Digital Business Ecosystem

Contract number° 507953

Workpackage 8
Population dynamics in the Evolutionary Environment

Deliverable 8.2
Report on population dynamics for variable-sized structures

Information Society Technologies

Project funded by the European Community under the "Information Society Technology" Programme

Contract number: 507953
Project acronym: DBE
Title: Digital Business Ecosystem

Deliverable N^{o}: D8.2
Due date: 31/12/2005
Delivery date: 31/12/2005

Short description: This deliverable is a report on our work on evolutionary dynamics for variable-sized structures. It is the first report of sub-task S4: Population dynamics in the evolutionary environment.

Author: UBHAM
Partners contributed: STU, LSE, HWU
Made available to: DBE Consortium and European Commission

Versioning		
Version	**Date**	**Author, Organisation**
1.0	02/12/2005	J. Rowe, D. Chu, J. Woodward (UBHAM)

Quality check
1st internal reviewer: Gerard Briscoe (HWU)
2nd internal reviewer: Thomas Heistracher (STU)

Contents

Executive Summary

This deliverable is a preliminary report on our work on evolutionary dynamics for variable-sized structures, in sub-task S4 (Population dynamics in the evolutionary environment). This sub-task will be completed by month 36, when a full report (D8.3) will be delivered. The report is in two main sections. In the first part, we report on various ideas for the representation of services and collections of services in the DBE. This work has come about through discussions with STU about the implementation of the Evolutionary Environment. This part includes a proposal to extend the formal representation of the set cover problem to a dynamic scenario, in which user requirements can change from time to time. We are currently working with STU to develop this model using SBVR data structures, and this will form a key part of our further work.

The second part concerns population dynamics for variable-sized structures and describes some theoretical and empirical work on the causes of *bloat*. This is the phenomenon in which larger and larger structures evolve for apparently no purpose. We look at some very simple scenarios and show that bloat can be caused by very small features of the fitness function. Moreover, finite population behaviour can be quite intriguing in such systems. This part is under submission to the Artificial Life conference. We conclude by describing the possibilities for bloat, and its avoidance, within the EvE.

Chapter 1

Introduction and relationship to the project

This deliverable is the first report for sub-task S4 (Population dynamics in the evolutionary environment), which is scheduled to run from month 19 to month 36, and builds on the work of S3, already reported [18]. The objectives of this sub-task are:

1. To study population dynamics of variable-sized structures. This will involve both empirical studies, as well as a theoretical extension of existing work on limit theorems for variable-sized strings.

2. To investigate the effects of a changing environment on evolution. Such changes may come from external forces (e.g. changes in users' requirements) or internally (e.g. from the migration of information).

3. To inform the development of DBE Evolutionary Environment, by liaising with STU and TCD with regard to representations, operators and fitness definitions.

We have done a number of empirical studies of an abstract evolutionary system with variable-sized elements in relation to the first objective (reported in the second main part of this report). Dominique Chu was employed for three months to work on this topic. He has now left UBHAM, and the remaining theoretical work will be done by Jon Rowe. Dominique also liaised with STU helping them in the development of their EvE simulator. Since his departure we have now recruited John Woodward, who continues to work on objective 3. This work is reported in the first main part of the report. We have just begun to consider the second objective, with an abstract model called the *dynamic set cover problem*. This is described briefly at the end of the first part.

This work provides a foundation of a fundamental "scientific value chain" by investigating from an abstract point of view, the dynamics that can be expected within the EvE, and a consideration of the types of problems that might arise in both its implementation and execution. In accordance with the third objective, we have maintained a close link with STU (less so with TCD, who are no longer involved in the EvE

development). We have also worked closely with Gerard Briscoe of HWU in the development of the basic architecture of the EvE. As part of the second objective, we need to consider the impact of moving from the ontology-based BML 1.0, to the more complex BML 2.0 (SBVR). While this is of primary concern to STU and ISUFI, we are assisting in assessing the impact of this move for the evolutionary fitness function. The models we have studied so far, however, (and described below) are independent of the particular BML version to be used: we simply assume that there will be some description of services and requirements, and some means to match these.

Chapter 2

Variable-size representations of service combinations in the EvE

2.1 Description of the problem

A user makes a request to the system, and the system searches for a set of services which meet the request and at minimal cost. We give a number of definitions which are illustrated with everyday examples and should aid understanding.

Definition 1 (feature) *A feature, t_k, is considered atomic and cannot be decomposed into subcomponents. It exists as part of a service, specified by a BML expression.*

For example an aeroplane flight from London to Paris maybe part of a package holiday to Paris. The flight is atomic as it is considered as a unit (i.e. you cannot take half a flight). The user, once he has purchased the package holiday he could use whatever components he likes.

Note that there can be additional constraints. For example if a return flight is booked, and the user does not board the outbound flight, in some case airlines will invalidate the return flight.

Definition 2 (service) *A service s is a set of features $\{t_1, t_2, ..., t_n\}$. A service may contain one or multiple features. It is therefore described by a collection of BML statements associated with each feature, together with any constraints and conditions of use. A cost is associated with a service. A service is atomic in the sense that it can be purchased or not purchased, but is decomposable in the sense that one of its features may or may not be used.*

For example a package holiday to Paris may include flights to and from London, a week in a Parisian hotel and transfers from Charles de Gaulle International Airport to the hotel. This package holiday could be considered as three features (a return aeroplane flight, a week in a Parisian hotel and transfers to and from the airport).

A user can decide to purchase a certain service or not. If a particular service is selected, then all of the features contained in the service are availably to satisfy the user's request. However, the user does not necessarily require all of the features of the service.

Definition 3 (request) *A request r is a set of features $\{t_1, t_2, \ldots, t_m\}$. A request may contain one or multiple features. It is specified as a BML query.*

It is important to notice that the features required by a user may not actually exist as features of services that exist. In this case, the user will, in the first instance, have to be satisfied with the closest match available. However, in the longer term, this may trigger the generation of a service to match the requirement (perhaps partially automatically).

2.2 Representations of service combinations

When a user presents a request to the EvE, it is unlikely that a single service will be able to satisfy all the requirements. The system must therefore seek a combination of services. We need some way to represent such combinations. There are a number of candidate representations we should consider.

2.2.1 Set Theory Representation

Given that a service is a set of features, and combining a set of services results in an overall service which consists of the union of the services, it appears that representing services as sets of features maybe a sensible thing to do, if there is no consideration of additional structure (such as the order of the component services).

For example a request $r = \{t_a, t_c\}$ could be satisfied by the set of services $s_1 = \{t_a, t_b\}$ and $s_2 = \{t_b, t_c\}$ as $s_1 \cup s_2 = \{t_a, t_b, t_c\}$ and this satisfies the request (t_b being redundant as it is not needed to meet the request).

2.2.2 Graph Representation

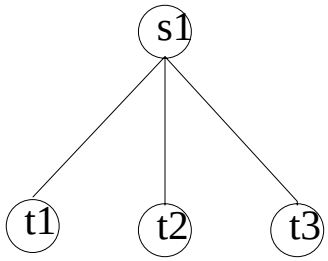

Figure 2.1: A graphical representation a service consisting of 3 features. Here it is shown with its components parts, however in the system it will only be available as a single node s_1 as we cannot cherry pick individual features from a service.

While a set theory representation may be a satisfactory representation in the simplest case, using set notation can lose information about sets of services (i.e. which feature in a potential solution came from which service). This information is important if we want to be able to recombine collections of services (using a crossover operator) in a meaningful way. In a tree based representation, we can represent services which are available as leaf nodes, and a combination of services as a node linking leaf nodes (i.e. non leaf nodes) — see figure 2.1. A composite service is represented as a tree (figures 2.2 and 2.3). Standard Genetic Programming [11] makes use of such tree structures, and there is a wealth of research into appropriate operators for this kind of representation.

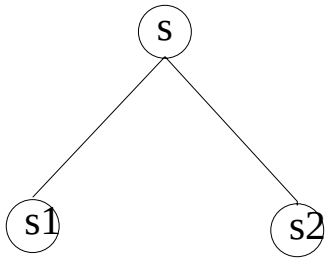

Figure 2.2: A combination of services s is obtained by combining services s_1 and s_2.

2.2.3 Ordered Representation

One problem with a tree based representation is that there is a many to one mapping between the genotype space (that is, the set of possible trees) and the phenotype space (the actual composite services). There are many ways of representing the combination of services in figure 2.3, but the result will be $\{s_1, s_2, s_3\}$. Similarly, if we were using the set theory representation there are many ways to represent the result of a union of sets (e.g. in this case 3 sets can be combined in 6 different way).

One potential way to reduce this redundancy would be to order the services so that there is only one representation of a combination of services. As the size of a subtree in the proposed solution grows, the number of ways it can be expressed grows exponentially. However the cost of reordering it will only grow as $nlog(n)$ (the complexity of quick sort). See [3] for information on ordered and unordered trees.

2.2.4 Hierarchical Sets

If the order of the services does not matter when they are combined, we can construct a representation which retains the hierarchical nature implicit in the tree representation. (It may be important to retain groupings of services in order to improve the system with reuse). In this representation, a new service is created by combining already existing services (by grouping them with an ellipse), and this newly created service can then be combined with other services (see figure 2.4). As the representation stores the way

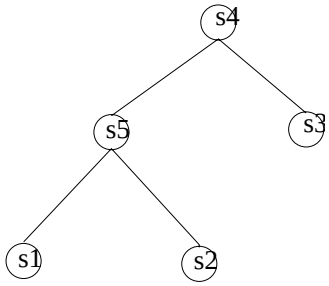

Figure 2.3: A service s_1 and s_2 are combined to give a novel service s_5. s_5 is then combined with s_3 to make a new service which we call s_4.

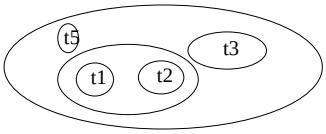

Figure 2.4: A combination of services in hierarchical set representation.

services are combined, we retain potentially useful hierarchical information. The use of *unordered trees* in Genetic Programming has been considered in [1].

2.3 Representation constraints on genetic operators

The type of representation chosen will to some extent decide what genetic operators are suitable. A mutation operator takes a single instance of a representation and transforms it into a new instance of the representation. A crossover operator takes a pair of instances of a representation, and exchanges 'sub parts' of the representation to produce two new instances of the representation.

Given that we are dealing with a hierarchical sets representation, we need to invent sensible operators tuned for this type of representation. As the hierarchical sets representation is essentially the tree based representation with an ordering on the members of any set (to reduce redundancy), it seems sensible to take standard tree based genetic operators and adapt these to our hierarchical sets representation.

A mutation operator can add or delete services from the hierarchical set. Hence any collection of services could be deleted or added. Thus in figure 2.4, any single service could be deleted or added to, but t_1 and t_2 could be deleted as a pair as they belong to the same service (the atomic level at which the operators can act). t_1 and t_3 could not be deleted by the same mutation operator as they belong to different services. Typically mutation operators use a uniform selection method, however given the nature of the problem in this project it is worth attempting to allow the mutation operators to self adapt (that is, tuning the probability of mutation depending on how successfully the evolution process is progressing). In addition to this, it is worth mentioning that if a service does not contribute to the current request then it may be worth engineering a

mutation operator to remove non-contributing services. However, given that we choose a fitness function which penalises more costly solutions (with the same number of requests met), then these types of services will naturally be lost.

The crossover operator works under similar constraints as the mutation operator (i.e. only operating on services in the same set). This is an adaption of sub-tree crossover used in standard tree style genetic programming, where two subtrees in two separate parents are exchanged. Thus, in the case of a hierarchical sets representation, some subset of services in one candidate solution can be exchanged with some subset of services in a second candidate solution.

2.4 Possibilities for Fitness Functions

There are a number of possibilities and constraints affecting the choice of fitness function. The topic has been looked at in depth by STU (see deliverable 9.1 [17]). Here we summarise some of the basic options.

2.4.1 Simple Fitness Functions

One possibility for a fitness function is to return the fraction of features in the request that are met. For example if the request consists of 9 features and 5 of features are present in a solution, then 5/9 is the fitness of the solution.

A refinement of this is to weight each of the features in the request with a value indicating their importance. It is of course not immediately clear what these weight values should be, and could be user defined and will depend on the problem. Another possibility is to order the features in terms of their importance, and this would avoid the weighting issue. However, the order which is used to differentiate final solutions is not necessarily a good ordering to drive the evolution.

2.4.2 Taking Cost into Account

The above fitness functions does not take cost of the services into consideration. One possibility is to add a term to the fitness function, but it is not immediately clear what is the best or even a good way to achieve this.

One possibility is to have a two stage fitness function, where two solutions are compared firstly on the number of features in the request they satisfy, and if they meet the same number then they are compare on cost. With situations like this, where there are multiple objectives (i.e. the cost and number of features), we should return a *Pareto set* of solutions.

Alternative, we could use *binary tournament selection*. Here, to select an individual from the population, we first choose two individuals completely randomly. We then compare them, and select the best one. The advantage of this is that we do not need to assign a numerical fitness value to solutions. We only need a procedure for comparing them. We propose:

1. If both solutions satisfy all the requirements, choose the cheapest one.

2. If neither solution satisfies all the requirements, choose the one that satisfies most of them.

3. If one solution satisfies all the requirements and the other doesn't, choose the one that satisfies them.

2.4.3 Diversity of Solutions

Typically a fitness value is a scalar value (i.e. a single number) indicating how good a solution is. It may be better, however, for the fitness function in the EvE to return a vector indicating which of the features are satisfied. For example, consider a request consisting of 10 features $r = \{t_1, ..., t_{10}\}$, and three solutions A, B and C, which each have a fitness of 5 (i.e. they meet half the requested features). A and B both satisfy the first 5 features and C satisfies the last 5 features. Combining A and B into a solution$A \cup B$ will still give a solution with a fitness of 5 (i.e. meeting the first 5 features). Combining A and C into a solution$A \cup C$ will still give a solution with a fitness of 10 (i.e. meeting all the features).

Given these above potential fitness functions, it is clear that proportional selection may not be the best choice of selection method. It may be difficult to quantify how much better one solution is compared to another. In this case either rank selection or tournament selection would be appropriate as these selection methods just require a knowledge of whether one solution is better than another.

There are a number of assumptions which we can make at this stage. While these are not critical assumptions, they may help when simulating the system to evaluate its performance.

When a user makes a request, the number of features in the request is likely to be much smaller than the total number of features availably in all of the services. For example, a user is likely to request only a few item off a menu rather than the entire menu.

It is likely that the user will make a request which can be met by the services avaliable i.e. a request is likely to be realistic and therefore we expect it to be met.

It is expected that there will be a highly non-uniform distribution of requests coming from different users at different times. For example, people do not randomly select items off a restaurant menu (e.g. 4 starters), but are more likely to make certain types of selection (e.g. a starter, main course, dessert and a drink). This gives us the potential to bias the search procedure, by taking into account common correlations in users' requests (see next section).

2.5 The dynamic set cover problem

We have previously (see deliverables 8.1, 11.1 [18, 19]) described the problem faced by DBE habitats as a *set cover* problem. That is, each service provides a set of features and, given a user's specification, we need to find the cheapest collection of services which will cover the specified features. We have also seen that evolutionary methods are generally superior heuristics for solving this class of problem (see chapter 3 of

deliverable 8.1). However, in addition to solving the particular requirements of a user at a particular time, we also would like the system to adapt in the long-term to changing user requirements. Moreover, we would like the solutions found in one habitat for one user to be readily available to other users with similar sets of requirements.

We formally define this long-term problem as the *dynamic set cover problem*. Let $\mathcal{F}$ be the total set of possible features. Then, as explained above, a service s can be thought of as a (probably small) subset of $\mathcal{F}$. Similarly a user's requirements R is a subset of features. We suppose that we have an effective algorithm for solving the set cover problem, given a particular set R and a collection of services C comprising the service pool of a DBE habitat. Now suppose we have a *sequence* of requirements $R_1, R_2, R_3, \ldots$, which we have to solve one after the other. If these requirements sets are drawn randomly from $\mathcal{F}$ then there is possibly nothing we can do to make the search more efficient. However, if some features occur more frequently in the requirements sequence than others, then we can potentially speed up the solution of requirements by binding services together (see above for different ways of doing this).

For example, consider the following algorithm:

1. Given R, find a set of services that solve it, S.
2. Randomly choose two services $s, t \in S$.
3. Add $s \cup t$ to C as a new service in its own right. This creation of a new composite service occurs with probability p, which will be a parameter of the algorithm.
4. Go to 1.

The idea is that even though the binding probability p may be small, if pairs of services are repeatedly used together, eventually they will be aggregated into a larger service. This will then make the solution of similar requirements more efficient. Eventually, if a particular R is repeated many times, a complete solution to it will exist in the habitat as a single entity, and the solution will therefore be directly available.

There is a trade-off, however. Adding new items to the pool C will in general make each set cover problem harder, as the search space size will have grown. We may therefore make it much harder to solve for very rare combinations of features. A further problem is that services might be bound together by chance when they are not really needed in the future. We might therefore have a *decay* probability by which combinations of services break apart if they are not used.

As composite services are created in a habitat, they can then be re-used within that habitat to address future similar requests from the same user. They can also be migrate as a single entity to other habitats. If the neighbouring habitats in the EvE network correspond to users with similar profiles, it is likely that these composite services will prove useful in meeting their needs (without having to reassemble a similar solution from the atomic components). The overall efficiency of the EvE will improve then, if the network is configured (or adapts itself) so that users with similar profiles are more likely to be connected together. Similarly, composite solutions which do not prove useful in the long run should decay. It is necessary to stop the spread of such useless composites through the network. The expected time to decay (given that there is no reinforcement from within the native habitat) must be less than the expected time to

migrate. A balance must be sought here between how easy it is for habitats to share ideas, and the prevention of the spread of poor solutions. This balance is addressed by Gerard Briscoe in the EvE Architecture Requirements document (appendix B of [13]).

2.6 Implications for the EvE

We have looked at some possible ways in which services in the EvE may be composed. A fundamental distinction must be made between compositions in which the order matters (which leads to a tree-based representation) and those where it does not (leading to a hierarchical set structure). The services themselves are to be considered as atomic with respect to the genetic operators, but the tree and set structures which represent their compositions can be manipulated by them.

We have not addressed in any detail the specific representations of services and requests in terms of BML. We assume that each atomic service has such a description, and that there exists (or will exist) a mechanism for comparing such descriptions with user requirements, to assess the extent to which there is a match. Such a mechanism is relatively straightforward for the ontology-based BML 1.0. However, it is less simple for SBVR (BML version 2.0 - see ISUFI's report on SBVR [10] and the discussion document [4]) and STU are currently working on this matching problem. We have assumed, perhaps naively, that once a matching mechanism exists for atomic services, then the extension to composites should be straightforward. The extent to which this is true will depend on how complex these composites are allowed to be.

The dynamic set cover problem is a further abstraction, which looks at what happens if composite services, once created, are allowed to persist within an EvE habitat. If so, it is assumed they will also be able to migrate as a single entity, which will potentially increase the efficiency of the evolutionary algorithm. However, it is also important that composites be allowed to *decay* if they do not achieve sufficient success in their native habitat (see the EvE Architecture Requirements document [13]). Without this feature, a number of problems could emerge, not least of which is *bloat*; the generation of large composites made up of apparently useless components. The general study of the bloat phenomenon is the subject of the second part of this report.

Chapter 3

Population dynamics for simple variable-length problems

3.1 Introduction

Standard genetic algorithms (GA) [5] are traditionally used with a fixed length genome. This is suitable for many optimization tasks. In the context of Artificial Life or even biological problems the requirement of a fixed length genome is often very restrictive. There is strong evidence that genomes in real organisms exhibited very much growth over time [9]; beside that there are other features of organisms that are subject to size changes over time. In order to model/simulate the evolution of those features *in silico* variable sized GAs will often be a useful methodological tool.

There are relatively few attempts to use variable length GAs. Harvey introduced the species adaptation genetic algorithm [8] which allowed certain variations of genome sizes in GAs; this work was recently further developed by Bull [2]. Poli *et al.* presented theoretical results for GAs with variable length genomes [14, 15]. Other applications include Grefenstette *et al.* who constructed a GA to learn tactical decision rules [6], Wu and Garibay introduce the "Proportional Genetic Algorithm" [20]; this is a more biologically motivated version of classical GAs that uses explicit genes to encode information about the genome.

A common problem of variable sized genomes is that solutions tends to get infested with non-functional parts that hitchhike with fit solutions, a phenomenon commonly known as *bloat*. Bloat might lead to substantial increases of the genome size relative to what would actually be required. There are two main methods to prevent bloat. Firstly, one could apply a fitness penalty that correlates with genome size. This method is not entirely without its problems. A wrong choice of exactly how length is penalized might lead to good (but long) solutions being missed; this is particularly a concern in circumstances where the length of good solutions is not known.

A second possibility is to reduce redundancy at run-time by removing junk-entries in the genomes. The problem here is that it might not be obvious whether or not a particular part of the genome is actually redundant or not. Precisely how feasible this

approach is will depend on the specific circumstances of the application.

Instead of focusing on these methods to control bloat at "run-time," we will look at possibilities to reduce the effect in the context of variable length GAs through careful choice of the crossover operator. We define three fitness functions; each of these functions has a different bias for solutions of various sizes. We then compare the observed growth of the genomes for various crossover operators. This gives some indications about the inherent tendency of this operators to cause bloat.

The actual fitness functions we use are simple. In all experiments discussed below the optimal solution has been found within very short time (except for one case; see below). Hence, what varies throughout the simulations is not the fitness of the solutions but rather the length of the genomes as the GA explores neutral mutants of the optimal solutions. Throughout this contribution we will therefore concentrate on this aspect of genome length rather than on the fitness of the solution.

What is absent from these experiments is mutation. In all simulations shown below the mutation rate was set to zero. The reason for neglecting mutations is that they introduce a number of second order effects. Our studies have shown that these do lead to interesting effects; yet it is not clear how these effects are to be interpreted in a more general context. We decided therefore not to include the role of mutations into this report.

3.2 Description of the model

The model is a simple implementation of a GA. In all the experiments reported here we used a population size of 1000 and a tournament selection with tournament size 10. In all simulations we performed 5 million tournaments. For practical reasons it was necessary to set an upper limit for the length of the genome; this was necessary in order to prevent the occurrence of too large genomes that would exhaust the available computational resources. This limit was kept constant for all runs and set to 200000. The population was initialized with random strings of 1's and 0's. The initial length of strings was randomly chosen between 3 and 2000. The figures illustrating the changes of genome length over time were produced as follows: At every 1000 time steps the sizes of all genomes in the population at this time were recorded. After the simulation this data file contained 5 million entries. The figures in this article were produced by plotting every tenth point in these files. This reduction did not qualitatively change the graphs but substantially reduced the computational resources needed to produce and handle the relevant figure files.

3.2.1 Fitness functions

We experimented with three different fitness functions and crossovers. The first fitness function, ff_1, was inspired by the Ising model [7]. If s_i is the i-th entry on the genome

string of length L, then ff_1 is

$$ff_1(s_1, \ldots, s_L) = -\sum_i^{L-1} \frac{diff(s_i, s_{i+1})}{L-1}$$

$$diff(x, y) = \begin{cases} 0 & \text{if } x \neq y \\ 1 & \text{otherwise} \end{cases}$$

Note that we follow the convention that small fitness values are better than high fitness values. Note further that in the case of ff_1 the fitness is independent of the length, in the sense that the fitness contribution is averaged over the length of the genome.

The second fitness function simply measures the distance of the solution from a target contents of 18 entries of 1's in the genome. So, for example, a string entirely consisting of zeros would have a fitness of 18, whereas a string that is 100 long with exactly 18 1's would have the optimal fitness of 0. The second fitness function is:

$$ff_2(s_1, \ldots, s_L) = \sqrt{\left(\sum_i^L s_i - 18\right)^2}$$

Other than ff_1 this fitness function does have an certain degree of length bias. There is a minimum length (here: 18) which is required in order to get maximal fitness.

Finally the last fitness function ff_3 equates the fitness with the number of leading 1's multiplied by -1.

$$ff_3(s_1, \ldots, s_L) = -\max_k : \prod_i^k s_i = 1$$

Again, the better the solution the lower the fitness. In the case of ff_3 there is a strong length bias. The maximal fitness possible is equal to the length of the string.

3.2.2 Crossover operators

In the experiments described here we will use three different crossover operators. If the crossover points on both parents are chosen randomly then the length of the offspring will very often be different to the length of both parents. The exact difference and the influence of this on the time evolution of the GA will depend on the specific crossover operator chosen. In this article we considered three possible crossover operators. These will be described in what follows.

The first crossover operator O_1 is a straightforward extension of the fixed length case. For each of the two parent strings p_1 and p_2 the crossover point is chosen between 1 and the length of the string $L(p_x)$; denote the respective crossover points by p_{1,n_1} and p_{2,n_2}. The offspring will then be the new string composed of the left part of p_1 and the right part of p_2: $p_{\text{off}} = p_{1,1}.p_{1,2} \ldots p_{1,n_1}.p_{2,n_2}.p_{2,(n_2+1)} \ldots p_{2,L(p_2)}$.

One property of O_1 is that the offspring p_{off} might substantially differ in length from its parents. While some variation in length is desirable in variable length GAs,

too much of it may not be. An alternative crossover operator O_2 works according to the same principle as O_1 but the choice of the crossover points is constrained so that the length of the offspring does not differ from $L(p_1)$ by more than a fixed number; in all experiments reported here this number was kept fixed at 10. The third crossover operator works the same way as O_2 but the difference between $L(p_1)$ and $L(p_{\text{off}})$ may be up to 10 percent of p_1. In practice the operators O_2 and O_3 are implemented as follows:

1. Choose a target length for the offspring.
2. Choose uniform random crossover points in parent 1 and parent 2. The crossover point of parent 1 is chosen so that the segment of the genome to the left of the point is shorter than the target length.
3. The first part of the offspring genome is the left part of parent 1 (as in O_1).
4. Fill up the remainder of the offspring genome with part to the right of the crossover point of parent 2.
5. If the offspring genome has not reached the target size, then the rest of the genome is filled with random entries (i.e. 0s and 1s chosen with equal probability). We will call this *padding*.

3.3 Empirical results for simple fitness functions

3.3.1 Flat fitness

We performed a number of simulations to understand the behavior of the variable length GAs. In what follows we are primarily interested in the lengths of the solution rather than in their fitness. In order to understand the inherent biases of the crossover operators we performed a number of simulations with a flat fitness function (that is all genomes have equal fitness).

Figure 3.1 shows results from example runs for the time evolution of the GA in a flat fitness-landscape under the three crossover operators. The behavior of operator O_1 shows strong quantitative variations both between runs and over the course of a single simulation, although the qualitative behavior does not change between runs. The standard deviation is typically close to the mean; for example in the particular run shown in figure 3.1 the mean length of genomes taken over the entire simulation is just under 380 with a standard deviation of 346. This large deviation of the actual behavior from the mean behavior is a consequence of the clearly visible (in figure 3.1) intermittent explosions of the genome size. Larger genome-sizes are distributed roughly exponentially over the course of a simulation. Figure 3.2 shows the histogram of the distribution of the genome length in a simulation of O_1 in a flat fitness landscape; note that this simulation is from a different run to that in figure 3.1. It is in agreement with a theoretical prediction by Rowe and McPhee [16] of the behavior of an infinite population.

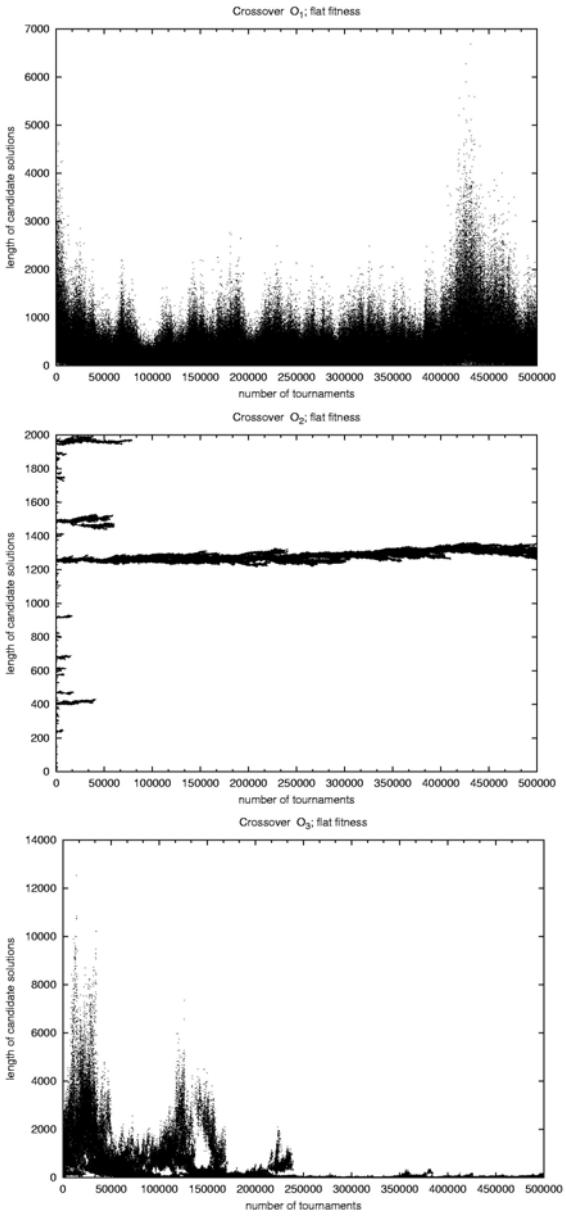

Figure 3.1: Time evolution effects of the three crossover operators in a flat fitness landscapes. From top to bottom are shown O_1, O_2 and O_3 respectively. The figures display the lengths of the entire populations recorded at regular time intervals; see main text for a precise explanation. The horizontal axis represents time and the vertical axis is the length of the candidate solutions. The figures show results from single runs.

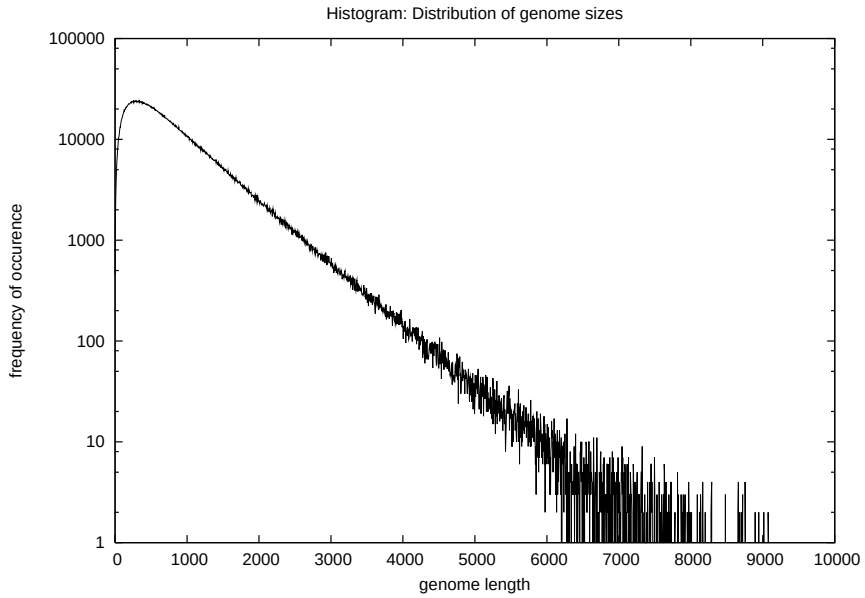

Figure 3.2: Histogram of the genome length recorded in a simulation of O_1 in a flat fitness landscape; note that this is a linear-log plot. Large genome lengths are roughly exponentially distributed.

Crossover O_3 shows qualitatively different behavior. As shown in figure 3.1 there are large genomes at early stages of the simulation. Over time the maximum sizes go down, although variations persist. Roughly in the second half of the run the genome sizes have substantially reduced. Closer inspection shows that in this area the largest sizes are around 200, apparently remaining stable over time from then on. Note again that this refers to a particular run; other runs show qualitatively similar behavior. This behavior can be explained by the properties of O_3. The possible size of the offspring is limited to be within 10 percent of the length of the parent. Hence the longer the genomes in the population the more variation one would expect; this variation can go in both directions, towards longer and shorter genomes, but once the population consists of short genomes only, the possible variations per crossover event are smaller; as a results there will be less growth in absolute terms; short genomes thus act as sinks.

Finally, the second crossover operator shows a similar effect yet with a different outcome. One genome size "hoovers" up all others. A closer inspection of figure 3.1 shows that O_2 starts with the lengths well distributed over the initially allowed range between 0 and 2000. After a short time, only 4 relatively narrow bands of genome sizes remain; after about a fifth of the simulation time all but one of them have died out and all genomes are in one single narrow band.

This effect is explained by a process similar to size related growth. Note that independent of how long a genome is, it's offspring can always only differ from parent 1 by at most 10. Furthermore, note that the offspring created replaces a randomly chosen

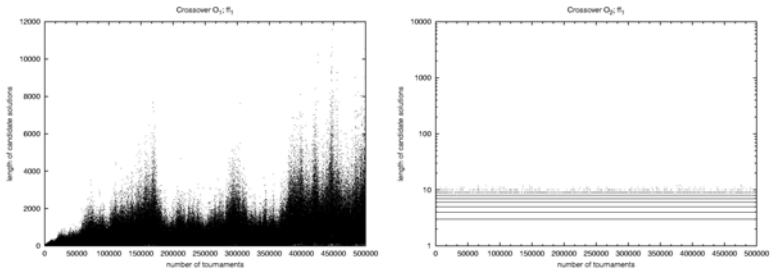

Figure 3.3: Time evolution of the genome sizes for the operators O_1 (left) and O_2 right.

member of the population. After inserting the offspring into the population, the number of genomes that are within 10 of parent 1 has either stayed the same or increased by 1. Remember that parent 1 is also chosen at random. The more genomes there are in a particular size band the more likely it becomes that the next chosen parent 1 is from this band and hence the more likely it is that the number of genomes in this band grows by 1. The effect observed here is closely related to well known examples of spontaneous symmetry breaking in complex systems.

The width of the observed band depends on the allowed absolute length change between parent 1 and offspring. In the limit of very large allowed changes, the system would approach the behavior of operator O_1. By the same token, smaller allowed variations lead to narrower bands.

In summary, experiments with the three crossover operators show their different characteristics. Operator O_1 approaches an exponential distribution. Operator O_3 on the other hand does have a bias for shorter solutions, in the sense that once there are only short genomes in the population, genome lengths will remain short. Finally, operator O_2 has a limited ability to explore various genome sizes, particularly once the population has converged.

3.3.2 Introducing Fitness

In this sections we will describe results obtained with the three crossover operators and the three fitness functions.

The first fitness function does not have a strong length bias as solutions of all sizes can acquire maximal fitness. Simulations with O_1 show that the optimal fitness is found within very short time (data not shown); from then on only individuals with the optimal solution appear in the population. At early stages of the simulation only short genomes are retained. This is readily explained by the fact that long genomes are very unlikely to have uninterrupted long stretches of either only 1's or only 0's (and therefore good fitness); very short random genomes are not only more likely to have good fitness but it is also easier to improve their fitness by a few crossovers only. This corresponds to initially very short genome sizes in the simulations with O_1 (in figure 3.3). In due course the GA also explores longer solutions. Due to the particular characteristics of the Ising model fitness function, once solutions are found they can easily be combined

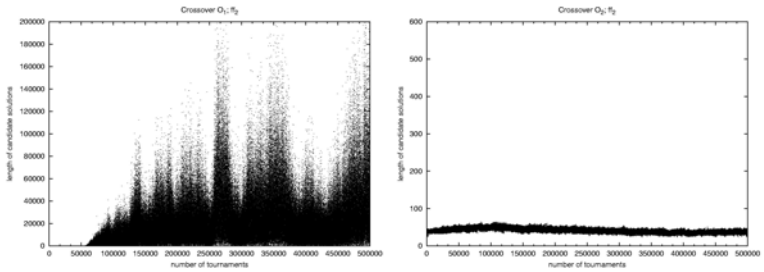

Figure 3.4: Time evolution of the genome sizes for the operators O_1 (left) and O_2 right.

via crossover to give new optimal fitness solutions. Once all sub-optimal solutions have been removed from the population, the application of ff_1 is essentially the same as a flat fitness for the operator O_1. This is also confirmed by the distribution of the genome sizes; this distribution (data not shown) is (after an initial period) identical to the flat fitness case.

The qualitative behaviors of O_2 and O_1 are very similar to each other but different to O_1. They lack a substantial exploration of longer optimal solutions and throughout the simulation remain essentially restricted to short solutions. The reasons for this "hoovering" effect described above (in the case of O_2) and the bias for short sequences (in the case of O_3). Another causal factor is padding. Padding makes growth difficult because padded solutions will normally be less fit than unpadded solutions (due to the added random entries in the genome). Experiments with variations of O_2 and O_3 that lack padding, however, show very similar results (data not shown) to the ones with padding. We conclude from this that padding has only a small influence on the genome length.

The second fitness function has a minimum length required in order for the genome to acquire optimal fitness; above this minimal genome size there are many solutions with optimal fitness. When using ff_2 there is thus a minimum size for the genome below which solutions cannot compete (at least after the short initial period required for the system to find one optimal solution). Using O_1 with this fitness-function leads again to an exponential distribution of the genome sizes, however, the maximal solutions reaches the cap of 200000. It is unclear whether or not the maximal solution would be bound in an uncapped version of the GA. Operators O_2 (see fig. 3.4) and O_3 (data not shown) show qualitatively similar behavior. During the first half of the simulation they settle on a symmetric distribution around a mean of about 45. This then falls to somewhat lower mean lengths between 25 and 35 (depending on the run). Common to all simulations is that (after a transitional period) the system never shows genome sizes that are substantially longer than that.

The third fitness function has an inherent bias for the long solutions; the optimal solutions to ff_3 must be the longest allowed in the system. In the present case this is a genome with the length equal to the cap size (200000). Figure 3.5 shows that both operators O_1 and O_3 quickly lead to this optimal fitness solution. Closer inspection shows that in both cases the population is dominated by genomes equal in length to the

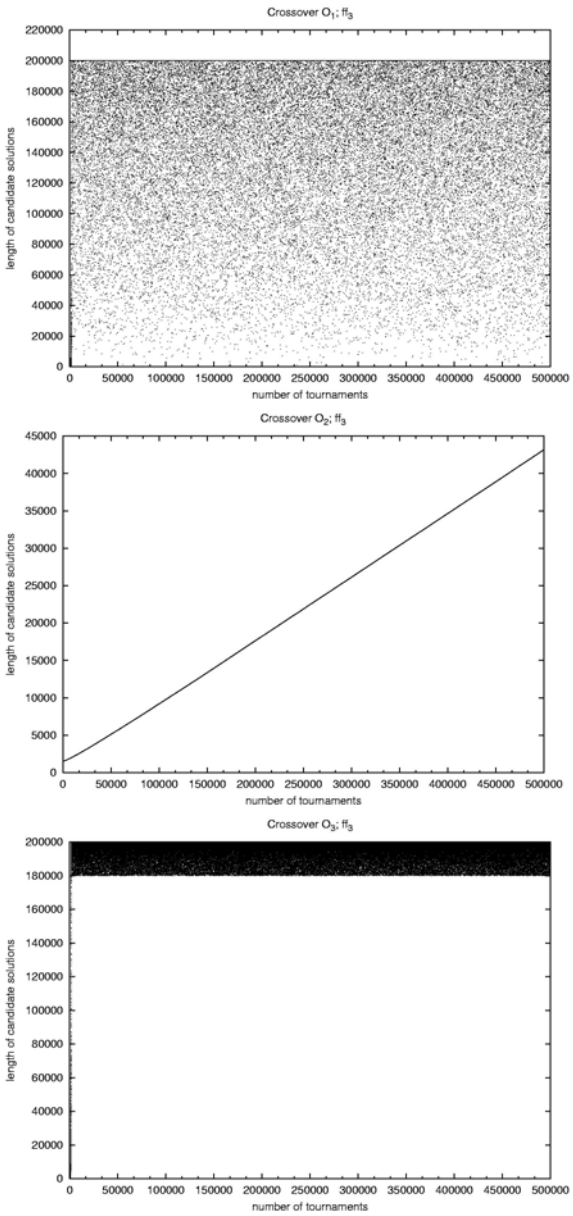

Figure 3.5: Time evolution of the genome sizes for the operators O_1 (top), O_2 (middle) and O_3 (bottom).

capsize; all other sizes are substantially less frequent (data not shown). In the case of O_3 explorations of alternative solutions is restricted to a relatively small band around the capsize; note that every solution shorter than 200000 will be immediately weeded out. So, the band represents solutions that are one crossover away from the optimal solution; the minimal genome length is thus 180000 which represents a length change of 10 percent of 200000. Similarly, in the case of O_1 we also only see solutions that arise by one crossover, yet the possible change of length is greater in this case; this is reflected by the wider range of sizes in figure 3.5. A very different picture is offered by O_2. The population shows linear growth over the course of the simulation. Yet, the possible increases in size do not allow the system to find the best possible solution within the simulation time.

3.4 Discussion

One of the foreseeable practical problems of variable length GAs is bloat. Whether or not this is desirable is mostly a question of the specific problem and the requirements to a solution. Furthermore, whether or not bloat will occur in a particular application of a variable size GA will also depend on the specific circumstances, the fitness-function and the density of good solutions among longer genomes. As such, the present results are limited in their generality as would be any general study. Despite those shortcomings, we believe that the chosen fitness functions and operators give at least some indications about their behavior in a real world application.

The experiments with a flat fitness function (see figure 3.1) indicate the main characteristics of the chosen operators. The first operator settles on a roughly exponential distribution; this confirms a recent theoretical result by Rowe and McPhee [16]. Even though the probability for very long genomes falls exponentially, the actually observed genomes are nevertheless very much longer than the ones that occur in the case of O_2 and O_3. In practice this means that O_1 does have an innate tendency for bloat. In the case of the flat fitness function the actually observed longest genomes were in the order of 10000 long. Applying a bias for longer genomes changes the situation drastically: The fitness-function ff_3 has a very strong bias for long sequences, in that the optimal fitness can only be achieved by the longest possible solutions; in this case it is therefore not surprising that nearly all genomes actually take the maximal length. The bias in ff_2 is much more subtle in the sense that there is a minimum required length for the optimal fitness but above that no better solutions can be found (although there are many genomes with the optimal size). This minimum length of the optimal solutions was set to 18. This is relatively short compared to the longest and even compared to the mean length observed in the flat fitness case. One way to think about it is to interpret the flat case as having a minimum length of 2. The added bias of ff_2 would then be very small relative to the mean length observed in the flat case particularly small relative to the maximum genome size observed in that case. One would therefore expect that this slightly stronger bias has at most a weak effect.

The simulations here indicate, however, that this is not so. In the case of O_1 even the modest added bias in ff_2 led to an increase of the sizes of the longest observed genomes by several orders of magnitude. In fact, the maximum genome size was equal

to the capsize. Note that this increase in size did not confer any fitness benefit to the solutions but were neutral. Altogether this suggests that in O_1 bloat does occur even in the absence of a bias for longer solutions but much stronger in the case of even a modest bias for longer sequences.

In the case that long solutions are a problem, operator O_1 is likely to be a sub-optimal choice; this might be different, however, in situations where good solutions are known to be restricted in length to a narrow band of values or even to a few values only. In this case, bloat will not be a problem, but the inherent properties of O_1 will lead to the exploration of a wide range of solutions of various sizes. In such a setting O_1 might be a good choice.

A crossover operator that would not do very well in such a situation is O_3. The experiments show that the capability of this operator to explore solutions of various lengths is rather limited. In the case of a flat fitness functions O_3 locks itself into a narrow range of values; similar behavior is observed when fitness functions are introduced. Once the population has converged to a certain genome length no big length variations can happen any more. This has the effect that bloat is substantially reduced; but it also leads to an inflexibility in the case where optimal solutions are outside the range of initial values of the population and/or outside the range of an initial convergence of the population size. This is particularly well demonstrated by the simulations of O_3 with ff_3 (see figure 3.5); here the operator cannot keep up with the size changes required to find the best possible solutions within the given time. This operator thus seems to be fairly good at avoiding bloat, at least when compared to O_1, but does so at the expense of not being able to explore larger intervals of genome sizes.

In a sense in-between operators O_1 and O_3 is operator O_2. At least in the test problems investigated here it avoided bloat in the case of fitness functions ff_1 and ff_2 but was able to quickly find the best possible and longest possible solution in the case of the fitness function ff_3.

3.5 Outlook and Conclusion

Altogether it thus appears that of the three operators investigated here, O_2 represents a useful combination between flexibility to explore solutions of various sizes and an inherent bias for shorter genomes that avoids bloat, at least in some circumstances. On the other hand, our experiments indicate that operators O_1 and O_3 are perhaps not useful except for applications that have very specific requirements. There may be applications where the user wants to restrict the length variations of the solutions or would like to explore very long solutions.

Only real practical applications can show to what extend the results presented here will generalize to arbitrary fitness functions. These experiments however do indicate some general tendencies of the operators under investigations; this will be useful as a general guideline for the practitioner who wishes to choose a crossover operator for a specific optimization problem.

There are several ways in which the current work can be extended. First of all it is desirable to mathematically formulate and prove properties of the behavior of the population under various operators and fitness functions. This is most likely only

possible for the case of flat fitness and very simple fitness functions. At least for the case of an infinite population and the operator O_1 this has already been done [16].

Future experimental work will need to explore the effects of various population sizes. The experiments presented here assume a rather large population size of 1000. Such population sizes might not be realistic in practical applications. Finally, and most importantly the present experiments need to be compared to harder problems. The fitness functions used here are very much toy functions; they were chosen to investigate the specific aspects of the GAs. Real problems will be normally very different in that good solutions will be rather rare. It is still unclear to what extend this influences the present conclusions.

Chapter 4

Bloat within the EvE

4.1 Causes of bloat

The second part of this report has so far considered a very abstract model of the evolution of variable-sized structures. We conclude with some comments about the relevance of these observations within the setting of the DBE's evolutionary environment.

The first, and most obvious, comment is that it is essential to take into account the *cost* of providing services with an appropriate accounting model. Not taking cost into account is a prime cause of bloat. For if services can be added to a composite for free, then there is nothing to stop this happening. And since there are many more large composites than small ones, the service pool will quickly be dominated by the large bloated composites. For example, if a user requests a hotel in Paris, then certainly the service "Grande Hotel, Paris" will satisfy this request. But so will the composite service "Grand Hotel, Paris + Hire car", and the composite "Grand Hotel, Paris + Hire car + restaurant booking". More bizarrely, the composite "Grande Hotel, Paris + Metro Hotel, Lisbon" will also work. These useless composites will quickly swamp the pool if they are not correctly costed.

Secondly, it would help control bloat considerably if the *decay* mechanism described earlier was implemented. That is, there must be some way to break apart composites as well as put them together. Otherwise, once they are formed, the system is stuck with them.

Thirdly, the choice of operators (crossover and mutation) must be carefully made. As described above, and in [16], different operators have different biases towards generating certain sized objects. The operators will naturally create composites around their preferred size distribution, unless kept in check by a relatively strong selection pressure. One should at least seek operators that are "size neutral" in that the expected size of the offspring is the same as the average size of the parents.

Lastly, there is a subtle influence creating bloat that cannot be controlled for. This is the phenomenon of *hitch-hiking*. Suppose a service s_1 gets combined with service s_2. Additionally, suppose that, by chance, a number of users make request that can be satisfied by s_1, but for which s_2 is irrelevant. The composite $s_1 + s_2$ therefore has a

high fitness, and so service s_2 gets selected for despite its apparent irrelevance to the problem. It is said to be hitch-hiking on s_1, and is a form of bloat. This can be hard to detect, as it is not always easy to determine which aspects of a complex composite are really responsible for the gain in fitness and which are hitch-hiking. For example, the measure of population complexity developed by HWU in deliverable 6.2 [13] is blind to this type of bloat, and therefore not able to control it. The use of a targeted (as opposed to random) decay operator may provide some solution to this problem.

4.2 Conclusions

Bloat is a potentially serious problem for a distributed evolutionary system such as the DBE. If the growth of service chains gets out of hand, the efficiency of finding good solutions to user requirements will be considerably impaired. Three recommendations to help control the problem are:

- A correct costing scheme. This is essential, although the system will be open to malicious attack by, for example, virus writers who add their code to the DBE for free. One possibility is to charge providers a fee.
- Getting the migration-decay balance right. This will need careful experimentation.
- Implementing size neutral genetic operators. This cannot of itself prevent bloat, but at least will not encourage it.

Even with these in place, there is still the possibility that hitch-hiking may lead to bloat. We will need to monitor the long-term evolution of the habitats to investigate the degree to which this is happening.

Bibliography

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, January 1998.

[2] L. Bull. Coevolutionary species adaptation genetic algorithms: A continuing saga on coupled fitness landscapes. In M. Capcarrere, A. Freitas, P. Bentley, C. Johnson, and J. Timmis, editors, *Advances in Artificial Life : 8th European Conference, ECAL 2005, Canterbury, UK, September 5-9, 2005, Proceedings*, pages 845–853. Springer, September 2005.

[3] Narsingh Deo. *Graph theory with applications to engineering and computer science*. New Delhi : Prentice-Hall of India, 1974.

[4] LSE (P. Dini). SBVR consensus document. Version 6.

[5] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[6] J. Grefenstette, C. Ramsey, and A. Schultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5:355–381, 1990.

[7] G. Grimmett. *Percolation*. Springer Verlag, 1989.

[8] I. Harvey. Species adaptation genetic algorithms: a basis for a continuing SAGA. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems*, pages 346–354, Paris, France, 1992. MIT Press, Cambridge, MA.

[9] L. Hsieh, L. Luo, F. Ji, and H.C. Lee. Minimal Model for Genome Evolution and Growth. *Physical Review Letters*, 90(5):101–104, 2003.

[10] ISUFI. BML framework, second release. DBE Deliverable 15.3.

[11] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[12] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pan, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 1997.

[13] ICL (now HWU). Control of self-organisation and a performance measure. DBE Deliverable 6.2.

[14] R. Poli, N. McPhee, and J. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, 2004.

[15] R. Poli, J. Rowe, C. Stephens, and A. Wright. Allele diffusion in linear genetic programming and variable-length genetic algorithms with subtree crossover. In *EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming*, pages 212–227, London, UK, 2002. Springer-Verlag.

[16] J. E. Rowe and N. F. McPhee. The effects of crossover and mutation operators on variable length linear structures. Technical Report CSRP-01-7, University of Birmingham, School of Computer Science, January 2001.

[17] STU. Report on the fitness landscape. DBE Deliverable 9.1.

[18] UBHAM. Report on the evolution of high-level software components. DBE Deliverable 8.1.

[19] UBHAM. Report on the flow of software components in a static network. DBE Deliverable 11.1.

[20] A. Wu and I. Garibay. The proportional genetic algorithm: Gene expression in a genetic algorithm. *Genetic Programming and Evolvable Hardware*, 3(2), 2002.