Genetic Programming for Reverse Engineering

Mark Harman*, William B. Langdon* and Westley Weimer†
*University College London, CREST centre, UK
†University of Virginia, Virginia, USA

***Abstract*—This paper overviews the application of Search Based Software Engineering (SBSE) to reverse engineering with a particular emphasis on the growing importance of recent developments in genetic programming and genetic improvement for reverse engineering. This includes work on SBSE for re-modularisation, refactoring, regression testing, syntax-preserving slicing and dependence analysis, concept assignment and feature location, bug fixing, and code migration. We also explore the possibilities for new directions in research using GP and GI for partial evaluation, amorphous slicing, and product lines, with a particular focus on code transplantation. This paper accompanies the keynote given by Mark Harman at the 20^{th} Working Conference on Reverse Engineering (WCRE 2013).**

I. Introduction

The term 'search based software engineering' was introduced in 2001 [42] to capture the (then emerging) interest in the use of computational search as a means of optimising software engineering problems. The motivation was that search based optimisation was ideal for the multiple conflicting and competing objectives with which software engineers routinely contend. The algorithms used to conduct search based optimisation are also known to perform well in the presence of partial, noisy and missing data. This makes them attractive tools with which to approach the complex, noisy and incomplete world in which the software engineer has to find engineering solutions.

Since 2001, SBSE has been applied to almost every aspect of software engineering activity. A more detailed survey of the entire field of SBSE can be found elsewhere [44], while a tutorial introduction is also available [46] that assumes no prior knowledge of computational search techniques. However in this paper, we focus on reverse engineering and the considerable potential for the development of new forms of Genetic Programming (GP) and Genetic Improvement (GI) to reverse engineering. Section II presents a summary of the application of SBSE to reverse engineering. Section III briefly reviews the relationship between the SBSE and RE publication venues and trends. Section IV sets out some directions for future work that form part of a 'GP4RE' research agenda; genetic programming applications for reverse engineering.

II. Applications of SBSE to Reverse Engineering

A large number of problems in reverse engineering are amenable to SBSE. In this section we highlight some of the many existing approaches, focusing on the application of SSBSE to the problems of re-modularisation, refactoring, regression testing, slicing, and concept assignment in particular.

We also discuss the general applicability of genetic programming to reverse engineering problems. Our overall conclusion is that these are vibrant and active areas of research with multiple open problems remaining to be tackled.

A. Re-Modularisation

Software structure degrades making periodic re-modularisation important [8]. The search based modularisation approaches discussed in this section assume that we start with some form of module dependence graph, from which we seek to construct suitable module boundaries.

Mancoridis *et al.* were the first to address the problem of software modularisation using SBSE leading to the development of a tool called Bunch [69] for module clustering. They experimented with several search based algorithms including genetic algorithms, hill climbing and simulated annealing. Hill climbing tended to produce the best results for the single objective of improving the Module Quality (MQ) metric.

MQ was introduced in 1998 [65], and refined in subsequent papers. All versions of MQ are combinations of cohesion and coupling into a single weighted fitness function, used to guide the search. Other authors explored other ways to capture cohesion and coupling in different metrics [47]. The choice of the cohesion and coupling metric has a critical impact on the results obtained.

Search based clustering has also been applied to package coupling [1], to reduce overall package size [17] and to explore the relationship between design and code level software structure [49]. The same overall search based clustering approach can also be adapted for clustering heap allocation [22] and for reducing memory fragmentation [23].

There were many attempts to find algorithms that produced better modularisation results (in terms of cohesion, coupling and faithfulness to some 'gold standard' modularisation). Many of the earlier attempts to improve on hill climbing, simply provided further evidence to suggest that it was a simple, fast and effective algorithm for module clustering [39], [64], [68].

In 2003, Mahdavi *et al.* [64] introduced a multiple hill climbing approach, which performs repeated hill climbing and combines the best results found to reduce the search space size for subsequent hill climbs. Mahdavi *et al.* also used parallelisation to improve the performance of the multiple clustering, something first explored by Mitchell *et al.* [70] who used parallel computing to distribute the task of computing module clusters (but did not combine results to reduce the search space).

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany
Invited Paper

This multiple hill climbing approach can improve the performance of simpler approaches. Performance is improved both through parallelisation and through the identification of high-quality building block sub-solutions that become fixed in subsequent searches to reduce search space size.

More recently, in 2011 Praditwong *et al.* introduced a multi-objective approach to module clustering [77]. The multi objective approach was the first to find module clustering solutions that improved significantly on the hill climbing algorithm, in terms of cohesion and coupling. The approach also found results that improved on the single objective, MQ, thereby demonstrating that multi objective techniques can find better single objective solutions than single objective approaches. This is a phenomenon observed elsewhere in the optimisation literature [71]; by supporting diversity and maintaining a sub-population of candidate solutions that meet sub-objectives, a multiple objective approach can beat a single objective approach 'at its own game'.

Results by Praditwong *et al.* [77] yielded good quality, but they came at a high computational price; the multi objective search required two orders of magnitude more time than the humble hill climber. Fortunately, recent results by Barros [7] provide evidence that multi objective approaches may also be capable of outperforming single objective approaches in terms of both quality and computation time.

There has thus been a significant amount of SBSE work tackling the problem of modularisation, and many problems remain open. We draw particular attention to, and encourage future work in, multi objective module clustering, since it also allows the developer to optimise other objectives than just cohesion and coupling.

Such additional objectives might include closeness to original module structure, business goals, technical constraints, testability, and other metrics that may be important in finding a good module structure. As well as providing candidate module structures, the multi objective approach, as with all multiple objective SBSE [82], allows the engineer to explore the trade offs and tensions between the objectives, something often advocated as an advantage of SBSE [42], [34].

B. Refactoring

In this subsection we highlight some SBSE approaches to optimisation and refactoring.

Early work on search based techniques for program transformation tended to focus on improving program execution time (through parallelisation) [72], [81], [91] and size [24]. These early search based approaches sought to go beyond the peep-hole optimisations available to compilers. Search based transformation seeks higher level transformations of source code that improve space and time performance.

More recently, authors have focused on the potential of search based approaches to suggest sequences of refactoring steps that might be applied to a system in order to automate or partially automate the refactoring process. There are both single [18], [74], [83] and multiple objective [48] approaches to search based refactoring.

Most of this work has concerned the code level (source-to-source) transformation problem, but there is also work on search based transformation at the model level [30]. Search based refactoring has tended to focus on imperative languages. There is work on refactoring for declarative languages [58], [63], making search based refactoring for declarative languages an interesting topic for future work. Search based refactoring has also been suggested as a means of improving testability [37] through testability transformations [35].

Search based refactoring is a widely-studied automated refactoring approach. It can also be used to experimentally validate metrics [73] by searching for agreement and disagreement between metrics on which refactorings should be performed. In this work, the refactorings themselves are unimportant; what is important is the behaviour of the refactoring process (guided by the metrics), which is used to asses the quality of guidance offered by each metric.

We encourage more cross-fertilisation between refactoring approaches (improving non-functional properties related to maintenance and evolvability) and the genetic improvement approaches (which can be thought of as improving non-functional properties such as power consumption or memory traffic) discussed in Section II-F4.

C. Regression Testing

In this subsection we briefly mention SBSE approaches to regression testing. In practice, when applied to realistic programs, this becomes an optimisation problem:

> *What is the test set (or order) that will maximise the positive aspects of regression testing, while minimising the negative aspects, subject to constraints?*

Positive aspects include coverage of the system and early fault discovery, while the negative aspects include cost drivers to be minimised, such as execution time and oracle cost [45]. The constraints come from technical concerns (for example, one test must be executed first since it creates a resource consumed by subsequent test) or business concerns (for example, we prioritise tests that exercise business-critical features).

Not surprisingly, there has been a great deal of work on regression test optimisation in the SBSE literature. Elsewhere, there is a comprehensive survey on regression test optimisation [92] and a shorter 'manifesto' for the specific case of the multi-objective regression test optimisation paradigm [36]. These topics, particularly the multi objective incarnations and the regression test prioritisation formulation, remain highly active.

D. Slicing and Dependence Analysis

In this subsection we relate SBSE and program slicing. Program slicing and dependence analysis has been widely studied in the reverse engineering literature, because slices have applications on comprehension [15], re-use [9], specification mining [4], understanding bug reports and tool traces [50], code salvaging [21] and testing [38], [67]. Dependence analysis also can be used to identify anti-patterns or code smells that may be the trigger for reverse engineering interventions [13], [14].

There has been work on the construction of program slices using search based approaches to slice construction [25]. There has also been work on identifying dependence structures using slicing as an input to a search for sets of slices that exhibit properties that may be of interest to the reverse engineer [51]. More generally, there has been work on search based program transformation and refactoring (mentioned in Section II-B). We see a need for more work of search algorithms to find interesting dependence structures, where 'interesting' can be captured in a measurable fitness function.

E. Concept Assignment and Feature Location

Concept assignment [10] and feature location [90] have found widespread application in the reverse engineering literature. We argue that there should be more work applying SBSE to concept assignment. The problem of locating the code corresponding to a concept or feature in a legacy system is a natural starting point for comprehension and, therefore, plays a critical role in many reverse engineering problems.

The underlying challenge is to find a good fit, where 'goodness of fit' is a direct target for formulation as a fitness function. Despite the inherent optimisation flavour of the problem, the authors are aware of only one approach to search based concept or feature location [31]. In this work, goodness of fit is formulated as a measurement of signal-to-noise ratio: how much of the reported code corresponds to the features to be located, and how much is merely 'noise'? We believe there is room for additional search based approaches to be profitably applied to the problem.

F. Applications of GP to Reverse Engineering

The topics in search based reverse engineering briefly outlined so far have used a variety of search based algorithms, (most notably genetic algorithms, but also hill climbing and simulated annealing). In this subsection we briefly review topics in reverse engineering for which *genetic programming* has proved to be particularly applicable, to set the scene for Section IV, which presents our GP4RE agenda focusing on novel topics in genetic programming for reverse engineering.

1) Bug Fixing: Work on GP for automated bug fixing [6], [32], [62] has demonstrated that some bugs in existing programs can be automatically patched so that the regression test suite passes. This may be useful to 'buy time' for the developers who could use an automated patch to keep a system live while they investigate longer term solutions. Ultimately, it is also possible that classes of bugs might be automatically patched in ways that replace the need for human bug fixing altogether. Of particular note to a reverse engineering community, the maintainability of GP-produced fixes has been evaluated in a human study [28]. A recent survey of work in this area can be found elsewhere [61].

2) Code Migration: Work on code migration [59] was used to automatically port the core algorithm of a UNIX utility (`gzip`) from standard programming environment written in the C language to the CUDA language on a general purpose graphics processing card (GPGPU) environment.

This work suggest the possibility of using GP more widely, to assist with the difficult and long-standing challenge of reverse engineering a system from one platform, language and/or environment to another.

3) Code Composition, Reuse and Model Extraction: Genetic programming has been used to construct behavioural models, allowing the reverse engineer to predict the response time of an assembly of re-used components in a novel architecture into which they are to be deployed. Krogmann *et al.* [55] evaluated this approach to GP for reverse engineering behaviour predictive models by predicting the performance of a (partly reverse-engineered) file sharing application. Their model monitors data and runtime bytecode counts and uses static bytecode analysis to combine a behavioural model with platform-specific benchmarking. Fredericks and Cheng [27] also recently used GP for module reuse.

More generally, we believe that many existing model extraction approaches could benefit from search based techniques to reduce manual overhead. As an early example, while using semantics-preserving operations to extract from a program an equivalent high-level specification, Ward observed [88]:

> *This process can never be completely automated — there are many ways of writing the specification of a program, several of which may be useful for different purposes. So the tool must work interactively with the tedious checking and manipulation carried out automatically, while the maintainer provides high-level "guidance" to the transformation process.*

We believe a GP search could allow such guidance to be codified in a fitness function, automatically favouring orderings of transformations that yield high-fitness specifications.

4) Genetic Improvement and Program Synthesis: Genetic improvement (also called Evolutionary improvement) [5], [43], [75], [76], [89]) is a form of program synthesis in which an existing program is improved, using GP. Much of this work has focussed on the improvement of the system's execution time, though it can also be applied to other non-functional properties of the system [43]. Since genetic programming tends to produce an optimised 'program' of some form, GI has also tended to focus on improving source code. However, it has also been applied at the architectural level too [79]. GI has even been applied to shader programs in the domain of computer graphics, producing tradeoffs between execution time and visual fidelity [85] via transformations that are not semantics-preserving (e.g., [80]).

Recently, Langdon and Harman reported that GI found improved versions with speed-ups of up 70x for a 50 KLoC C/C++ system [60]. Even more attractive was that this improvement could be achieved with relatively few interventions in the code. These interventions are not necessarily semantics-preserving, so they cannot be found using traditional compiler optimisation (which must be semantics-preserving by construction). However, although not semantics-preserving by construction, Langdon and Harman were able to demonstrate reasonable, even improved, faithfulness to required behaviour.

III. SBSE and the Working Conference on Reverse Engineering (WCRE)

According to a recent survey [44], by 2010, SBSE papers had appeared in 201 different publication venues. Ranking these 201 venues by number of SBSE papers, the International Conference on Software Maintenance (ICSM) was 9th (with 9 papers) and the Conference on Software Maintenance and Reengineering (CSMR) was 14th (with 5). The Working Conference on Reverse Engineering (WCRE) is not ranked in the top 20. However, there have been three papers on SBSE in WCRE [1], [25], [52]. Not surprisingly the single venue in which the largest number of SBSE papers have appeared is the Genetic and Evolutionary Computation Conference (GECCO), which has featured a dedicated SBSE track since 2001.

However, other well-known software engineering venues such as ASE, ICSE, ISSTA, IST, JSS, TSE and STVR all appear in the top twenty by number of publications on SBSE. Some of these papers are also on topics that concern the application of SBSE to reverse engineering. Given the number of reverse engineering applications for SBSE, it is surprising that there are not more SBSE papers appearing in WCRE.

Hopefully, this keynote paper will encourage authors to consider WCRE as a potential venue for SBSE work on reverse engineering. We also hope that it may stimulate reverse engineers to consider SBSE techniques.

IV. GP4RE: New Genetic Programming Applications for Reverse Engineering

This section proposes new research directions into Genetic Programming (GP) for Reverse Engineering (RE): GP4RE.

A. GP for Partial Evaluation

In partial evaluation [53] a program is partially evaluated by applying it to only some of the full range of its inputs (the inputs are partially applied). The program is evaluated with respect to be available inputs to produce not an output, but a new program. This new program is specialised to those inputs; the inputs become hardwired into the source code as a result of this partial evaluation. The hardwiring process consists of transforming the source code to optimise performance with respect to the known inputs. Typical transformations involve evaluation of expressions, constant folding and loop unrolling. As well as improving performance, partial evaluation may have other applications in reverse engineering such as program comprehension [16].

By fixing the values of some inputs, GI can be used to search for a genetically improved version of the original program that is specialised to these inputs. This can be achieved using any existing approach to GI, simply by fixing some of the input values so that the genetically improved candidates are only ever executed with these specific choices. Overfitting becomes a distinct advantage in this case; it will be one mechanism by which GI can partially evaluate the code to be improved, fitting it to the specific cases denoted by the set of inputs to which the program is partially applied.

Existing partial evaluation approaches often rely on particular backtracking search strategies (e.g., [2]), to which GP provides an alternate. Similarly, in the presence of an adequate test suite, GI holds out the promise of reducing or eliminating the manual annotation burden of many popular partial evaluation approaches (e.g., DyC [33]).

Generalised partial evaluation [29] (which partially evaluates with respect to conditions on the input) could be achieved simply be enclosing the code to be evolved in a harness that checks the conditions as pre-conditions. However, there would, of course, have to be sufficiently many test cases that meet the required pre-conditions. If there were not, then search based test data generation tools [3], [26], [41], [57] can be used to enhance the test suite.

B. GP for Slicing

There are known relationships between partial evaluation and program slicing [11]. For example, a partially evaluated program can be formulated as an amorphous conditioned slice with respect to the 'all variables at all points' slicing criterion that requires complete faithfulness to the original program's semantics. Given these relationships and the discussion in Section IV-A, it should come as no surprise that there is additional scope for GI to implement various forms of slicing (over-and-above the limited initial work overviewed in Section II-D).

Using these relationships, we can extend the approach to GI for partial evaluation (Section IV-A) to GI for amorphous conditioned slicing, by restricting the test suite to those test cases that capture the slicing criterion. This would be a form of observation-based slicing [12]. It would be amorphous slicing because GP will not necessarily simply delete statements, but may add new code. It would be conditioned slicing if generalised partial evaluation were used (and more like dynamic slicing otherwise). It would be observation-based because the preservation of behaviour at the slicing criterion is tested through observation (not built-in by construction).

C. GP for Transplants

We believe that there is considerable potential for search based techniques to be used to achieve software transplantation. In this section we introduce the concept of search based translation, setting out some initial definitions and possible approaches. Much more work is required to develop the idea of software transplantation, but we believe that this could prove to be a valuable application of SBSE in the future.

Suppose we could automatically incorporate a desired feature from one program into another program. We currently perform such 'reuse' entirely manually. We propose that GP be used to re-implement the new feature in a more automated fashion. Such an approach may yield a transplant that is syntactically re-constructed (through search) to ensure that the transplantation is successful. We could even transplant from one language and platform into another using this approach. We know that this is theoretically possible because of previous work on code migration (see Section II-F2).

Definition 1 (Transplant): We informally define a *software transplant* as the adaptation of one system's behaviour or structure to incorporate a subset of the behaviour or structure of another.

We could view a transplant as a more sophisticated (and ideally much more benevolent) form of the kind of 'software injection' that occurs when a virus copies itself into a host. Had the same human effort and ingenuity that has been targeted on virus design have been re-directed towards investigation of software transplantation, we might be currently experiencing far better software systems.

For example, suppose System D, a potential 'donor' of a transplant, includes a hotkey combination feature that exits the program without saving any state. The hotkey feature is a behaviour of System D that we might like to transplant into some other System H, the 'host' of the transplant. Conceptually, it should be fairly straightforward to transplant the hotkey feature into any such System H that also handles keyboard input. We suggest that a GP system should be able to perform such transplants *almost* entirely automatically.

At a high level, we identify key steps likely to occur in a software transplant algorithm to add feature F from source System D (the donor) to destination System H (the host):

1) **Localise**: Identify and localise the code $D_F \subseteq D$ that implements F (this might use, for example, concept and feature location (see Section II-E))
2) **Abstract**: Construct an abstraction $\mathcal{A}_F$ from D_F, retaining control and data flow directly related to F in the donor but abstracting references to D-specific identifiers so that these become parameterised.
3) **Target**: Find locations H_F in the host, H, where code implementing F could be located.
4) **Interface**: Construct an interface, $\mathcal{I}$ and add it to the host, H, allowing the resulting combination $H \cup \mathcal{I}$ to act as a 'harness' into which candidate transplants can be inserted and evaluated.
5) **Insert**: Instantiate and concretise a candidate transplant $\mathcal{A}'_F$ (concretised from $\mathcal{A}_F$) at H_F.
6) **Validate**: Validate the resulting transplanted system $H \cup \mathcal{I} \cup \mathcal{A}'_F$.
7) **Repeat**: Repeat the above steps until a suitably well tolerated transplant is found.

We call this seven-step approach the LATIIV^R approach[1]: Localise, Abstract, Target, Interface, Instantiate and Verify.

In the remainder of this subsection we discuss these seven steps in more detail, highlight existing GP, RE or SBSE techniques that might be applicable.

1) Localise: Localise the Feature in the Donor System: Ideally, the user of a transplant system would be able to identify the interesting behaviour in the donor via the system's functionality and not by code inspection. As a first step, we propose a combination of test cases and manual annotation to identify the feature F.

[1]Pronounced 'Lativar'.

For example, the user might specify a regression suite T testing non-F behaviour and additional tests T_F testing F behaviour. In such a scenario, mutations to the donor that cause T_F to fail while allowing T to pass are associated with the location of the feature F. Additionally, the user might explicitly annotate particular lines of code, variable definitions or uses, or dependencies as part of (or not part of) F.

These localisation actions would require a trained software engineer. However, for some (relatively simple) transplants, it may be possible for an untrained user to identify desired behaviour in the donor that they would like to see transplanted into the host simply through the user interface.

For example, if the donor has a feature that renders the screen purple in response to a GUI button, then this 'purplisation' feature might be identified by the user and transplanted into the host simply by extending the events to which the host can respond and incorporating the purplisation action code.

Given some identification of the feature of interest, existing fault localisation [78], slicing [40], [56], [93] and feature extraction (Section II-E) approaches could be used to find the code $D_F \subseteq D$ that implements F.

2) Abstract: Abstract the Feature: The source code D_F implementing feature F in System D is almost certain to contain explicit references to variables, functions and code specific to the donor system. For example, consider the case where F is a simple local null check:

$$\texttt{if (ptr != NULL) \{ foo(*ptr); \}}$$

The host system H is likely to use a different identifier and perform different behaviour in the non-null case. Indeed, System H might be written in a different language in the most extreme case. We thus desire an abstraction $\mathcal{A}_F$, such as:

$$\texttt{if (}\square_1 \texttt{ != NULL) \{ }\square_2(\square_1)\texttt{; \}}$$

The syntax suggests a *transplantation template* where $\square_1$ is an unspecified local pointer variable and $\square_2$ is a use of it. Similar abstractions are already learned automatically in other domains (e.g., documentation synthesis [20]) and such templates have already been successfully applied to automated bug fixing [54].

3) Target: Find Candidate Destinations in the Target System: In theory this step could be completely automatic, with a search based algorithm trying out many possible locations H_F in the target System H to host the transplant. In practice, feasibility will likely require a combination of test cases and manual annotations for System H, akin to those used to localise the feature in the source system.

As a simple example, consider the case of transplanting additional GUI- or keyboard-handling behaviour. Ideally, this step would be as simple as locating the switch statement that dispatches code based upon GUI or keyboard events and adding an additional case branch. This localisation can additionally take advantage of the structure of the abstract feature template $\mathcal{A}_F$. For instance, in the local null check example above the template should favour destinations where a local pointer is in scope.

4) Interface: Construct an Interface Between Transplant and Host: In the case where donor and host systems are written in different language we may need a parameter passing mechanism (perhaps through abstracted variable instances) so that we can evaluate candidate re-implementations of the feature against the original (which would act as oracle). A similar approach has previously been used in search based code migration (Section II-F2).

However, such interface construction will also be useful in cases where the same platform, environment and programming language dialect are used for the host into which the transplant is located and the donor from which it is taken. It will assist with testing and on-going maintenance.

The process of defining the interface may be automated, given that the localisation and abstraction steps have produced an abstracted feature and the target step has identified a location in which to insert the transplant into the host. Some interfaces may be trivial. However, for maintainability and ease of testing, it will probably make sense to capture the transplant as a function or procedure, with a defined interface (through formal and actual parameters). This will assist in testing, for example, using unit testing approaches to test the transplant (with the original feature playing the role of oracle).

5) Insert: Instantiate and Concretise the Transplant: Given a destination location H_F and the abstract template for the feature $\mathcal{A}_F$, we want to apply the transplantation template at that destination. While our examples have been phrased in terms of code additions, interesting transplants are likely to involve deletions and replacements as well. For example, the null check template above might be phrased as a replacement:

$$\square_2(\square_1) \quad \Longrightarrow \quad \texttt{if } (\square_1 \texttt{ != NULL}) \texttt{ \{ } \square_2(\square_1)\texttt{; \}}$$

This phrasing suggests that an unguarded use of a pointer should be replaced by a null-checked version of the same use. However, since such templates likely contain 'holes' or other abstract references, we must concretize them, choosing functions and identifiers specific to the host System H. For example, one concretization of the previous replacement template might be:

$$\texttt{host\_action(*host\_ptr);} \quad \Longrightarrow$$
$$\texttt{if (host\_ptr != NULL) \{ host\_action(*host\_ptr); \}}$$

Once this template is concretized, it can then be instantiated at any occurrence of `host_action(*host_ptr);` at H_F, replacing that use with the guarded null check. Thus each template will likely have multiple concretizations, and each such concretization can likely be instantiated at (applied to) multiple points in H_F, phrasing both such choices as search problems.

The instantiation of such templates has been studied in the bug fixing context [54] and has a direct formulation as a search problem considering possible mappings from template holes to local instantiations. In practice, instantiations are likely to require assignments, casts, coercions and other 'glue code' to form an interface between the behaviour $\mathcal{A}_F$ from the donor and the context H_F of the host.

For example, a template involving the Eclipse IDE may require an ASTNode object where only an IFile object is in scope. Ideally, this glue code will be capture by the interface. Algorithms such at Prospector [66] solve exactly this issue, synthesising such glue code from input and output types. Similarly, 'programming by sketching' or 'storyboard' approaches [84] synthesise working algorithms in the presence given structured but incomplete information about the contents. Finally, if user annotations take the form of contracts or specifications, recent advances in automated synthesis admitting the creation of non-trivial algorithms from scratch (e.g., Bresenham's, Dijkstra's, etc.) [86] could be applied.

6) Validation: Validating the Resulting Transplantation: In a search based setting, many candidate transplants may be considered before an acceptable one is found. We highlight a number of dimensions along which the quality of a transplant might be evaluated during the validation stage.

1) **Passes new feature tests**: The transplant implements the new desired behaviour correctly. Ideally, all tests of the host (System H) for the new feature F should pass after the transplant.
2) **Passes regression tests**: The transplant does not disrupt existing behaviour by introducing side effects. There are two cases of side effects:
   a) **The truth, the whole truth**: The transplant retains required existing behaviour of H. Ideally, all tests (and invariants, contracts, etc.) associated with System H (the host) that do not directly conflict with feature F from the donor (System D) should pass after transplantation. Standard regression testing might be used to determine if the transplant sacrifices existing behaviour [92].
   b) **... and noting but the truth**: The transplant does not introduce new undesired behaviour. This might be measured in terms of anomaly detection or fuzz testing [62]. Automated test data generation [3], [26], [41], [57] could be used to augment the host's existing test suite for higher coverage of the changed area H_F.
3) **Passes quality tests**: The transplant is readable, maintainable and/or acceptable. If the post-transplant system is to be maintained by humans, metrics or user studies of readability [19], maintainability [28] or acceptability [54] should be used.

These criteria are similar to the use of test cases in automated bug fixing [6], [32], [62]. Indeed, automated bug fixing is a special case of transplantation and transplantation as a special case of program improvement, which is, in turn a special case of program synthesis.

That is, in traditional search based program repair, the donor and host are the same system. We seek a transplant that replaces the buggy code with code that implements the new feature of 'correct behaviour'. In this sense, repairs are special kinds of transplants. All transplantation seeks to improve an existing program.

In this sense, transplantation is a special case of program improvement: it seeks to improve the behaviour of an existing program. Finally, since program synthesis is concerned with automated code generation, program improvement can be regarded as a special case of synthesis. The synthesised code is not written from scratch, but augments some existing program or system.

7) Discussion: While not every approach to software transplanting need necessarily follow the LATIIV^R process outlined above, we believe it highlights the challenges that will be encountered. Localisation, abstraction, targeting, interfacing, inserting and validation are all likely to be critical to any such approach. We also believe that an SBSE formulation is also very natural: given a fixed set of abstract templates and targets from the localisation, search through that space until a concretization and instantiation are validated.

We note that not all software transplantation tasks are of equal complexity. Since the most direct and well-studied approaches to localisation and validation are based on testing, we propose that researchers begin with independent implementations of the same API or specification, thus allowing the donor and host to share tests. In this formulation, transplanting is akin to automated bug-fixing [6], [32], [62] with the bug being that host fails the tests related to feature F and the possible fixes all coming from the source code of the donor.

Independent implementations also make attractive testbeds. For example, it is easy to imagine experiments to transplant encryption code from one ZIP compression library to another (or one audio or video codec library, etc.). Similarly, researchers might make use of the multiple similar-but-not-identical implementations of the various core Unix utilities (e.g., Ultrix vs. Solaris vs. GNU) or the various secure socket libraries (e.g., OpenSSL vs. GnuTLS vs. NSS vs. CyaSSL, etc.), not all of which support the same subset of protocols.

Some code will be simply untransplantable, because the two systems are just too different. For example, we cannot meaningfully transplant a payroll update procedure into a disk controller; they are simply incompatible *species* of software. Defining the systems that can be mutual hosts and donors might be an intellectual mechanism for defining *species of software systems*. Such a definition of software species may be a valuable contribution in itself (and one with surprising findings). This alone might justify the effort to explore research opportunities in software transplantation.

It seems clear that some of the socket libraries mentioned above will be similar enough to admit transplants; it is equally easy to imagine that some code will be simply *untransplantable* into some hosts. For example, we cannot meaningfully transplant a payroll procedure into a disk drive controller; they are incompatible species of software.

Even intra-species transplants might be rejected, perform poorly or cause side effects. A transplant is *rejected* if the resulting host-plus-transplant simply fails to compile. The transplant *performs poorly* if it fails some of the tests which check the new desired functionality. It *causes side effects* if some regression test fails.

As with animal transplantation, a rejected transplant is useless, but poorly performing and side-effecting transplants may be acceptable. For example, a transplant may add sufficient value to the host to be retained until something better can be found. If it causes side effects that can be tolerated or ameliorated then it may also be retained. For example, a transplant that causes a slight degradation in performance might be tolerated, while one that adds 'junk' to a log file could be ameliorated by incorporating a filter.

D. GP for Software Product Lines

There has recent been work on regression test optimisation for Software Product Lines (SPLs) [87] using SBSE techniques related to those overviewed in Section II-C, but GP has not been used for the generation and merging of SPL branches.

The idea of GP for SPLs was proposed in the 'GISMOE Challenge' keynote paper at ASE 2012 [43], but has yet to be fully explored. The idea is to use GI to create new branches automatically and to merge versions when the product family becomes large or unwieldy.

A new branch can be thought of as a transplant in the sense defined in Section IV-C above. If a donor system can be found then the SPL branch can be extended using a transplanting approach. However, even where there is no donor, an extension could be provided by GI, so long as there are sufficient test cases available that capture the additional desired functionality.

For small extensions, it would make sense to put effort into defining test cases, rather than into hand-writing code that might otherwise be found by a GP. After all, a test suite will be required to check that the branch is correct, so why not define it up front?

A merge of several branches can be performed by seeking to locate a parameterisation of the branches that allows a single fragment of code to implement all branches. GI can be used to find the smallest set of parameters that allows the smallest set of code to be used to suffice for all branches; a natural multi objective formulation.

Observe that there always exists a solution with maximal code size and small parameter set size: it is simply the switch of all branches, parameterised by a single enumeration-type parameter that selects between them. Of course, this would be a trivial solution that adds no value. The goal of GI SPL automated branch merging is to improve on this.

E. GP for Reverse Engineering Strategies

One overall reverse engineering scenario could be characterised as a three step process: decompose, evolve, re-compose. That is, a system is first decomposed into models. There is some existing work on refactoring and extraction of models mentioned in Sections II-B and II-F3 that could assist here.

The components extracted through decomposition can then be re-evolved into improved versions using the GI approach outlined in Section II-F4. Finally, there is the remaining problem of how to re-compose the system into a new composition of (improved) modules. This reassembly may be done by GP.

Currently, research on reverse engineering proceeds in problem-specific silos that target parts of each of these phases. In the RE literature, we can find work on extraction of reusable components and work on techniques to help engineers improve these components. We also find (once again, separate) work on reuse and integration of components. What we need is a coherent overall process that combines all of these phases — as well as suitable benchmarks for evaluating them in tandem — since each is clearly dependent upon and affects the others.

Since all phases of the overall RE process could potentially be partly automated as optimisation problems for SBSE, there is also the eventual opportunity to combine all three phases into an over-arching strategy. Using GP to evolve effective strategies for RE, we could envisage a future for RE in which the whole reverse engineering process is treated as a single optimisation problem, for which we seek an optimal overall strategy. The phases of this process are also sub-problems, each of which can be formulated as an optimisation problem.

V. Conclusion

In this paper we have briefly described the application of Search Based Software Engineering to reverse engineering. The literature shows a recent upsurge in papers on Genetic Programming (GP) for reverse engineering, with exciting results on automated software repair, migration and improvement using GP. We believe that there are many more equally fruitful applications of GP in problems of slicing, transformation and partial evaluation.

We also outlined, in more detail, the use of GP to achieve automated and semi-automated software transplants and advocate its use in software product line extension and branch merging. We believe GP may even provide a means to search for overall reverse engineering strategies. Much of this research agenda (a 'GP4RE research agenda') remains to be explored. We are very interested to collaborate with other researchers and practitioners on these and related topics.

Acknowledgements: Thanks to the many people whose work has influenced and inspired the ideas for GP4RE set out in this paper, including Enrique Alba, Andrea Arcuri, Peter Bentley, Lionel Briand, Edmund Burke, Betty Cheng, John Clark, Prem Devanbu, Robert Feldt, Stephanie Forrest, Carlo Ghezzi, Sumit Gulwani, Mike Holcombe, Yue Jia, Kiran Lakhotia, Emmanuel Letier, Claire Le Goues, Phil McMinn, Tim Menzies, Gabriela Ochoa, Justyna Petke, Riccardo Poli, Ralf Reussner, Martin Rinard, Moshe Sipper, Zhendong Su, Paolo Tonella, David White, John Woodward, Xin Yao, Shin Yoo, Yuanyuan Zhang & Andreas Zeller. Many apologies to any of those whom we neglected to mention.

Langdon and Harman are supported by EPSRC Grant EP/I033688 (GISMO: Genetic Improvement of Software for Multiple Objectives). Harman is additionally supported by EPSRC Grants EP/J017515 and EP/I010165 and a platform grant (EP/G060525) for the CREST centre at UCL. Weimer is partly supported by NSF grants CCF 0954024 CCF 0905373 and DARPA grant P-1070-113237.

References

[1] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In 16^{th} *Working Conference on Reverse Engineering (WCRE '09)*, pages 103–112, Lille, France, 13-16 October 2009. IEEE.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. *ACM Trans. Program. Lang. Syst.*, 28(4):696–714, 2006.

[3] Nadia Alshahwan and Mark 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.

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *The* 29^{th} *SIGPLAN-SIGACT Sxymposium on Principles of Programming Languages (POPL 2002)*, pages 4–16, Portland, OR, USA, January 2002.

[5] Andrea Arcuri, David Robert White, John A. Clark, and Xin 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.

[6] Andrea Arcuri and Xin 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.

[7] Márcio de Oliveira Barros. An analysis of the effects of composite objectives in multiobjective software module clustering. In *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO '12)*, pages 1205–1212, Philadelphia, USA, 7-11 July 2012. ACM.

[8] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Software re-modularization based on structural and semantic metrics. In 17^{th} *Working Conference on Reverse Engineering, WCRE 2010*, pages 195–204. IEEE Computer Society, 2010.

[9] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518, Los Alamitos, California, USA, 1993. IEEE Computer Society Press.

[10] T. J. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, California, USA, May 1993. IEEE Computer Society Press.

[11] David Binkley, Sebastian Danicic, Mark Harman, John Howroyd, and Lahcen Ouarbya. A formal relationship between program slicing and partial evaluation. *Formal Aspects of Computing*, 18(2):103–119, 2006.

[12] David Binkley, Nicolas Gold, Mark Harman, Jens Krinke, and Shin Yoo. Observation-based slicing. Technical Report RN/13/13, Computer Sciences Department, University College London (UCL), UK, June 20th 2013.

[13] David Binkley, Nicolas Gold, Mark Harman, Zheng Li, Kiarash Mahdavi, and Joachim Wegener. Dependence anti patterns. In 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability (Evol'08)*, pages 25–34, L'Aquila, Italy, September 2008.

[14] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance*, pages 177–186, Los Alamitos, California, USA, 2005. IEEE Computer Society Press.

[15] David Binkley, Mark Harman, L. Ross Raszewski, and Christopher Smith. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension*, pages 161–170, Los Alamitos, California, USA, June 2000. IEEE Computer Society Press.

[16] Sandrine Blazy and Philie Facon. Partial evaluation for program comprehension. *ACM Computing Surveys*, 30(3), September 1998. Article 17.

[17] Thierry Bodhuin, Massimiliano Di Penta, and Luigi Troiano. A Search-based Approach for Dynamically Re-packaging Downloadable Applications. In *Proceedings of the 2007 Conference of the IBM Center for Advanced Studies on Collaborative Research (CASCON '07)*, pages 27–41, Richmond Hill, Ontario, Canada, 22-25 October 2007. ACM.

[18] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A Novel Approach to Optimize Clone Refactoring Activity. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM.

[19] Raymond P. L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558, 2010.

[20] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *34^{th} International Conference on Software Engineering (ICSE 2012)*, pages 782–792, Zurich, Switzerland, June 2012.

[21] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance*, pages 424–433, Los Alamitos, California, USA, September 1994. IEEE Computer Society Press.

[22] Myra Cohen, Shiu Beng Kooi, and Witawas Srisa-an. Clustering the Heap in Multi-Threaded Applications for Improved Garbage Collection. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1901–1908, Seattle, Washington, USA, 8-12 July 2006. ACM.

[23] Christian Del Rosso. Reducing Internal Fragmentation in Segregated Free Lists using Genetic Algorithms. In *Proceedings of the 2006 International Workshop on Interdisciplinary Software Engineering Research (WISER '06)*, pages 57–60, Shanghai, China, 20 May 2006. ACM.

[24] Deji Fatiregun, Mark Harman, and Rob 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, September 2004. IEEE Computer Society Press.

[25] Deji Fatiregun, Mark Harman, and Rob Hierons. Search-based amorphous slicing. In *12^{th} International Working Conference on Reverse Engineering (WCRE 05)*, pages 3–12, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, November 2005.

[26] Gordon Fraser and Andrea 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.

[27] Erik M. Fredericks and Betty H.C. Cheng. Exploring automated software composition with genetic programming. In *Proceeding of The 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '13)*, pages 1733–1734, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[28] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis, (ISSTA 2012)*, pages 177–187, Minneapolis, MN, USA, July 2012.

[29] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In D. Bjrner, Andrei P. Ershov, and Neil D. Jones, editors, *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 133–151. North–Holland, 1987.

[30] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. Model refactoring using interactive genetic algorithm. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, volume 8084, pages 96–110, St. Petersburg, Russia, 24-26 August 2013. Springer.

[31] Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. A search based approach to overlapping concept boundaries. In *22^{nd} International Conference on Software Maintenance (ICSM 06)*, pages 310–319, Philadelphia, Pennsylvania, USA, September 2006.

[32] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley 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.

[33] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.

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

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

[36] Mark Harman. Making the case for MORTO: Multi objective regression test optimization. In *1^{st} International Workshop on Regression Testing (Regression 2011)*, Berlin, Germany, March 2011.

[37] Mark Harman. Refactoring as testability transformation. In *Refactoring and Testing Workshop (RefTest 2011)*, Berlin, Germany, March 2011.

[38] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[39] Mark Harman, Robert Hierons, and Mark Proctor. A New Representation and Crossover Operator for Search-based Optimization of Software Modularization. In *Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02)*, pages 1351–1358, New York, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[40] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[41] Mark Harman, Yue Jia, and Bill Langdon. Strong higher order mutation-based test data generation. In *8^{th} 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.

[42] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[43] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *27^{th} IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, September 2012.

[44] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):Article 11, November 2012.

[45] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[46] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In Bertrand Meyer and Martin Nordio, editors, *Empirical software engineering and verification: LASER 2009-2010*, pages 1–59. Springer, 2012. LNCS 7007.

[47] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1029–1036, Washington DC, USA, June 2005. Association for Computer Machinery.

[48] Mark Harman and Laurence Tratt. Pareto optimal search-based refactoring at the design level. In *GECCO 2007: Proceedings of the 9^{th} annual conference on Genetic and evolutionary computation*, pages 1106 – 1113, London, UK, July 2007. ACM Press.

[49] Sunny Huynh and Yuanfang Cai. An Evolutionary Approach to Software Modularity Analysis. In *Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)*, pages 1–6, Minneapolis, USA, 20-26 May 2007. ACM.

[50] Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (POPL 2005)*, pages 38–47, Chicago, IL, USA, June 2005.

[51] Tao Jiang, Nicolas Gold, Mark Harman, and Zheng Li. Locating dependence structures using search based slicing. *Information and Software Technology*, 50(12):1189–1209, 2008.

[52] Tao Jiang, Mark Harman, and Youssef Hassoun. Analysis of procedure splitability. In *15^{th} Working Conference on Reverse Engineering (WCRE'08)*, pages 247–256, Antwerp, Belgium, October 2008.

[53] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, England, 1993.

[54] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *35^{th} International Conference on Software Engineering (ICSE '13)*, pages 802–811, San Francisco, CA, USA, May 2013.

[55] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using genetic search for reverse engineering of parametric behaviour models for performance prediction. *IEEE Transactions on Software Engineering*, 36(6):865–877, November-December 2010.

[56] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7:49–76, 2002.

[57] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An open source tool for search based software testing of C programs. *Journal of Information and Software Technology*, 55(1):112–125, January 2013.

[58] Ralf Lämmel. Towards generic refactoring. In *ACM SIGPLAN workshop on rule-based programming (RULE'02)*, pages 15–28, 2002.

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

[60] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 2013. To appear.

[61] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[62] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[63] Huiqing Li and Simon J. Thompson. Comparative study of refactoring haskell and erlang programs. In 6^{th} *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE Computer Society, 2006.

[64] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *IEEE International Conference on Software Maintenance*, pages 315–324, Los Alamitos, California, USA, September 2003. IEEE Computer Society Press.

[65] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, pages 45–52, Ischia, Italy, 24-26 June 1998. IEEE Computer Society Press.

[66] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 48–61, Chicago, IL, USA, June 2005.

[67] Phil McMinn, Mark Harman, Youssef Hassoun, Kiran Lakhotia, and Joachim Wegener. Input domain reduction through irrelevant variable removal and its effect on local, global and hybrid search-based structural test data generation. *IEEE Transactions on Software Engineering*, 38(2):453 – 477, March&April 2012.

[68] Brian S. Mitchell and Spiros Mancoridis. Using Heuristic Search Techniques to Extract Design Abstractions from Source Code. In *Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02)*, pages 1375–1382, New York, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[69] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems using the Bunch Tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.

[70] Brian S. Mitchell, Martin Traverso, and Spiros Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *IEEE/IFIP Proceedings of the Working Conference on Software Architecture (WICSA '01)*, pages 181–190, Amsterdam, Netherlands, 2001. IEEE Computer Society.

[71] Frank Neumann and Ingo Wegener. Minimum spanning trees made easier via multi-objective optimization. *Natural Computing*, 5(3):305–319, 2006.

[72] Andy Nisbet. GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In Peter M. A. Sloot, Marian Bubak, and Louis O. Hertzberger, editors, *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN '98)*, volume 1401 of *Lecture Notes In Computer Science*, pages 987–989, Amsterdam, The Netherlands, 21-23 April 1998. Springer.

[73] Mel Ó Cinnédie, Laurie Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In 6^{th} *IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, pages 49–58, Lund, Sweden, September 2012.

[74] Mark O'Keeffe and Mel Ó Cinnédie. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, March 2006.

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

[76] Justyna Petke, William B. Langdon, and Mark Harman. Applying genetic improvement to minisat. In *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE '13)*, volume 8084, pages 257–262, St. Petersburg, Russia, 24-26 August 2013. Springer.

[77] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, 2011.

[78] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis (ISSTA '13)*, pages 191–201, Lugano, Switzerland, july 2013.

[79] Outi Räihä, Kai Koskimies, and Erkki Mäkinen. Multi-objective genetic synthesis of software architecture. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '11)*, pages 249–250, Dublin, Ireland, 12-16 July 2011. ACM.

[80] Martin C. Rinard. Living in the comfort zone. In *Proceedings of the 22^{nd} Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2007)*, pages 611–622, Montreal, Quebec, Canada, October 2007.

[81] Conor Ryan. *Automatic Re-Engineering of Software using Genetic Programming*, volume 2. Kluwer Academic Publishers, 2000.

[82] Abdel Salam Sayyad and Hany Ammar. Pareto-optimal search-based software engineering (POSBSE): A literature survey. In 2^{nd} *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2013)*, San Francisco, CA, USA, 2013.

[83] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based Improvement of Subsystem Decompositions. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*, pages 1045–1051, Washington, D.C., USA, 25-29 June 2005. ACM.

[84] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *19^{th} ACM SIGSOFT Symposium on the Foundations of Software Engineering European Software Engineering Conference (ESEC/ESEC '13)*, pages 289–299, Szeged, Hungary, September 2011.

[85] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Trans. Graph.*, 30(6):152, 2011.

[86] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 313–326, January 2010.

[87] Shuai Wang, Shaukat Ali, and Arnaud Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Proceeding of The 15th Annual Conference on Genetic and Evolutionary Computation (GECCO '13)*, pages 1493–1500, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[88] Martin P. Ward. Reverse engineering through formal transformation: Knuths 'polynomial addition' algorithm. *Comput. J.*, 37(9):795–813, 1994.

[89] David Robert White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[90] N. Wilde, J.A. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 200–205, Los Alamitos, California, USA, 1992. IEEE Computer Society Press.

[91] Kenneth Peter Williams. *Evolutionary Algorithms for Automatic Parallelization*. Ph.D. thesis, University of Reading, UK, Department of Computer Science, September 1998.

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

[93] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *International Workshop on Automated and Algorithmic Debugging(AADEBUG 2005)*, pages 33–42, Monterey, California, USA, September 2005. ACM.