Modelling biological evolvability: implicit context and variation filtering in enzyme genetic programming

Michael A. Lones, Andy M. Tyrrell

Intelligent Systems Research Group, Department of Electronics, University of York, York YO10 5DD, United Kingdom

Abstract

This paper describes recent insights into the role of implicit context within the representations of evolving artefacts and specifically within the program representation used by enzyme genetic programming. Implicit context occurs within self-organising systems where a component's connectivity is both determined implicitly by its own definition and is specified in terms of the behavioural context of other components. This paper argues that implicit context is an important source of evolvability and presents experimental evidence that supports this assertion. In particular, it introduces the notion of variation filtering, suggesting that the use of implicit context within representations leads to meaningful variation filtering whereby inappropriate change is ignored and meaningful change is encouraged during evolution.

Key words: genetic programming, evolvability, implicit context, variation filtering

1 Introduction

Enzyme genetic programming [Lones and Tyrrell, 2001a,b, 2002b, 2003] is a form of genetic programming (GP) [Koza, 1992] which uses a program representation modelled upon biological enzyme systems. The approach is motivated both by the limitations of conventional GP representations and by the premise that biological representations are well adapted for representing entities undergoing evolutionary processes, irrespective of whether these entities are biological or non-biological in nature. The logic behind this reasoning

Email addresses: Michael.Lones@bioinspired.com (Michael A. Lones), Andy.Tyrrell@bioinspired.com (Andy M. Tyrrell).

has been addressed in earlier papers on enzyme GP [e.g. Lones and Tyrrell, 2001b]. These papers also present comparative analysis of the performance of enzyme GP [Lones and Tyrrell, 2002a], the development of the enzyme model [Lones and Tyrrell, 2001b, 2002b] and the evolution of solution size [Lones and Tyrrell, 2002a,b].

The aim of this paper is to give further insight into the properties of the program representation used by enzyme GP, and in particular to verify that the representation used by enzyme GP is able to support meaningful program evolution. An important issue addressed by this paper is the evolvability of different forms of representation in GP and the potential sources of evolvability within enzyme GP. For more information on evolvability in GP and biology, see Conrad [1990], Kirschner and Gerhart [1998], Altenberg [1994] and Wagner and Altenberg [1996].

The paper is structured as follows. Section 2 compares approaches to program representation in GP and develops the notion of implicit context as an important mechanism for preserving the meaning of program components during evolution. Section 3 describes how implicit context is implemented in the program representation of enzyme GP. Section 4 presents experimental results and observations that support the ideas developed in section 2; showing how implicit context in enzyme GP leads to behaviours that promote evolvability. Section 5 offers conclusions and speculates about the role of enzyme GP and evolutionary computation in understanding biological evolution.

2 The Role of Context in Program Representation

The evolution of an entity is a result of processes of variation acting upon its representation. The way in which an entity evolves depends upon both the way in which variation changes the representation and the extent to which change in the representation leads to change in the entity. This, in turn, depends upon the relationship between representation and entity.

In genetic programming, we are interested in evolving programs. Enzyme GP models computation as a network of interacting functional elements, input nodes and output nodes. This model is general enough to capture most of the programs that are evolved by other GP systems, including the tree-structures evolved by standard GP. The role of each of the nodes within the network can be defined by the nature of its outputs. For a functional element, this is the result of applying its function to the data supplied at its inputs i.e. it is determined by both its function and its inputs. Its inputs, more specifically its input connections from other network nodes, determine the context within which its function is applied. The manner in which this context is expressed,

however, depends upon the program representation.

2.1 Explicit Context

The most common representations used by GP declare explicit connections between components. Consequently, a node's context is also recorded explicitly. Examples of this form of representation include the parse tree of conventional genetic programming and linear GP representations [e.g. Nordin, 1994].

This use of explicit context has a number of implications for these representations when subjected to evolution. First, a node's context is typically determined by its position within the representation. Crossover operators (unless designed to be context-preserving and therefore more computationally intensive) tend not to preserve this positional information and, consequently, tend to disrupt context. The result of this is that crossover is more likely to cause macro-mutation rather than meaningful recombination [Angeline, 1997].

Second, where explicit context is used there is usually a one-to-one mapping between representation and program, such that changes to the representation lead directly to changes within the program. The variation operators therefore act directly upon the program, implying that evolution is determined solely by processes of variation and selection. The representation, by comparison, has no dynamic role to play within the evolutionary process.

2.2 Indirect Context

In some other GP representations, connections between components are specified using indirection. Typically each component is assigned a reference (which may be a location, a number or an arbitrary code) and other components specify their input connections using these references. A good example of this is Cartesian GP [Miller and Thomson, 2000], where components are assigned to locations in a cartesian co-ordinate system and input connections are expressed in terms of the co-ordinates of components they wish to receive inputs from.

An interesting facet of representations that use indirect context is that a component only becomes active in the program if another component expresses a connection to it. Otherwise it is recessive. This has some important implications. If a new component is added to the representation during recombination, it will only become active in the program if its reference is addressed by an existing component or it is an output node. In this paper, this effect is termed

variation filtering, since the representation only expresses certain variation events in the program whilst filtering out others.

The trouble with most representations that use indirect context is that components are assigned references arbitrarily. Usually there is no correlation between component reference and component behaviour within, and more significantly, between programs. Accordingly, components with different behaviours can have the same reference and components with the same behaviour can have different references. This implies that the meaning of a component's input context depends upon which other components are present in its environment. If this environment changes following recombination, this will cause the component's context — and therefore its role within the program — to change. Hence, indirect context does not maintain its meaning following recombination.

However, it is conceivable that the population might be able to evolve a correlation between component reference and component behaviour over the course of time, especially as the population becomes more homogenous. In this sense, the meaning of indirect context is evolvable. Representations that use indirect context also have a number of other interesting evolutionary behaviours. For instance, mutation is able to change the structure of the program by targeting the references that specify connections. Furthermore, mutation or recombination can lead to recessive components becoming re-activated. This may encourage back-tracking behaviours during evolutionary search. Also, recessive components are still exposed to evolution: encouraging useful forms of neutral evolution. These ideas have been explored further in [Lones and Tyrrell, 2001a].

2.3 Implicit Context

Implicit context is an innate property of biological representations and, as we shall see, the program representation used by enzyme GP. The principle behind implicit context is that a component's input context is defined in terms of behavioural properties of the component it would like to receive input from. For example, the behaviour of a bio-chemical is dependent upon its physical shape and chemical properties. Enzymes, which conceptually receive their input in the form of bio-chemicals, express their preference for these inputs by the shape and chemical properties of their binding sites. Since the bio-chemicals that they bind have physical shape and chemical properties complementary to these binding sites, the binding sites are implicitly describing the context of the substrates they expect to bind.

Implicit context within program representations captures many of the rich

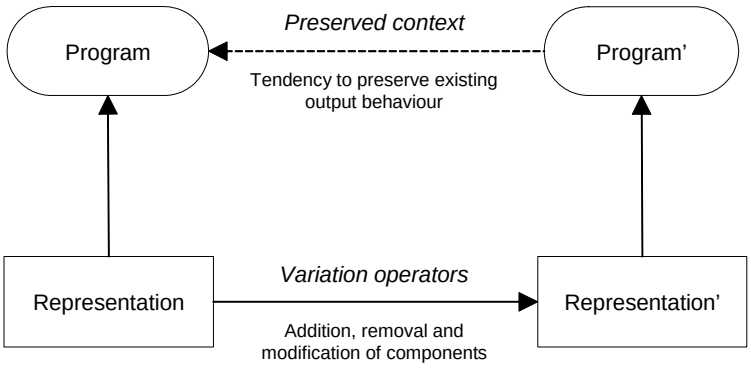

Fig. 1. The combination of implicit context and variation filtering causes a tendency to preserve existing context and program output behaviour.

evolutionary behaviours possible with indirect context, but — because context is defined in a manner which is independent of any particular environment — does not suffer the same problems with loss of meaning following recombination. In particular, representations with implicit context also experience variation filtering. However, unlike with indirect context representations where loss of meaning makes this process arbitrary, variational filtering in representations with implicit context may be a significant source of evolvability: filtering out inappropriate variation whilst preserving meaningful variation. If a new component is added to the representation (during recombination), it will only become active in the program if it either fulfills an existing component's preferred context better than any existing component or it is an output node. If it does not fit into the context of the program, it will become recessive. If an active component is removed from a program (during recombination), then the components which received input from it will receive input from components that have functional properties most similar to the one that was removed — which may be components that are currently recessive. If an existing active component is modified (by mutation), it may become recessive if the context it provides no longer fits with the rest of the program. If a recessive component is modified, it may become active if the context it provides now fits better into the program than the context provided by an active component.

In effect, the combination of implicit context and variation filtering could result in a tendency to preserve existing contexts within the program following the application of variation operators. In turn, this will tend to preserve the existing behaviour at the program outputs (which may come from more than one program following recombination). This idea is illustrated in figure 1.

However, there are certain pre-conditions for implicit context being useful in enabling meaningful recombination and variation filtering. These were originally identified in Lones and Tyrrell [2002a] and concern the relationship between implied context (the context declared in the representation) and actual context (the context which occurs in the program). Precision is the generality

of the implied context: the degree to which it suggests an actual context. If implicit context is too general, then the behaviour of the program will not be obvious from the representation, making the mapping between representation and program unstable and therefore easily disrupted by the application of variation operators. In turn, this will make meaningful recombination unlikely. Related to precision is the issue of specificity: how well an implied context is able to identify an actual context. If a component's implied context is too unspecific, it is likely to form different actual contexts within different programs, meaning that variation filtering will tend to carry out arbitrary behaviours. Finally, there is the issue of accuracy: the ability of implicit context to match actual context given the availability of components within the representation and any constraints upon connections between components that are placed upon the behaviour of the program. Given that these limitations are unavoidable, there is little that implicit context can do to overcome them. Nevertheless, it is important that when the preferred context is not available, the implied context should match the nearest available actual context. In turn, this requires that a distance metric can be defined between implied and actual contexts, such that the greater the distance between contexts, the greater is the difference in component behaviour.

3 Implicit Context in Enzyme Genetic Programming

In enzyme GP, program components are modelled upon biological enzymes. In addition to having an activity, each component has a shape which declares its expected role within the program and analogues of binding sites whose shapes declare the component's implicit context — the shapes of its expected inputs (or substrates) within the program. However, unlike biological enzymes, individual program components carry out only a limited range of functional activities and typically more than one component within a program would carry out the same function. Consequently, for implicit context to confer sufficient precision and specificity, it would be insufficient for shape to capture only the component's activity. Accordingly, shape captures the entire form of the component: both its activity and the shapes of its binding sites. Likewise, binding sites describe the complete shape of the components they expect to bind.

This has an interesting implication. Since a component's shape captures the shapes of its binding sites, and these shapes capture the entire form of its expected substrates, it is also capturing the activity and binding site shapes of these expected substrates: and recursively, the shapes — and therefore the activities — of all the components in the program fragments that develop below its inputs. This is not a behaviour that occurs in biological enzyme systems, since enzymes interact through chemical intermediaries and even then

do not capture the entire shape of their substrates. Nevertheless, it would appear to be a useful mechanism in increasing the precision and specificity of implicit context within enzyme GP.

The shape of a program component is defined using a measure called functionality: a vector which describes the component's position within an activity reference space. This reference space has one dimension of unit length for each member of the GP function and terminal sets. The functionality, F, of a program component is defined as follows:

$$F(component) = (1 - k) \cdot F(activity) + k \cdot F(binding\_sites) \tag{1}$$

where k is a constant that biases the functionality towards either the component's activity or the component's binding sites; $F(activity)$, the functionality of the component's activity, is a unit vector situated in the dimension corresponding to the enzyme's function; and $F(binding\_sites)$, the functionality declared by the component's binding sites, is defined:

$$F(binding\_sites) = \frac{\sum_{i=1}^{n} F(site_i) \cdot strength(site_i)}{\sum_{i=1}^{n} strength(site_i)} \tag{2}$$

i.e. the average of the functionalities corresponding to its binding sites weighted by the strength (a number between 0 and 1) of each binding site.

In effect, a component's functionality declares an expected activity profile of the components that occur in the program fragment of which it is the root, weighted by depth and biased by the strength of binding sites. Functionality space itself is continuous and the distance between functionalities is calculated using vector subtraction. This reflects the difference between their activity profiles, meeting the requirement (outlined in section 2.3) that the greater the distance between contexts, the greater is the difference in component behaviour.

3.1 Mapping Implicit Context to Actual Context

A program representation is the enzyme GP analogy of a biological genome: stored as a linear array of program components sub-divided into input terminals, functional elements and output terminals. In addition to an activity declaration (a member of the function or terminal set), each component that processes inputs, i.e. functional elements and output terminals, contains a list of potential binding sites, each of which has a functionality and a strength.

The objective of mapping program representation to program is that each

active input of each active component will be connected to the output of the component for whose shape its corresponding binding site has the highest specificity i.e. the shortest distance between functionalities. A number of different mapping processes are possible. The mapping process used for the results in this paper begins with expression of the output terminals, which then choose substrates whose shapes are most similar to the shapes of their strongest binding sites. These substrates are now considered expressed and, if they require inputs, attempt to satisfy them by binding their own substrates. This process continues in hierarchical fashion until all expressed program components have satisfied all of their inputs. Each output terminal is expressed exactly once, whilst input terminals and functional elements can be expressed once or not at all. Where more than one component chooses the same substrate, the output of the substrate is shared between them.

The programs evolved for this paper are required to have strict tree-structures. This constraint is handled within the mapping process by checking for the presence of cycles before a new connection is made. If a connection to a substrate would result in a cycle, then an alternative (less preferred) connection must be made. Unfortunately this cycle checking brings the time complexity of the mapping process above linear, but is an unavoidable overhead of developing a cyclic graph into a tree structure. An alternative mapping process, based upon transforming the program representation into a network random key representation [Rothlauf et al., 2002], has also been used within enzyme GP, but has a higher average-case complexity than the process outlined above.

3.2 Evolving Program Representations

Evolution of program representations occurs within a spatially-distributed parallel genetic algorithm, details of which can be found in Lones and Tyrrell [2001b]. Bounded-size program representations are constructed randomly to fill the initial population. New program representations are created through crossover and mutation of existing program representations.

Crossover in enzyme GP is somewhat different to sub-tree crossover in conventional GP and takes advantage of the fact that added components need not replace existing components. In sub-tree crossover, one sub-tree is always replaced by another. In enzyme GP crossover, a contiguous group of components is copied from one solution to another without removing any existing components (with the exception of output terminals, whose numbers remain constant). This transfer operation is complemented by a converse operation which removes contiguous groups of components from solutions. Each of these operations is used, non-deterministically, for 50% of crossover events. The effect of the remove operation is to balance solution sizes so that recombination

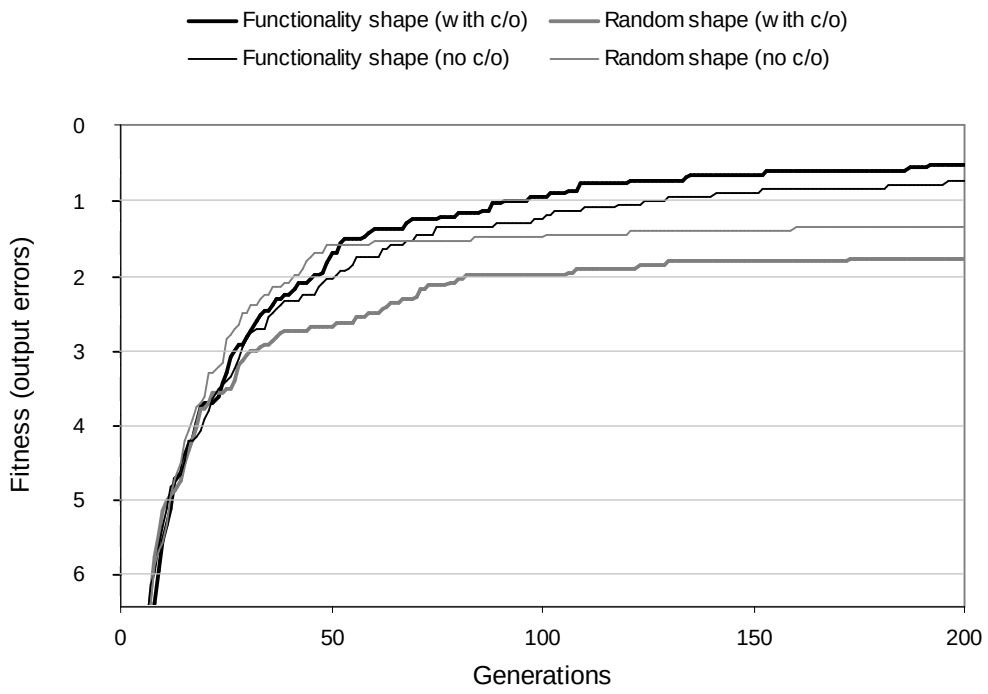

Fig. 2. Comparing mean number of output bit errors when evolving two-bit multipliers. (Averages taken over 50 runs. Parameters are given in table 1).

has an overall neutral effect upon solution size within a population. For both transfer and remove operations, the number of components targeted is chosen randomly within an upper limit (t_u). The benefits of this form of crossover over uniform and tree-based crossovers is discussed in Lones and Tyrrell [2002b]. Mutation targets both functionalities (m_f) and activities (m_a).

4 Properties of the Representation

4.1 Meaningful context

Earlier papers have presented comparative analysis of the performance of enzyme GP upon a range of problems in combinational logic design [Lones and Tyrrell, 2002a]. These have shown that the approach is able to compete favourably against indirect context representations upon most problems that it has been applied to.

A more direct comparison is shown in figure 2, where fitness curves for enzyme GP with functionality shapes are shown against fitness curves for enzyme GP with randomly generated shapes. These random shapes effectively specify an indirect context, where there is no relationship between pattern and behaviour. It can be seen that the fitness curves for enzyme GP with random

Table 1
Success rate and average solution time (in generations) for evolving two-bit multipliers in a population of size 324 with $k = 0.3$, $m_f = 2\%$, $m_a = 1.5\%$ and $t_u = 5$.

Shape	Variation operators	Success rate	Average
Functionality	Mutation and recombination	70%	98
Functionality	Mutation only	55%	79
Random	Mutation and recombination	26%	69
Random	Mutation only	30%	64

shapes initially grow at a high rate, but fall to a relatively low rate once they reach a certain fitness level. This suggests a high level of disruptive variation, which initially benefits search but later becomes a hindrance to effective exploitation. The curves for enzyme GP with functionality shapes, by comparison, demonstrate a steady rate of decay in fitness growth, indicating a more structured search which continues until the optimum is found. Performance statistics for enzyme GP with both types of shape are listed in table 1; and indicate that performance is considerably better with functionality shape than with random shape. This supports the notion that implicit context captures more meaningful context than indirect context.

4.2 Context preservation in recombination

Table 1 also compares the performance of enzyme GP both with and without recombination. When functionality shapes are used, recombination leads to a significant improvement in performance. This occurs even though the problem's fitness landscape is not particularly suited to recombination [Vassilev et al., 1999], and illustrates that functionality shapes are able to preserve context both within and between program representations. When random shapes are used, however, recombination impairs performance: suggesting that recombination disrupts rather than preserves context.

Figures 3 and 4 show examples of the behaviours that occur when transfer and remove operations are applied during recombination. The transfer operation cannot directly replace any components within a program representation (other than outputs). However, as figure 3 illustrates, it can still lead to behaviours at the program level that look like replacement. This happens because the new component has subsumed the role of the former component, offering a closer match to the input context declared by the parent node. Nevertheless, the former component is still present within the program representation and, if the new component was removed, could resume its former role in the program. Figure 3 also shows how subsumption is able to preserve the behaviour of the program below the component that has been replaced, indicating that the new

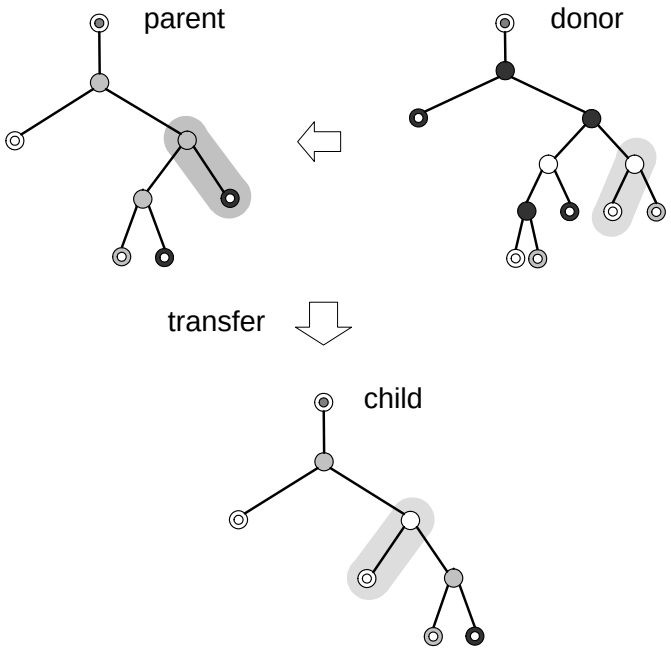

Fig. 3. A simple example of subsumption resulting from a transfer operation. Different node and terminal shades represent different members of the function and terminal sets.

component has a functionality similar to that of the former component, and illustrating the context preserving nature of recombination in enzyme GP.

Figure 4 shows two other behaviours which often occur during recombination: insertion and deletion. In the example on the left of the figure, a transfer operation leads to a new sub-tree being inserted between two existing components, something which is not possible using sub-tree swapping recombination in standard GP. Again, the existing sub-tree below the root of the inserted sub-tree is preserved, and from the perspective of the component (the output terminal) above the inserted sub-tree, the new context is related to the former context: and presumably a closer fit to the implicit context declared by its binding site. In fact, insertion is a special case of subsumption where the subsuming component happens to declare an input context which matches the role of the component that it subsumed. In the other two example in figure 4, removal operations lead to parts of programs being deleted. In the example in the centre, two sub-trees are affected by a single removal operation. This shows how recombination operators in enzyme GP operate upon groups of components at the representation level rather than structures (in this case sub-trees) at the program level: a behaviour which is presumably more disruptive [1], but also more expressive, than conventional sub-tree crossover. In

[1] Although, given that recombination targets contiguous groups of components in the representation, the development of genetic linkage over time might naturally reduce the level of disruption in the later stages of search.

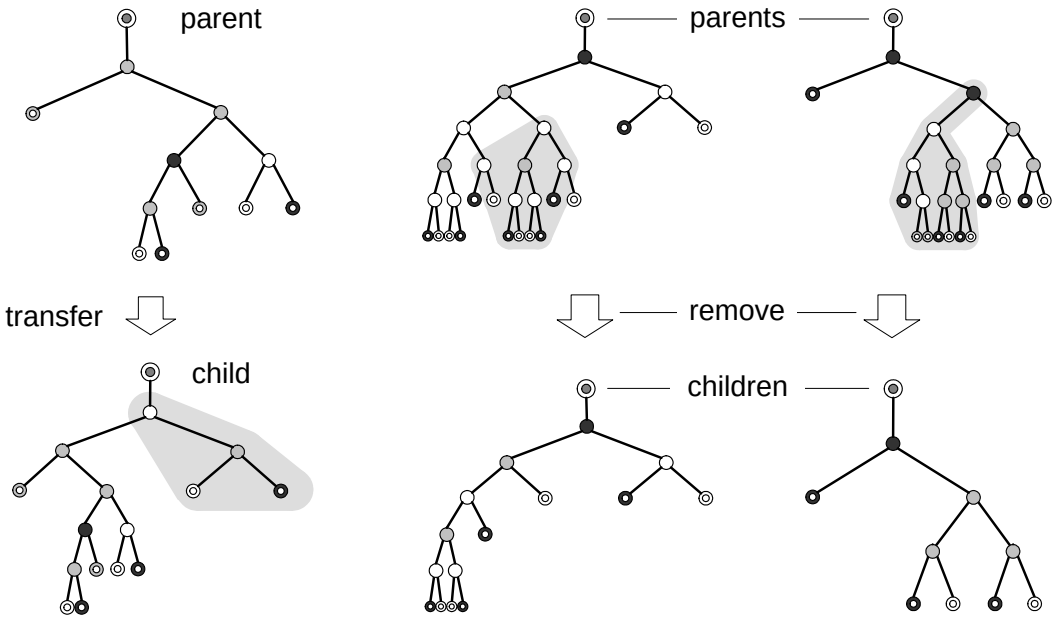

Fig. 4. Behaviours resulting from transfer and remove operations.

the example on the right of the figure, an entire sub-tree is removed from between two nodes. This is the complement of the behaviour which occurs in the transfer example on the left of the figure, and shows how a subsumption operation might become undone. In both of these removal examples, the program below the deletion remains unaffected, showing how the remove operation also encourages context preservation.

4.3 Variation filtering

Conceptually, all the components defined in a representation occupy some position in a subsumption hierarchy. At the top are the expressed components: those which appear in the program. Below these are any redundant copies of the expressed components. Below these are components that have been subsumed by the expressed components; and further down, components that were subsumed by components further up the hierarchy which have since themselves also been subsumed. At the bottom are components which have never been expressed but which could in principle be expressed if all the components above them in the subsumption hierarchy were removed. All the behaviours that occur in enzyme GP are a result of variation operators modifying this subsumption hierarchy: either by adding new entries, by removing entries or by re-ordering existing entries (which is what mutation operators are essentially doing). Nevertheless, the operators are not aware of this subsumption hierarchy. They blindly add, remove and re-order entries — only causing change in the program when they happen to add, remove or re-order entries at the top of the hierarchy. All other changes are absorbed into the lower echelons of the

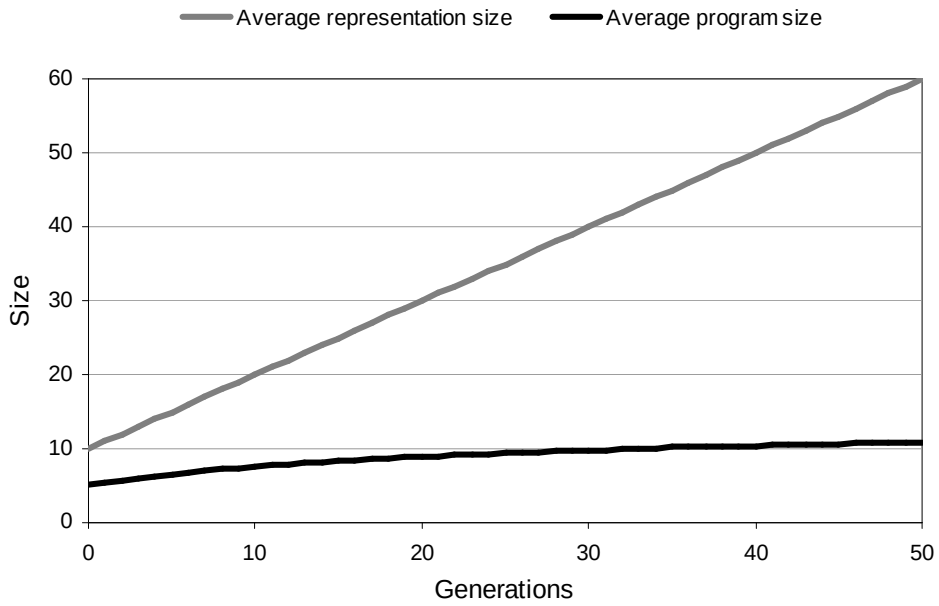

Fig. 5. Comparing representation size growth against forced program size growth in a population of size 100 with no selection pressure, no removal operation and no mutation.

hierarchy.

Subsumption supports the idea that the program representation used by enzyme GP promotes meaningful variation filtering: since it is clearly the way in which the representation describes the program, rather than the action of the variation operators, that allows subsumptive behaviours to occur. Further evidence for the existence of variation filtering can be seen in the relationship between patterns of growth in the program and in the representation. Earlier papers [Lones and Tyrrell, 2002a,b] have described this relationship in the case where the population is undergoing fitness-based selection. These have shown that program growth is considerably lower than representation growth. Figure 5, by comparison, compares program and representation size growth when the population is not undergoing selection and only the insertion operator is used. Consequently, any problem-specific program size influence is removed. Again, program size growth is considerably lower than representation size growth, implying that the proportion of un-expressed components in a program representation, on average, increases over time. This effect can be accounted for by a process of variation filtering, where the un-expressed portion of the representation contains those components which have been inserted into the representation but not propagated to the program.

This un-expressed portion of the program representation constitutes the recessive part of the subsumption hierarchy described above. Although it would not normally grow at the rate seen in this example, it is interesting to consider whether it could have a function other than filtering out components that

do not fit into the context of the program. One such function is evolutionary back-tracking. If the population were to evolve towards a local optimum, these recessive subsumption hierarchies contain information that could be used as a means of escape to some previous point before the population converged upon the local optimum. Other functions concern behavioural plasticity: the possibility of recessive components becoming expressed during operation to compensate for failure of expressed components or changes in the program's requirements (an idea related to work reported in Tyrrell et al. [2001]).

5 Conclusions

Most genetic programming systems represent the programs they are evolving using either explicit or indirect context. This paper introduces an alternative, biologically motivated, approach to program representation that uses implicit context; arguing that this form of representation is more able to confer evolvability than those which use explicit or indirect context. This paper also introduces the notion of variation filtering: the tendency of a representation to promote certain types of change whilst filtering out others. In particular, it is suggested that implicit context representations can carry out a form of variation filtering that promotes evolvability by filtering out inappropriate change whilst maintaining meaningful change — and that this process is instrumental in enabling meaningful recombination.

In biological representations, shape is an innate form of implicit context. The program representation used by enzyme genetic programming is modelled upon the representations used by biology to represent enzyme systems, and models shape as a pattern which describes the role of each component defined within the representation. This paper presents results which demonstrate the ability of this pattern to capture meaningful context and therefore enable meaningful recombination. It also introduces some of the behaviours that occur during recombination and discusses how these are the result of the representation's variation filtering behaviours.

Whilst enzyme GP has so far been used as a tool for understanding computational evolution, it is interesting to consider whether it might also have a role in understanding biological evolution; particularly the early stages of biological evolution. It seems evident that variation filtering behaviours do occur in biological systems, although simple behaviours of the kind described in this paper may only have had an influence before active forms of genetic and enzymatic regulation evolved. Likewise, it seems plausible that the non-coding portions of DNA, rather than being junk, have a role similar to the recessive subsumption hierarchy described in this paper — providing a source of error recovery and evolutionary back-tracking. Therefore, we would like to

end this paper by posing the question: could evolutionary computation have a role within the understanding of biological evolution similar to the role played by neural computing in the understanding of brain function: to validate biological theories, to test the generality of biological constructs and to identify holes in our understanding of how biology functions.

References

Lee Altenberg. The evolution of evolvability in genetic programming. In K. Kinnear, Jr, editor, *Advances in Genetic Programming*. MIT Press, 1994.

P. Angeline. Subtree crossover: Building block engine or macromutation? In John R Koza, Kalyanmoy Deb, Marco Dorigo, David B Fogel, Max Garzon, Hitoshi Iba, and Rick L Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference, GP97*, pages 240–248. Morgan Kaufmann, 1997.

M. Conrad. The geometry of evolution. *BioSystems*, 24:61–81, 1990.

M. Kirschner and J. Gerhart. Evolvability. *Proceedings of the National Academy of Science (USA)*, 95:8420–8427, July 1998.

John Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, 1992.

M. A. Lones and A. M. Tyrrell. Biomimetic representation in genetic programming. In H. Kargupta, editor, *Proceedings of the 2001 Genetic and Evolutionary Computation Conference, Computation in Gene Expression Workshop*, pages 199–204, July 2001a.

M. A. Lones and A. M. Tyrrell. Enzyme genetic programming. In J.-H. Kim, B.-T. Zhang, G. Fogel, and I. Kuscu, editors, *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 2, pages 1183–1190. IEEE Press, May 2001b.

M. A. Lones and A. M. Tyrrell. Biomimetic representation in genetic programming enzyme. *Genetic Programming and Evolvable Machines*, 3(2): 193–217, June 2002a.

M. A. Lones and A. M. Tyrrell. Crossover and bloat in the functionality model of enzyme genetic programming. In *Proceedings of the 2002 World Congress on Computational Intelligence*. IEEE Press, 2002b.

M. A. Lones and A. M. Tyrrell. Enzyme genetic programming. In M. Amos, editor, *Cellular Computing*, Genomics and Bioinformatics Series. Oxford University Press, 2003. (To appear).

J. Miller and P. Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Third European Conference on Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*. Springer, 2000.

P. Nordin. A compiling genetic programming system that directly manipulates

the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

F. Rothlauf, D. Goldberg, and A. Heinzl. Random network keys — a tree network representation scheme for genetic and evolutionary algorithms. *Evolutionary Computation*, 10(1):75–97, 2002.

A. M. Tyrrell, G. S. Hollingworth, and S. L. Smith. Evolutionary strategies and intrinsic fault tolerance. In D. Keymeulen, A. Stoica, J. Lohn, and R. S. Zebulum, editors, *Proceedings of the Third NASA/DoD Workshop on Evolvable Hardware*. IEEE Computer Society, July 2001.

V. Vassilev, J. Miller, and T. Fogarty. On the nature of two-bit multiplier landscapes. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 36–45. IEEE Computer Society, July 1999.

G. P. Wagner and L. Altenberg. Complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976, 1996.