Experiments in Genetic Divergence for Emergent Systems

Christopher McGowan
Lancaster University
christopherdavidmcgowan@gmail.com

Alexander Wild
Lancaster University
a.wild3@lancaster.ac.uk

Barry Porter
Lancaster University
b.f.porter@lancaster.ac.uk

ABSTRACT

Emergent software systems take a step towards tackling the ever-increasing complexity of modern software, by having systems self-assemble from a library of building blocks, and then continually re-assemble themselves from alternative building blocks to learn which compositions of behaviour work best in each deployment environment. One of the key challenges in emergent systems is populating the library of building blocks, and particularly a set of *alternative implementations* of particular building blocks, which form the runtime search space of optimal behaviour. We present initial work in using a fusion of genetic improvement and genetic synthesis to automatically populate a divergent set of implementations of the same functionality, allowing emergent systems to explore new behavioural alternatives without human input. Our early results indicate this approach is able to successfully yield useful divergent implementations of building blocks which are more suited than any existing alternative for particular operating conditions.

ACM Reference Format:
Christopher McGowan, Alexander Wild, and Barry Porter. 2018. Experiments in Genetic Divergence for Emergent Systems. In *IEEE/ACM 4th International Genetic Improvement Workshop, June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194810.3194813

1 INTRODUCTION

Emergent software systems use a paradigm of continuous runtime self-assembly to automate the construction and runtime optimisation of complex software systems. Starting from a goal, these systems use a large library of micro building blocks to dynamically discover all possible behaviours that can be composed together to meet this goal; this will include alternative implementations for many such building blocks which carry out the same functionality but do so in a different way, such as alternative search or sort algorithms. Online learning is then used to experiment on the live, running system, by seamlessly adapting between different possible compositions of behaviour to learn which ones best satisfy a reward function of interest (such as performance) against periodic observations of the system's current deployment environment [1, 2].

Emergent systems partially solve the challenge of manual design and optimisation of complex software systems by automatically discovering potential building blocks for those systems and continuously learning which particular combination of alternative building blocks offers the best performance in each environment range that is encountered at runtime. At present, however, these building blocks must be manually developed – and in particular, variations of building blocks (such as alternative search or sort algorithms) must be manually written by human programmers in the hope that they will be useful in certain deployment environment conditions. Because the range of environments is virtually infinite, and the correlation of a particular set of deployment conditions to its ideal implementation of sub-behaviours for a given system is difficult to predict, the development of these building blocks represents one of the largest open challenges in emergent software systems.

We present an approach to using genetic improvement to help automatically populate a design space of software building blocks; emergent systems can then learn the runtime characteristics of these new behaviours by seamlessly integrating them into the live system and observing their characteristics, thus discovering increasingly optimal compositions of behaviour for each set of deployment conditions encountered by the running system. Because emergent systems are inherently modularised at a fine granularity, we apply genetic improvement at the level of an individual building block (typically 100 lines of code). As each individual building block does not have a large amount of code from which to source genetic modifications, we use a blend of improvement and synthesis to explore genetic material from both existing and newly synthesised logic.

In detail, our specific contributions are:

- To identify emergent software systems and genetic improvement as natural partners and a rich area for future research: emergent systems are inherently composed of fine-grained modules which limit the search space for genetic improvement, and emergent systems require a large pool of alternative implementations of behaviours which are likely to perform differently in different operating environments.
- To demonstrate an approach that blends genetic improvement with synthesis of new behaviour to gain genetic divergence. We use mutations that can create highly generalised new logic, together with a two-phase algorithm which cycles between a traditional *improvement* phase and a distinct *synthesis* phase that prefers to expand a component's source code. All of our genetic operators work at the source code level and guarantee not to produce compiler errors.
- To show promising initial results in an emergent software system, using a web server as an example, in which the hash map sub-behaviour of the web server's caching module is mutated to alternative implementations that better suit specific deployment environment conditions.

The source code of our genetic improvement framework is made available along with instructions on how to replicate our results at [3]. In the remainder of this paper we first survey related work in Sec. 2, then describe our approach in detail in Sec. 3. We present our initial evaluation results in Sec. 4 and conclude in Sec. 5.

GI'18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5753-1/18/06…$15.00
https://doi.org/10.1145/3194810.3194813

2 RELATED WORK

The genetic approach to code modification is one of the most popular approaches to automated software improvement [4] with recent progress in automated bug-fixing [5–7], optimising non-functional properties [8–10], and automatic test-case generation [6].

Our work falls into the category of genetic improvement, as we take an input component and attempt to produce a functionally equivalent one which has improved non-functional properties for a particular set of operating conditions. Because our approach fuses traditional improvement with new code synthesis, we survey recent work in both genetic improvement and program synthesis.

Genetic Improvement. Genetic improvement has been applied to source code [9, 11], abstract syntax trees [5, 12, 13], intermediate compiled code forms such as Java bytecode [14], and compiled machine code [15]. It is unclear whether any particular level of operation across this spectrum has quantitative benefits (in terms of the resulting program) over any other level, though in qualitative terms the modification of source code may have benefits in human legibility (and verification) of genetic modifications. We choose to operate on the abstract syntax tree of our input components, resulting in modifications that are more likely to be legible to human programmers if desired.

One of the most compelling results in genetic improvement is that of Forrest *et al* [5], which demonstrates effective automated repair of bugs. The approach uses a negative test case which activates the bug, together with positive test cases which verify correct behaviour. During the improvement process, mutation operations are preferentially applied to the execution path that is activated by the negative test case, which reduces the search space to a size tractable for genetic improvement to navigate in reasonable time. The specific genetic operators used are delete, swap, and insert; delete and swap only take place within the negative execution path, while insert may take a statement from anywhere in the program and insert it into the negative execution path. Our work differs in that: we do not use a preferential execution path to bias our search; we restrict our improvement to a single component rather than the entire program; we seek to derive behaviours that demonstrate a useful un-filled point in the divergent optimality space rather than fix bugs; and we blend improvement operations with a set of genetic operators that can synthesise brand new logic.

Building on the technique used by Forrest *et al*, Langdon and Harman use genetic improvement to enhance performance of a large C program [9], using the same set of genetic operators (delete, swap and insert), where insertions are sourced from existing code in the program being modified. The code to which genetic operators are applied is similarly weighted to code activated by test cases, though a formal grammar is additionally used to aid in generating only syntactically valid code. The resulting program is demonstrated to be significantly faster than the original. Similarly, Haraldsson *et al* [16] uses genetic improvement for performance enhancement of the ProbAbel bioinformatics program written in C and C++, using delete, replace, swap and copy as its genetic operators, where the swap operation can either swap entire lines or operators from common groups known to be type compatible (like − and +). In both of these approaches, when source material is needed for additive operations such as copy, this material is randomly (and exclusively) sourced from the existing code of the program being improved. These approaches are closer to our goal of deriving higher-performance component variants for a given environment, but our work differs in that we restrict our operations to a single component rather than a selected execution path through the whole program, and we blend improvement operations with the synthesis of brand new logic.

Finally, we note that our aim to produce multiple functionally equivalent variants of each component is similar to the concept of N-version development [17], traditionally used for fault tolerance. It has recently been suggested by Petke that genetic improvement may be a good approach to generate a search space of runtime variants [18], but we are not aware of any other work besides ours that has attempted this to date.

Code Synthesis. Because we apply genetic improvement to small components that have relatively little genetic material, we combine traditional genetic improvement operators (which take genetic material exclusively from existing code), with operators that synthesise brand new logic, aided by a set of heuristics. In this section we review recent research in pure code synthesis.

Code synthesis traditionally attempts to start from an empty or near-empty program file, either to produce a program which matches a desired input-output mapping [19], or which approximates a function (such as a regressor or classifier) [20].

Gulwani *et al* propose an approach to generate a computer program that transforms a set of input examples to a set of corresponding output examples [19]. This is done by first defining a domain-specific language for a certain class of program (like string manipulation) which helps to restrict the search space for synthesising a program. For each input-output pair, all possible programs are synthesised in the language, and then an intersection operation is applied across the set of programs for all input-output pairs to isolate a program that works for all examples. While the approach of generating all variants from which to select is appealing, doing this to generate even small components in a fully generalised (rather than specialised) programming language would be prohibitively expensive. We instead perform synthesis operations heuristically and incrementally, deriving the set of possible single synthesis actions from an existing piece of code and applying one at random.

Swan and Burles [21] propose a mixed synthesis approach with genetic improvement, using a concept of template functions by which the programmer writes a generic version of a function, such as a sorting algorithm, and defines variation points within that function to be automatically generated. This is close in spirit to the aims of emergent systems in generating a variant of that algorithm which is efficient to a particular input data class. Our approach operates at the level of a multi-function component, and does not use a special template description language that the programmer must learn. The actual mutations applied are also relatively simple, adjusting the pivot point of a quicksort algorithm, where our approach is capable of synthesising complex generalised source code, from variable declarations to nested for-loops.

As far as we are aware, our work represents a new point in the design spectrum of improvement and synthesis, able to incrementally synthesise new, fully-generalised, logic and blend it with existing genetic source material in a component to improve.

3 GENETIC DIVERGENCE FRAMEWORK

3.1 Background

We use the emergent systems framework reported in [1] as the domain to which we apply genetic improvement, and we specifically use the emergent web server example from the same paper as the target application. The source code of both is publicly available at [22]. Here we briefly summarise the way in which the emergent web server works before describing our genetic framework.

The web server is assembled from a collection of building blocks written in the Dana programming language [23], a highly adaptive component-based language which can hot-swap ('adapt') behaviour in a running system. These adaptations are safe to the state machine of the executing program and occur very quickly (a few microseconds) which makes runtime behavioural exploration very cheap. Emergent systems are uniformly built from such adaptable components, so that any element of a running program can be adapted, and it is assumed that all candidate compositions of components made available to a running emergent system are correct, such that adapting to them will not normally create erroneous conditions.

The components that form an emergent system are interconnected via interfaces, such that each component will provide one or more interfaces and may require one or more interfaces. Each required interface of a component must be connected to a type-compatible provided interface of another component to satisfy the dependency. These inter-interface connections can later be adapted at runtime to change the implementation of the system, by loading a different implementation of a provided interface into memory, adapting a required interface to connect to that implementation, and unloading the previously-used one.

The emergent systems concept is able to optimise software at runtime because different (functionally equivalent) implementations of a provided interface have different performance characteristics under different operating environment conditions. During execution, an online learning algorithm discovers which components can be used to compose a particular system and then learns which composition works best under each detected operating environment range. This learning is performed on the live system, using the cheap adaptations mentioned above, so that learned information is relevant to the actual conditions experienced by the system in its real production environment and the actual effects that those conditions are observed to have on its behavioural alternatives.

As an example, the emergent web server that we target here exhibits divergent optimality around its caching and compression components. In its resource-fetching subsystem, the web server has different building blocks that can either serve content directly from the disk, can cache a limited amount of content in memory and serve from this cache, and/or can compress content before returning it to a client. In operating conditions where a large number of requests are for the same content, compositions that include the caching behaviour tend to be faster because reading from memory is faster than reading from the disk. Conversely, in operating conditions that have very few requests in common, compositions without caching tend to be faster (as checking the cache takes time). Within these behavioural variations there are further sub-variations, such as alternative cache algorithm implementations, which themselves perform differently under different kinds of input patterns.

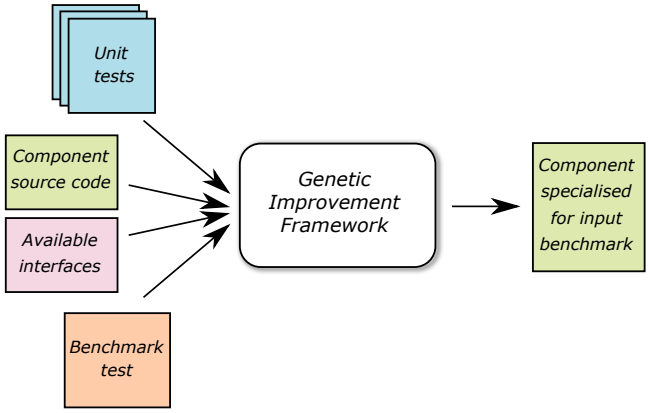

Figure 1: Our genetic improvement framework.

3.2 Genetic improvement procedure

Our overall genetic improvement procedure is illustrated in Fig. 1. As input to our framework, we assume that we are given four kinds of information: (i) the source code of a component implementing one particular interface; (ii) a set of test cases for this interface which confirm that inputs to functions of the interface produce the correct outputs; (iii) a specific *benchmark* of inputs, for which the framework will attempt to derive a higher-performing implementation of the given component; and (iv) a set of other interfaces that can be used by the generated code. This set of other interfaces is typically the required interfaces that are already used by the input component, plus any other standard library interfaces believed to be of potential use towards the improved component. The benchmark test that is input to the framework is assumed to have been derived from the running system in its actual operating environment, collected by triggering a phase of capturing inputs via dynamically-inserted probes, for offline replay and analysis.

Our genetic improvement framework operates by cycling between two main phases: *improvement* and *synthesis*. In the improvement stage, we reward shorter code and faster performance in completing the benchmark. In the synthesis stage we reward longer code and place less emphasis on performance. The framework always starts in an improvement phase, using only the submitted component as its existing genetic source material, and runs in this phase until it appears to have reached a stable reward value. The framework then moves to a synthesis phase for a fixed number of generations, before returning to an improvement phase. If, once this successive improvement phase has reached a stable reward value, the framework has ended up with a worse reward value than that achieved prior to the synthesis phase, we revert to the best entire population from before that synthesis phase. The framework then cycles between these improvement and synthesis stages for a fixed amount of wall-clock time (provided as a parameter).

The intention of the synthesis stage is to augment the relatively small amount of genetic source material (because components in emergent systems tend to be quite small) with generated new material. This may allow new implementation permutations to be reached during mutations that would otherwise not have been reachable using only the code present in the source material.

3.3 Genetic algorithm

Our genetic algorithm uses a population size of 14, for which all members of the first population are a copy of the input source code. We use elitism whereby the best two members of a population are preserved, such that they are not themselves affected by mutation or crossover. Within a population we use a mutation probability of 80% and a crossover probability of 20%[1].

Our algorithm then proceeds in generations by first running our benchmark test (and test cases) on all members of the current generation, then applying our fitness function which takes into account the number of test cases passed, the performance against the benchmark measured in a component-specific way, and the total code length of the component, preferring shorter code in improvement phases and longer code in synthesis phases. These three metrics are arithmetically combined into a single overall fitness score for the genetic algorithm. Following testing, the population members are then rank-ordered by fitness and mutation and crossover take place among the (non-elite) population members according to their respective probabilities, using a roulette wheel strategy to select population members [24]. The new population members are compiled and we then return to the testing and ranking stage for the next generation using this new population.

3.4 Genetic operators

As discussed in Sec. 2, the majority of genetic improvement research to date uses delete, insert and swap operators, using only the genetic material of the program being improved as needed for insert and swap. Because our framework works at the level of individual, fine-grained software components, we take a different approach which is significantly biased towards genetic operators that synthesise new genetic material. This synthesis is done randomly, but using heuristics that reduce the possible search space, and in such a way that synthesised code is likely to integrate with existing source code by (for example) re-using existing declared variables in newly synthesised behaviour. In addition, all of our genetic mutation operators are designed to result in compilable code, for example avoiding syntactic, type compatibility and scope errors.

We use the mutation operators *insert*, *modify* and *delete*. When a mutation occurs we choose uniformly at random from among these operators, then randomly choose a specific sub-operator and apply this sub-operator at a randomly-selected location in the source code (if possible, according to the language grammar). We now describe each operation in detail.

Insert Operations

All insert operations work at the level of lines-of-code. The sub-operators are: declare variable (local of global); assign a variable; insert control statement; insert return statement.

When declaring a variable, our framework is currently able only to declare a variable of primitive type (int, bool, char, dec) or an array of one of these types. When variable declaration is chosen, at the randomly selected point in the source code, we randomly select a type for the variable and give the variable a name that does not collide with any other variable already declared in this scope.

The remainder of our insert operations, in addition to the statement being inserted, may require an *input* which is used alongside that statement, such as the expression to which a variable is assigned or from which a parameter is passed. For this input, we always make a random choice from the set F_i, which includes function calls (selected at random from among the available interfaces passed into our framework); in-scope type-compatible variables; constants declared in the source file; or a literal value (i.e., true/false for boolean types; or a randomly chosen integer for integer types). Note that if the chosen member of F_i is a function call with parameters, we randomly select inputs to those parameters from F_i.

When inserting a variable assignment, an in-scope local or global variable is randomly selected and then a type-compatible member of F_i is chosen as the expression to which the variable is assigned. If a variable being accessed or assigned is an array, and the assignment is occurring within the scope of a for-loop, we always use the loop variable as the index for assignment/access. Otherwise, if not in a for-loop, we assume that the array itself is being assigned/accessed.

When inserting a control statement, we can insert a for-loop, while-loop, break statement, or if-statement; alternatively we can insert an else-branch of an existing if-statement. The insertion of a for-loop follows the additional rules that it must use an integer as its loop variable, its loop condition must return a boolean value, and its 'increment' operation must involve the loop variable in some way (but not necessarily as an increment). Once a control statement has been selected and any parameters chosen, along with its insert location, we then randomly choose how much code below the insert location should become part of the control statement's scope (if any). Control statements can be arbitrarily nested inside other control statements, to any nesting depth.

When inserting a new return statement, we again elect to return something randomly chosen from F_i.

Our general inclusion of *any in-scope variable* in F_i helps to allow newly-synthesised code to integrate with existing code, and our restrictions on for-loops, and the way in which variable accesses / assignments are used within them, encourage this kind of loop to be used for array iteration. Note that the framework is still able to synthesise arbitrary kinds of loop via the use of while loops.

Modify Operations

For modify, our sub-operators are assignment; function call; function parameter; or return statement. If an assignment is being modified, the right-hand-side of the assignment is changed to a randomly chosen member of F_i. For function calls, we simply swap a function call to a type-compatible alternative function call and randomly populate any parameters of that alternative call from F_i. The modification of a function parameter similarly chooses from among F_i, as does the modification of a return statement's operand. In addition to local/global variable assignments, our modify operators can also alter declared constants in the component's scope.

Delete Operations

For delete, we have sub-operators for declaration; assignment; control statement; and return statement. When deleting a declaration, we first check that the declared variable is not used. The

[1]The particular parameters used in our genetic algorithm were chosen because they were empirically observed to work well among several alternatives. However, we have not yet conducted a methodical evaluation of the available parameter space and the effects of different parameters on performance.

deletion of assignments has no restrictions. When deleting a control statement, if the statement under consideration is a for-loop we simply delete the entire for-loop due to the frequency with which its loop body will use the loop variables. In the case of a while-loop or if-statement, we make a random choice between deleting the entire control statement, or deleting the control statement header but retaining the body of the statement as an unconditionally-executed piece of code. When keeping the body of a control statement we check all variables declared within the body and, for any name clashes with variables declared outside of the original control statement's scope, we rename those variables to avoid clashes.

Language operators as function calls

A key part of the generality of our approach is that we make language operators synonymous with function calls. To do this, before the genetic algorithm starts its first generation, the input source code passed into the framework is first parsed so that all syntactic language operators (such as '+') are transformed to pseudo-function-calls (such as 'add(a, b)') which are considered to be members of a built-in 'operator interface'. As such, all *language operators* are then simply equivalent to *function calls* for the purposes of all of the genetic operators described above, allowing a '+' to be changed to a '-' by altering the 'add(a, b)' call to any function call with the same parameter list (such as 'sub(a, b)'), or indeed to any function call with a type-compatible return value.

3.5 Crossover

In genetic programming, the crossover stage attempts to mimic chromosomal crossover in biological reproduction. We chose to implement this for source code as a line-by-line crossover mechanic in which a random line of code is selected in two source files (population members) A and B. The line at this point in A is then inserted into the chosen point in B. If the line in A is a control statement header, the entire control statement is inserted, otherwise the single line is inserted. If the insertion would mean that the inserted line refers to any variables that are not declared in the destination scope, declarations of these variables are also inserted from the original code in A. Any duplicate variable declarations that would collide with variables already declared in B are then renamed.

The random nature of line selection inherently allows crossover to insert either genetic material that already existed in the source component, or genetic material that was synthesised from one of our mutation operators. This crossover-by-line-insertion can occur N times, where N is a parameter of our framework; for $N > 1$ we ensure that the same line from A is not taken twice.

3.6 Infinite loops

Our genetic operators allow us to insert brand new loops with arbitrary termination conditions, which presents the possibility of non-terminating functions. We use a simple solution to mitigate this, via the built-in `halt` operation in the Dana language. If the execution of our benchmark tests takes more than twice the amount of time taken by the original component, we use the `halt` operation to forcibly stop all execution in the object (any in-progress function calls return with an exception) and mark this population member as having a very poor fitness value.

3.7 Limitations

The main limitation in our approach is the set of data types that can be used by our genetic operators, which focus on primitive types and arrays of those types. Our genetic operators cannot use composite `Data` types (analogous to structs in C) and cannot instantiate Object types. This limits the kinds of code that our approach can currently evolve, and is a key area of future work.

4 EVALUATION

We focus our evaluation on the hash map component of the emergent web server's set of potential building blocks, used by its cache component. The default hash map component is 92 lines of code long, which represents the entirety of the genetic source material over which our framework operates. We use this component in particular because different hash algorithms are known to perform differently over different sets of input data, producing a better or worse distribution of hashed keys across buckets. This component also does not use any composite data types and so is a good target as our genetic operators cannot yet work with these types.

We submit the default hash map component to our framework with a variety of different benchmarks and observe the resulting level of genetic divergence that is achieved. Each experiment is executed on a server of equal performance for a duration of six hours, after which we take the best-performing member of the final generation as the output of our framework that has been specialised for the given benchmark.

Our evaluation explores the following specific questions:

(1) To what extent we see genetic improvement, aided by our operators that are biased towards code synthesis, starting from a relatively small pool of genetic source material.
(2) The effect that the distinct synthesis phase has on genetic improvement, measured by running experiments in which entering this phase is prevented.
(3) The level of divergence we see when applying our approach to *different* benchmarks, measured by testing specialised output for each benchmark against all other benchmarks.

Our results are divided into three sections which provide initial data towards answering each of these questions.

4.1 Results

Core algorithm performance

We first focus on the extent to which we see genetic improvement of the hash component in a certain direction, given a particular benchmark test. In these tests our framework operates in its standard mode, with the synthesis phase enabled using an exploration period of 6 generations, and crossover taking one line of source code (or an entire control statement, if that line is a control statement header). Fig. 2 shows the performance of executing the benchmark over successive generations, averaged across 24 executions of the framework, focusing specifically on the performance of the best population member from each generation. The Y-axis of the graph is measured in percent, which represents how close the solution is to a perfectly even distribution of hash keys across buckets – a measure of optimal performance in a hash component.

The graph shows that the component's original implementation is relatively poor for this benchmark. Average performance of the

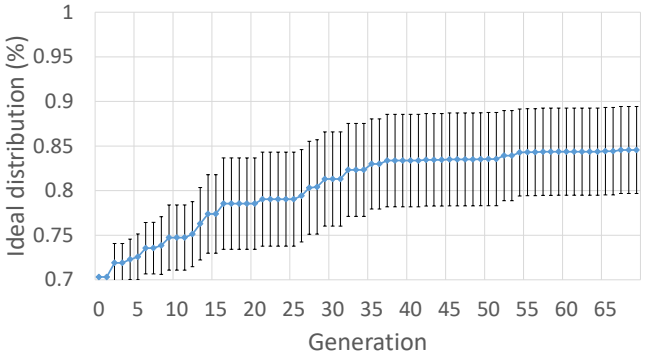

Figure 2: Average distribution fitness of the best population member across generations, with standard error.

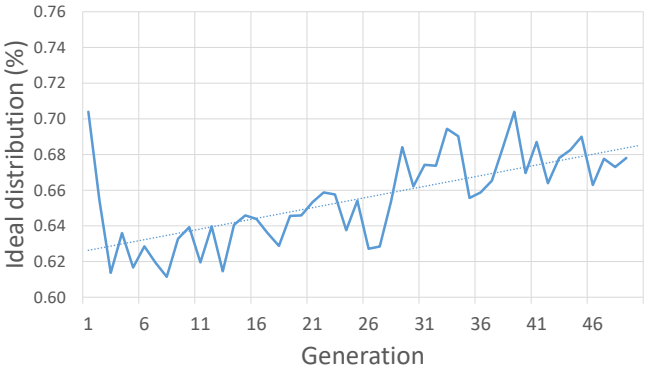

Figure 3: Average distribution fitness of all population members across generations.

mutated component across the experiments increases steadily up to generation 35, then increases more slowly up to generation 50, after which no further improvement is observed. The standard error, also plotted on the graph, indicates a significant range of variation between different experiment runs; from analysis of the raw data we see that this is predominantly caused by some experiments in which no improvement is made at all over the entire experiment.

Fig. 3 shows the average performance of *all* population members from the above experiments (not just the best-performing one). Here we see a more mixed result, with highly varying performance of individual members of a population, though the overall upward trend in improved performance is still present. This suggests that elitism has the desired effect in maintaining a high-performing member of each generation (as show in Fig. 2), that there is significant exploration of alternative solutions even towards the end of the experiment, and that positive attributes spread to the majority of population members as the experiment progresses.

Examining the code that is output by our framework in more detail, over the course of these experiments the average code length of the hash component increases from 92 lines of code up to 107. Considering the best-performing population member across each generation from the best-performing experiment, an average of 56 mutations occurred, and 6 crossovers; this is expected due to

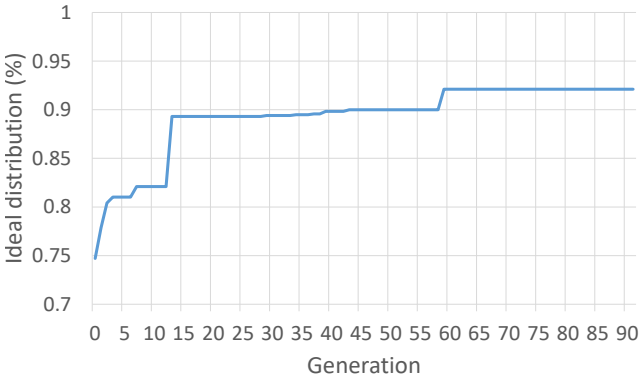

Figure 4: Average distribution fitness best population member with no synthesis phases.

the way in which we configured our algorithm with an 80% bias towards mutations (see Sec. 3). Of the mutations that took place, the best-performing population member experienced an average of 17.3 insertions, 17 modifications, and 22 delete operations.

At the end of the experiments, the resulting code of the best-scoring population members includes additional for-loops iterating over the key to hash, new declared variables, and new arithmetic operators that use these variables to store intermediate values. In addition to the creation of new algorithmic logic, an unexpected result of these experiments was that in some cases the 'bucket count' of the hash table was changed to a different value (because our mutators are permitted to modify constants). Our data therefore reflects the fact that the genetic search is simultaneously optimising both the bucket count and the hash distribution function for that bucket count towards a maximally even distribution of keys.

Synthesis phase evaluation

In the experiments reported in the previous section, an average of two synthesis phases were triggered – in which our framework observed a stable fitness value for an extended period and so switched from using reduced code length as part of its fitness value to instead using increased code length as an indicator of greater fitness, and discarding performance as a contributing factor to fitness.

To further investigate the effect of this mechanism, we ran experiments in which the synthesis phase was disabled. We also explore the effects of changing the number of generations for which a synthesis phase executes before returning to improvement mode.

Fig. 4 shows the average performance of the best population member across generations with the synthesis phase disabled. This shows similar upward trends in performance, finishing at a similar (very slightly higher) final fitness level. We also note that the number of generations executed is higher than in experiments with the synthesis phase enabled. These results suggest that the synthesis phase does not have a marked effect in reaching more distant areas of the search space, and that the inherent bias in our genetic operators towards those that synthesise new logic is already sufficient to have this effect. The increase in generations observed in these experiments indicates that the synthesis phase itself is computationally more expensive than the improvement phase.

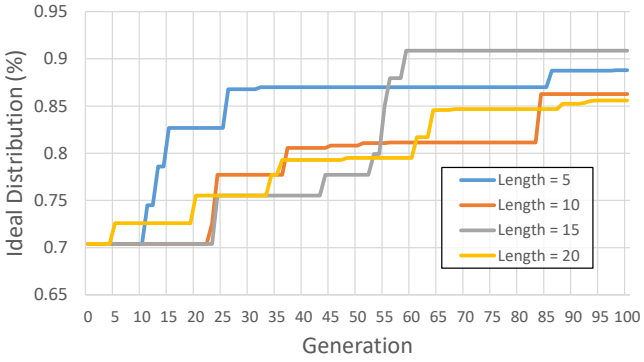

Figure 5: Average distribution fitness of best population member using different synthesis phase lengths.

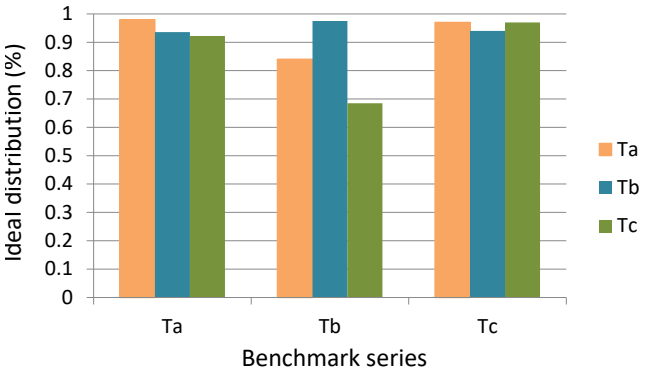

Figure 6: Distribution fitness of best population member, specialised against one of three different benchmark tests and then run against all three benchmarks.

We further examine the effect of using a distinct synthesis phase by varying the amount of generations it is permitted to use to explore longer programs, to reveal whether the synthesis phase is more effective when given more time. For these experiments we alter the exploration length of the synthesis phase to 5, 10, 15 and 20 generations. The results of these experiments are shown in Fig. 5. These results indicate that different synthesis phase lengths have little effect, even when given significantly more generations over which to operate, reinforcing the above observation that the distinct synthesis phase does not greatly increase the level of improvement seen. Although further investigation of this point is needed, we speculate that at least one reason for these negative results is that all of our experiments were bounded by the same total wall-clock time, such that the use of longer synthesis phases equates to spending proportionately more time exploring larger programs rather than improving programs (i.e., making the improver's job harder).

Divergence degree

In our final set of experiments we examine the extent to which our approach is able to achieve useful implementation *divergence* for different benchmarks. This is our primary goal in contribution to emergent systems, which require that different building block implementations are available to better suit different operating environment ranges detected at runtime.

For this experiment we take three different benchmark traces representing three different operating environment conditions; we refer to these three benchmark traces as T_A, T_B and T_C. For each one of these traces we then carry out a set of experiments in which we input the original hash map component into our framework with the corresponding benchmark trace. The output of an experiment run should then be a variant of the hash map component which is specialised towards that particular benchmark.

With the resulting three divergent implementations, optimised against these different benchmarks, we then run each of the three implementations against *all three* benchmark tests to observe the degree of specialisation that has taken place. Fig. 6 shows the results, grouped by experiment series on the x-axis.

Here we see clear specialisation within in each benchmark series. The best component produced by our framework for T_A clearly specialises towards that benchmark, achieving a fitness of 98% when used against this benchmark compared to 93% and 92% when tested against benchmarks T_B and T_C respectively. The best component produced in specialisation for benchmark T_B demonstrates a more dramatic difference, showing that the best component derived by our framework against this benchmark has 98% fitness when tested against T_B, with only 84% and 67% fitness when this component is then tested against benchmarks T_A and T_C respectively.

Finally, the best component produced in specialisation for benchmark T_C presents a more mixed picture, with a final component that performs relatively well against all three benchmarks, with a slightly worse performance against T_B. This suggests that the content of the benchmark T_C may be representative of generally common input sequences from other benchmarks, though we also note that this component does still perform slightly worse than the best component in the other two experiment groups against their specialised benchmarks.

Discussion

Returning to our evaluation questions, for the first question our results indicate that a mixed synthesis approach is consistently able to improve component performance towards a particular benchmark despite the small amount of genetic source material available; in other words, that we can usefully augment the limited genetic source material with randomly-selected, newly-synthesised logic in a general-purpose programming language.

For the second question, we see that the use of *distinct* synthesis phases – in which longer-length programs are rewarded – does not seem to be a useful approach in itself. From this data we conclude that the inherent bias towards synthesis in our set of genetic operators is sufficient to reach useful parts of the search space, without the need for dedicated synthesis phases.

Finally, for the third question, we see useful genetic divergence towards specific benchmarks, indicating that our approach may be a promising way of automatically populating a valuable set of diverse building blocks for emergent software systems that are likely to be useful in optimising towards different sets of operating environments observed at runtime.

5 CONCLUSION

We have presented an approach to achieving genetic divergence in sub-components of emergent software systems, in which we use a set of genetic operators that have a bias towards new code synthesis as well as being able to use existing genetic material. Emergent software systems are a natural fit for genetic improvement because they require a large pool of building blocks, where many such building blocks must alternative (but functionally equivalent) implementations that are better suited to different operating environment conditions detected at runtime. By capturing various observed operating conditions from the running system, we can recreate these conditions offline to specialise new components towards operating more optimally within each set of conditions.

Our initial results demonstrate that improvements in fitness are observed despite the relatively small amount of genetic source material available to our framework, indicating that our generalised synthesis operators are able to successfully augment this genetic source material to improve it in useful directions that would otherwise be unreachable. We also observe positive results in genetic divergence, where component implementations are seen to become specialised towards a specific benchmark (reflecting one particular set of operating environment conditions) and away from other benchmarks. While our experiment repetition count is limited in this paper, the trends shown in the results are sufficiently compelling to warrant further investigation of the technique.

In our future work we will enhance our genetic improvement framework to be able to deal with composite data types, object instantiation, and any other features needed so that it can operate with generality on any source code. We will then explore the effectiveness of our mixed synthesis approach in a much broader set of examples, enabling us to draw more general conclusions. We also intend to explore the ability of our framework to divide and combine logic from multiple components, so that it can automatically 'refactor' the broader design of a system; this will enable the improvement framework to present an emergent software system with different design options, in terms of software architectures, in addition to different implementations of individual building blocks.

ACKNOWLEDGEMENTS

This work was partly supported by the Leverhulme Trust Research Project Grant *The Emergent Data Centre*, RPG-2017-166.

REFERENCES

[1] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "RE^X: A development platform and online learning approach for runtime emergent software systems," in *Symposium on Operating Systems Design and Implementation*. USENIX, November 2016, pp. 333–348.

[2] R. Rodrigues Filho and B. Porter, "Defining emergent software using continuous self-assembly, perception, and learning," *Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 3, pp. 1–25, September 2017.

[3] Source code from this paper with instructions: http://research.projectdana.com/gi2018mcgowan.

[4] J. Petke, S. Haraldsson, M. Harman, william Langdon, D. White, and J. Woodward, "Genetic Improvement of Software: a Comprehensive Survey," *IEEE Transactions on Evolutionary Computation*, no. c, pp. 1–1, 2017.

[5] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 947–954.

[6] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, pp. 162–168, 2008.

[7] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," *Pldi*, pp. 43–54, 2015.

[8] V. Mrazek, Z. Vasicek, and L. Sekanina, "Evolutionary Approximation of Software for Embedded Systems: Median Function," *Genetic Improvement 2015 Workshop*, no. 1, pp. 795–801, 2015.

[9] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, Feb 2015.

[10] J. Landsborough, S. Harding, and S. Fugate, "Removing the Kitchen Sink from Software," *Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference - GECCO Companion '15*, pp. 833–838, 2015.

[11] B. Cody-kenny, E. Galván-lópez, and S. Barrett, "locoGP : Improving Performance by Genetic Programming Java Source Code," *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 811–818, 2015.

[12] S. O. Haraldsson and J. R. Woodward, "Automated Design of Algorithms and Genetic Improvement : Contrast and Commonalities," *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, pp. 1373–1380, 2014.

[13] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pp. 306–317, 2014.

[14] M. Orlov, "Evolving Software Building Blocks with {FINCH}," *Gi-2017*, 2017.

[15] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 317–328.

[16] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, A. V. Smith, and V. Gudnason, "Genetic improvement of runtime and its fitness landscape in a bioinformatics application," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 1521–1528.

[17] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliablilty of software operation," in *25th International Symposium on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years.*, Jun 1995, p. 113.

[18] J. Petke, "New operators for non-functional genetic improvement," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 1541–1542.

[19] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, p. 97, 2012.

[20] M. Orlov and M. Sipper, "Flight of the FINCH through the java wilderness," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 2, pp. 166–182, 2011.

[21] J. Swan and N. Burles, *Templar – A Framework for Template-Method Hyper-Heuristics*. Cham: Springer International Publishing, 2015, pp. 205–216.

[22] Dana language: http://www.projectdana.com.

[23] B. Porter, "Runtime modularity in complex structures: A component model for fine grained runtime adaptation," in *Component-Based Software Engineering*. ACM, June 2014, pp. 26–32.

[24] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998.