Satisficing the Conflicting Software Qualities of Maintainability and Performance at the Source Code Level

Bill Andreopoulos
Department of Computer Science, York University, Toronto, Ontario, Canada, M3J 1P3
billa@cs.yorku.ca

Abstract. The major contributions of our work include adopting the *NFR framework* to represent and analyze two software qualities that often conflict with each other: *maintainability* and *performance*. We identified and described many heuristics that can be implemented in a system's source code to achieve either quality. We implemented some of the heuristics in two medium-sized software systems and then collected measurements to determine the effect of the heuristics on maintainability and performance. A general methodology is described for evaluating and selecting the heuristics that will improve a system's software quality the most. The results of our research were also encoded in XML files, and made available on the World Wide Web for use by software developers. The WWW address is: http://www.cs.yorku.ca/~billa/SIG/SIG.xml

Keywords: Maintainability, Performance, NonFunctional Requirement, Software Quality, Optimization.

1. Introduction

Developers should consider the conflicting qualities of *maintainability* and *performance* early in the process of developing a software system. Failing to integrate either quality into the system will later increase costs exponentially when maintaining or expanding the system. Thus, it is necessary to be able to represent and analyze software quality effectively throughout the entire software lifecycle. We propose a general methodology for evaluating and improving the quality of a software system. We focus on the *maintainability* and *performance* software qualities, because these qualities often conflict with one another and achieving one quality might affect the other negatively. An important part of our work was to examine how the two qualities conflict with each other.

In requirements engineering, a requirement can be described as a condition or capability to which a system must conform, and which is either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document. [1] Requirements can be classified into:

- Functional requirements, which are externally visible behaviors, showing what the system must do, and
- Non-functional requirements (or *software qualities*), which are constraints on the design and/or implementation of the solution.

Software qualities describe not *what* the software will do, but *how* the software will do it, by specifying constraints on the system design and/or implementation. [1] Unfortunately, software qualities are usually specified briefly and vaguely for a particular system.

We adopt the NFR Framework to represent the qualities of maintainability and performance, the software characteristics that affect them and the *heuristics* that can be implemented in source code to achieve them. The *NFR framework* for representing software qualities was developed by Lawrence Chung, Brian Nixon, Eric Yu and John Mylopoulos at the University of Toronto. [1] The NFR framework represents quality requirements as *softgoals*. Softgoals are goals with no clear-cut criterion for their fulfillment. Instead, a softgoal may only contribute positively or negatively towards achieving another softgoal. By using this logic, a softgoal can be *satisficed* or not; *satisficing* refers to satisfying at some level a goal or a need, but without necessarily producing the optimal solution. [1] The NFR framework represents information about softgoals using primarily a graphical representation, called the *softgoal interdependency graph*. A *softgoal interdependency graph* represents each softgoal as an individual node (or cloud). A softgoal interdependency graph records all softgoals being considered, as well as the interdependencies between them. [1] Figure 1 shows an example of a softgoal interdependency graph.

The results of our research were also encoded in XML files, and made available on the World Wide Web for use by software developers. The WWW address is: http://www.cs.yorku.ca/~billa/SIG/SIG.xml . The purpose of this website is to provide a tool that can be used by software developers for optimizing a system‘s source code. This website provides descriptions of all heuristics and can assist in selecting the subset of heuristics that will benefit the system's maintainability and/or performance the most, while minimizing the negative side-effects.

2. Maintainability and performance

We used the NFR framework to analyze the maintainability and performance qualities. Sections 2.1 and 2.2 describe the *softgoal interdependency graphs* that we built for maintainability and performance respectively. Section 2.3 explains how the qualities of maintainability and performance can be satisficed in a system by implementing selected *heuristics* in source code.

2.1 Decomposing maintainability into softgoals

Maintainability is defined as the characteristics of the software, its history, and associated environments that affect the maintenance process and are indicative of the amount of effort necessary to perform maintenance changes. It can be measured as a quantification of the time necessary to make maintenance changes to the product. [2, 5] Figure 1 shows the full *softgoal interdependency graph* for maintainability. This graph attempts to illustrate the specific software attributes that affect maintainability.

In some cases there exist conflicting views of how attributes affect the maintainability of software.

It is important to note that in this work we only describe softgoals relevant to the source code of the target system. It is possible to identify softgoals irrelevant to source code that contribute towards satisficing maintainability. Such softgoals may be related to other environmental factors, such as `Management' or the `Operational Environment'. [2] However, identifying such heuristics would require knowledge about the specific environment in which the software system is embedded, and thus describing them is outside the scope of our work.

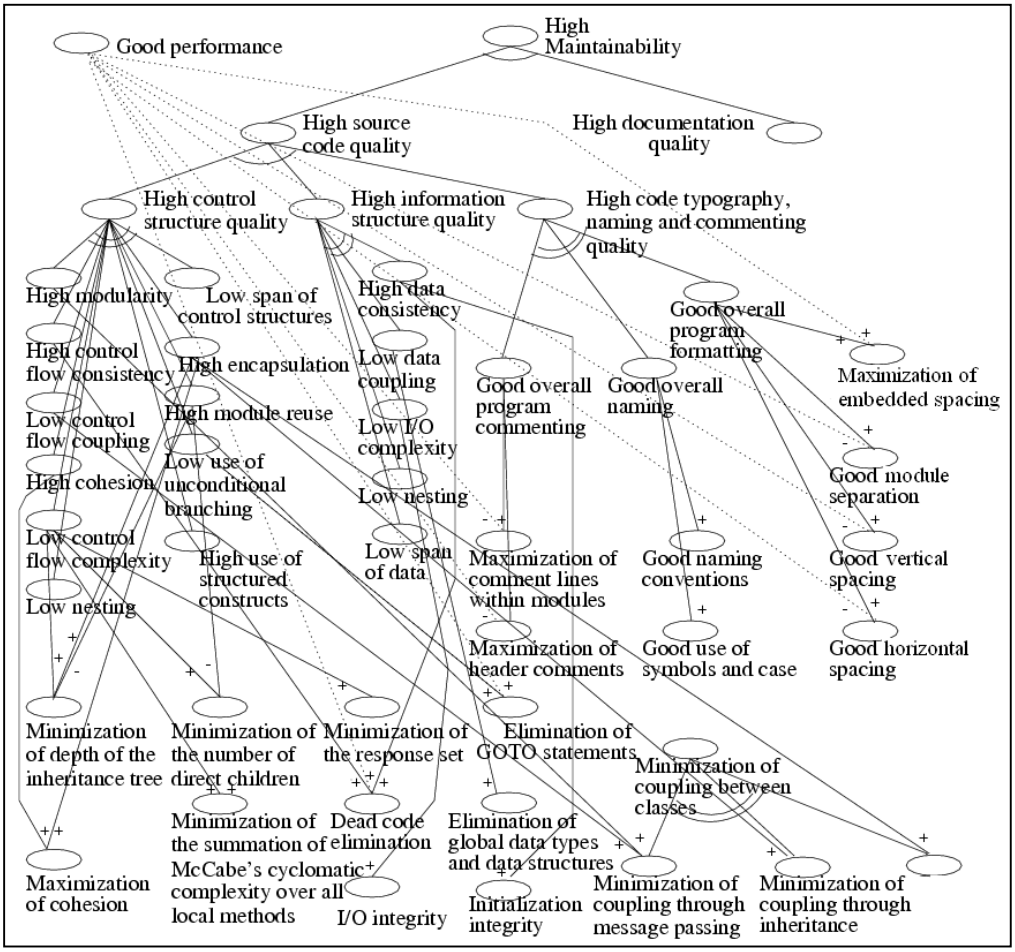

Figure 1 - Maintainability softgoal interdependency graph, including all heuristics.

The maintainability quality is decomposed into the softgoals

- high source code quality [2], and
- high documentation quality [3].

This decomposition is shown in Figure 1. Both softgoals of high source code and documentation quality must be satisficed for a system to have high maintainability. This is referred to as an *AND* contribution of the offspring softgoals towards their parent softgoal, and is shown by grouping the interdependency lines with an arc. The rationale behind this *AND* contribution is that a software system with clear source code but bad documentation will be hard to maintain, since maintainers will need to study requirements and design documents in order to understand how the system works. A software system with clear documentation but badly-written code will also be hard to maintain, since maintainers will need to understand how the source code works in order to make changes to it.

The *high source code quality* softgoal is further decomposed into the softgoals

- high control structure quality [2],
- high information structure quality [2], and
- high code typography, naming and commenting quality [6, 7].

This decomposition is shown in Figure 1. As shown, this is also an *AND* contribution, i.e. all three sub-softgoals must be satisficed to achieve the *high source code quality* softgoal. The rationale behind this *AND* contribution is that source code will be hard to understand if it is badly commented, or is laid out in a bad manner (typography qualities). But source code will also be hard to understand if characteristics such as modularity, encapsulation or cohesion have not been achieved (control structure and information structure qualities).

2.2 Decomposing performance into softgoals

As with maintainability, we also view performance as a *softgoal* (see Section 1.2) that can be broken down into more specific softgoals. Figure 2 shows the full *softgoal interdependency graph* for performance.

The *high performance* quality can be decomposed into softgoals

- good time performance [4], and
- good space performance [4].

This decomposition is shown in Figure 2. As shown, this is an *AND* contribution, i.e. both softgoals must be satisficed to achieve the *performance* softgoal. The rationale behind this *AND* contribution is that both softgoals of good time and space performance must be satisficed for a system to achieve good performance. It is inconceivable for a system that is fast but makes bad memory-utilization to be characterized by good performance. It is also inconceivable for a system that makes good memory-utilization but is slow to be characterized by good performance. Thus, software developers must try to satisfice both softgoals in a system. If there is a tradeoff involved between achieving both of them then that tradeoff must be balanced.

In turn, the *good space performance* softgoal can be decomposed into the following sub-softgoals:

- low main memory utilization, and
- low secondary storage utilization.

This decomposition is shown in Figure 2. As shown, this is also an *AND* contribution, i.e. both sub-softgoals must be satisficed to achieve the *good space performance* softgoal. The rationale behind this *AND* contribution is that the system may be stored

either in main memory or in secondary storage, and the term "space" is used interchangeably to refer to both types of storage.

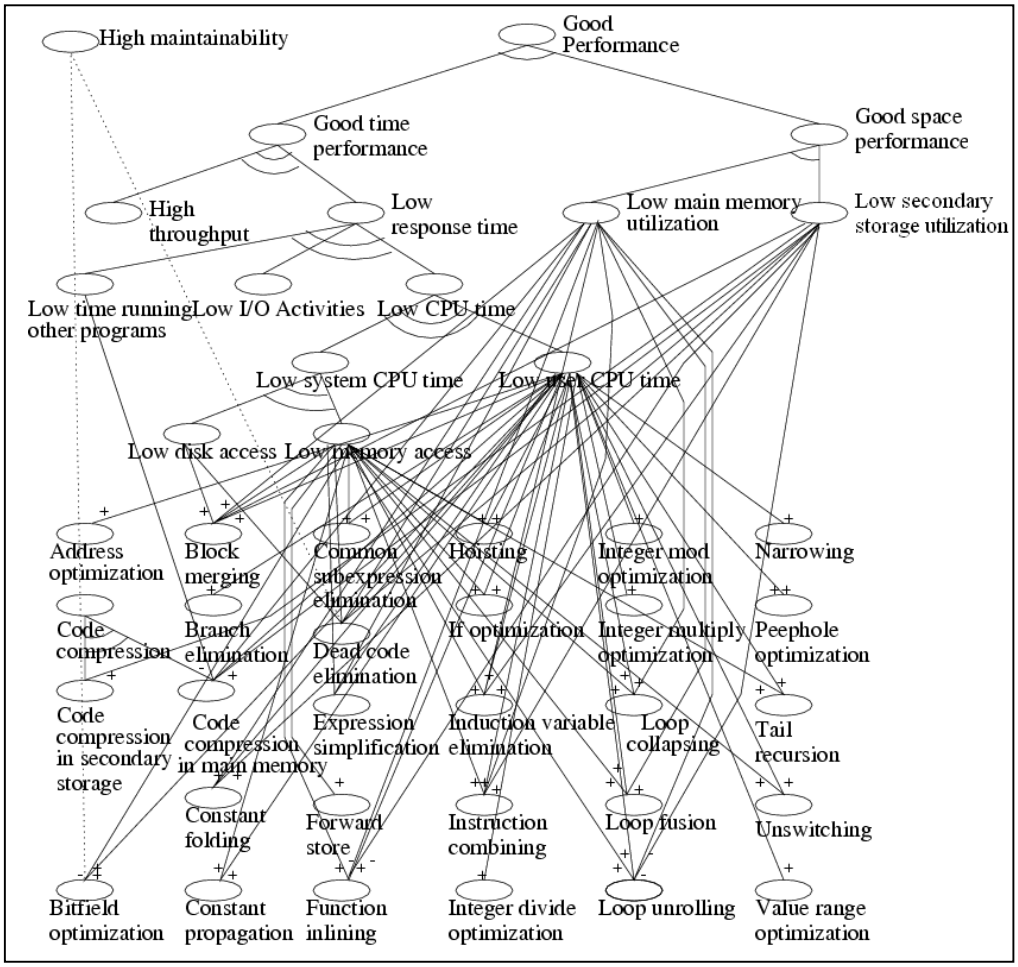

Figure 2 - Performance softgoal interdependency graph, including all heuristics.

2.3 Identifying heuristics to achieve software quality

Up to now we have been providing more precise definitions for the broad qualities of maintainability and performance. There exist heuristics that can be implemented in source code to satisfice the software quality requirements of high maintainability and performance. The NFR framework treats these heuristics as *softgoals* because this allows developers to decompose heuristics into more specific ones. Heuristics are often referred to as *operationalizing softgoals*. Like other softgoals, heuristics also make a contribution towards one or more parent softgoals. In this case the

contribution types are positive/negative. This is represented with a *"+", "++", or "-", "--"* symbol. [1]
As shown in Figure 1, an example of a heuristic that can be implemented in a system's source code to contribute towards satisficing the maintainability quality requirement is *elimination of global data types and data structures*. This means to make global data types and data structures local. Implementing this heuristic makes a "++" contribution towards meeting the *low data coupling* softgoal.
As shown in Figure 2, an example of a heuristic that can be implemented in a system's source code to contribute towards satisficing the performance quality requirement is *integer divide optimization*. This means to replace integer divide instructions with power-of-two denominators and other bit patterns with faster instructions, such as shift instructions. Implementing this heuristic makes a "+" contribution towards meeting the *low user CPU time* softgoal.
As shown in Figures 1 and 2, some heuristics such as *dead code elimination* and *elimination of GOTO statements* contribute to both maintainability and performance and they might affect one quality positively while affecting the other negatively. We examine these conflicting heuristic contributions towards different qualities in the next section.

3. Maintainability and performance measurements

We performed maintainability and performance optimization activities, by implementing different heuristics at the source code level. Each optimization activity that we performed corresponds directly to a specific heuristic that is shown in Figures 1 and 2. We evaluated the effect of applying each optimization heuristic on the maintainability and performance of the source code. For each optimization activity, a set of maintainability metrics models were applied to the source code, both before and after the optimization activity took place.
The C++ source code of two different software systems was modified for our experiments; WELTAB, an election tabulation system, and the AVL GNU tree and linked list libraries. Both systems were originally written in C, but a reengineering tool was used to migrate the procedural C code to the object-oriented C++ language. The primary reason for reengineering WELTAB and AVL from C to C++ was our desire to produce object-oriented code that was of very low quality. This low quality was desirable for our experiments, because it gave us many opportunities to improve the source code by implementing optimization heuristics.

3.1 Maintainability metrics models

In order for maintenance processes to be improved and for the amount of effort expended in software maintenance activities to be reduced, it is first necessary to be able to measure software maintainability. [8] Certain maintainability metrics were extracted from the WELTAB and AVL C++ source code to evaluate the effects of optimizations. In each case the metrics were extracted automatically using DATRIX, a tool for assessing the quality of C and C++ source code.

MI1 is computed as: *125 - 10 * LOG(avg-E)*

The term *avg-E* is the *average Halstead Volume V per module.*

MI2 is computed as: *171 - 5.44 * ln(avg-E) - 0.23 * avg-V(G) - 16.2 * ln(avg-LOC) + 50 * sin(sqrt(2.46 * (avg-CMT / avg-LOC)*
The coefficients are derived from actual usage. Avg-E is the average Halstead Volume V per module. Avg-V(G) is the average extended McCabe's cyclomatic complexity per module. Avg-LOC = the average count of lines of code (LOC) per module. Avg-CMT = average percent of lines of comments per module

MI3 is computed as: *171 -3.42*ln(avg-E) - 0.23*avg-V(G) - 16.2*ln(avg-LOC) + 0.99*avg-CMT*
The coefficients are defined above.

3.2 A study of the optimization activities

The analysis of the maintainability and performance metrics was performed on nine different source code optimization heuristics, as described below. Five of these heuristics focused on improving maintainability and the other four focused on improving performance.
Some of these activities were applied to WELTAB only, others to AVL only, and others to both systems. We first extracted maintainability and performance metrics on the original WELTAB and AVL C++ source code before any of the optimization activities took place. After each distinct optimization activity took place, we extracted maintainability and performance metrics on either WELTAB or AVL or both. Detailed results are described next for each optimization.

Elimination of GOTO statements. The objective of this maintenance optimization activity was to minimize the number of GOTO statements in WELTAB. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added and no defects were fixed.
The maintainability measurements taken on the new optimized version of WELTAB are shown in the Table below.

Metric	Pre-value	Post-value
M1	71.9263	71.6085
M2	36.6910	35.4542
M3	61.3768	60.2877

All the Maintainability Indexes (MIs) decreased. These descreases can be attributed to the fact that all Halstead's metrics, McCabe's Cyclomatic Complexity and lines of code (variables that affect the MIs) increased. It is important to note that maintainability did get improved by eliminating GOTO statements. Elimination of GOTO statements is the only way to minimize the number of unconditional branches in source code. Decreasing the number of unconditional branches is a key factor in improving maintainability, as it can assist a maintainer in understanding the source code of a system. [2]
The performance measurements showed that performance was improved in some cases and was affected negatively in other cases. Thus, the results do not provide

sufficient evidence that elimination of GOTO statements affects performance in a specific way.

Dead Code Elimination. The objective of this maintenance optimization activity was to eliminate dead code that was unreachable or that did not affect the program. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added and no defects were fixed.

It is important to note that the original WELTAB C++ source code contained a large amount of dead code. It cannot be certain that all dead code was eliminated. However, after dead code was eliminated on some source files, the size of the files decreased by almost half their original size. This fact alone points out the importance of dead code elimination, not only for maintainability purposes, but also for space performance purposes.

This heuristic was implemented in WELTAB only. The maintainability measurements taken on the new optimized version of WELTAB are shown in the Table below. As can be seen, *dead code elimination* had as a result that maintainability was affected positively in the optimized system.

Metric	Pre-value	Post-value
M1	71.9263	77.2713
M2	36.6910	56.6653
M3	61.3768	78.8650

Some of the performance measurements taken are shown in the Table below. As we can see, performance was improved after applying this heuristic.

WELTAB Function	Pre-performance	Post-performance
weltab	0.69	0.61

Elimination of Global Data Types and Data Structures. The objective of this maintenance optimization activity was to turn global data types and data structures to local. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added and no defects were fixed.

This heuristic was implemented in WELTAB only. The maintainability measurements taken are shown in the Table below. All measurements show an increase in maintainability after eliminating global data types and data structures.

Metric	Pre-value	Post-value
M1	71.9263	71.9391
M2	36.6910	36.7616
M3	61.3768	61.4414

Some of the performance measurements taken are shown in the Table below. As we can see, performance was affected negatively after applying this heuristic.

WELTAB Function	Pre-performance	Post-performance
weltab	0.78	0.79

Maximization of Cohesion. The objective of this maintenance optimization activity was to split a class with low cohesion into many smaller classes, each of which has higher cohesion. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.
This heuristic was implemented in AVL only. The maintainability measurements taken are shown in the Table below. All measurements show an increase in maintainability after maximizing cohesion.

Metric	Pre-value	Post-value
M1	70.43	76.32
M2	36.22	55.32
M3	61.43	73.67

Some of the performance measurements taken are shown in the Table below. As we can see, performance was affected negatively after applying this heuristic.

AVL Function	Pre-performance	Post-performance
SampleRec	0.67	0.69

Minimization of Coupling Through ADTs. The objective of this maintenance optimization activity was to eliminate variables declared within a class, which have a type of ADT that is another class definition. This optimization falls into the category of perfective maintenance since the software environment was not changed, no new functionality was added, and no defects were fixed.
This heuristic was implemented in AVL. The maintainability measurements taken are shown in the Table below. All measurements show an increase in maintainability after this optimization.

Metric	Pre-value	Post-value
M1	76.86	79.31
M2	98.77	102.67
M3	108.44	111.45

The performance measurements taken are shown in the Table below. As we can see, this heuristic affected performance negatively.

AVL Function	Pre-performance	Post-performance
Ubi_cacheRoot	0.67	0.68
Ubi_idbDB	0.56	0.58

Hoisting and Unswitching. The objective of this performance optimization activity was to optimize run-time performance by minimizing the time spent during FOR loops, by moving loop-invariant expressions out of FOR loops and transforming

a FOR loop containing a loop-invariant IF statement into an IF statement containing two FOR loops..
This heuristic was implemented in WELTAB only. The maintainability measurements taken are shown in the Table below. All MIs decreased. These descreases can be attributed to the fact that all Halstead's metrics and lines of code (variables that affect the MIs) increased. Thus, *Hoisting and Unswitching* affected maintainability negatively in the optimized system.

Metric	Pre-value	Post-value
M1	71.9263	71.9256
M2	36.6910	36.6757
M3	61.3768	61.3618

The performance measurements taken are shown in the Table below. As we can see, performance was improved after applying the heuristic.

WELTAB Function	Pre-performance	Post-performance
Report-canv	0.32	0.28
Baselib-smove	0.83	0.69

Address Optimization. The objective of this performance optimization activity was to fit all the global scalar variables of WELTAB in a global variable pool. Then, each of the global scalar variables gets accessed via one pointer and an offset, instead of via constant address. This way more expensive load and store sequences are avoided and code size is reduced.
This heuristic was implemented in WELTAB only. The maintainability measurements taken are shown in the Table below. All MIs decreased. Thus, *Address Optimization* affected maintainability negatively in the optimized system.

Metric	Pre-value	Post-value
M1	71.9263	71.8982
M2	36.6910	36.6559
M3	61.3768	61.3547

Some of the performance measurements taken are shown in the Table below. As we can see, performance was improved after applying the heuristic.

WELTAB Function	Pre-performance	Post-performance
cmprec-vfix	0.98	0.87
report-chead	0.76	0.63

Integer Divide Optimization. The objective of this performance optimization activity was to replace integer divide expressions with power-of-two denominators with faster integer shift instructions.
This heuristic was implemented in both WELTAB and AVL. The maintainability measurements taken on WELTAB are shown in the Table below. All MIs decreased

slightly after this heuristic was applied. Thus, this optimization affected the maintainability negatively.

Metric	Pre-value	Post-value
M1	71.9263	71.9256
M2	36.6910	36.6902
M3	61.3768	61.3763

The performance measurements taken on WELTAB are shown in the Table below. As we can see, performance was improved after applying this heuristic.

WELTAB Function	Pre-performance	Post-performance
wcre-showdone	0.76	0.65
weltab-showdone	0.33	0.28

Function Inlining. The objective of this performance optimization activity was to eliminate the overhead associated with calling and returning from a function, by expanding the body of the function inline.
This heuristic was implemented in both WELTAB and AVL. The maintainability measurements taken on WELTAB are shown in the Table below. All MIs decreased after this heuristic was applied. These decreases can be attributed to the fact that all Halstead's metrics and lines of code increased. Thus, this optimization affected the maintainability negatively.

Metric	Pre-value	Post-value
M1	71.9263	71.4982
M2	36.6910	35.5612
M3	61.3768	60.4460

The performance measurements taken on WELTAB are shown in the Table below. As we can see, performance was improved after applying this heuristic.

WELTAB Function	Pre-performance	Post-performance
Weltab-poll	0.81	0.42
Weltab-spol	0.32	0.23

4. Discussion

For any potential optimization heuristic, a software developer should examine the number of source code locations to which the heuristic can be applied, and the chances that these source code locations will be maintained during the maintenance process (for a maintainability optimization heuristic) or executed during run-time (for a performance optimization heuristic). The *80-20* rule is often used to describe such situations [4]. This rule states that 20% of the source code will be executed 80% of the time; and similarly that 20% of the source code will be maintained 80% of the time. Thus, in selecting the best combination of optimization heuristics, a developer

should attempt to select heuristics that can be applied to many source code locations falling under the 80-20 category.
The set of heuristics, selected for implementation, must be the ones that will benefit the system the most, by maximizing the ratio of gains to losses. When choosing a set of heuristics to be implemented in the target system, an *evaluation procedure* can be used to determine the degree to which each top-level quality requirement (i.e. maintainability) will be achieved.
In the NFR Framework, the heuristics that are chosen to be implemented in the target system are indicated by "√". On the other hand, rejected candidates are represented as "**X**". Heuristics for which a decision has not been made are simply left blank. Figure 3 shows examples of these selections as check-marks ("√") inside the nodes.
The developer has to evaluate the precise impact of the selected heuristics on top-level quality requirements (i.e. maintainability) to find out if the top-level quality requirements are achieved or not. The evaluation process can be viewed as working bottom-up, starting with bottom leaves of the graph representing heuristics. The evaluation process works towards the top of the graph, determining the impact of offspring softgoals on parent softgoals. This impact is represented by assigning labels (**"√"** and "**X**") to the higher-level parent softgoals. The impact upon a parent softgoal is computed from the contributions that all the offspring softgoals make towards it.

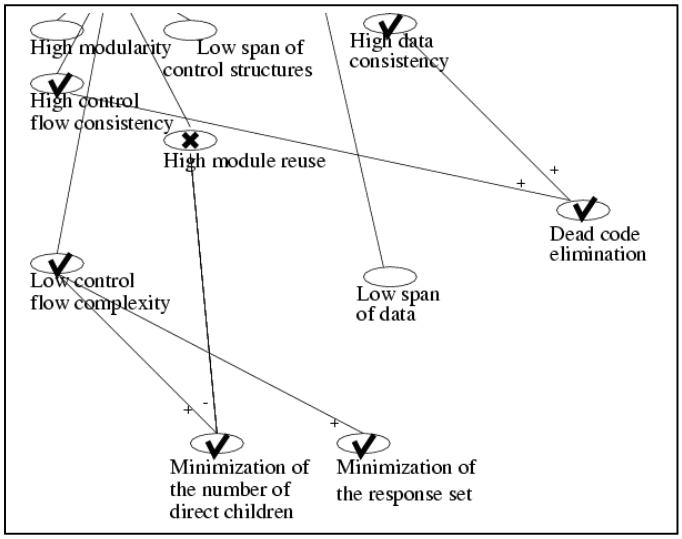

Figure 3 - Selecting among alternative combinations of heuristics.

As shown in Figure 3, the heuristic *minimization of the number of direct children* that is satisficed (**"√"**) makes a negative contribution towards its parent softgoal *high module reuse*, which is denied ("**X**"). On the other hand, it makes a positive contribution towards *low control flow complexity*, which is satisficed (**"√"**). The heuristic *dead code elimination* that is satisficed (**"√"**) makes a positive contribution towards its parent softgoals *high control flow consistency* and *high data consistency*. Thus, both softgoals are satisficed (**"√"**).

5. Conclusions

Our framework can be viewed as a generic methodology for selecting the set of optimization heuristics that will improve the system's software quality the most, while minimizing negative side effects. The major contributions of this work include using the *NFR framework* to model two particular software qualities, maintainability and performance. We identified and described many heuristics that affect these software qualities and that can be implemented in a target system's source code. We conducted experiments by implementing some of the heuristics in two medium-sized software systems and then collecting measurements. Finally, we presented an evaluation procedure for evaluating the effect of heuristics on software quality.

References

[1] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *NonFunctional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[2] J. R. Hagemeister, "A Metric Approach to Assessing the Maintainability of Software", Master's thesis, University of Idaho, Moscow, Idaho, 1992.

[3] J. Arthur and K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators", in *Proceedings Conference on Software Maintenance*, pp. 40-49, IEEE CS Press, 1989.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, San Mateo, CA, 1990).

[5] IEEE, *Standard Glossary of Software Engineering Terminology*, 1990.

[6] R. M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs*, (Addison Wesley, 1989)

[7] P. W. Oman and C. R. Cook, Typographic Style is More than Cosmetic, Communications of the ACM (CACM)", Vol.33, pp. 506—520, Communications of the ACM (CACM), 1990.

[8] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", in *Proceedings 1995 International Conference on Software Maintenance,* pp. 295-303, IEEE CS Press, 1995.