Software refactoring guided by multiple soft-goals

Yijun Yu, Julio Cesar Leite, John Mylopoulos, Linda Lin Liu, Eric Yu, Erik D'Hollander*

CS Department, University of Toronto, M5S 2E4 Canada
*ELIS Department, University of Ghent, B9000, Belgium

Abstract

Software refactoring is intended to enhance the quality of a software by improving its understandability, performance, as well as other quality attributes. We adopt the modelling framework of [14] in order to analyze software qualities, to determine which software refactoring transformations are most appropriate. In addition, we use software metrics to evaluate software quality quantitatively. Our framework adopts and extends work reported in [15].

1 Introduction

Fowler et al [7] define software refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure". "Improvements to its internal structure" amount to improvements to the quality of the software system (also known as non-functional requirements). Examples of such improvements are "making the code easier to understand and cheaper to modify". Their refactoring framework was proposed mainly for improving understandability and modifiability. However, the idea can also be applied to other qualities [14], such as performance, security, usability and more.

In this study, we adopt the process-oriented framework proposed in [3] for modelling software qualities. In this framework, qualities and the factors that affect them are modelled as *soft-goals*, while functionalities are modelled as *goals*. Specifically, they can be *operationalized* into *tasks* and *resources*. The dependencies among goals, soft-goals, tasks and resources are represented in a soft-goal interdependence graph (SIG).

Besides the functionality goals, we focus on performance and code complexity soft goals that are associated with a set of software metrics [6]. Constrained by conflicting resources, multiple soft-goals have to be traded off, in situations such as "apply transformations to speedup the program 20 times without sacrificing the code complexity 4 times and introducing new expense on hardware"; or "adding functionalities to the program until the performance is getting slow (e.g., longer than one hour) or the code becomes too complex to understand and maintain (e.g., more than 10 classes in one header file)".

In a case study, we will show how SIG guides the refactoring towards high performance and code simplicity while keeping implementing more functionalities. We have applied this approach in our header refactoring project [18] observing the progress (functionality), health (code complexity) and quality (performance) evolutions in the software development [4].

2 Software refactoring process

The refactoring process [7] is a sequence of small transformation steps: "while each refactoring step is simple, yet the cumulative effect of these small changes can radically improve the design". To achieve this success, our approach is an iterative process that meets the soft-goals gradually, as illustrated in Figure 1. The process is subdivided into four consecutive steps:

1. Setting up the goal-reasoning model as a SIG [3], quantifying the satisfy or denial attributes of each soft-goal in a [0,1] metric [8];

2. Quantitatively measuring software metrics so as to claim which alternative soft-goal should be applied first;

3. Picking an effective refactoring among various transformations that contribute to the claimed soft-goal;

4. Applying the selected refactoring technique, which leads to iterative evaluations back in step 2,

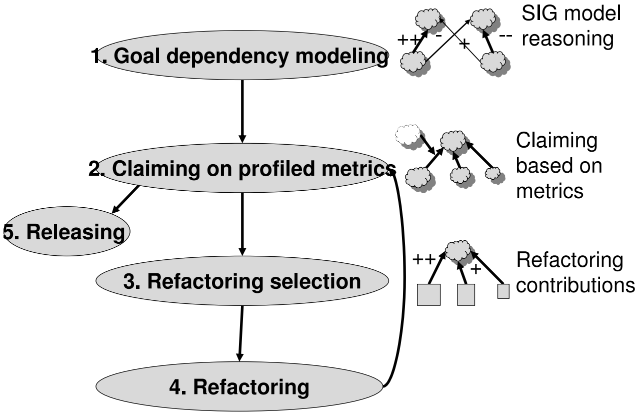

Figure 1: The overview of the soft-goal directed software refactoring process.

until every top-level soft-goals are met to release the software product.

3 A case study

In this section, we report on a case study for performance and complexity soft-goals on a fixed functionality – consider the following Fortran program for multiplying two matrices $A \in R^{m \times l}$ and $B \in R^{l \times n}$ into a matrix $C \in R^{m \times n}$.

```
real*8 A(512,512),B(512,512),C(512,512)
M = L = N = 512
do i = 1,M
  do j = 1, L
    do k = 1, N
      C(i,k) = C(i,k) + A(i,j) * B(j,k)
```

3.1 Goal Modelling

Software development aims to improve the speed of software (performance) and to reduce the code complexity. Achieving the performance soft-goal reduces the operational cost while achieving the code simplicity soft-goal reduces the develop and maintenance cost [10].

Here we establish a soft-goal interdependence graph for the possibly conflicting higher performance and lower code complexity soft-goals and show how much the operationalized soft-goals contribute to them.

Less code complexity Code complexity has effects on how difficult to test and to maintain, while good testability and maintainability lead to less defect rate. In literature [10], code complexity can be measured in lines of code (LOC), McCabe's cyclomatic number $V(G) = e - n + 2$ [12] or in Halstead's information science metrics [9] as $(N_1 + N_2)log_2(n_1 + n_2)$ where n_1 is the number of unique operators, N_1 is the total number of operators, n_2 is the number of unique operands and N_2 is the total number of operands N_2. Their SIG are shown in Figure 2.

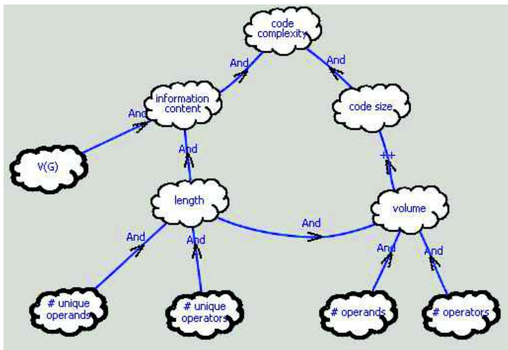

Figure 2: A SIG model for code complexity

Higher time performance In order to achieve a higher time performance system "Perf[System]", both hardware ("Perf[Architecture]") and software ("Perf[Algorithm]" and "Perf[Coding]") improvements are useful. For software refactoring, we consider mostly on the codings that fit programs on the given architecture. The "Perf[architecture]" soft-goal is decomposed into "Perf[Processor]" and "Perf[Storage]", and "Perf[Processor]" is decomposed into "Faster CPU frequency[Processor]", "More CPU[Processor]" and "Deeper pipeline[Processor]"; "Perf[Storage]" is decomposed into "Perf[Main Memory]", "Perf[Cache]" subgoals, where "Perf[Cache]" can be further decomposed into "Larger cache size[Cache]", "Larger cache line[Cache]" and "More set associativity[Cache]" soft-goals. Soft-goals for coding have corresponding hardware constraints. For example, "More CPU[Processor]" soft-goal is required to implement parallelism, which can be improved by "Loop partitioning[coding]" [1, 5, 19] to achieve "More loop parallelism[coding]" soft-goal; "Larger cache[Cache]" soft-goal to resolve capacity cache misses; "Loop tiling and fusion[coding]" shortening the stack reuse distances [2] so as to avoid capacity misses for a given cache size [16, 13, 2]. A detailed decomposition of time performance soft-goal is shown in Figure 3.

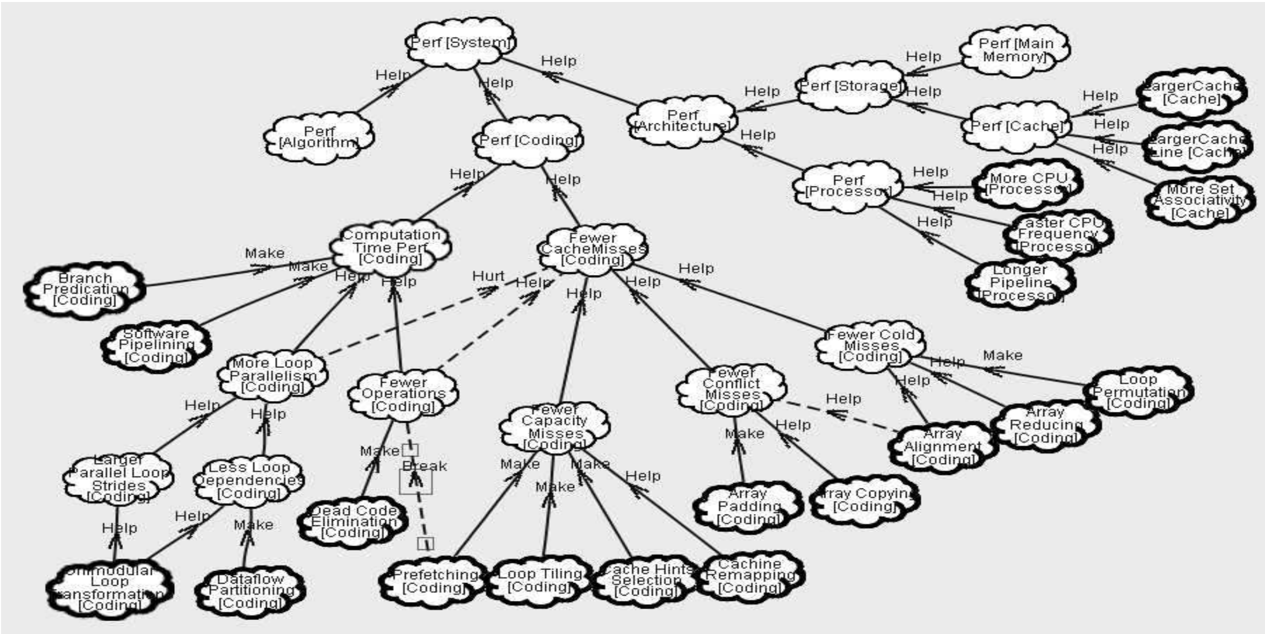

Figure 3: A SIG decomposed for time performance. The thinner cloud nodes are soft-goals while the heavier cloud nodes are operationalized (measurable or implementable) soft-goals. The solid arrows reflect top-down decomposition of the soft-goal into "And", "Or" or supporting soft subgoals such as "Make(++)" and "Help (+)". The dashed arrows reflect correlational links between soft-goals on different decomposition paths to represent both positive and negative ("Hurt(−)/Break(−−)") contributions.

3.2 Claiming bottlenecks soft-goals

In the SIG, transformations are the operationalized soft-goals. Depending on the the problem and the resources, different transformations contribute differently to the parent soft-goals. In our case study, for instance, we don't welcome algorithm or architecture changes because they are *claimed* less return of investment (expensive in human creativity or in machine availability). Thus we consider the children of the "Perf[coding]" soft-goal. Which of them should be addressed? According to Intel VTune white paper, hardware counter metrics help programmer to decide whether the execution time is intensive in memory, I/O, communication or computation. One of the decision point is the ratio of the L2 cache misses over the number of load instructions: whenever the L2 cache miss ratio is higher than 20%, memory is the bottleneck of performance. In our case study, this ratio is over 79.45%, which indicates that "Fewer cache misses[coding]" is *the* soft-goal to address[1]. Using a cache visualizing tool [17], we found that the dominate cache misses are capacity misses as well as conflict misses, which narrows us down to the "Fewer Capacity Misses[coding]" and the "Fewer Conflict Misses[coding]" categories. Now just a few transformation techniques are suited.

[1]All experiments are performed on an Intel P4 1.2GHz CPU with 512KB cache and 128MB main memory

3.3 Selecting refactoring transformations

Bottleneck measurement is still not enough to decide which refactoring should be made. Another factor is meeting multiple soft-goals as the general aim of the refactoring. For example, the aim can be expressed as "apply transformations to speedup the program 20 times without sacrificing the code complexity 4 times". One needs to see which transformation is most effective.

In the first step of refactoring process, we consider what operational soft-goals are applicable. Fixed hardware platform and algorithm already narrowed our search to the "Perf[coding]" subcategory in Figure 3. Initially we considered five applicable transformations to this program. Besides the above transformations we also investigated the impact of different optimization techniques such as dynamic memory allocation, dot-product.

Using the experiment data, the log scale view of

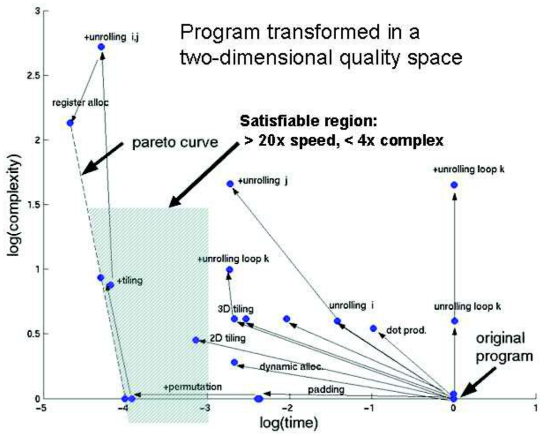

Figure 4: For the example programs, the quality space with time performance and code complexity indices are shown at log scale. Each program in table 1 is projected to a point in the space, each transformation produces an arrow from the program to another. It is clear that array padding, loop permutation, tiling and unrolling are effective optimizations when used properly, however, the last two increase complexity.

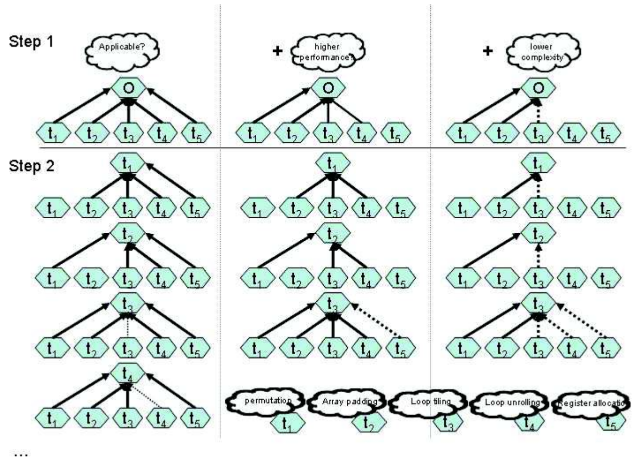

Figure 5: Two steps of refactoring are shown. Each step identifies the applicable tasks (operational soft-goals) for the given program (context), then decides on appropriate tasks for performance and complexity. The weights given to a branch result from multiplying the distribution weight through measurement with the soft-goal dependence weight in the quantified SIG.

the quality space of time and complexity indices are shown in Figure 4.

Some techniques improve one soft-goal but harm severely to the other, while others provide net improvement to both soft-goals. The decision maker can choose the one suited for the intention. Figure 5 shows five major transformations as decision making alternatives. It is based on the initial soft-goals as "apply transformations to speedup the program 20 times without sacrificing the code complexity 4 times", which is indicated in Figure 4 as a shadowed region.

4 Application

By laws of software evolution, a software is naturally subjected to continuing change (law 1), increasing complexity (law 2) and declining quality (law 7) [11]. We applied the proposed refactoring process to an on-going joint project with IBM Toronto lab [18]. This project aims at delivering a restructuring tool that speeds up the build processes within a reasonable time budget. We monitored the growth of the tool from scratch and plotted the metrics of changing functionality, performance and complexity in Figure 6, as it evolves from a code dependency graph partitioning algorithm (version 1 to 3), to 269 KLOC VIM 6.2 in C (version 4 to 9) and to a 1580 KLOC commercial software package in C++ (version 10 to 13). The functionality is measured as the number of program units being correctly preprocessed. Here we use the number in version 13 as unit to normalize those at earlier versions. Performance and complexity metrics are also normalized as ratios against the largest data point and observed the versions where performance tuning and complexity refactoring taken places.

5 Related work and summary

Ladan Tahvildari et al [15] first applied the NFR framework for comparing performance and maintainability of OO software with/without design patterns. Her work emphasizes on comparing design patterns on maintainability issue, while this paper focuses on the performance tuning issues along with refactoring in the develop process. In order to make the trade-off between the two issues clear, we plot the metrics in an quality space so that the refactoring process can be traced as one of the paths leading to the satisfiable region by the soft-goals.

Functionalities of a software system are concerned with changes to the state of data, while non-functional requirements are with the changes to the state of the program without touching the state of the data.

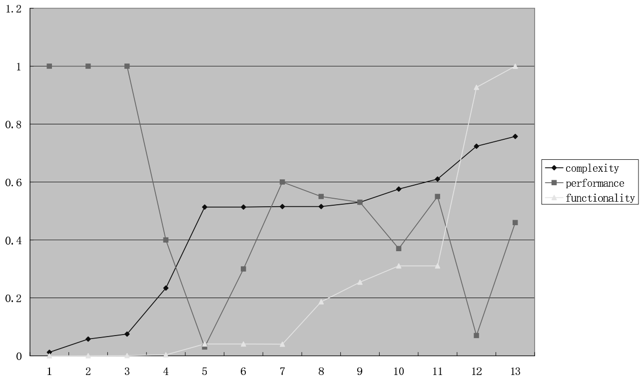

Figure 6: Goal guided refactoring on the software development for C/C++ header restructuring project.

In this perspective, Zou et al [20] consider software migration practice as a state transition system of the quality attributes. The assumption is that a legacy procedural program as a given product can be migrated to leverage qualities promised by object-oriented paradigm.

A case study in our work has shown that refactoring can be measured as the transformation on the state of program in the quality space. By further monitoring the process developing a new software from scratch, we suggest measuring the quality space along with the progress so that the refactoring goal is balanced with the productivity goal on demand. The satisfiable region in the quality space should be adjusted dynamically during the software evolution, i.e., development can be centric with a different top-priority at a different development phase. Because the refactoring changes are non-functional, they are invertible if no functionality change happens. Mixing with the functionality changes, however, the invertibility does not hold. Therefore the steps of refactoring must be small enough so that neither the productivity nor the invertibility would endanger the development goals.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, feb 1993.

[2] K. Beyls and E. H. D'Hollander. Reuse distance-based cache hint selection. *Lecture Notes in Computer Science*, 2400:265–274, 2002.

[3] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-functional requirements in software engineering*. Kluwer Academic Publishers, 2000.

[4] H. Dayani-Fard. *Quality-based software release management*. PhD thesis, Queen's University, 2003.

[5] E. H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, jul 1992.

[6] N. Fenton and S. L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 edition, 1996.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with goal models. *Lecture Notes in Computer Science*, 2503:167–??, 2002.

[9] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1 edition, 1977.

[10] D. L. Lanning and T. M. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, 27(9):35–40, Sept. 1994.

[11] M. M. Lehman. Laws of software evolution revisited. *Lecture Notes in Computer Science*, 1149:108–120, 1996.

[12] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[13] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, Jul 1996.

[14] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, Jun 1992. Special Issue on Knowledge Representation and Reasoning in Software Engineering.

[15] L. Tahvildari and K. Kontogiannis. Requirements-driven software re-engineering framework. In *WCRE 2001*, pages 71–80, 2001.

[16] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, 1991.

[17] Y. Yu, K. Beyls, and E. D'Hollander. Visualizing the impact of the cache on program execution. In *5th International Conference on Information Visualization (IV '01)*, pages 336–341, Washington - Brussels - Tokyo, July 2001. IEEE.

[18] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software development processes. In *Proceedings of the 13th CASCON conference*, pages 288–297, Oct. 2003.

[19] Y. Yu and E. D'Hollander. Partitioning loops with variable dependence distances. In *Proceedings of 29th International Conference on Parallel Processing*, Toronto, Canada, Aug. 2000. Ohio State Univ.

[20] Y. Zou and K. Kontogiannis. Migration to object oriented platforms: A state transformation approach. In *International Conference on Software Maintenance (ICSM'02)*, pages 530–539, Oct. 2002.