DOI: http://dx.doi.org/10.7551/978-0-262-32621-6-ch013

Reflective Grammatical Evolution

Christopher Timperley Susan Stepney

York Centre for Complex Systems Analysis, University of York, UK, YO10 5DD
ct584@york.ac.uk susan.stepney@york.ac.uk www.yccsa.org

Abstract

Our long term goal is to develop an open-ended reflective software architecture to support open-ended evolution. Here we describe a preliminary experiment using reflection to make simple programs evolved via Grammatical Evolution robust to mutations that result in coding errors.

We use reflection in the domain of grammatical evolution (GE) to achieve a novel means of robustness by autonomously repairing damaged programs, improving continuity in the search and allowing programs to be evolved effectively using *soft* grammars. In most implementations of GE, individuals whose programs encounter errors are assigned the worst possible fitness; using the techniques described here, these individuals may be allowed to continue evolving.

We describe two different approaches to achieving robustness through reflection, and evaluate their effectiveness through a series of experiments carried out on benchmark regression problems. Results demonstrate a statistically significant improvement on the fitness of the best individual found during evolution

Keywords: reflection, open-ended evolution, grammatical evolution, robustness

Introduction

Our long term goal is to develop an open-ended reflective software architecture to support open-ended evolution *in silico*, as outlined in Stepney and Hoverd (2011). As a first step, we are making evolutionary systems more robust to evolved programs that contain errors. Here we describe an experiment using reflection to make simple programs evolved via Grammatical Evolution robust to mutations that result in coding errors.

Grammatical evolution (GE) is a metaheuristic search method belonging to the family of evolutionary algorithms. The evolutionary algorithm finds solutions to problems by evolving a population of individuals, each representing a potential solution to the given problem. In GE the individual genomes are typically represented as integer lists, decoded through a provided grammar into syntactically correct computer program phenotypes. Evolution operators may nevertheless produce semantically faulty programs. If these faulty programs are removed from the population, any implicit heuristic information they may have gained will be lost to future generations. The alternative, examining the program phenotype to fix errors, is both complicated and grammar-dependent.

We instead use *reflection*, the ability of a program to inspect and modify its own code at run-time, to autonomously correct or *heal* the errors or *defects* that occur during the evolutionary process, in a grammar-independent manner. By being able to tolerate defects, a layer of *robustness* is incorporated into the search, allowing a wider range of mutations to be performed with less risk of losing good individuals from the population. This also allows less restrictive *softer* grammars to be used, which may improve locality in the search; traditionally such grammars would produce too many errors to be used effectively.

We demonstrate that reflection offers a viable mechanism for automatically correcting the dynamic population of individuals in GE. Using reflection, rather than a complicated grammar-dependent analysis of an individual's genotype and phenotype, allows the evolutionary process to remain unaware of the errors within its individuals' phenotypes and independent of how those errors are corrected. We achieve this isolation through a decentralised *robustness layer* that logically rests on top of the virtual machine of the programming language and below the layer of its programs.

The structure of the rest of this paper is as follows. We provide relevant contextual material on GE and reflection. We next describe two robustness layer designs, a global and a local one, to provide a robustness layer in Ruby programs. We then describe some evolutionary experiments, comparing these apporaches with a control case of no robustness layer, where faulty phenotypes are discarded.

Background

Grammatical Evolution

Grammatical Evolution (Ryan et al., 1998) is an evolutionary computation technique, implemented as a variant of the Genetic Algorithm (GA). The genotype typically takes the form of a list of integers, and the phenotype is produced by

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

mapping the genotype to a derivation of a given grammar. Typically this grammar is used to model the grammar of a given programming language and the resulting derivations form syntactically valid programs in that language.

Unlike Genetic Programming (GP), another EA variant that directly represents and operates on individuals as computer programs (typically using a tree-like structure), GE's genotype separates the program structure from the evolutionary process. This allows programs to be evolved using arbitrary forms and languages, beyond LISP-style tree structures, and allows the GE representation to be used with non-evolutionary meta-heuristics to produce programs (Dempsey et al., 2009), such as particle swarm optimisation (O'Neill and Brabazon, 2006).

Mapping Process Individuals are mapped to their phenotype through the use of a Backus-Naur Form grammar. The values v from the list of integers are sequentially read off from left-to-right and used to select the next production rule to use from the grammar; at each step, the rule at index i is selected, where $i = v \bmod m$, and m is the number of production rules available.

This process of derivation continues until one of the following situations occur:

- The mapping is completed, and all non-terminals have been transformed into terminals, producing a complete phenotype.
- The list of integers has been exhausted, but the mapping is incomplete. Usually in this case, the list is wrapped and its alleles are reused, allowing the derivation process to continue (Dempsey et al., 2009); however, this reduces locality within the chromosome, since codons may possess different meanings. An alternative is to cease the mapping process, but this produces a large number of incomplete and invalid derivations. Another alternative is to sample new codon values when they are required and to append these values to the end of the genome.
- A critical number of codons have been consumed. The derivation process is halted and the individual is assigned the worst possible fitness value; this mechanism prevents extremely long derivations from occurring.

Grammars Traditionally, GE uses heavily restricted, or *hard*, grammars designed to ensure syntactically and, as far as possible, semantically valid programs are always produced. Often these grammars are tailored to the domain of the specific problem being solved.

The performance of GE is highly sensitive to the choice of grammar down to its finest details (O'Neill et al., 2001). This problem, combined with the large domain of possible grammars, leads practitioners to use ad-hoc measures to design their grammar, if such considerations are given at all. Whilst *hard* grammars restrict the domain of the search, and hence reduce the scale of the problem, their inflexibility can leave them trapped in a local optima and unable to escape.

We propose instead the use of *soft* grammars, that allow smaller changes to be made to the overall program. These grammars result in a larger number of semantically incorrect programs, however, so we combine this approach with a *self-repair* mechanism. This broadens the genetic landscape and introduce flexibility into the phenotype. By incorporating such flexibility into the phenotype, neutral genetic drift is made possible; small changes to genotype may occur without dramatic consequence to the fitness of the individual. This transformation of the genetic landscape may allow the search to avoid the local optima of *hard* grammars, and may ultimately improve the performance of the algorithm.

Autonomic Computing

Self-healing is a form of *autonomic computing*, a term introduced by IBM in 2001 (Kephart and Chess, 2003), and used to describe computer systems capable of inspecting and modifying their behaviour in response to changing requirements. Inspiration for these systems came from the autonomic nervous system of the human body; an incredibly complex system composed of many interconnected components that seamlessly manage themselves to hide their complexity from us.

Kephart and Chess (2003) give four functional features that an autonomic computing system should possess:

1. **Self-optimisation:** The ability to profile the usage and control the allocation of resources, to provide optimal service quality. Self-optimisation could also include rewriting software functions to reduce resource usage.
2. **Self-configuration:** The ability to automatically configure components according to the context and environment of the system.
3. **Self-healing:** The ability to automatically diagnose and correct faults.
4. **Self-protection:** The ability to recognise and protect itself from malicious attacks.

Whilst the grand vision of autonomic computing is yet to come to fruition, its individual features are beginning to be realised. Much of this effort has been directed towards the self-healing aspect of autonomic computing, in the field of *resilient software*, which seeks to address the inherent fragility suffered by all computer systems, by developing simple and lightweight alternatives to the difficult robustness and fault tolerance techniques employed in safety-critical system, which often involve additional hardware.

Adaptive Software Systems Haydarlou et al. (2005) proposes the use of *adaptive software systems*, which modify their own behaviour in response to errors and environmental

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

changes with the help of an adaptive plan describing what modifications should be made. Where such systems have been used, they have typically employed a fixed adaptive plan, pre-designed by a human programmer. An alternative, outlined in Haydarlou et al. (2005), involves using genetic programming to evolve an adaptive plan to allow the system to autonomously adapt itself to new situations.

Fraglets Tschudin (2003) describes a radically different approach to creating a robust environment for hosting resilient software, capable of addressing the problems of soft faults and program adaptability from a pure software perspective, without the need for hardware fault tolerance measures. This is achieved by using a novel software execution model inspired by the interaction of chemical molecules, which places the ability to self-modify at the core of its design.

This model is composed of small computational atoms called *Fraglets*, which when combined through reactions, process both code and data. Each time program code is required, a new instance is dynamically loaded and executed rather than being stored in the memory of the computer. This allows the code to be easily manipulated by the program without the problems of accessing live code in the memory. Lost and failed Fraglet executions are handled by creating and executing a new instance of the affected Fraglet.

Metaprogramming and Reflection

Metaprogramming is the technique of writing computer programs which manipulate or write the code of other programs (Perrotta, 2010). These meta-programs are written using a *metalanguage*, which facilities the writing and manipulation of programs written in the target *object language* (Stump, 2009).

Reflection is the ability of an object language to act as its own metalanguage (Smith, 1982; Maes, 1987; Stump, 2009; Ferber, 1989). Reflective systems may exploit the ability of reflection to inspect and manipulate the functionality and implementation of the system itself. In some programming languages, such as Lisp and Ruby, this reflective system may be the programming language itself, allowing many aspects of the implementation and semantics of the language to be inspected, extended and redefined at run-time.

Design

Here we describe two designs for implementing a robustness layer through reflection: a centralised or "global" approach, and a "local" approach. We have implement both, and compared them through evolutionary experiments.

Following an analysis of several popular reflective programming languages, we have concluded that Ruby offers the richest suite of reflective capabilities and a natural tilt towards meta-programming, making it an ideal language to explore the robustness layer approach.

ARITHMETIC FUNCTIONS

⟨*op*⟩ ::= 'add' | 'sub' | 'mul' | 'div'

SOFT GRAMMAR

⟨*program*⟩ ::= ⟨*block*⟩

⟨*block*⟩ ::= ⟨*fcall*⟩ | ⟨*float*⟩ | ⟨*var*⟩

⟨*fcall*⟩ ::= ⟨*string*⟩ '(' ⟨*fargs*⟩ ')'

⟨*fargs*⟩ ::= ⟨*block*⟩ ',' ⟨*fargs*⟩ | ⟨*block*⟩

⟨*string*⟩ ::= ⟨*char*⟩⟨*string*⟩ | ⟨*char*⟩ | ⟨*char*⟩

⟨*float*⟩ ::= ⟨*nz_digit*⟩⟨*integer*⟩ '.' ⟨*integer*⟩
  | ⟨*digit*⟩ '.' ⟨*integer*⟩

⟨*integer*⟩ ::= ⟨*digit*⟩⟨*integer*⟩ | ⟨*digit*⟩

⟨*char*⟩ ::= 'a'..'z'

⟨*digit*⟩ ::= '0'..'9'

⟨*nz_digit*⟩ ::= '1'..'9'

⟨*var*⟩ ::= 'x' | 'y'

Figure 1: Soft grammar used throughout paper.

sdd(afg(mua(x, mua(x, x, y)), juli(x, x)), x) →
add(add(mul(x, mul(x, x)), mul(x, x)), x) →
$x^3 + x^2 + x$

Figure 2: An example of an evolved program, its effective form when using robustness measures, and its equivalent symbolic expression.

Here we investigate the use of these robustness measures with Ruby programs created via GE using a soft grammar (Figure 1). The only methods actually supported are the arithmetic functions (Figure 1), which take two arguments. The soft grammar can produce programs with method names that are arbitrary character strings, taking different numbers of arguments. (The long-term aim is to allow methods to mutate gradually into different methods; this is a first study.) To support this grammar, the robustness measures handle missing method errors, by mapping to an existing method, and incorrect argument errors, by truncating or padding the argument list. An example of a program evolved using the soft grammar is given in Figure 2.

Global Approach

The global approach achieves robustness by autonomously intercepting all exceptions thrown by monitored methods and applying fixes aimed at removing the root cause of the problem, calculated using details of the fault. This process can be seen as a form of *self-healing*, where the robustness layer *diagnoses* problems based on their details and automatically *heals* those problems by modifying the *damaged*

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

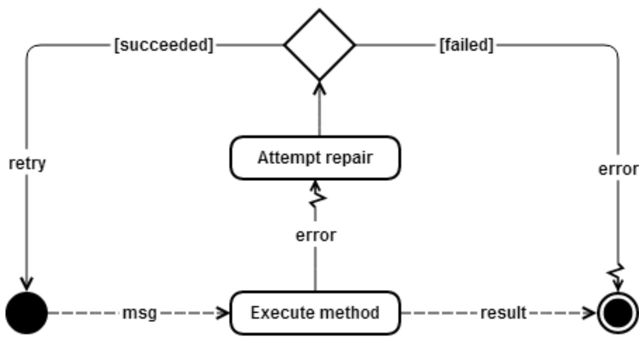

Figure 3: An overview of the global robustness layer design.

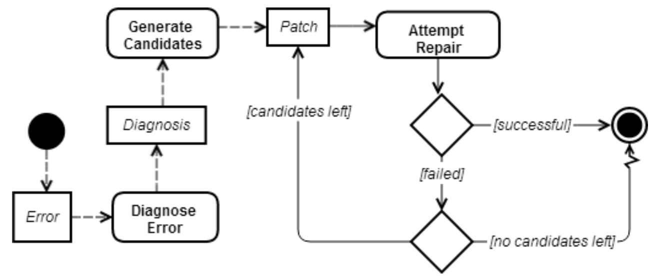

Figure 4: Architecture of the global robustness layer.

source code.

All calls to a *protected* method are routed through the global robustness layer, which attempts to execute the method according to its normal semantics.

- In the event of a fatal exception, the robustness layer becomes active and diagnoses the fault before calculating a suitable candidate fix to the issue.
- After the fix has been applied, the kernel attempts to call the modified method once again. In the event a subsequent error occurs, the process of diagnosis and repair is repeated, until either the method no longer produces errors during its execution, or till a maximum number of successive repairs has been reached.

Modifications made to the affected method during its "protected execution" may then be saved by the original method, removing the need to repair the method for successive calls. This ability proves useful for problems where the candidate solution is executed a number of times during evaluation, such as in regression problems.

Architecture

This layer is composed of three components which interact to examine the behaviour of protected method calls and to apply corrective measures in the event of their failure to prevent the program from crashing. These three components are:

Detection: Detects the occurrence of a specific type of fatal error within the affected method.

Diagnosis: Uses details of the error to determine its root cause within the source code of the affected method.

Repair: Uses the error diagnosis to apply corrective modifications to the source code of the affected method.

Instructions on how to detect, diagnose and repair specific types of errors are supplied to these components in the form of *error strategies*.

Immediately after an exception is thrown by a protected method, its details are passed to each of these error strategies, which try to detect if a certain type of error has occurred, the region of the source code responsible for producing the error, and how that code can be modified to remove the cause of the error.

After passing details of the error and method call to each error strategy, a set of candidate fixes are produced which are exhaustively tried until either the original error is no longer present, or there are no fixes left, in which case the original exception is thrown again, or a threshold number of tries is reached (10 in the work reported here).

Global Error Handling

The grammar as defined can produce two specific errors.

Missing Method Calls. Calls to non-existent methods are dealt with by changing all instances of the method name used to the name of an existing method that has with the lowest Levenshtein distance to the original.

Incorrect Number of Arguments. This is handled by reducing calls with too many parameters to the correct size, and padding calls with too few parameters with zeroes.

This transformation process manipulates the deepest method calls first (i.e. method calls that are used as parameters to other calls) to ensure that the resulting program is well-formed and that all calls to the affected method use the correct number of arguments.

Local Approach

Whereas the global robustness layer waits for a protected method to encounter a fatal error before attempting to repair the region of source code responsible for its cause, the local robustness layer is designed to achieve robustness by adapting the Ruby kernel itself to ensure that fatal exceptions are never raised in the first instance.

Instead of using a set of strategies to heal from specific errors, a set of changes are made to areas of the Ruby kernel capable of producing such errors, via a process known as "monkey patching" (Perrotta, 2010). By applying modifications to certain aspects of the programming language,

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

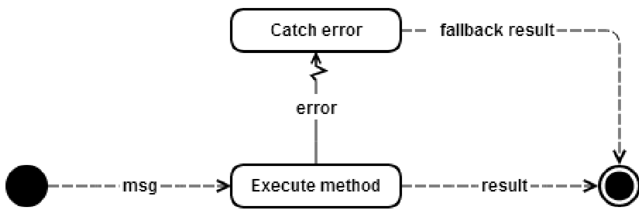

Figure 5: An overview of the local robustness layer design.

Selection	Tournament
Mutation	Uniform Random
Crossover	Two Point
Replacement	Generational
Evaluation Limit	10,000

Table 1: EA setup used for all experiments.

there is no need for a central entity to handle the execution and repair of protected methods; instead each fix acts independently, and the combination of them produces a thin robustness layer.

Local Error Handling

Missing Method Calls. To deal with calls to non-existent functions, the local approach implements Ruby's `method_missing` hook in the `Module` class, which is called when a requested method cannot be found inside a particular class.

The implementation of this function is similar to the approach taken to handling missing method calls in the global robustness layer; the function call is routed to the suitable function with the lowest Levenshtein distance to the requested function.

This re-routing mechanism also enforces a maximum distance constraint, which prevents the call from being routed to a function whose name is too far away from the requested method.

Incorrect Number of Arguments. Although the `missing_method` callback is capable of routing calls to non-existent functions towards their closest candidate function, this behaviour alone does not prevent an error being raised when the wrong number of arguments are then supplied.

To provide robustness against such errors, the implementation makes use of the `method_added` callback and the `remove_method` magic functions. Since there are no callbacks in place to handle function calls to existing functions with the incorrect number of arguments, the implementation instead captures and stores the object for every method added to the module in a private hash (indexed by the name of the method), via the `method_added` callback, before immediately removing the method from the module via the `remove_method`.

Every method call made to the module then invokes the `missing_method` callback, since methods are removed immediately after being added. The `missing_method` callback checks if the requested method "exists" by looking for its entry in the method hash; if there is no entry for the method then the closest candidate function is selected instead (if this is not possible then an exception is raised), before adjusting its arguments to agree with the arity of the selected method.

The arguments provided with the method call are made to fit with the expected argument structure by removing excess arguments from the end of the arguments array or by padding the arguments with zeroes to the right to compensate for missing arguments.

Experiments

To determine the effect of the robustness layer on the evolution trajectory during GE, and to test the design hypothesis, we have conducted a series of benchmark evolutionary experiments. To perform these experiments, we used our own EA implementation, written in Ruby, whose components are detailed in Table 1. (After some preliminary testing, we found that two-point crossover yielded better results than the more traditional single point crossover.)

These experiments compare the performance and behaviour of GE using global, local and no robustness measures, across a number of benchmark multi-modal symbolic regression problems from McDermott et al. (2012), listed in Table 2.

The genome is a fixed-length list of 100 integers, each taking a value between 0 and $2^{31}-1$. The mutation operator uses uniform random replacement, replacing the value of a codon with a uniform random value from the legal range. Each genome is initialised with 100 uniform random values from the legal range.

The evaluation function measures the fitness of individuals as the sum of squared differences between their actual and expected results using data from a pre-generated training set. Hence low values are better than higher values, and the best achievable fitness is 0.

Calibration

Good parameter choices for each robustness measure, chosen from a given set of possible parameter values (Table 3), were determined through calibration based on the Keijizer-15 benchmark.

Due to limits upon time and computational resources, an exhaustive search of the space of all possible parameters is not possible. Instead, we employed the Relevance Estimation and Value Calibration of Evolutionary Algorithm

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

Name	Vars	Objective Function	Training Set
Koza-1	1	$x^4 + x^3 + x^2 + x$	$U[-1, 1, 20]$
Koza-3	1	$x^6 - 2x^4 + x^2$	$U[-1, 1, 20]$
Keijzer-12	2	$x^4 - x^3 + \frac{y^2}{2} - y$	$U[-3, 3, 20]$
Keijzer-14	2	$8/(2 + x^2 + y^2)$	$U[-3, 3, 20]$
Keijzer-15	2	$\frac{x^3}{5} + \frac{y^3}{2} - y - x$	$U[-3, 3, 20]$

Table 2: Regression problems used to perform comparison. $U[a, b, c]$ is c uniform random samples drawn from a to b, inclusive, for the variable.

Parameter	Range
Tournament Size	$[2, 10]$
Elitism	$[0.0, 0.5]$
Mutation Rate	$[0.001, 0.5]$
Crossover Rate	$[0.1, 1.0]$
Population Size	$[10, 200]$

Table 3: The space of evolutionary parameter values.

Parameters (REVAC) method (Nannen and Eiben, 2007) to calibrate the evolutionary parameters of our algorithm.

We used the REVAC parameters of Smit and Eiben (2010), except with fewer evaluations (1000 instead of 5000) to reduce computational effort (Table 4). To measure the expected performance of each parameter vector, we use the mean best fitness (MBF) across a number of runs.

The results of calibration are shown in Table 5, and are the evolutionary parameters used for the evolutionary experiments.

Main Experiments

After determining good evolutionary parameters to use for each of the robustness measures, we compared the performance of each of the measures over 100 runs of each of the benchmark functions.

Results from these experiments (Figure 6 and Table 6) show an improvement in both the median and minimum best fitness values when using either local or global measures, compared to using no robustness measures, for all benchmarks except Koza-3.

In all cases, the use of local robustness measures gave a minimum best fitness that was at least as good as that obtained using global robustness measures, and was usually better; this may be due to the local robustness layer's ability to cope with an unlimited number of errors within any given program.

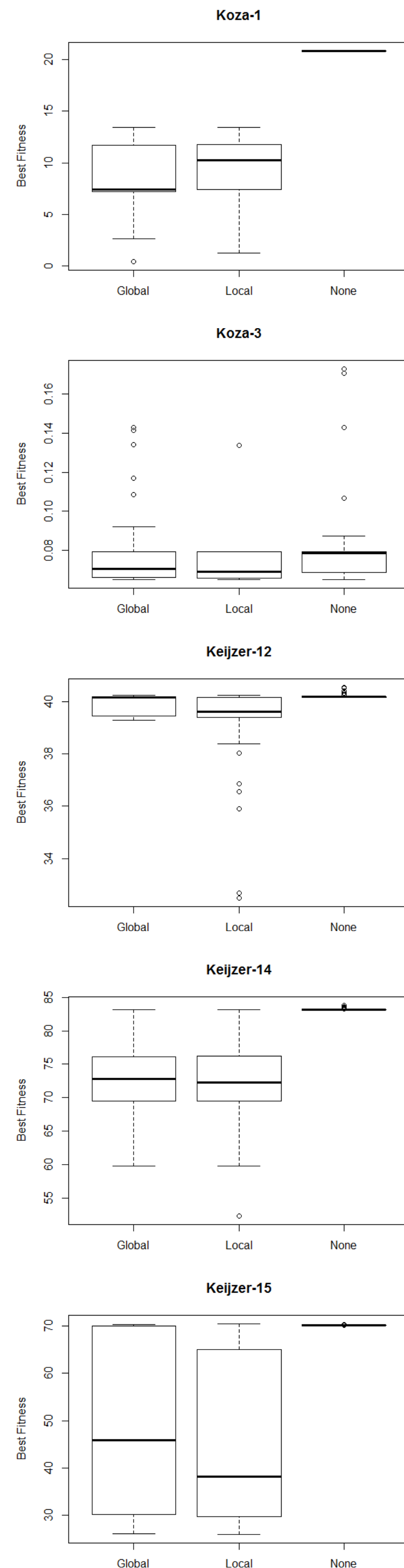

Figure 6: Distribution of best fitness values across 100 runs for each benchmark (low fitness values are better).

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

Population size	80
Best size	40
Smoothing coefficient	10
Repetitions per vector	10
Maximum number of vectors tested	1000

Table 4: REVAC parameters used during all tuning sessions.

Parameter	Global	Local	None
Tournament Size	2	4	2
Elitism	0.440	0.459	0.438
Mutation Rate	0.425	0.363	0.375
Crossover Rate	0.705	0.544	0.764
Population Size	161	161	189

Table 5: The parameter vectors for each measure.

Statistical Significance

We used the non-parametric Mann-Whitney U test to measure the statistical significance of the results, and to test the null hypothesis, H_0: "the use of robustness measures have no effect on the fitness of the best individual found during evolution, compared to using no measures, when evolving a soft grammar."

We test to the standard 95% confidence level. We are performing 10 benchmark comparisons (Table 7), so we apply a Bonferroni correction and require each comparison to have a confidence level of 99.5%, or $p < 0.005$. Results given in Table 7 show that $p < 0.005$ on each benchmark except Koza-3. We reject the null hypothesis at the 95% confidence level, except for Koza-3.

Since using a large enough sample is usually enough to produce a statistically significant result, we also measure the effect size; we use the non-parametric Vargha-Delaney A value (Vargha and Delaney, 2000). From the results shown in Table 7, we can observe a large effect size between the use of global or local measures and no measures on almost of the benchmark functions, again except for Koza-3. These results suggest that the use of robustness measures produce a significant difference to the best fitness value found during evolution.

Conclusions

We have designed, implemented and tested two approaches that exploit reflection to perform self-repair in programs evolved using GE. We have demonstrated that both these measures have a statistically significant effect on the performance of GE using soft grammars in a series of benchmark functions.

	Max	UQ	Median	LQ	Min
Global Measures					
Koza-1	13.415	11.682	7.387	7.291	0.460
Koza-3	0.143	0.794	0.071	0.066	0.0652
Keijzer-12	40.246	40.170	40.156	39.439	39.274
Keijzer-14	83.195	76.049	72.756	69.552	59.833
Keijzer-15	70.280	70.012	45.935	30.223	26.100
Local Measures					
Koza-1	13.412	11.792	10.267	7.387	1.246
Koza-3	0.134	0.079	0.069	0.066	0.065
Keijzer-12	40.232	40.157	39.620	39.390	32.502
Keijzer-14	83.195	76.208	72.318	69.541	52.359
Keijzer-15	70.443	64.435	38.239	29.914	25.987
No Measures					
Koza-1	20.801	20.801	20.801	20.801	20.801
Koza-3	0.173	0.079	0.078	0.069	0.065
Keijzer-12	40.537	40.185	40.185	40.158	40.156
Keijzer-14	83.787	83.195	83.195	83.195	83.195
Keijzer-15	70.280	70.162	70.162	70.156	70.153

Table 6: Results from the benchmark experiments (low fitness values are better)

Measure	Benchmark	p	A
Global	Koza-1	3.293×10^{-30}	1.000
	Koza-3	0.0173	0.595
	Keijzer-12	3.761×10^{-12}	0.781
	Keijzer-14	1.003×10^{-35}	0.982
	Keijzer-15	1.304×10^{-19}	0.864
Local	Koza-1	4.163×10^{-39}	1.000
	Koza-3	0.0011	0.630
	Keijzer-12	4.483×10^{-20}	0.872
	Keijzer-14	2.670×10^{-37}	0.995
	Keijzer-15	3.044×10^{-30}	0.962

Table 7: Statistical significance (Mann-Whitney U test p) and effect size (Vargha-Delaney A value) of use of global and local robustness measures each compared to no measures. An A value > 0.56 is a "small" effect; an A value > 0.71 is a "large" effect.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024

Limitations

Although the local robustness layer is capable of handling the errors investigated here, to handle each error, a series of changes to the Ruby kernel has to be made. This approach is thus limited by the ability of the Ruby kernel to catch such errors. More complex errors, such as name errors raised when an unreferenced variable is used, require the robustness layer to know their context and to modify the original source code, and cannot be intercepted and dealt with on the fly. Since the global robustness layer is capable of examining the context of errors, and has the required ability to apply changes to the source code, it is better equipped to diagnose and repair a far wider category of errors.

Whilst the local robustness measures performed well across all of the problems, an unfortunate consequence of their changes to the Ruby kernel restricts them to a single thread during evolution. However, this shortcoming may be addressed using Ruby's recently introduced *refinement* functionality (Cooper, 2010), which allows kernel changes to be localised within a given context.

Another weakness of the local approach is that its inherent handling of errors on the fly means that the semantics of programs produced during evolution may not be the same when the local robustness layer is not deployed, since the code is never changed and errors will no longer be dealt with. One solution to this problem may be to pass individuals to a finaliser at the end of the evolutionary process, which analyses the execution of their associated program and monitors exceptions caught by each of the patches to determine the effective source code of the program.

Future Work

Having demonstrated the effectiveness of robustness measures when combined with soft grammars, we are now investigating the use of radically more expressive grammars, modelling subsets of an entire programming language, capable of universal computation.

We believe that by taking this approach, we can free the programmer from performing arbitrary decisions on their choice of grammar, and that entirely novel solutions to problems, beyond the scope of conventional hard grammars, may be discovered.

We are also investigating the use of techniques which allow the use of alternative meta-languages to perform correction, instead of the same language as the program, allowing programs to be evolved in arbitrary languages, regardless of their support for reflection.

Our long term goal is to use such techniques in the development of an open-ended an open-ended reflective software architecture to support open-ended evolution *in silico* (Stepney and Hoverd, 2011).

References

Cooper, P. (2010). Ruby Refinements: An Overview of a New Proposed Ruby Feature. http://www.rubyinside.com/ruby-refinements-an-overview-of-a-new-proposed-ruby-feature-3978.html.

Dempsey, I., O'Neill, M., and Brabazon, A. (2009). *Foundations in Grammatical Evolution for Dynamic Environments*. Springer.

Ferber, J. (1989). Computational reflection in class based object-oriented languages. In *OOPSLA '89*, pages 317–326. ACM.

Haydarlou, A. R., Overeinder, B. J., and Brazier, F. M. T. (2005). A self-healing approach for object-oriented applications. In *Proc. 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems*, pages 191–195.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Maes, P. (1987). Concepts and experiments in computational reflection. *ACM Sigplan Notices*, 22(12):147–155.

McDermott, J., White, D. R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., and O'Reilly, U.-M. (2012). Genetic programming needs better benchmarks. In *GECCO '12*, pages 791–798. ACM.

Nannen, V. and Eiben, A. E. (2007). Relevance estimation and value calibration of evolutionary algorithm parameters. In *IJCAI'07*, pages 975–980. AAAI Press.

O'Neill, M. and Brabazon, A. (2006). Grammatical Swarm: The generation of programs by social programming. *Natural Computing*, 5(4):443–462.

O'Neill, M., Ryan, C., and Nicolau, M. (2001). Grammar defined introns: An investigation into grammars, introns, and bias in grammatical evolution. In *GECCO 2001*, pages 97–103.

Perrotta, P. (2010). *Metaprogramming Ruby*. Pragmatic Bookshelf.

Ryan, C., Collins, J., , and O'Neill, M. (1998). Grammatical Evolution: Evolving programs for an arbitrary language. In *Proc. 1st European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95. Springer.

Smit, S. K. and Eiben, A. (2010). Beating the 'world champion' evolutionary algorithm via REVAC tuning. In *CEC 2010*, pages 1–8. IEEE.

Smith, B. C. (1982). *Reflection and semantics in a procedural language*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science.

Stepney, S. and Hoverd, T. (2011). Reflecting on open-ended evolution. In *ECAL 2011, Paris, France, August 2011*, pages 781–788. MIT Press.

Stump, A. (2009). Directly reflective meta-programming. *Higher Order Symbol. Comput.*, 22(2):115–144.

Tschudin, C. F. (2003). Fraglets – a metabolistic execution model for communication protocols. In *Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park*.

Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educ. Behav. Stat.*, 25(2):101–132.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alife2014/26/71/1901889/978-0-262-32621-6-ch013.pdf by guest on 22 February 2024