf, gMate, 1);  % RAR-1 operator
                        Mutate(gPop, 0.02);             % set mutation
                        Squash(gPop);                   % evaluate the new population

                endstructfor

                % now possibly migrate the best member of each island to a random island
                % and replace a random member of an island with the best immigrant

                gEmigrant := ReduceRawBest(gPop, bMaxIsBest);
                MigrateRandom(gsImmigrants, gEmigrant, 0.1);
                gBestImmigrant := SelectRawBest(gsImmigrants, bMaxIsBest);
                ProjectRandom(gPop, gBestImmigrant);

        endstructfor
endfor
endplan
```