The GISMOE challenge: Constructing the Pareto Program Surface Using Genetic Programming to Find Better Programs*

Mark Harman[1], William B. Langdon[1], Yue Jia[1], David R. White[2], Andrea Arcuri[3], John A. Clark[4]
[1]CREST Centre, University College London, Gower Street, London, WC1E 6BT, UK.
[2]School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland, UK.
[3]Simula Research Laboratory, P. O. Box 134, 1325 Lysaker, Norway.
[4]Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK.

ABSTRACT

Optimising programs for non-functional properties such as speed, size, throughput, power consumption and bandwidth can be demanding; pity the poor programmer who is asked to cater for them all at once! We set out an alternate vision for a new kind of software development environment inspired by recent results from Search Based Software Engineering (SBSE). Given an input program that satisfies the functional requirements, the proposed programming environment will automatically generate a set of candidate program implementations, all of which share functionality, but each of which differ in their non-functional trade offs. The software designer navigates this diverse Pareto surface of candidate implementations, gaining insight into the trade offs and selecting solutions for different platforms and environments, thereby stretching beyond the reach of current compiler technologies. Rather than having to focus on the details required to manage complex, inter-related and conflicting, non-functional trade offs, the designer is thus freed to explore, to understand, to control and to decide rather than to construct.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Algorithms, Design, Experimentation, Human Factors, Languages, Measurement, Performance, Verification.

*This position paper accompanies the keynote given by Mark Harman's at the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12) in Essen, Germany. It is joint work with Bill Langdon, Yue Jia, David White, Andrea Arcuri and John Clark, funded by the EPSRC grants SEBASE (EP/D050863, EP/D050618 and EP/D052785), GISMO (EP/I033688) and DAASE (EP/J017515/) and by EU project FITTEST (257574).

Keywords

SBSE, Search Based Optimization, Compilation, Non-functional Properties, Genetic Programming, Pareto Surface.

1. INTRODUCTION

Humans find it hard to develop systems that balance many competing and conflicting non-functional objectives. Even meeting a single objective, such as execution time, requires automated support in the form of compiler optimisation. However, though most compilers can optimise compiled code for both speed and size, the programmer may find themselves making arbitrary choices when such objective are in conflict with one another.

Furthermore, speed and size are but two of many objectives that the next generation of software systems will have to consider. There are many others such as bandwidth, throughput, response time, memory consumption and resource access. It is unrealistic to expect an engineer to decide, up front, on the precise weighting that they attribute to each such non-functional property, nor for the engineer even to know what might be achievable in some unfamiliar environment in which the system may be deployed.

Emergent computing application paradigms require systems that are not only reliable, compact and fast, but which also optimise many different competing and conflicting objectives such as response time, throughput and consumption of resources (such as power, bandwidth and memory). As a result, operational objectives (the so-called non-functional properties of the system) are becoming increasingly important and uppermost in the minds of software engineers.

Human software developers cannot be expected to optimally balance these multiple competing constraints and may miss potentially valuable solutions should they attempt to do so. Why should they have to? How can a programmer assess (at code writing time) the behaviour of their code with regard to non-functional properties on a platform that may not yet have been built?

To address this conundrum we propose a development environment that distinguishes between functional and non-functional properties. In this environment, the functional properties remain the preserve of the human designer, while the optimisation of non-functional properties is left to the machine. That is, the *choice* of the non-functional properties to be considered will remain a decision for the human software designer.

However, all decisions concerning the implementation details that will maximise or minimise non-functional properties will be delegated to the machine. Different versions of the program can be constructed for different environments (even those yet to be built), while maintaining the functional properties originally set out by the software designer.

Non-functional properties will often be in conflict with one another. For example, faster response times may increase power, memory or bandwidth consumption. Therefore, we do not propose that there will be a single solution program offered. Rather, we propose that the development environment will produce a Pareto surface of candidate programs, each of which shares the functional properties specified by the human, while presenting different non-dominated solutions with respect to non-functional properties.

A solution is non-dominated if no other solution is superior according to all non-functional properties. Each non-dominated solution thus denotes a candidate trade off between the non-functional properties. The surface of solutions might contain thousands, perhaps millions of such candidate programs, all of which perform the same function, but each of which does so with different non-functional trade offs. The programmer's task is to define the functional properties and to explore the space of non-functional solution alternatives provided by the automated development environment.

In this paper we explain how a combination of advances in software test data generation, genetic programming and multi objective optimisation can be combined to realise this vision. We present an architecture and principal features of this proposed development environment, which we call GISMOE: Genetic Improvement of Software for Multiple Objective Exploration.

The GISMOE approach draws its description of functional characteristics either from a pre-existing program or from a declarative specification of functional requirements. We focus on pre-existing programs as the source of functional descriptions in this paper (with a brief discussion of the alternative in Section 6.2). GISMOE uses automated testing to assess the degree to which the functional properties are preserved and to measure the achievement of non-functional requirements. Measurements of both functional and non-functional requirements are used to provide the fitness functions that guide a multi objective optimisation and visualisation process. We envisage that the optimisation will be achieved by a Genetic Programming system, though there may be other possibilities.

We present both a 'conformant' and 'heretical' version of our GISMOE development environment. The heretical version is merely an extension of the conformant version, but it crosses a rubicon that may be unacceptable to some software engineers, and will certainly be inappropriate for some application areas (such as safety critical systems).

The conformant version is faithful to the long-cherished belief that software engineers should strive for correct systems. The heretical view is that even correctness, as measured approximately (through testing), could become just another measurable objective to be traded off against non-functional properties.

In order to re-cast functional properties such as optimisation objectives, within the same framework as the non-functional properties, we shall need testing at a sufficiently fine level of granularity and with a sufficiently rich mapping from test cases to meaning.

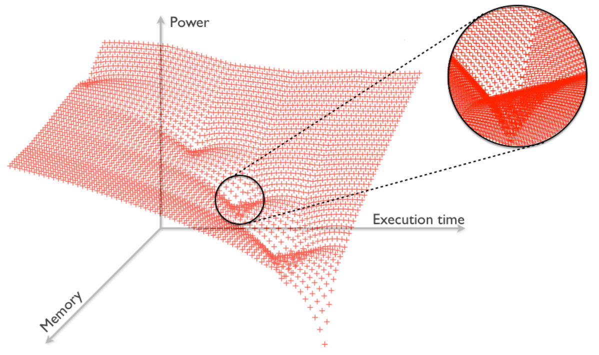


Figure 1: The GISMOE Pareto Program Surface

We review (in overview) recent results, which we believe provide the initial empirical evidence on which we rest our belief that the GISMOE vision is achievable. Though we believe this evidence to be promising, we claim only that it demonstrates feasibility at the level of a 'proof of concept'. We recognise that the GISMOE research agenda currently remains more of a 'grand challenge' than a reality. We hope that this paper will provide a sufficiently compelling argument for the GISMOE agenda that others will feel motivated to take up this challenge.

2. THE PARETO PROGRAM SURFACE

Suppose we plot a large set of programs on a surface. Each program on this surface is plotted in three dimensions (x, y, z), where x is the execution time of the program, y is the power consumed by the program and z is the memory consumed by the program (see Figure 1 for an illustration).

For each of these three dimensions, there are many questions about how we shall define the measurements: do we want the average, the worst case, the most common case? All of these (and more) are possibilities. For different GISMOE applications, each will be useful. Let us suppose that we have agreed a definition of each of these non-functional properties.

To make the discussion concrete, let us suppose (for illustration purposes) that we are concerned with an embedded controller and that we wish to reduce worst case power consumption, peak memory use and execution time (even should this lead to increases in average and best case performance for these properties).

Suppose we have an existing implementation of the controller, designed by human programmers and for which there is a well-established test suite collected from human-designed test cases combined with test cases extracted from the logs of in-situ execution in the field. Suppose the software designer is presented with a Pareto surface like that depicted in Figure 1. Each point on this surface represents an implementation of the controller that passes all of the test cases.

However, each point also denotes a trade off between the three non-functional objectives of power, speed and memory. That is, no point on the surface is superior to any other point according to all three non-functional criteria, so each represents a valid choice; sacrificing one objective for the other, the designer could choose an implementation that best suits the problem in hand.

For instance, if one version of the controller has to be used in a low power environment then power can be favoured, whereas, for some other instantiation of the controller, worst case response time may be paramount (even at the expense of increased memory use and power consumption).

Like all Pareto fronts, the Pareto program surface may also reveal so-called knee points; the presence of regions of the solution space in which a rapid change in the trade offs between objectives is observed. Whether or not the software engineer ultimately selects for deployment any solution in such a region of the Pareto program surface, he or she will be able to analyse the surface gaining insight into the underlying programming problem and the trade offs inherent in any solution. The **GISMOE challenge** is thus:

> *to create an automated program development environment in which the Pareto program surface is automatically constructed to support dialog with and decision making by the software designer concerning the trade offs present in the solution space of programs for a specific programming problem.*

Where a solution program on the surface is deemed to be acceptable to the designer, it can simply be selected as the solution. Where the designer is unwilling to vest this degree of trust in an automated program-space exploration tool, the GISMOE Pareto program surface will still offer value: the designer can 'zoom in' on regions of the surface that offer interesting and perhaps insightful trade offs between the objectives (as illustrated in Figure 1).

The software designer can request that the system produce more solutions in this region to flesh out the shape of the surface in this neighbourhood. The designer can also ask the system to report a code template for solutions in a region of interest, where the template captures the syntax that all programs in the region share. Through this mechanism the designer may recognise a pattern in the coding solutions that had not previously been considered and which offers a novel and 'clever' way to balance competing and conflicting non-functional requirements.

The end result of interaction with the GISMOE development environment may therefore not be a program constructed by the tool at all. Rather, it may be that the human engineer merely gains a better understanding of the underlying programming problem and the possible solutions that the search space offers.

The engineer can also be brought into the optimisation process itself as an interactive contributor to the fitness assessment for candidate solutions. This is the open challenge of 'Human in the loop' set out in Section 6.4.

In this example scenario, we considered only three objectives for a hypothesised embedded controller with a human generated test suite. However, the example was merely intended as an illustration: the GISMOE vision is far more general than this. We seek to use automated test data generation to provide test cases (using the original program as an oracle). This imbues GISMOE with the luxury of a fully automated testing process. We also constrain neither the number nor type of non-functional properties that can be considered, though there will be additional visualisation issues when we seek to optimise for more than three such objectives (See Section 6.3). We also do not constrain ourselves to situations in which there is a pre-existing version of the program (See Section 6.2), though we confine ourselves to this scenario for the majority of the remaining discussion.

3. THE GISMOE MOTIVATIONS

In this section we set out the motivation for the GISMOE research agenda. We describe how each of the components of the overall approach fit together to form a coherent vision of a new way of developing software. We propose a 'next generation' software development environment that offers intelligent decision support to the software engineer. With this proposed development environment the software engineer makes the transition from programmer to software designer.

Non-Functional Measureability: The code to meet unknown functional properties cannot be automatically generated. This requires human effort and creativity to describe the requirements and to turn the users' functional requirements into an executable form (which GISMOE uses as an oracle). By contrast, the non-functional requirements are easier to describe and handle.

For example, a user may spend many person months in requirements specification for functional requirements, but the non-functional requirements might merely state that the resulting program should be fast and compact[1]. How fast? How compact? The answer is 'as fast and compact as possible', but what is possible? This is where GISMOE can provide answers, by exploring the space of non-functional trade offs between speed and compactness; the Pareto program surface.

The two non-functional properties of speed and compactness can be measured and can therefore be 'optimised into' the programs produced to explore the range of implementations that meet the functional requirements, while offering different balances of trade off between speed and compactness. Similarly, any set of non-functional properties (bandwidth, throughput, response time, memory consumption, resource access, etc.) can be measured and used as an optimisation objective.

The human will always be required to elicit and understand the functional requirements and to decide upon their translation into test cases that capture and delimit what is expected of the functional behaviour of the system. Nevertheless, most systems will have many other aspects of their behaviour that are non-functional and equally important. For such non-functional requirements, the goal is not the achievement of a specific, detailed set of 'discrete' (functional) requirements. Rather, the goal is to make progress to a solution that performs better with regard to the more 'continuous' measurements observed for the non-functional requirements, when executed in the intended environment.

Non-Functional Sensitivity Analysis: It has been argued [7, 40, 60, 63] that Genetic Programming (GP) can scale more readily when provided with a guide to identify the code that should be modified. This is effective when evolution starts with an existing system, seeking to evolve it to meet new requirements or to better perform with respect to existing requirements.

In the GISMOE approach, we advocate the use of a 'sensitivity' analysis to identify those parts of the code that

[1]We recognise that there will be many issues in the capture of these non-functional requirements and their effective measurement [64, 85]. However, our observation is that non-functional properties, when measurable, can be 'optimised into' the system; something that is much harder for functional aspects of the system.

are sensitive to each of the non-functional requirements. By 'sensitivity' we mean the degree to which the observed measurements of a non-functional property are affected by changes to an element of the program. That is, those elements for which changes lead to larger effects are considered to be more sensitive.

This analysis is independent of the assessment of the functional properties of the system. For example, we can identify which code elements most affect speed of execution without considering whether changing any such element will affect the functional computation performed; it surely will, but that is not the point of the sensitivity analysis.

In this way, the sensitivity analysis can produce an ordering of program elements according to the degree to which they are observed (through the sensitivity experiments) to affect the non-functional property of interest. We shall construct one such ordering for each non-functional property. The ordering can be used to inform the selection of code elements to evolve by the GISMOE GP engine.

Advances in Automated Test Data Generation: Recent advances in automated software test data generation [6, 10, 29, 34, 44, 47, 56, 81] mean that it is possible to generate a large number of test inputs that achieve coverage of structural features of code. While the problem of test data generation for structural coverage is yet to be fully solved [57], recent advances in Search Based Software Testing (SBST) [6, 29, 44, 47] and Dynamic Symbolic Execution (DSE) [34, 81] and their combination [10, 56] have led to publicly available systems that can generate reasonable quality coverage-based test suites.

In particular, SBST allows us to target specific test objectives as fitness functions, thereby generating suitable test inputs. SBST can provide the GISMOE approach with inputs for functional testing by targeting coverage of the initial version of the program to be developed. However, SBST can also target non-functional properties [3], thereby providing a way to improve the targeting of testing towards the specific non-functional properties of interest.

Since test data generation and the evolutionary search for new implementations are both population-based optimisation processes we can develop a co-evolutionary approach to GISMOE. In the co-evolutionary GISMOE model, the test cases (both functional and non-functional) are evolved to try to find faults and worst case performance with the current best solutions on the Pareto program surface. At the same time, the programs on the Pareto program surface are evolved to try to minimise failing functional tests, while maximising achievement of the non-functional objectives

Plastic Surgery: Previous work on genetic programming for bug fixing [63] has used a technique sometimes referred to as 'plastic surgery' [40], in which seeds to the genetic programming systems are provided by locating fragments of related code elsewhere in the system under evolution. This fixes faults in much the same way that a plastic surgeon heals wounds using skin grafts from unaffected areas of the body.

We propose to extend this code-search from the program under evolution to all programs that can be found by searching local repositories or the internet. There are large corpuses of available code from which the GISMOE system can seek code. We envisage a situation in which the tool automatically 'scavenges' for fragments of code in source code repositories, choosing fragments that appear promising.

It remains an interesting topic for future work to adapt techniques for syntactic and semantic similarity measurement to guide such a scavenging process. There is much work on code search and analysis [21, 73] from which we might draw inspiration. This scavenging approach may also raise issues concerning code provenance, which is known to be an issue with conventionally constructed code [54]. However, there is no reason why provenance tracking cannot be built into the GISMOE system: at least it will be more thorough and complete than the records typically maintained by human programmers concerning code provenance.

Existing Oracles: The code evolved using traditional GP is only ever as good as the test cases used to define the fitness function that guides the evolution. This raises the question of how the engineer can be expected to provide a high quality test suite, against which the evolution can be evaluated and guided by the fitness function. SBST can be used to generate test inputs, but a test suite is made up of test cases, each of which is a pair containing the test input and its corresponding expected output.

Constructing good quality test cases represents a significant problem in its own right. It involves knowing what the expected output should be for any given input; the so–called oracle problem [13, 89]. In traditional software development of a new system there is no oracle. Fortunately, however, there exists a wealth of well–tested code that could be used as a specification of the functional properties of a desired system.

Each such program p may satisfy the functional objectives, though it may not satisfy the multiple emerging additional properties, many of which have a non–functional character. We can therefore use automated test data generation to create a large pool of inputs for which the programming problem is well-covered structurally and from which the corresponding output can be obtained simply by running p.

We can use the original program as the oracle, thereby producing as many test cases as are needed in order to satisfy 'correctness up to a certain level of testing'. As part of this overall research agenda, one might also explore the extent to which it is possible to use other techniques to ensure that the evolved code is faithful to the original, such as model checking and assertion checking.

Observe that this approach is at least as stringent as the test–based assurances offered by most existing, entirely human–based industrial development processes. This leaves open the question: 'but will the software developer accept evolved code?'; maybe the evolved code will not be as easy to understand as that generated by humans[2]? This is the question to which we now turn.

Human In the Loop: Even with automated documentation of the generated code, coupled with extensive and thorough testing, the software engineer may be reluctant to leave the code development entirely to an automated GISMOE tool. This is not unreasonable.

[2]This may not be the barrier one might suppose: recent work on automated documentation of evolved code suggests that evolved code can be documented to the point at which it becomes competitive, even for human understandability, when compared to human constructed patches [31].

While we may do all we can to capture all of the functional and non-functional requirements the human has in mind, the engineer is unlikely to be able to precisely specify all of the ways in which they would seek to constrain the evolution; they may not even be aware of their implicit coding style requirements nor their cognitive needs.

Human programmers are notoriously resistant to the task of reading other humans' code. Such code is foreign and treated with skepticism and mistrust. Will they not treat GISMOE generated code with at least this degree of hostility? Such code may not only seem foreign, it may even be regarded as downright *alien*.

This is why we propose a 'human in the loop' approach to GISMOE. There is a wealth of research on interactive evolution [32] (and its application to SBSE [38, 41, 83]) from which we can draw inspiration. If the human forms part of the fitness computation for (samples of) the GISMOE candidate solutions, this will ensure that human judgements can be incorporated into the solutions proposed on the Pareto program surface. Human judgement may also be essential for some of the non-functional properties (such as code readability and maintainability). Nevertheless, the development of this 'human in the loop GISMOE' remains an open problem (discussed in more detail in Section 6.4).

Even with the human in the loop and with extensive testing prior to deployment, the software engineer may remain nervous. Early adopters of GISMOE prototypes will surely want some assurance that the GP-evolved solutions they select from the Pareto program surface will not cause embarrassment; no one ever got fired for using human programmers. Fortunately, the GISMOE approach has a built-in 'try before you buy' option:

Try Before You Buy: A human developer may naturally feel more comfortable managing their own code than any code created using an evolutionary algorithm. Fortunately, when the developer is unsure of the evolved code there is a simple solution. That is, if the developer is unsure of the evolved code then there can be a trial period during which both the evolved code unit and the original are run in parallel.

In this way, the developer is never required to allow the evolved code to take over until sufficient confidence and trust is attained. Furthermore, any deviance during this trial period can simply serve as a stimulus for further evolution and perfection of the evolved code.

Since the entire process can be automated, the human need never devote precious time to consideration of any of the evolved code unless and until sufficient benefit has accrued. The benefits of using the evolved code can be automatically assessed as the evolution progresses through the repeated computation of fitness. Therefore, the software designer can set a threshold above which the solutions found become interesting, and simply continue to use the 'dual code' version until such a threshold is reached.

Insight: The GISMOE approach does not *require* the software designer to accept any of the evolved solutions in order to be useful. The software designer can also use GISMOE to explore the multi-objective candidate solution space, gaining insight into what can be achieved by balancing several competing constraints. In this way the technique can be thought of as a source of stimuli for 'eureka moments'.

It may be that GISMOE has found an unexpected innovation that can effect a dramatic improvement in the trade off between objectives. Such behaviour is very common for evolutionary algorithms, because they simply optimise for the objectives they have, unconstrained by assumptions and biases that may lead a human towards certain solutions.

The software designer can identify particularly interesting programs on the Pareto program surface. The GP engine can then generate many new programs close to the ones the designer prefers and can pool these to automatically identify those aspects that the programs share. The common code in all such solutions may be relatively small and thereby reveal the insightful 'coding trick' used to achieve the interesting results observed. Even when the common code is large, it can be winnowed by further considering the results of the sensitivity analysis; code can be presented to the human in a prioritised order, depending on the effect it has on the non-functional properties of interest.

Component by Component: One does not need to attempt to evolve a whole system at once. Components of the system can be identified and re-evolved one at a time. Using sensitivity analysis at a coarse level of granularity, we can identify components that would most affect the non-functional properties of interest and focus on these first. In this way, we can gradually evolve the overall system, component-by-component, focussing first on replacing those components most critical to the non-functional properties of interest. Over time, the deployed system will gradually include more and more evolved code, though some original human code may always remain. This is another way in which GISMOE might achieve 'scalability by stealth'.

4. PROPOSED GISMOE ARCHITECTURE

The proposed GISMOE architecture is depicted in Figure 2. Dotted lines indicate choices of components. Solid lines indicate data flows. The 'fixed software' is the code that is not changed by GP. The choice of platform and fixed software constitute the environment in which the non-functional properties are evaluated. Together, these non-functional evaluators constitute the collections of multiple objective fitness values for which GISMOE seeks Pareto optimal solutions (thereby creating the Pareto program surface).

The software developer can choose to combine arbitrary sets of non-functional-property fitness functions with different environments to make a harness that provides fitness data as an input to the GP engine and sensitivity analysis. Test cases can come from any test data generation technique, for which there are many options (already available) that can be used as a component to GISMOE.

Sensitivity information can be pre-computed before the GP improvement process commences. The sensitivity of the program to each non-functional property is computed using the non-functional evaluation harness. This process requires no knowledge of the functional test cases, since it seeks to identify those parts of the program that are non-functionally sensitive, irrespective of functional properties.

The result of the sensitivity analysis is a prioritised list of program elements for each non-functional property of interest. This is used by the GP engine to prioritise for evolution those parts of the program that are more sensitive to the non-functional property of interest.

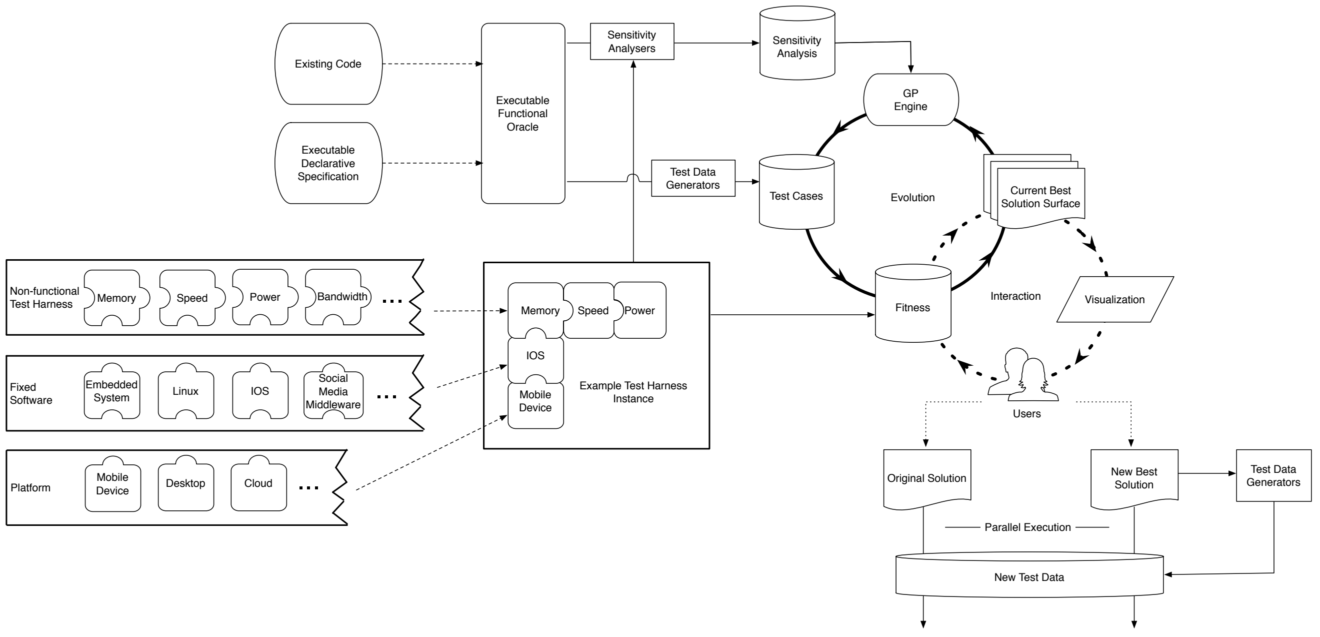


Figure 2: The GISMOE Architecture

If the program becomes very different to the original, then the sensitivity analysis may be recomputed. In the co-evolutionary GISMOE model, test case generation and GP evolution proceed in tandem.

The description of the functional properties comes from either an initial version of the program (code reuse) or from a newly written declarative specification of functional behaviour, which also serves as an initial version (albeit, one that will be constructed with no attempt to optimise for non-functional properties).

The definition of a suitable declarative language for expressing functional oracles to guide the GISMOE evolution, remains an open problem discussed in more detail in Section 6.2. This description of the functional properties (however obtained) is used to provide the oracle for testing, so that the test inputs from the tests input generation can be computed with expected outputs to form test cases.

The GP engine uses the test cases and the non-functional harness to compute its fitness. In the conformant version of GISMOE, the only solutions considered are those that pass all test cases (these programs are functionally conformant and the approach is conformant to the view that 'correctness is paramount').

4.1 The Heretical Version of GISMOE

In the heretical version of GISMOE, the human is permitted to see versions of the program that do *not* pass all test cases. In the heretical world view, functional correctness is merely another dimension on which to plot programs on the Pareto surface, thereby allowing the software designer to explore the ways in which the functional and non-functional properties might be traded off against each other.

Perhaps future generations of software engineers will wonder what all the fuss was about: why would our standpoint on functional correctness be considered so very *heretical*? How could functional correctness ever have trumped all other system properties?

Anyone who believes that functional correctness should be paramount (surpassing all non-functional properties in importance) should try to execute proven functionally correct software on a device with a flat battery. We may thus sum up our heresy more succinctly as follows:

Availability of sufficient resources for computation is the ultimate correctness concern.

Previous work has already been undertaken to demonstrate the feasibility of optimising programs for a combination of functional and non-functional properties, in which the functional properties of the system are traded for improvements in power consumption [92] and speed [82].

However, this work was, arguably, *not* heretical because the functional properties in question were based on continuous quality assessments that can be degraded gracefully (randomness [92] and graphical smoothness [82]). If a human cannot perceive the difference between the quality of each solution, then there may be no 'down side' to the trading off functional properties such as these, against improvements in performance.

The heresy proposed by the 'heretical' version of GISMOE is that one might even countenance the loss of correctness where it is painfully noticeable, in order to achieve better performance for a non-functional objective. Perhaps, in some circumstances, even a program that fails to meet all of the functional test cases (spectacularly crashing without warning) may be acceptable (even desirable) should it perform quickly and with minimal heat dissipation. That is, a user may prefer this 'crasher' to a correct program that never crashes but which is slow and consumes more power.

For example, suppose one is running a cloud data centre in which a crashed computation can be re-started efficiently and effectively and without loss of data. A faster, lower power consumption, version of the computation that crashes a little more often may be attractive and cost-effective in this situation.

Similarly, on a long haul flight from London to New York, a hot laptop that runs out of battery half way through the flight is less usable than one that remains cool and lasts the entire flight. Of course, though one might want heretical GISMOE software running on one's personal computing devices during the flight, one is unlikely to want it controlling the aircraft itself.

4.2 Dynamic Adaptive Search Based Software Engineering

In the bottom right-hand corner of Figure 2 we see the deployment model we envisage for GISMOE's software products. The original and the evolved version selected by the software designer for a chosen environment are executed (for a trial period) 'in parallel' within the selected deployment environment. Testing can thus continue once the code is deployed, using user sessions with the system as a source of test cases.

This is the 'Try Before You Buy' model of adoption that we described in Section 3. Different versions of the system will be deployed in different environments, using the original program as a continuing oracle. The user need not accept an evolved alternative unless and until they are satisfied with the performance and behaviour of the system in its intended environment.

In some situations, where spare power is available, we can also perform further testing, in-situ, using a test data generator. In this way, we can deploy a self-testing version of the evolved software, together with its test harness and test generator.

Of course this will increase deployed code size and so it may not be appropriate for some systems, where compact code was an important non-functional requirement. In such situations a client-server approach may allow the deployed code to retain communication with a server that continues to monitor performance. This would be possible in circumstances where spare communication bandwidth is available.

Suppose that we could embed into the deployed software the entire GISMOE system or to run a monitoring service using some form of client-server approach. If we could find a way to continue to operate the entire GISMOE engine and its supporting systems (measurement and analysis of non-functional sensitivity and the oracle) together with the deployed software system, then we would have found a way to achieve Dynamic Adaptive Search Based Software Engineering [43].

Dynamic adaptivity is also the goal of the 'Dynamic Adaptive Automated Software Engineering' (DAASE) project. This is a large UK Engineering and Physical Sciences (EPSRC) 'programme grant' with a programme of work on dynamic adaptivity running from 2012 to 2018. More details about the project and opportunities for (part-funded) collaboration with the DAASE programme can be found in the ESEM 2012 keynote paper [43].

It may not prove possible to embed into deployed software the entire GISMOE system. We may need to tailor the system to achieve this or to find smart ways to continue an on-going evolutionary improvement process once the GISMOE-evolved software is deployed. Nevertheless, we hope that parts of the approach can be 'compiled into' the deployed code so that the code can adapt to changes in environment that affect non-functional performance.

5. WHY GISMOE IS FEASIBLE

Recent advances in Genetic Programming have indicated that it is able to balance many different competing and potentially conflicting objectives in other engineering paradigms and to scale the applications of GP from small code fragments to the evolution and re-evolution of components of larger systems.

It is increasingly recognised that optimisation in general (and computational search in particular) also has the potential to meet Software Engineering objectives, through the recent rapid growth in work on Search Based Software Engineering [37]. In this section we summarise some of the recent work that, we believe, provides evidence that the GISMOE vision is feasible.

Bug Fixing: Genetic Programming is evolving. Work on GP for automated bug fixing [8] has rapidly developed, with results that demonstrate that it is already possible to find and fix non-trivial bugs [36, 63, 88], generating patches that are reasonably human-readable [31].

For the GISMOE agenda, the important observation from this work is:

> *The original program serves as an ideal oracle for the re-evolution of fragments of new code.*

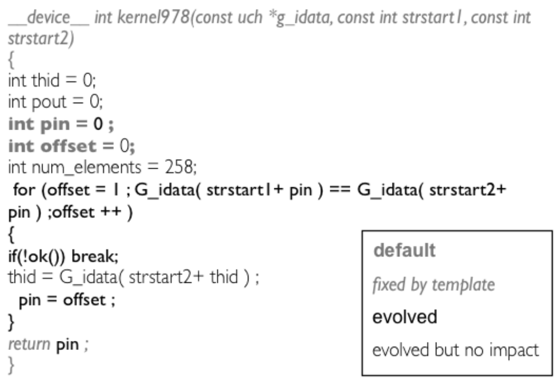
In work on patching, the amount of code generated by GISMOE is very small, but the larger size of the program to which the patches have been applied is encouraging. The work shows the value of seeding the search with fragments of existing code and pinpointing the areas of the existing program that need to be targeted for evolution.

Migration: Recent work on code migration using a subset of the GISMOE approach [60] has demonstrated that non-trivial, real world programs (in this case the UNIX utility `gzip`) can be re-evolved, by focusing on the component that plays a primary role in the computation. The GP engine was able to port this kernel component of `gzip` to an entirely new platform (from desktop to graphics processing card).

The GP engine had available only fragments of CUDA code that were entirely unrelated to compression and the `gzip` original. The original served as the oracle for testing purposes, yet the evolved version was found to be functionally correct (as evaluated by extensive regression testing and detailed human examination of the resulting code). These results are encouraging because they involve the evolutionary generation of larger fragments of code than patches and also migration to unfamiliar and unseen platforms and operating environments. The primary observation of interest for the GISMOE agenda arising from this work is:

> *Code can be re-evolved from one environment to an entirely new environment and programming language.*

However, many more problems remain to be fully solved: the test suites were the result of unautomated human ingenuity, as was the kernel component identification process. We hope that future work on the GISMOE agenda will incorporate state-of-the art test data generation techniques and advances in non-functional sensitivity analysis on which we are presently working.



```
__device__ int kernel978(const uch *g_idata, const int strstart1, const int
strstart2)
{
int thid = 0;
int pout = 0;
int pin = 0 ;
int offset = 0;
int num_elements = 258;
 for (offset = 1 ; G_idata( strstart1+ pin ) == G_idata( strstart2+
pin ) ;offset ++ )
{
if(!ok()) break;
thid = G_idata( strstart2+ thid ) ;
  pin = offset ;
}
return pin ;
}
```

Figure 3: A fragment of evolved gzip code[60].

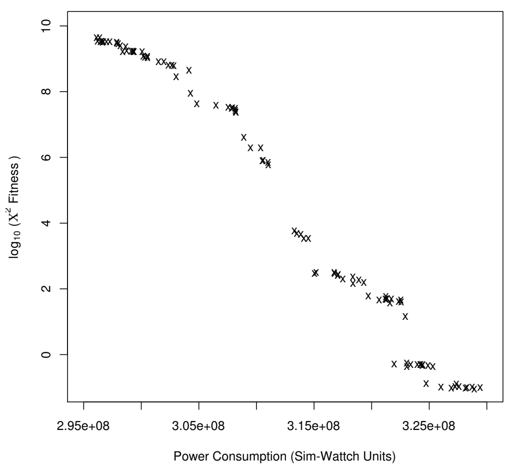


Figure 4: Functionality (vertical axis) vs. power consumption (horizontal axis) [90].

Trading Functional & Non-Functional Requirements: Previous work on searching for alternative balances between functional and non-functional requirements has also been promising. White et al. [92] evolved different versions of a pseudo random number generator with a range of power-consumption characteristics. The important observation of this work for GISMOE is that

Functional properties are 'just another optimisation objective', like non-functional properties.

This previous work was not an instance of 'heretical GISMOE' because the non-functional property in question was the degree to which the outcome of random number generation could be said to be 'random'. Figure 4 shows the results. In this case the resulting set of programs trade-off pseudo random number quality (as measured by the Strict Avalanche Criterion) against the likely power consumption of the implementation (as estimated by the Sim-Wattch simulator). The figure shows the results of the generation of programs for pseudo random number generation, tested on 4,096 test cases for power consumption and randomness.

The results yield a two dimensional Pareto surface (a 'Pareto front') with knee points. The lower on the vertical axis a program is plotted, the more 'random' are the numbers it generates. The further to the right are the programs on the horizontal axis, the more power is consumed when the program executes. The knee points indicate that there are regions of the Pareto program front for which a considerable increase in randomness can be achieved with a modest increase in power. It also reveals other regions where much more power is required for a relatively small improvement in randomness.

Compatible Crossover: Recent experimental results from the FINCH project [71] demonstrated that the use of a 'compatible crossover' could dramatically increase the effectiveness of GP when evolving novel implementations of Java bytecode. The use of smarter typed and context-respecting genetic operators such as this may improve the applicability of code scavenging approaches, such as the 'plastic surgery scavenging' that we propose in this paper. The lesson we learn from this work for GISMOE is that

Type awareness reduces the search space and makes genetic operators more effective.

Software Uniqueness: Recent empirical analysis of a large (420 million lines of code) corpus of code by Gabel and Su [33] reveal that code may not be as unique as one might suppose. The code produced by human programmers does not even *begin* to explore the conceptually infinite (and practically enormous) space of possible programs expressible in a language. Source code is far more constrained and less varied than natural language [51]. For the GISMOE agenda the important finding is that

The space of candidate programs is far smaller than we might suppose.

This observation means that the search space for candidate solutions to programming problems may turn out to be far less challenging than previously thought. Furthermore, the existence of such large (half billion LoC) repositories of available code, in which one has to write approximately 6 statements before one is writing unique code (up to consistent identifier replacement) [33], provides a ready source of seed material for GISMOE's 'plastic surgery' (See Section 3).

Dynamically Discovering Static Truths: Work by Ernst et al. [22, 23] has demonstrated that likely program invariants can be inferred from test cases for programs. Though there remain challenges in finding useful, insightful and compact sets of invariants using this approach to dynamic assertion discovery, the fundamental observation that underpins this work has profound implications for software engineering (and our GISMOE agenda):

A small amount of dynamic information is sufficient to approximate (and sometimes precisely capture) static information.

It is upon this observation of the surprising power of testing that we rest our claim that testing may prove to provide a sufficing guide to the GP engine with respect to functional properties, and ultimately, the correctness of the code evolved by GISMOE.

Multiplicity, Product Lines & N-Versions:

There has been much recent interest in the concept of 'multiplicity computing' [16], in which many versions of a system are deployed. By engineering diversity into different versions of a system we may generate solutions that can defeat some kinds of malware attack [78]. Multiplicity has also been advocated as a means of achieving resilience in the presence of change [16]. Techniques developed for multiplicity computing can also be incorporated into a GISMOE approach: they may provide alternatives to GP as a means of searching the space of candidate solutions.

Diversity and dynamic adaptivity [43] can also be sources of resilience (against change and attack). GISMOE may have a contribution to make to this agenda. That is, by generating a range of solutions, we are automatically working within a 'multiplicity computing' paradigm. We can make implementation diversity one of the non-functional properties for which we optimise, thereby creating a diverse pareto surface of solutions. All those found within acceptable tolerance can be deployed simultaneously, leading to a GISMOE-inspired approach to diversity and multiplicity.

Pioneering early work by Feldt on the problem of using GP to generate multiple instances of a solution demonstrated that GP could be used to generate an automated form of N-version computing [25]. The use of GP in this work overcame the principle obstacle to wider use of N-version computing, by dramatically reducing the cost involved in hiring multiple development teams. GISMOE seeks to take this further: In the case of the Pareto program surface, N can be very large, while the incorporation of different non-functional properties allows the spread of solutions to address multiple conflicting and competing objectives.

There is also increasing interest in software engineering for Software Product Lines (SPLs) [18, 35] for which there are many variants of the systems under development for different platforms and instantiations. The design and management of SPLs is increasingly common in software for consumer products such as automotive and white goods, where all products in a product family share a common core set of program features, but for which each member of the family may exist on a different branch of the product tree. Each branch selects or de-selects features, leading to different implementations at the leaves.

SPLs are also increasingly common because many applications run on multiple platforms. Managing and controlling the plethora of different versions of a system is a challenge. The GISMOE approach may also offer solutions to some of the issues raised by SPLs. For example, using GISMOE, we can create new branches automatically: the GP engine will evolve the new versions of the product family from existing members of the family. We may also be able to merge versions when the product family becomes large or unwieldy.

6. OPEN CHALLENGES

This paper sets out our vision for the GISMOE approach, but there remain several challenges to be overcome in order to see this vision realised. We hope that this paper illustrates the value of the GISMOE research agenda and goes some way towards decomposing the overall grand challenge into more manageable components. Many of these components are current topics for the SBSE, GP, optimisation, and testing communities.

In this section we summarise some of the open challenges for the realisation of the GISMOE grand challenge, highlighting areas of current work that offer promising sources of potential solutions.

6.1 Measuring Non-Functional Properties as Fitness Functions

We have rather glibly dismissed the engineering problems involved in accurately, efficiently and effectively measuring non-functional properties of software in its intended deployment environment. However, many challenges and open problems reside within this part of the GISMOE research agenda.

Some non-functional properties are notoriously hard to measure using current techniques, such as power consumption and heat dissipation. In order to optimise for any of these properties we must be able to measure them. Measurements must be sufficiently precise to inform the sensitivity analysis and the GP evolution engine. These challenges must be addressed, not only because they are a requirement of the GISMOE agenda, but because without effective measurement, these properties cannot be controlled by any approach. Indeed, it has long been a rallying cry of the software metrics community (paraphrasing Kelvin [55]) that

You cannot control what you cannot measure.

This observation is no less true of non-functional properties than it is of any other aspects of the software development process and its products; if a non-functional property is important and we seek to improve performance with respect to this property, we cannot hope to find any engineering solution, should we find ourselves unable to even measure the property of interest.

6.2 Using Declarative Languages to Describe Functional Properties

It was the founding principle of functional programming [27] that one should design programs in a declarative style, concentrating on the correctness of one's program, and leaving the efficiency of implementation to an automated 'improvement' system. This vision of 'correctness first; efficiency second' inspired much early work on automated program transformation systems [20, 72].

For these early pioneers of functional programming, the non-functional property of concern was execution time [20]. Systems were implemented in large stand alone mainframes, and there was little concern for other properties such as power consumption and heat dissipation, which were seen merely as the necessary evils of computation. In fact, the very foundations of algorithmic complexity are based on the belief that the primary computational resources of concern are time and space and all efforts at program improvement have, hitherto, tended to focus on these two aspects.

The GISMOE agenda shares the separation of concerns between functional and non-functional properties of a system. However, we widen the pool of non-functional properties to be considered and we propose the use of computational search (such as GP) to find the efficient solutions we seek, rather than the transformation of the original into a more efficient version (though search-based transformation might also be considered [24, 77] to be an alternative vehicle to achieve GISMOE).

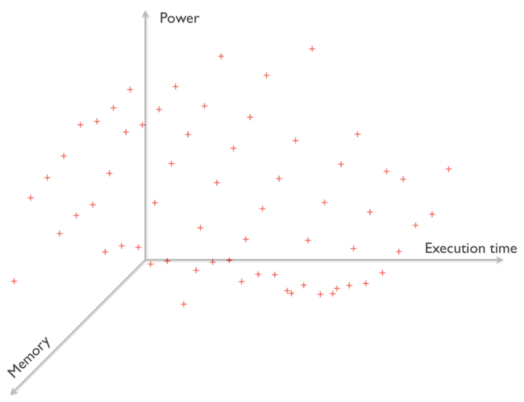


Figure 5: A more realistic Pareto program surface

For entirely new computational problems, where there is no existing code to be used as an oracle, the GISMOE approach will require a description of the functional properties of the system to be evolved on the Pareto program surface. In this situation, a declarative programming notation would seem to present the ideal programming language in which to express the oracle.

One simply wants the functional properties, without consideration of the non-functional properties, which will subsequently be 'optimised in'. This is precisely what functional programming was designed to achieve. Initial results indicate that GP-evolution of code in entirely different languages from those in which the oracle is written is possible [60]. Therefore, we can envisage the specification of functional properties in a functional language, and the evolution of instances on the Pareto program surface in any one of many more imperative languages.

6.3 Visualisation of Complex Pareto surfaces

Another aspect of the GISMOE vision that has been rather glossed over in this paper is the visualisation of the Pareto program surface itself. Where there are only three objectives to be considered, we can suppose that these will be viewable on a three dimensional surface of solutions. However, where the software designer is interested in more than three properties, a higher dimensionality will be required, posing challenges for the visualisation of the Pareto program surface.

Even where the surface is to be rendered in only two or three dimensions, the presence of programs on the surface may be more sparse than we might like, making it hard to identify knee points and the overall shape of the surface itself. Rather than a nice, evenly spread and dense Pareto program surface like that idealised in the illustration in Figure 1, we might find ourselves attempting to visualise and explore a surface more like that depicted in Figure 5.

To address this, we might need interpolation of solutions between points to aid visualisation even where such points do not exist. We might also present non-pareto-optimal points, merely to fill in areas near the surface with candidate non-dominated solutions. Where the pareto program surface consists of more than three dimensions, we may use projection to visualise the trade offs between any three of these, with the engineer considering multiple views.

Other, more advanced visualisation techniques using graphics and virtual reality may also help in this GISMOE visualisation agenda. The problem of visualising multi dimensional space is one that finds application in many fields; GISMOE can exploit any such advances.

6.4 Human in the loop: Interactive Optimisation

We have proposed the use of a 'human in the loop' approach to incorporating aesthetic and other subjective judgements about software design, architecture and coding style into GISMOE. We have experimented with an interactive evolutionary pretty printer [38] to produce a tool that explores the software designer's preferences with respect to program layout [59]. However, this is very early work and does not address the complexities involved in bringing the human designer fully into the evaluation of fitness.

One of the primary challenges of all work on interactive evolution lies in the issue of fatigue; no human designer will want to look at many evolved solutions, though the system may generate billions. This requires smart selection and presentation techniques that can quickly and unobtrusively extract human assessments. One approach to this problem might be to use eye tracking systems to capture the human judgements about code without interfering with the engineers' reading and examination of the solutions offered.

6.5 Test data generation

For each program generated we shall need to identify worst cases, which will involve test data generation for all properties (rather than merely relying on the test cases generated to cover the functional properties with which the process commences). This will pose a challenge for test data generation. We shall need fast construction of test cases for a rapidly changing suite of programs, all of which are related to one another, targeting the worst case for some non-functional property of interest.

Fortunately, search based test data generation can be targeted at non-functional properties (such as temporal properties [74]), and there is also much work on the problem of regression test optimisation [93]. For GISMOE, we shall need to develop techniques that can adapt and develop new test cases on-the-fly as the evolution process takes place. Promising approaches to this problem include co-evolution of test cases and code [1, 7] and test regeneration and augmentation [80, 94].

Our approach puts great stress on the importance of testing. Ultimately, if the GISMOE vision is achieved, then much of what we now regard as programming will be automated, with the result that human attention will move to the earlier stages of the software development process.

In a 'GISMOE software engineering world', the locus of human ingenuity will traverse the interface between requirements and testing. The designer will thereby concentrate on the assurance that software does what it is intended (functionally) and that it does so as best as possible and within acceptable tolerance (for non-functional properties), given the operational environment in which it is deployed.

We shall require better automated testing than we currently have available in order to achieve greater confidence in the code automatically produced by GISMOE. Fortunately, the presence of an automated oracle, will mean that test generation can be entirely automated.

Consequently, there will be orders of magnitude improvement in the testing resources available, at a fraction of the cost of current laborious manual test processes.

We shall also need a tighter coupling between the user requirements (their elicitation, negotiation and change) and the test process (the generation, selection and prioritisation of test cases). Work on the interactions between requirements and testing as the primary vehicle for describing and ensuring software usefulness is a long overdue (and hitherto under-explored) topic. There has been work on the development of approaches to optimise the requirements negotiation and elicitation process [28, 49, 79, 98] and to optimise the selection and prioritisation of testing [93]. We need an optimisation approach that seamlessly encompasses *both* requirements and test optimisation [43].

6.6 Incorporating the Operational Profile

We have focussed on the non-functional aspects of the different environments into which we may deploy GISMOE programs. We might also consider the way in which some operating environments do not exercise the code on all possible inputs. In one environment the operational profile might be very different from another. For instance, two different users of a weather app on a smart phone might be concerned with different locations, seldom if ever considering location inputs outside a narrow range.

We can use profiling to exploit the operational profile of the environment in the evolutionary process [91]. Through profiling we can select or modify the test cases that are used to test both functional and non-functional aspects of the GP-evolved code. In so-doing we shall be creating specialised programs that are optimised for their operating environment (an idea that resembles the early work on partial evaluation [12, 14] as a means of program specialisation).

7. RELATED WORK

The GISMOE approach set out in this paper draws heavily on previous Search Based Software Engineering (SBSE) work, particularly the work on test optimisation and genetic programming for software engineering. SBSE consists of the reformulation of software engineering problems to make them amenable to the application of computational search (as well as optimisation techniques more usually associated with Operations Research) [17, 45]. SBSE has grown rapidly in the past ten years [30], and has found many applications including Requirements [98]; Predictive Modelling [2, 41]; Non-Functional Properties [3]; Program Comprehension [38]; Design [76] and Testing [3, 5, 39, 65].

There has been industrial uptake of SBSE techniques at Berner and Mattner and Daimler [56, 87], Cisco [50], Ericsson [4, 97], Google [96] Microsoft [15, 58], Motorola [11] and NASA [19]. There are also a number of SBSE tools, many of which are publicly available, supporting SBSE applications right across the range of software engineering activities from release planning [69], through design [66] to testing [6, 29, 53, 56, 84] refactoring [67] and patching [63].

Surveys and tutorials on SBSE can be found elsewhere in the literature [2, 3, 37, 42, 46, 48, 65, 76]. The GISMOE approach essentially seeks to extend previous work on genetic programming for software engineering.

Decomposition has proved to be an important problem for work on GP [52, 86]. The GISMOE approach also seeks scalability through building block decomposition.

It uses building blocks at fine granularity (through plastic surgery) and also at the coarse granularity (through hybrid componentisation).

Various approaches have also been proposed to constrain the recomposition of GP fragments [9, 61, 70, 71, 68]. The GISMOE approach also seeks to constrain the way in which GP is applied to evolve new programs for non-functional properties through sensitivity analysis.

The GISMOE approach is also an inherently explorative and multi-objective GP approach. Feldt [26] was an early advocate of an explorative approach to GP. Although there has been some previous work on multi-objective GP ([61, 92]) most multi-objective optimisation research has concentrated upon other forms of evolutionary algorithm.

Scalability of GP approaches has also been addressed using novel parallel hardware platforms. For example, Poli [75] distributes the GP population across multiple computers (demes), while Langdon exploited the parallelism of GPGPU hardware [62]. Yoo et al. [95] were the first to use GPGPU for an SBSE problem, but their approach used a genetic algorithm, not genetic programming. The GISMOE approach may also benefit from parallel execution to help scale the multiple fitness and test evaluations required by the approach.

8. CONCLUSION

As non-functional requirements become ever more pressing, numerous and their interactions more complex, we simply cannot expect human ingenuity to cope; some form of automated exploration of programming space will be required. A step change in what we expect from a software development environment will be needed. The achievement of a development environment capable of producing a Pareto program surface addresses this problem. It is a grand challenge for the software engineering community.

We do not underestimate the magnitude of this challenge, but the potential benefits make it a worthy goal. When it is accomplished, it may radically transform our view of programming and, indeed, that entity we currently call ‘a programmer’.

Before 1950 the word ‘computer’ was a term used to describe a human who performed repeated (often tedious and error-prone) computations by hand and brain. How strange and dated this interpretation of the word ‘computer’ now seems. How much has human development been accelerated by the advantages of automated computation? Speed, precision, reliability and scale have combined to change the way our world works.

Will future generations look back on our use of the term ‘programmer’ with the same, perhaps nostalgic, but undoubtedly deprecating and even condescending eyes? How quaint it will seem, in an age of automated programming, to think of ‘programmers’ as rows of humans, packed into felt-covered boxes, working on such tedious, repetitive and error-prone tasks. How costly, inefficient and wasteful of human ingenuity that we would task some of our smartest individuals with the construction of perhaps one or two programs on a Pareto surface, when an automated programmer can, with a fraction of the time and cost, produce a multitude of solutions all of which outperform the human-generated solutions.

Acknowledgements: Thanks to the many people whose ideas and conversations have influenced and inspired the GISMOE vision set out in this paper, including Enrique Alba, Peter Bentley, Lionel Briand, Cristian Cadar, Prem Devanbu, Robert Feldt, Stephanie Forrest, Carlo Ghezzi, Mike Holcombe, Kiran Lakhotia, Emmanuel Letier, Claire Le Goues, Phil McMinn, Tim Menzies, Peter Pietzuch, Riccardo Poli, Zhendong Su, Joe Sventek, Paolo Tonella, Shin Yoo, Wes Weimer, Alex Wolf, Andreas Zeller & Yuanyuan Zhang. Many apologies to any of those whom we neglected to mention.

9. REFERENCES

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. Mutation testing using genetic algorithms: A co-evolution approach. In *Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103*, pages 1338–1349, Seattle, Washington, USA, June 2004. Springer.

[2] W. Afzal and R. Torkar. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems Applications*, 38(9):11984–11997, 2011.

[3] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[4] W. Afzal, R. Torkar, R. Feldt, and G. Wikstrand. Search-based prediction of fault-slip-through in large software projects. In *Second International Symposium on Search Based Software Engineering (SSBSE 2010)*, pages 79–88, Benevento, Italy, 7-9 Sept. 2010.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering*, pages 742–762, 2010.

[6] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3 – 12, Lawrence, Kansas, USA, 6th - 10th November 2011.

[7] A. Arcuri, D. R. White, J. A. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In 7^{th} *International Conference on Simulated Evolution and Learning (SEAL 2008)*, pages 61–70, Melbourne, Australia, December 2008. Springer.

[8] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*, pages 162–168, Hongkong, China, 1-6 June 2008. IEEE Computer Society.

[9] A. Arcuri and X. Yao. Co-evolutionary automatic programming for software development. *Information Sciences*, 2010. To appear. Available on line from Elsevier.

[10] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In 26^{th} *IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 53 – 62, Lawrence, Kansas, USA, 6th - 10th November 2011.

[11] P. Baker, M. Harman, K. Steinhöfel, and A. Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In 22^{nd} *International Conference on Software Maintenance (ICSM 06)*, pages 176–185, Philadelphia, Pennsylvania, USA, Sept. 2006.

[12] L. Beckman, A. Haraldson, O. Oskarsson, , and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4):319–357, 1976.

[13] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[14] D. Bjørner, A. P. Ershov, and N. D. Jones. *Partial evaluation and mixed computation*. North–Holland, 1987.

[15] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In 33^{rd} *International Conference on Software Engineering (ICSE'11)*, pages 1066–1071, New York, NY, USA, 2011. ACM.

[16] C. Cadar, P. Pietzuch, and A. L. Wolf. Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In G.-C. Roman and K. J. Sullivan, editors, *Workshop on Future of Software Engineering Research (FoSER 2010)*, pages 81–86. ACM, 2010.

[17] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

[18] P. C. Clements. Managing variability for software product lines: Working with variability mechanisms. In 10^{th} *International Conference on Software Product Lines (SPLC 2006)*, pages 207–208, Baltimore, Maryland, USA, 2006. IEEE Computer Society.

[19] S. L. Cornford, M. S. Feather, J. R. Dunphy, J. Salcedo, and T. Menzies. Optimizing Spacecraft Design - Optimization Engine Development: Progress and Plans. In *Proceedings of the IEEE Aerospace Conference*, pages 3681–3690, Big Sky, Montana, March 2003.

[20] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[21] P. T. Devanbu. GENOA: A customizable language- and front-end independent code analyzer. In T. Montgomery, L. A. Clarke, and C. Ghezzi, editors, 14^{th} *International Conference on Software Engineering (ICSE '92)*, pages 307–317, Melbourne, Australia, May 1992. ACM Press.

[22] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD Thesis, University of Washington, 2000.

[23] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[24] D. Fatiregun, M. Harman, and R. Hierons. Evolving transformation sequences using genetic algorithms. In 4^{th} *International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–74, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[25] R. Feldt. Generating diverse software versions with genetic programming: and experimental study. *IEE Proceedings - Software*, 145(6):228–312, 1998.

[26] R. Feldt. Genetic programming as an explorative tool in early software development phases. In *1st International Workshop on Soft Computing Applied to Software Engineering*, pages 11–20, University of Limerick, Ireland, 12-14 Apr. 1999. Limerick University Press.

[27] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, 1988.

[28] A. Finkelstein, M. Harman, A. Mansouri, J. Ren, and Y. Zhang. A search based approach to fairness analysis in requirements assignments to aid negotiation, mediation and decision making. *Requirements Engineering*, 14(4):231–245, 2009.

[29] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011.

[30] F. G. Freitas and J. T. Souza. Ten years of search based

software engineering: A bibliometric analysis. In 3rd *International Symposium on Search based Software Engineering (SSBSE 2011)*, pages 18–32, 10th - 12th September 2011.

[31] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA'12)*, Minneapolis, Minnesota, USA, July 2012. To appear.

[32] P. Funes, E. Bonabeau, J. Herve, and Y. Morieux. Interactive multi-participant task allocation. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 1699–1705, Portland, Oregon, 20-23 June 2004. IEEE Press.

[33] M. Gabel and Z. Su. A study of the uniqueness of source code. In 18th *ACM SIGSOFT international symposium on foundations of software engineering (FSE 2010)*, pages 147–156, Santa Fe, New Mexico, USA, 7-11 Nov. 2010. ACM.

[34] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[35] M. Goedicke, C. Köllmann, and U. Zdun. Designing runtime variation points in product line architectures: three cases. *Science of Computer Programming*, 53(3):353 – 380, 2004.

[36] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012.

[37] M. Harman. The current state and future of search based software engineering. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, pages 342–357, Los Alamitos, California, USA, 2007. IEEE Computer Society Press.

[38] M. Harman. Search based software engineering for program comprehension. In 15th *International Conference on Program Comprehension (ICPC 07)*, pages 3–13, Banff, Canada, 2007. IEEE Computer Society Press.

[39] M. Harman. Open problems in testability transformation. In *1st International Workshop on Search Based Testing (SBT 2008)*, Lillehammer, Norway, 2008.

[40] M. Harman. Automated patching techniques: The fix is in: technical perspective. *Communications of the ACM*, 53(5):108, 2010.

[41] M. Harman. The relationship between search based software engineering and predictive modeling. In 6th *International Conference on Predictive Models in Software Engineering (PROMISE 2010)*, Timisoara, Romania, 2010.

[42] M. Harman. Software engineering meets evolutionary computation. *IEEE Computer*, 44(10):31–39, Oct. 2011.

[43] M. Harman, E. Burke, J. A. Clark, and X. Yao. Dynamic adaptive search based software engineering. In 6th *IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, Lund, Sweden, 2012.

[44] M. Harman, Y. Jia, and B. Langdon. Strong higher order mutation-based test data generation. In 8th *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 212–222, New York, NY, USA, September 5th - 9th 2011. ACM.

[45] M. Harman and B. F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, Dec. 2001.

[46] M. Harman, A. Mansouri, and Y. Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 2012. To appear.

[47] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[48] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[49] W. Heaven and E. Letier. Simulating and optimising design decisions in quantitative goal models. In 19th *IEEE InternationalRequirements Engineering Conference (RE 2011)*, pages 79–88. IEEE, 2011.

[50] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Ttransactions on Software Engineering and Methodology (TOSEM)*. To appear.

[51] A. Hindle, E. Barr, Z. Su, P. Devanbu, and M. Gabel. On the naturalness of software. In *International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, 2012.

[52] D. Jackson. Self-adaptive focusing of evolutionary effort in hierarchical genetic programming. In A. Tyrrell, editor, *2009 IEEE Congress on Evolutionary Computation*, pages 1821–1828, Trondheim, Norway, 18-21 May 2009. IEEE Computational Intelligence Society, IEEE Press.

[53] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In 3rd *Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, August 2008.

[54] J. Krinke, N. Gold, Y. Jia, and D. Binkley. Cloning and copying between GNOME projects. In J. Whitehead and T. Zimmermann, editors, 7th *International Working Conference on Mining Software Repositories (MSR 2010)*, pages 98–101. IEEE, 2010.

[55] L. M. Laird and M. C. Brennan. Software measurement and estimation: a practical approach, 2006.

[56] K. Lakhotia, M. Harman, and H. Gross. AUSTIN: An open source tool for search based software testing of C programs. *Journal of Information and Software Technology*. To appear.

[57] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In 4th *Testing Academia and Industry Conference — Practice And Research Techniques (TAIC PART'09)*, pages 95–104, Windsor, UK, 4th–6th September 2009.

[58] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy — Search-based floating point constraint solving for symbolic execution. In 22nd *IFIP International Conference on Testing Software and Systems (ICTSS 2010)*, pages 142–157, Natal, Brazil, November 2010. LNCS Volume 6435.

[59] W. B. Langdon. Evo_indent interactive evolution of GNU indent options. In F. Rothlauf, editor, *Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 2081–2084, Montreal, Québec, Canada, 2009. ACM.

[60] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[61] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, 2010.

[62] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, Oct. 2008. Special Issue on Distributed Bioinspired Algorithms.

[63] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[64] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In 10th *ACM SIGSOFT symposium on foundations of software engineering (FSE '02)*, pages 119–128, 2002.

[65] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[66] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[67] I. H. Moghadam and Mel Ó Cinnéide. Code-Imp: A tool for automated search-based refactoring. In *Proceeding of the 4th workshop on Refactoring Tools (WRT '11)*, pages 41–44, Honolulu, HI, USA, 2011.

[68] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[69] A. Ngo-The and G. Ruhe. A systematic approach for solving the wicked problem of software release planning. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):95–108, August 2008.

[70] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[71] M. Orlov and M. Sipper. Flight of the FINCH through the java wilderness. *IEEE Transactions Evolutionary Computation*, 15(2):166–182, 2011.

[72] H. A. Partsch. *The Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer, 1990.

[73] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[74] H. Pohlheim and J. Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCDO '99)*, volume 2, page 1795, San Francisco, CA 94104, USA, 13-17 July 1999. Morgan Kaufmann.

[75] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 1(1/2):37–56, Apr. 2000.

[76] O. Räihä. A survey on search–based software design. *Computer Science Review*, 4(4):203–249, 2010.

[77] C. Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming.* Kluwer Academic Publishers, 1 Nov. 1999.

[78] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In 4*th EuroSys Conference (EuroSys'09)*, pages 33–46, Nuremberg, Germany, Apr. 2009. ACM.

[79] M. O. Saliu and G. Ruhe. Bi-objective release planning for evolving software systems. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the* 6th *joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) 2007*, pages 105–114. ACM, Sept. 2007.

[80] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In 23rd *Automated Software Engineering (ASE '08)*, pages 218–227, L'Aquila, Italy, 2008. IEEE.

[81] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In M. Wermelinger and H. Gall, editors, 10th *European Software Engineering Conference and 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE '05)*, pages 263–272. ACM, 2005.

[82] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In T. Gyimóthy and A. Zeller, editors, 19th *ACM Symposium on the Foundations of Software Engineering (FSE-19)*, pages 124–134, Szeged, Hungary, Sept. 2011. ACM.

[83] C. L. Simons, I. C. Parmee, and R. Gwynllyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering*, 36(6):798–816, 2010.

[84] P. Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 119–128, Boston, Massachusetts, USA, 11-14 July 2004. ACM.

[85] A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[86] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.

[87] J. Wegener and O. Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, pages 1400–1412, Seattle, Washington, USA, June 2004. LNCS 3103.

[88] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE 2009)*, pages 364–374, Vancouver, Canada, 2009.

[89] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, Nov. 1982.

[90] D. R. White. *Genetic Programming for Low-Resource Systems.* PhD Thesis, Dapartment of Computer Science, University of York, UK, 2010.

[91] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[92] D. R. White, J. Clark, J. Jacob, and S. Poulding. Searching for resource-efficient programs: Low-power pseudorandom number generators. In *2008 Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1775–1782, Atlanta, USA, July 2008. ACM Press.

[93] S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[94] S. Yoo and M. Harman. Test data regeneration: Generating new test data from existing test data. *Journal of Software Testing, Verification and Reliability*, 2012. To appear.

[95] S. Yoo, M. Harman, and S. Ur. Highly scalable multi-objective test suite minimisation using graphics cards. In 3rd *International Symposium on Search based Software Engineering (SSBSE 2011)*, pages 219–236, 10th - 12th September 2011. LNCS Volume 6956.

[96] S. Yoo, R. Nilsson, and M. Harman. Faster fault finding at Google using multi objective regression test optimisation. In 8th *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, Szeged, Hungary, September 5th - 9th 2011. Industry Track.

[97] Y. Zhang, E. Alba, J. J. Durillo, S. Eldh, and M. Harman. Today/future importance analysis. In *ACM Genetic and Evolutionary Computation COnference (GECCO 2010)*, pages 1357–1364, Portland Oregon, USA, 7th–11th July 2010.

[98] Y. Zhang, A. Finkelstein, and M. Harman. Search based requirements optimisation: Existing work and challenges. In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'08)*, volume 5025, pages 88–94, Montpellier, France, 2008. Springer LNCS.