Search Based Transformations

Deji Fatiregun, Mark Harman, and Robert Hierons

Department of Information Systems and Computing
Brunel University
Uxbridge, Middlesex, UB8 3PH
ayodeji.fatiregun@brunel.ac.uk

1 Introduction

Program transformation [1,2,3] can be described as the act of changing one program to another. The aim being an alteration of the program syntax but not its semantics, hence leaving both source and target programs functionally equivalent. Consider as examples of transformations the following program fragments:

T_1: `if (e1) s1; else s2;` $\Rightarrow$ `if (!e1) s2; else s1;`
T_2: `if (true) s1; else s2;` $\Rightarrow$ `s1;`
T_3: `x = 2; x = x - 1; y = 10; x = x + 1;` $\Rightarrow$ `x = 2; y = 10;`
T_4: `for (s1; e2; s2) s3;` $\Rightarrow$ `s1; while (e2) s3; s2;`

Program Transformations are generally written in order to generate *better* programs. In transformations, we apply a number of simple transformation axioms to parts of a program source code to obtain a functionally equivalent program. The application of these axioms is treated as a search problem and we apply a meta–heuristic search algorithm such as hill climbing to guide the direction of the search.

1.1 The Transformation Problem

An overall program transformation from one program p to an improved version p' typically consists of many many smaller transformation tactics [1,2]. Each tactic consists of the application of a set of transformation rules. A transformation rule is an atomic transformation capable of performing the simple alterations like those captured in the examples $T_1 \ldots T_4$. At each stage in the application of these simple rules, there are many points in the program at which a chosen transformation rule could be applied.

There are many points in a program; typically one per node of the Control Flow Graph. The set of pairs of possible transformation rules and their corresponding application point is therefore large. Furthermore, to achieve an effective overall program transformation tactic, many rules may need to be applied, and each will have to be applied in the correct order to achieve the desired result.

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 2511–2512, 2003.

2 Local Search-Based Transformation

Meta-heuristic search algorithms such as hill-climbing may be applied to arrive at an optimum result, or at least, a locally optimal result. Rather than apply the transformations manually, one after the other, we allow the algorithm to pick the best transformations to apply from a given set. We assume a search space containing all the possible allowable transformation rules and define our fitness function using an existing software metric [5,4], for example to be size of the program in Lines of Code (LoC).

An optimum solution would be the sequence of transformations that results in an equivalent program with the fewest possible number of statements. For instance, in the examples used in the introduction, transformation T_3 clearly shows a reduction in the size of the program from 4 nodes to 2 nodes and so would be selected as one which returned a better program, than, for example, the identity transformation.

Using a simple hill-climbing search algorithm and a size-based metric such as LoC, after a particular transformation is applied, the size of the new program is compared with that of the previous program. Any transformation which reduces size, is retained and the search continues from the new smaller program found. When no smaller program is found by the application of a rule, the search terminates.

3 Evolutionary Search-Based Transformation

Our approach is to use the *transformation sequence* to be applied to the program as the individual to be optimised. Using the transformation sequence as the individual makes it possible to define crossover relatively easily. Two sequences of transformations can be combined to change information, using single point, multiple point and uniform crossover. The result is a valid transformation sequence and since all transformation rules are meaning preserving, so are all sequences of transformation rules.

References

1. I. D. Baxter. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse*, Austin, TX, USA, January 1999.
2. Keith H. Bennett. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 247–254, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.
3. John Darlington and Rod M. Burstall. A tranformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
4. Norman E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1990.
5. Martin J. Shepperd. *Foundations of software measurement.* Prentice Hall, 1995.