

# Pareto Optimal Search-based Refactoring at the Design Level

Mark Harman, Laurence Tratt  
Department of Computer Science  
King's College London  
Strand, London, WC2R 2LS  
mark.harman@kcl.ac.uk,  
laurie@tratt.net

## ABSTRACT

Refactoring aims to improve the quality of a software systems' structure, which tends to degrade as the system evolves. While manually determining useful refactorings can be challenging, search-based techniques can automatically discover useful refactorings. Current search-based refactoring approaches require metrics to be combined in a complex fashion, and produce a single sequence of refactorings. In this paper we show how Pareto optimality can improve search-based refactoring, making combining metrics easier, and presenting multiple sequences of optimal refactorings to users.

## General Terms

Delphi theory

## Keywords

Search-based, software engineering, refactoring, Pareto optimality

## 1. INTRODUCTION

Software systems are subject to continual change and as they evolve to reflect new requirements, their internal structure tends to degrade. The cumulative effect of such changes can lead to systems that are unreliable, difficult to reason about, and unreceptive to further change. Refactorings aim to reverse this decline in software quality by applying a series of small, behaviour preserving transformations each of which improves a certain aspect of the system [Fowler 1999]. Typical examples of refactorings are moving a method from one class to another or inserting a class into the inheritance hierarchy.

Currently refactorings have to be determined and applied by hand. While some useful refactorings can be easily identified, often it is difficult to determine those refactorings which would improve a systems' structure. Unfortunately many seemingly useful refactorings, whilst improving one

aspect of a system, make another aspect worse. This problem is particularly acute with large systems, or with systems which the would-be refactoring is unfamiliar with. Therefore it has been proposed to view refactoring as a search-based technique, where an automated system can discover useful refactorings [O'Keeffe and Cinnéide 2006]. This can be achieved by determining an appropriate metric which measures the overall quality of the system and using that as a fitness function [Harman and Clark 2004]. Useful refactorings are those which improve the metric.

There are two problems with this approach. The first is determining which metric(s) are useful for a given system, and finding how best to combine multiple metrics [O'Keeffe and Cinnéide 2006]. The second is that the result of running the search is a single sequence of refactorings; while multiple runs of the search may produce different sequences of refactorings, the user is given no guidance as to which sequence may be best for their given system, beyond their relative fitness values.

In this paper we show how the concept of Pareto optimality can usefully be applied to search-based refactoring. Multiple runs of our search-based refactoring system lead to the production of a Pareto front, whose values represent Pareto optimal sequences of refactorings. Each value on the front in some sense maximises the multiple metrics used to determine the refactorings. Users can therefore choose a value on the front which represents the trade-off between metrics most appropriate to them, in the knowledge that it is Pareto optimal. We also show how the production of a Pareto front lessens the need for complex combinations of metrics, since differing fitness functions contribute different Pareto optimal values to the front.

## 2. BACKGROUND

### 2.1 Search-based refactoring and transformation

Program refactoring is closely related to source-to-source program transformation, since both are concerned with altering the concrete representation of the program, while preserving its semantics. In both cases, the program is restructured in order to improve some property of the way in which the program expresses the computation it denotes. Several authors have previously considered the problem of automating the process of transforming or refactoring a program, using Search Based Software Engineering. There are two broad approaches considered in the literature. To distin-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

guish between them in this paper, we call them the ‘direct’ and ‘indirect’ approach.

### 2.1.1 Direct Approach

In the direct approach, the program is directly optimized. That is, the possible variants of the program form the search space of the problem. Using this approach the transformation/refactoring steps are applied directly to the program, denoting moves from the current program to a near neighbour in the search space. This approach is best suited to a local search, because semantic preservation can only be ensured by applying a valid transformation to the individual. Therefore, with one exception, all work that has adopted this approach, has used local search techniques, such as hill climbing and simulated annealing.

The single exception is Ryan [Ryan 2000], who used Genetic Programming (GP) to automate parallelization for supercomputers. GP evolves versions of the program which do not necessarily preserve the semantics of the original, because GP crossover and mutation operations may (either subtly or radically) alter the behaviour of the original program. This means that it is possible that Ryan’s approach will produce a version of the program that increased parallelizability, but which is not faithful to the semantics of the original. To address this problem of semantic compliance, Ryan used a set of test cases to measure how close the GP-evolved solution was to the original. This formed a part of the fitness assessment for the GP.

However, the problem remains that subtle changes in semantics might mean that the program merely *appears* to be faithful to the semantics of the original according to the selected set of test cases. For refactoring, this is unacceptable, as the refactored (i.e. transformed) version of the program must be *guaranteed* to preserve the semantics of the original.

Apart from Ryan’s work on GP for parallelization, other work on the direct approach has avoided the use of GAs because of the problem of ensuring correctness when the cross over operator is applied. The parallelization problem was also addressed by Williams [Williams 1998] using the direct approach with local search techniques. More recently, O’Keefe and O’Cinnéide [O’Keefe and Cinnéide 2006] have applied the direct approach to the problem of refactoring. They use a collection of 12 metrics to measure the improvements achieved when methods are moved between classes. These 12 lower-level metrics are combined using various weightings to assess more subjective and higher level indirect metrics such as ‘flexibility’ and ‘understandability’. The weightings represent the various degrees to which the authors believe that these 12 metrics contribute to the higher level concepts indirectly measured.

### 2.1.2 Indirect Approach

In the indirect approach, the program is indirectly optimized, via the optimization of the sequence of transformations to apply to the program. In this approach the program is optimized by a sequence of transformations and it is the set of possible sequences of transformations that form the search space. Fitness is computed by applying the sequence of transformations to the program in question and measuring the improvement in the metrics of interest. In this way, the goal remains the refactoring/transformation of the program, but the optimization is performed on the transformation sequence and fitness is computed indirectly,

by applying the transformation sequence to the program.

In the indirect approach, it is possible to apply global search techniques (such as genetic algorithms) because the sequence of transformations can be subjected to arbitrary crossover and mutation operations. The result would be an arbitrarily changed sequence of transformations. However, since all transformations are meaning preserving, all such changes to the sequence are also guaranteed to be meaning preserving, when the changed sequence is applied to the program.

The first authors to use search in this way were Cooper et al. [Cooper et al. 1999], who used biased random sampling to search a space of high level whole-program transformations for compiler optimization. The order of application of optimization steps plays a crucial role in the quality of the results and so the search problem is to identify the optimal application order. Cooper et al. compare the results of their experiments with those obtained using a fixed set of optimizations in a predetermined order, showing that search can find better orders of application. Some whole program transformations may enable others and this is highly program-dependent. Therefore, no single ordering of whole program of transformations will suit all programs.

Williams [Williams 1998] also used a genetic algorithm to find sequences of whole-method transformations in order to optimize method parallelizability. Williams compared the direct and indirect approaches to the parallelization problem, when applied to small laboratory example programs, reporting that the direct approach outperformed the indirect approach. Nisbet [Nisbet 1998] also used a GA to find program restructuring transformations for FORTRAN programs to execute on parallel architectures.

Fatiregun et al. [Fatiregun et al. 2004, Fatiregun et al. 2005] showed how search based transformations could be used to reduce code size and construct amorphous program slices. Their approach also followed the indirect approach, treating the sequence of transformations to be applied as the individual to be optimized, allowing them to explore the relative value of greedy, GA, hill climbing and random algorithms. However, their transformation steps were smaller atomic level transformations than the transformation tactics used by Williams and Nisbet, or the transformation strategies used by Cooper et al. Also, for Fatiregun et al., the goal was to reduce program size rather than to improve performance.

Seng et al. [Seng et al. 2006] propose an indirect search-based technique using a genetic algorithm over sets of refactorings. In contrast to [O’Keefe and Cinnéide 2006], the multiple weighted metrics they combine into a single fitness function are based on well-known measures of coupling between program components.

The indirect approach is an example of a search based sequencing problem. Other related sequencing and prioritization problems have also been attacked using Search Based Software Engineering, for example sequencing of requirements implementations [Bagnall et al. 2001, Greer and Ruhe 2004, Karlsson et al. 1998], sequencing of work packages in project planning [Antoniol et al. 2005] and sequencing of test cases for regression test case prioritization [Li et al. , Nashat Mansour, Rami Bahsoon and Baradhi 2001, Rothermel et al. 2001, Walcott et al. 2006].

In all of these problems it is the sequence that is the final result of the optimization process. By contrast, in search

based transformation/refactoring the final result is the program obtained by applying the sequence of transformations to the original program, making this a very different kind of sequencing problem.

## 2.2 Problem Statement

In both the direct and indirect approach, previous work has considered the search based transformation and refactoring problems as single objective search problems. Where multiple metrics have been collected, for example, in the approach to refactoring due to O’Keefe and O’Cinneide [O’Keefe and Cinnéide 2006], the 12 directly computed source code metrics are combined, using weights, into a single objective fitness function. Weighting is a well-established approach to solving multiple objective problems, but it can suffer from several problems when the choice of weight coefficients is unclear and when the various metric values are not independent, as is the case with search based refactoring. This makes determining the relative quality of the fitness function difficult. Furthermore for some tasks, one may wish to add new metric into the fitness function (or remove certain metrics) to get best use from the system; this is currently an extremely daunting task. A second issue is that multiple runs of a search-based refactoring system may return different results; however the user is given no guidance as to which sequence of transformations may be best beyond their relative fitness values, which often.

We assert that a fundamental problem with existing search-based refactoring approaches is that the fitness functions are reliant on exceptionally well-crafted combinations of metrics. In this paper we show how the concept of Pareto optimality can lessen the need for well-crafted fitness functions and how it can allow users to choose the most appropriate output of multiple runs of a search-based refactoring system.

## 3. APPROACH

We have built a general search-based system in the Converse language [Tratt 2004] which reads in arbitrary Java systems, performs search-based refactorings upon them, and returns a sequence of refactorings as its output. As a system is read in, it is converted into a UML-like design model where low-level details in method bodies are largely ignored.

### 3.1 CBO

In the interests of brevity, we consider two metrics which measure the quality of a system. The first of those is the CBO metric from the [Briand et al. 1999] catalogue of metrics. CBO measures the coupling between classes in a system and is formally defined thus:

$$CBO(c) = |\{d \in C - \{c\} | uses(c, d) \vee uses(d, c)\}|$$

where  $C$  is the set of all classes in the system and  $uses(x, y)$  is a predicate which is true if there is a relationship between the two classes  $x$  and  $y$  e.g. an attribute or local variable of type  $y$  in  $x$ . As this suggests, although our system refactors at the design level, we record relationships between classes that occur within method bodies. In order to calculate the total CBO of an entire system we sum the CBO of each class:

$$CBO(C) = \sum_{d \in C} CBO(d)$$

Since it is considered to be desirable to have systems with

lower degrees of coupling, it is desirable to minimise the CBO value of any given system.

## 3.2 Refactorings

In similar fashion to [Seng et al. 2006] we consider only the move method refactoring, also reusing the following simple optimisations of the search:

- Each method can be moved a maximum of once.
- Don’t move a method if it breaks the system (e.g. don’t move a method from a non-abstract class if it overrides a method in an abstract superclass).
- Only move a method to a class there is already a relationship with.

A move method refactoring records three pieces of information: the class  $c$  the method is being moved from, the specific method  $m$  in  $c$ , and the class  $d$  that the method is being moved to.

## 3.3 Systems under analysis

For this paper we ran our experiments on the following three systems:

**JHotDraw v5.3** A GUI-based drawing application.

**Maven v2.04** A system building tool similar in spirit to `make`.

**XOM v1.1** An XML API.

All three systems are non-trivial real world systems, in the region of 20,000 to 40,000 lines of code. JHotDraw was chosen because it has been used in a previous search-based refactoring systems [Seng et al. 2006] as it is often considered an example of good design. Maven and XML represent very different styles of applications, and have very different styles of system design, than JHotDraw.

## 3.4 Determining search-based refactorings

Our search-based approach indirect in nature as it optimizes a sequence of refactorings. The search algorithm itself can be considered to be a variant hill climbing approach. In order to determine refactorings, we start with an unadulterated system and record the system’s fitness value. We then choose a random move method refactoring and apply the refactoring to the system. The fitness value of the updated system is then calculated. If the new fitness value is worse than the previous value, we discard the refactoring and try another. If the new fitness value is better than the previous, we add the refactoring to our current sequence of refactorings, and the system with that refactoring made to it becomes the new base for the next iteration. Since during each iteration an incredibly large number of possible refactorings can be tried, we set an arbitrary cut off point at which point we consider that no further refactorings can be usefully applied. For our experiments this cut off point is set at 500 attempted refactorings for an iteration, since beyond this point returns diminished to virtually zero. Note that the ordering of refactorings is important since some refactorings may only improve the fitness function after other refactorings have been performed.

The end result of our search is therefore a sequence of refactorings and a list of the before and after values of the various metrics involved in the search.

## 4. USING A SINGLE METRIC AS A FITNESS FUNCTION

As a simple first experiment, we use the CBO metric as the sole part of a fitness function to determine refactorings. Using the XOM system as an example, its initial CBO value is 351. A representative run of our search-based refactoring approach finds 68 move method refactorings such as:

```
Move makeProcessingInstruction from NodeFactory to Nodes
Move copy from FastReproducer to DocType
Move insertChild from NodeFactory to Element
```

The chain of refactorings found by the search reduces the CBO value to 272. In practical terms this seeming improvement is misleading since the improvement in CBO is at the expense of other desirable aspects of the system. As an example, figure 1 shows that, when the search is guided solely by CBO, the refactored system has emptied several classes of methods, while leading to the creation of a so-called ‘god class’ i.e. a class with a disproportionately large number of methods in it.

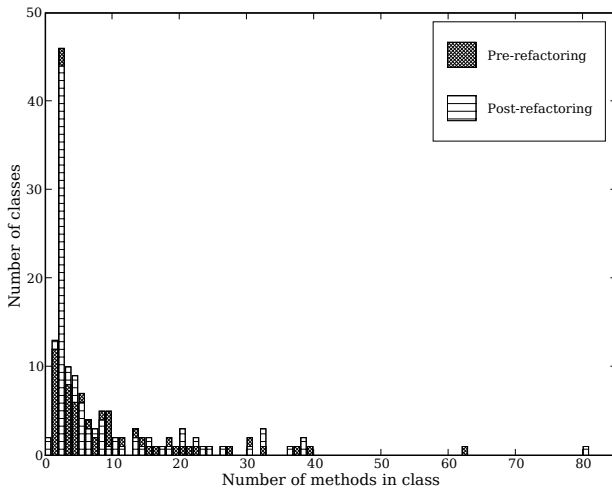


Figure 1: Method distribution in XOM before and after search-based refactoring using the CBO metric as the fitness function. The refactored system has more classes at the extreme ends of the scale.

## 5. COMBINING METRICS

Using a single metric to guide search-based refactoring has an obvious problems: optimizing only one aspect of the system can make other important measures of quality unacceptably worse. Therefore it is common to combine more than one metric when designing an appropriate fitness function, with the intuitive idea that the combination of metrics should prevent any one metric being unduly favoured.

In the case of the previous example, statistical theory provides a simple ‘counter metric’ to CBO’s tendency to bloat a small number of classes with large numbers of methods. The second metric we use is the standard deviation of methods per class in the system which we write as  $SDMPC(C)$  (note that the number of methods in the system stays constant no matter how many move method refactorings we use). We now come up against an immediate problem: how should

we combine these two metrics into one fitness function? Initial candidates include  $CBO(C) * SDMPC(C)$  or  $CBO(C) + SDMPC(C)$ , possibly with weightings attached to the individual metrics. Previous search-based refactoring approaches [O’Keeffe and Cinnéide 2006, Seng et al. 2006] combine metrics together in often complex fashions, and with the choice of weightings for various metrics often unclear. In similar fashion we initially arbitrarily define our new fitness function to be  $CBO(C) * SDMPC(C)$ .

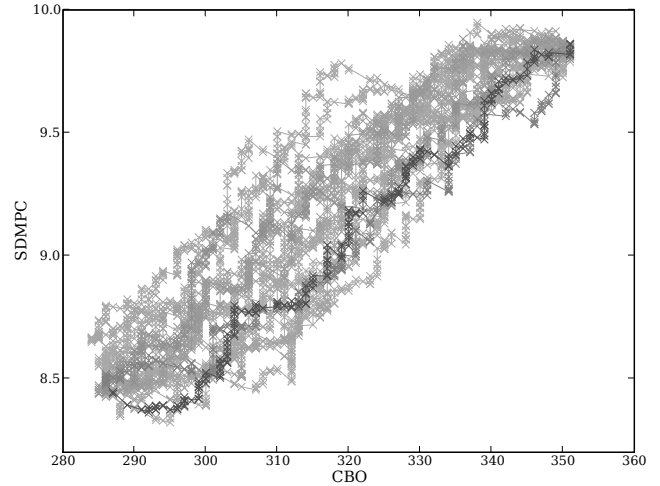


Figure 2: 20 runs of the search-based refactoring using the  $CBO(C) * SDMPC(C)$  fitness function where each run is in increasingly darker shades of grey. Lower values are better on both axes. The search started in the top-right corner of the graph.

Armed with this new fitness function, we run our search-based refactoring system again. Figure 2 shows the output from running the system 20 times on XOM; each run takes approximately 3 minutes on an AMD 4200+ desktop PC. As expected, the new fitness function improves the CBO value of the refactored system while also improving the SDMPC of the system. However figure 2 is more notable for two things that it does not tell the user. First which is the ‘best’ of the 20 sequences of refactorings discovered? Second is this a good fitness function? We tackle both questions in subsequent sections.

## 6. PARETO OPTIMALITY

In this section we show how the concept of Pareto optimality naturally applies to search-based refactoring. In order to do that, we first define the concept of Pareto optimality and a Pareto front.

### 6.1 Definition

The concept of Pareto optimality was originally defined by the Italian economist Vilfredo Pareto<sup>1</sup>. In economics a value is effectively a tuple of various metrics which can be made better or worse. A value is Pareto optimal if moving from it to any other value makes any of the constituent metrics better or worse; it is said to be a value which is not dominated by any other value. For any given set of values

<sup>1</sup>Also notable as the man responsible for the 80/20 rule!

there will be one or more Pareto optimal values. The sub-set of values which are all Pareto optimal is termed the Pareto front.

In the context of search-based systems, one additional concept is important. In theory for a search-based system there is a ‘true’ Pareto front — that is, the front from which no other combinations of refactorings can produce any more Pareto optimal values. We assume that production of this ‘true’ front is impossible analytically, and impractical through exhaustive search. Therefore the front of Pareto optimal values we can create after any given number of runs is considered to be an approximation to the ‘true’ front. Further runs of the system may improve the front approximation (note that improvements may only improve part of the front, leaving the rest of the front at its previous value). However we do not necessarily expect our approximation to ever reach the ‘true’ front, and indeed we are unlikely to be able to tell if this has happened, since we can not be sure if further runs of the system might improve the front further. Consequently in the rest of this paper we assume that any given Pareto front is an approximation of the ‘true’ Pareto front, since this is the reality of Pareto fronts in search-based systems.

## 6.2 Creating a Pareto front

Figure 2 showed the result of running the search-based system on XOM 20 times. However there is no obvious way to determine which is the ‘best’ value(s) from the resulting data. Since in this case lower values are better for both axes, intuitively one might expect that the value in the ‘bottom left’ would be the best possible value — however there are several values that one could argue are the ‘real bottom left’.

Figure 3 shows the Pareto front calculated from 2, with the front is in the bottom left of the graph. The user can thus select one of the values on the front knowing that, from the existing runs, he is guaranteed to be picking one of the Pareto optimal points.

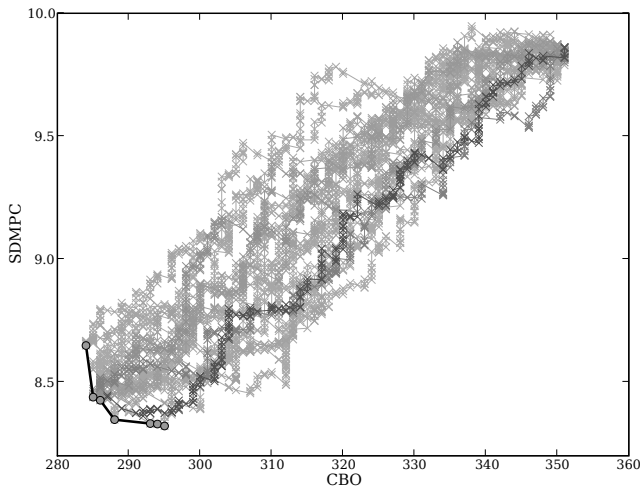


Figure 3: The XOM refactoring Pareto front from figure 2.

All the points on a Pareto front are, in isolation, considered equivalently good. In such cases, it might be that the user may prefer some of the Pareto optimal points over oth-

ers. For example user who are more interested in achieving a low CBO than they are in a low SDMPC then will pick a point on the left of the front.

Often one finds that the user often has further criteria which allows them to pick from one the Pareto optimal values. In our case an important additional criteria is the number of refactorings that it takes to get to a given point on the Pareto front: often a developer would prefer to perform as few refactorings as possible to achieve an improved system. In figure 3 for example, points on the front range from 211 to 231 refactorings.

## 6.3 Pareto fronts of data sub-sets

For search-based refactoring, the additional criteria of the number of refactorings that a developer would need to make is more important in practice than we have hitherto given credit. Because of the simplistic metrics we have used to guide our search, large numbers of refactorings are generated. For example, for JHotDraw our search will typically suggest over 350 move method refactorings. While more complex combinations of metrics would reduce this number significantly (see [Seng et al. 2006]), for large systems it is still likely to be the case that developers may not have sufficient resources available to make all suggested refactorings.

The concept of a Pareto front makes as much sense with subsets of data as it does for complete sets. This then allows us to allow developers to determine how much resources they are prepared to make available for refactoring, and to generate a Pareto front that respects that limit. Figure 4 shows the Pareto front resulting from restricting the search-based refactoring of JHotDraw to the first 50 refactorings; for comparison the Pareto front for the full search (which requires over 350 refactorings) is shown in the far bottom left.

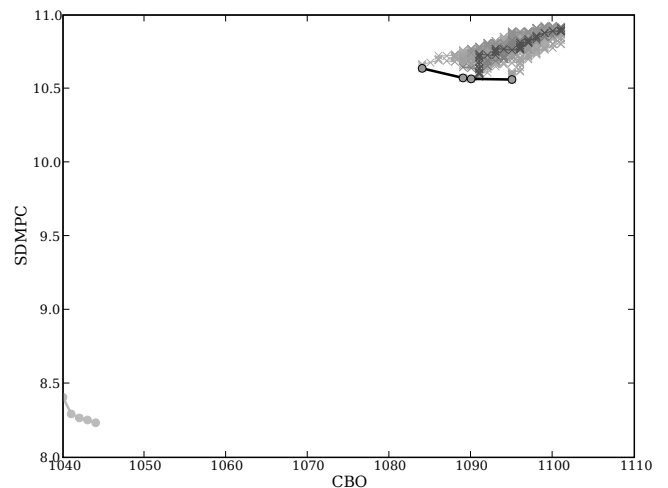


Figure 4: The Pareto front created by 20 runs on JHotDraw when limited to 50 refactorings. For comparison the front of the complete search, which involves several times as many transformations, is shown in light grey.

One thing to note about figure 4 is that the number of points on, and general shape of, the Pareto front for the sub-setted data has no inherent relationship to that of the

front for the full search. Although we do not show it here, in the interests of brevity, one can get surprisingly different shaped fronts for different sub-sets of the refactorings.

## 6.4 Evolution of the Pareto front approximation

Up until this point we have always used 20 runs of the search-based refactoring system to illustrate our points. Part of the reason for choosing this figure is that it is the point at which the visualizations used thus far are still reasonably uncluttered. However an important question is: how many runs of a search-based refactoring system does it take to produce a good approximation of the true Pareto front?

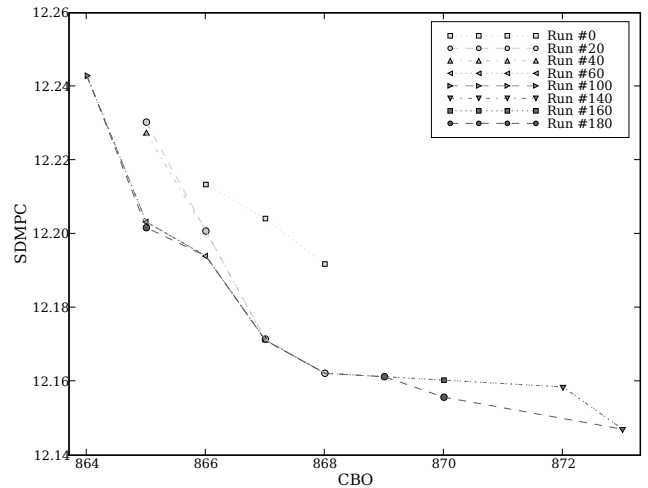
Figure 5 shows how the Pareto front approximation evolves for the Maven system, showing the evolving approximation after every 20 runs of the search-based system (if we show the front evolving after turn the graph becomes rather cluttered for publication). The first thing we observe is that the very first run of the search system creates a front with 3 points in it: depending on the fitness function, it is possible for a run of the search system to generate multiple points on the front approximation. Our next observation depends on the fact that the visualization captures not only the evolution of the Pareto front approximation, but also only shows the first run which discovered any given point on the front (i.e. if both run  $M$  and run  $N$ , where  $N > M$ , have point  $(x, y)$  then the graph shows run  $M$  as being the source of that point). Knowing this, we can see that by run 20 the Pareto front has evolved substantially over the first run; but from run 20 to run 40 there is relatively little improvement. Indeed, only runs 100 and 140 (and, to a lesser extent 60 and 180) make substantial updates to the front approximation.

As figure 5 shows, while longer runs of the search system create better front approximations, one gets surprisingly good results after relatively few runs. For very large software systems, where running the search system may be quite slow, this is an important result as it allows developers to get reasonable quality answers quickly. Furthermore since the front approximation is an additive process – in other words, subsequent runs of the search system may improve the front approximation, but can’t make it worse – developers are free to execute extra runs of the system if they feel they have not yet achieved points of sufficient quality on the front approximation.

## 7. MULTIPLE FITNESS FUNCTIONS AND PARETO OPTIMALITY

Pareto optimality allows us to determine if one fitness function is inferior or not to another: broadly speaking, if fitness function  $f$  produces data which, when merged with the data produced from function  $f'$ , contributes no points to the Pareto front then we define  $f$  to be inferior to  $f'$  (notice that we do not define a notion of superiority, simply non-inferior). While theoretically one needs to test all possible inputs to  $f$  to confirm this, one need only find one counter-example to show non-inferiority.

Earlier we noted that two of the more obvious ways to combine the CBO and SDMPc metrics into a fitness function are  $CBO(C) * SDMPc(C)$  and  $CBO(C) + SDMPc(C)$ . Thus far in this paper we have used the former of these two fitness functions to guide our search-based refactoring system. Using the latter fitness function on any of the systems



**Figure 5: The evolving Pareto front approximation for Maven after every 20 runs (iterations which do not evolve the front approximation are not shown). Each point on a front approximation represents the first iteration of the search-based system to discover that point.**

under examination in this paper, we quickly find that it is non-inferior i.e. it produces distinct Pareto optimal values.

Although it may not be immediately apparent, Pareto optimality confers a benefit potentially more useful than simply determining if one fitness function is worse than another. If two fitness functions generate different Pareto optimal points, then we can naturally combine the different points into a single front. As figure 6 shows, for JHotDraw the two different fitness functions give the user extra Pareto optimal refactoring sequences to choose from.

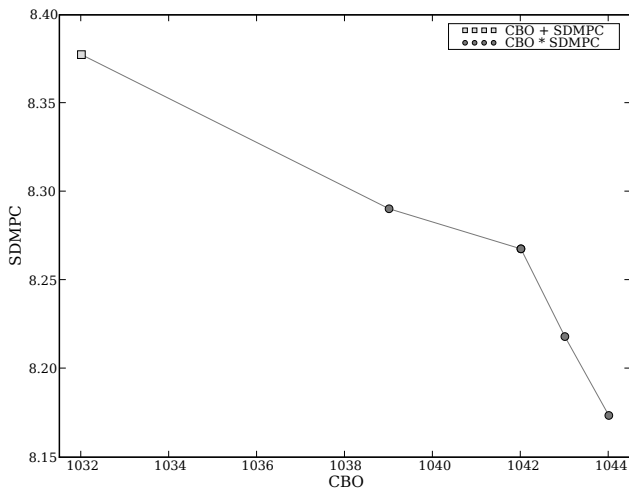
As this shows, Pareto optimality has many benefits for search-based refactoring. It lessens the need for ‘perfect’ fitness functions; indeed, as this shows, different fitness functions can increase the diversity of Pareto optimal values presented to the user.

### 7.1 Fitness values as tuple

The more diverse the points on the Pareto front, the more choice the user of our search-based refactoring system has. In this section we show one example of a non-traditional approach to a refactoring fitness function that can generate significantly different points on the Pareto front.

First we treat the fitness function not as a combined value of metrics, but as a tuple  $(CBO(C), SDMPc(C))$  of the two individual metrics. A fitness value  $(a, b)$  is considered better than  $(x, y)$  iff  $(a < x \wedge b \leq y) \vee (b < y \wedge a \leq x)$ ; intuitively this says that a fitness value is better than another provided it makes one metric better and the other metric no worse. As figure 7 clearly shows, this technique leads to significantly different Pareto optimal points than the previous two metrics. The purpose of this example is not necessarily to show that this particular fitness function is useful, but rather to show that non-traditional fitness functions may yield useful Pareto optimal values.

## 8. FUTURE WORK



**Figure 6:** The Pareto front generated by 100 runs of the two fitness functions  $CBO(C) * SDMPC(C)$  and  $CBO(C) + SDMPC(C)$  on JHotDraw. Notice that both fitness functions contribute different point(s) on the front.

While we believe that this paper provides a solid foundation for using the Pareto optimality concept in search-based refactoring, subsequent research could go in many different directions. The most obvious future direction is to extend our search-based refactoring system to measure more complex metrics and to investigate how different combinations of metrics effect the results of the search, and we are currently working to this end.

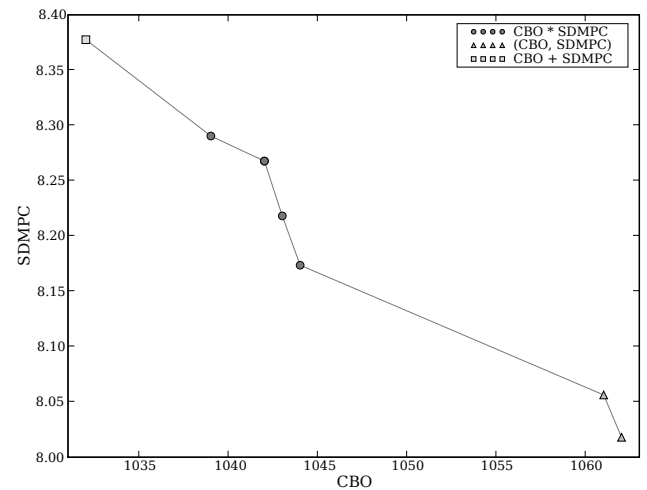
We also believe that treating refactoring fitness values as tuples may prove more effective than the traditional coalesced fitness values, since one can express more complex constraints such as ‘accept a refactoring if it makes metric  $M$  better but metric  $N$  no more than 1% worse than its previous value’. This may allow small number of metrics to provide equally convincing real-world results as a larger number of coalesced metrics.

## 9. CONCLUSIONS

In this paper we first defined the concept of ‘direct’ and ‘indirect’ approaches to search-based refactoring and discussed how existing search-based refactoring approaches rely on complex fitness functions with weighted combinations of metrics. We then presented a general-purpose search-based system running on several real-world open-source Java applications. By taking two simple metrics, we were able to show how the concept of Pareto optimality can be usefully applied to search-based refactoring, and how it allows multiple fitness functions to present different Pareto optimal values to the user.

## 10. REFERENCES

ANTONIOL, G., PENTA, M. D., AND HARMAN, M. 2005. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *21<sup>st</sup> IEEE International Conference on Software Maintenance* (Budapest, Hungary, September



**Figure 7:** The Pareto front created by 100 runs each of 3 different fitness functions on JHotDraw.

30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 240–249.

BAGNALL, A., RAYWARD-SMITH, V., AND WHITTLEY, I. 2001. The next release problem. *Information and Software Technology* 43, 14 (Dec.), 883–890.

BRIAND, L. C., DALY, J. W., AND WÜST, J. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Eng.* 25, 1, 91–121.

COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES’99)*. ACM Sigplan Notices, vol. 34.7. ACM Press, NY, 1–9.

FATIREGUN, D., HARMAN, M., AND HIERONS, R. 2004. Evolving transformation sequences using genetic algorithms. In *4<sup>th</sup> International Workshop on Source Code Analysis and Manipulation (SCAM 04)* (Chicago, Illinois, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 65–74.

FATIREGUN, D., HARMAN, M., AND HIERONS, R. 2005. Search-based amorphous slicing. In *12<sup>th</sup> International Working Conference on Reverse Engineering (WCRE 05)*. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 3–12.

FOWLER, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

GREER, D. AND RUHE, G. 2004. Software release planning: an evolutionary and iterative approach. *Information & Software Technology* 46, 4, 243–253.

HARMAN, M. AND CLARK, J. 2004. Metrics are fitness functions too. *Proc. International Symposium on METRICS*, 58–69.

KARLSSON, J., WOHLIN, C., AND REGNELL, B. 1998. An evaluation of methods for prioritizing software requirements. *Information and Software Technology* 39, 939–947.

LI, Z., HARMAN, M., AND HIERONS, R. Meta-heuristic search algorithms for regression test case prioritization.

*IEEE Transactions on Software Engineering*. To appear.

NASHAT MANSOUR, RAMI BAHSOON AND BARADHI, G. 2001. Empirical comparison of regression test selection algorithms. *Systems and Software* 57, 1, 79–90.

NISBET, A. 1998. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings*, P. M. A. Sloot, M. Bubak, and L. O. Hertzberger, Eds. Vol. LNCS 1401. Springer, 987–989.

O’KEEFFE, M. AND CINNÉIDE, M. Ó. 2006. Search-based software maintenance. In *Proc. Software Maintenance and Reengineering*.

ROTHERMEL, G., UNTCH, R., CHU, C., AND HARROLD, M. J. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct.), 929–948.

RYAN, C. 2000. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers.

SENG, O., STAMMEL, J., AND BURKHART, D. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proc. Genetic and Evolutionary Computation*.

TRATT, L. 2004. *Converge Reference Manual*.  
<http://www.convergepl.org/documentation/refmanual/>  
Accessed Jan 3 2006.

WALCOTT, K. R., SOFFA, M. L., KAPFHAMMER, G. M., AND ROOS, R. S. 2006. Time aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA 06)*. ACM Press, Portland, Maine, USA., 1 – 12.

WILLIAMS, K. P. 1998. Evolutionary algorithms for automatic parallelization. Ph.D. thesis, University of Reading, UK.