Information and Software Technology xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Source code optimization using equivalent mutants

Jorge López*[a], Natalia Kushik[a], Nina Yevtushenko[b]

[a] SAMOVAR, CNRS, Télécom SudParis, Université Paris-Saclay, 9 rue Charles Fourier, Évry 91000, France
[b] Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25 Alexander Solzhenitsyn street, Moscow 109004, Russia

ARTICLE INFO

Keywords:
Program / code optimization
Mutation (software) testing

ABSTRACT

Context: A *mutant* is a program obtained by syntactically modifying a program's source code; an *equivalent mutant* is a mutant, which is functionally equivalent to the original program. Mutants are primarily used in *mutation testing*, and when deriving a test suite, obtaining an equivalent mutant is considered to be highly negative, although these equivalent mutants could be used for other purposes.

Objective: We present an approach that considers equivalent mutants valuable, and utilizes them for *source code optimization*. Source code optimization *enhances* a program's source code preserving its behavior.

Method: We showcase a procedure to achieve source code optimization based on equivalent mutants and discuss proper *mutation operators*.

Results: Experimental evaluation with Java and C programs demonstrates the applicability of the proposed approach.

Conclusion: An algorithmic approach for source code optimization using equivalent mutants is proposed. It is showcased that whenever applicable, the approach can outperform traditional compiler optimizations.

1. Introduction

Source code optimization is a process which *enhances* a program's source code, in order to obtain a functionally *equivalent* program, i.e., a program which computes the same solution for the same problem but, possesses better non-functional aspects. Traditionally, source code optimization techniques are implemented on compilers [1].

Program mutants are used in mutation testing [2], a software testing technique whose main idea is to modify the original source code to obtain a *mutant* that should be later distinguished from the original program by a test case. The program modification is performed using a *mutation operator*; a mutation operator performs changes to the original source code. When applying a mutation operator, an equivalent program called an *equivalent mutant* can be obtained. Mutation testing attempts to detect and avoid equivalent mutants [3]. We note that detecting equivalent mutants using compiler optimizations is well established [4]. However, to the best of our knowledge, the first publication where *a novel use of equivalent mutants* is discussed, appeared recently [5]; the authors show that equivalent mutants can be used for static anomaly detection, e.g., to detect if the mutated code possesses better readability, better execution time, etc. However, the authors do not study nor outline a procedure where mutation operators are used for source code optimization.

Equivalent mutants can provide an optimized source code in terms of its (program/binary) execution time and other aspects. However, to effectively use the software mutation technique for source code optimization, several questions should be addressed: what are the mutation operators which can provide such optimizations? how to apply such mutation operators for optimization purposes? what is the benefit of the mutation-based source code optimization compared to traditional source code optimization? This paper is devoted to answer these questions; further, we perform preliminary experiments with a mutation software, μJava [2], which showcase the applicability and effectiveness of the proposed approach (Section 3).

2. Equivalent mutants for source code optimization

Given a (computer) program $\mathcal{P}$, we denote $\mathcal{S}_{\mathcal{P}}$ its associated source code. $\mathcal{P}$ is obtained from $\mathcal{S}_{\mathcal{P}}$ through a proper compilation process, i.e., a function C: $\Sigma^* \mapsto \{0, 1\}^*$ that maps a program's source code (a string over a particular programming language alphabet Σ) into a binary (or executable) code, i.e., $\mathcal{P} = C(\mathcal{S}_{\mathcal{P}})$. We denote the set of all possible inputs for $\mathcal{P}$ as I; correspondingly, O is the set of all possible outputs of $\mathcal{P}$. An input sequence is denoted as $\alpha \in I^*$; correspondingly, an output sequence $\beta \in O^*$ is the program's output response to this sequence, denoted as $out(\mathcal{P}, \alpha)$[1]. We consider a program's running time under a

* Corresponding author.
E-mail addresses: jorge.lopez@telecom-sudparis.eu (J. López), natalia.kushik@telecom-sudparis.eu (N. Kushik), evtushenko@ispras.ru (N. Yevtushenko).

[1] We assume that the program is deterministic and, therefore, such output is unique.

https://doi.org/10.1016/j.infsof.2018.06.013
Received 13 December 2017; Received in revised form 22 June 2018; Accepted 25 June 2018
0950-5849/ © 2018 Elsevier B.V. All rights reserved.

given input sequence α, in a common and predefined architecture, measured in milliseconds (*ms*) and denoted as $t(\mathcal{P}, \alpha)$. Correspondingly, we denote the *overall running time of a program* $\mathcal{P}$ with respect to a set of input sequences M as $\tau = \sum_{\alpha \in M} t(\mathcal{P}, \alpha)$.

A program $\mathcal{P}$ is *M-equivalent* to $\mathcal{P}'$ (written $\mathcal{P} \overset{M}{\equiv} \mathcal{P}'$) if $\forall\, \alpha \in M\ out(\mathcal{P}, \alpha) = out(\mathcal{P}', \alpha)$. We focus on program *M*-equivalence due to the fact that in the general case, the problem of checking the equivalence of two arbitrary programs is undecidable. However, in some cases, equivalence with respect to a finite set of inputs implies complete functional equivalence when having a behavior model [6]. Furthermore, many programs are used only within a context, receiving only a subset of possible (defined) inputs, or the program is only developed for a subset of inputs. Likewise, it is well-known that regression tests (a finite subset of the program inputs) are becoming an industry standard, and they somehow guarantee that a new version (including an optimized one) behaves as required.

A source code optimization process is a function which receives a source code and produces a new (optimized) source code O: $\Sigma^* \mapsto \Sigma^*$. The obtained source code compiles to a functionally equivalent program with respect to an input set M, i.e., $C(\mathcal{S}_\mathcal{P}) \overset{M}{\equiv} O(C(\mathcal{S}_\mathcal{P}))$. As it is not possible to derive an algorithmic approach to compute the time complexity of a program, optimality is considered with respect to the overall running time of a program, i.e, $\sum_{\alpha \in M} t(C(O(\mathcal{S}_\mathcal{P})), \alpha) < \sum_{\alpha \in M} t(C(\mathcal{S}_\mathcal{P}), \alpha)$.

Arcaini et al. [5] showcased that mutants can be *better* than the original source code, including the case when the mutated source code has better time complexity than the original one. However, no discussion was performed on how equivalent mutants can be exploited. Therefore, the problem stated and solved in this paper is as follows: how can source code optimization be forced by the use of source code mutation? It is important to highlight that the use of source code mutants to enhance the source code's non-functional properties is limited in the literature; for a comprehensive survey on the subject the interested reader can refer to [7].

We assume that there exist certain mutation operators which are more likely to provide source code optimization due to their nature. Operators as statement deletion can optimize the source code by performing a *dead code elimination*, arithmetic operator replacement can optimize the source code by performing operators' *strength reduction*, etc. [1]. Nevertheless, compiler optimizations are likely to be more effective while performed on target by a compiler. Therefore, the question arises: are there any mutation operators that can produce source code optimizations which are different from the known compiler optimizations? Indeed, we collected the following set of mutation operators based on the method-level mutation operators of μJava [2]:

- Relational Operator Replacement (ROR): replaces relational operators with others, e.g., >= with >. In certain cases, avoiding to execute the code when the condition reaches equality can enhance the performance (as shown in [5]), for example, when searching for the maximum number within an array as shown in the following code snippet (hereafter Δ denotes the difference/replacement, i.e., the obtained mutant).

```
for (int i = 0; i < arr.length(); i++)
    if (arr[i] >= max)
Δ   if (arr[i] > max)
        max = arr[i];
```

- Shortcut Assignment Operator Replacement (ASR): replaces shortcut assignment operators with other shortcut assignment operators, e.g., += with *=. In certain cases, advancing faster in the progression can avoid the execution of loop cycles, for example, when working over the powers of a given number as shown in the following code snippet.

```
    for(int i = 1; i <= N; i+=3)
Δ   for(int i = 1; i <= N; i*=3)
        if(i > 0 && 1162261467 % i == 0)
            //If-body
```

- Arithmetic Operator Replacement (AOR): replaces arithmetic operators with others, e.g., from + to *; similar to ASR, AOR can help advancing faster in the progressions.

We are interested in the set of mutation operators that perform different optimizations from traditional compiler optimizations, and can be applicable to different programming languages. Let $\mu = \{ROR, ASR, AOR\}$ be the set of mutation operators of interest. This set can be always extended by adding other mutation operators that can also perform compiler optimizations. We aim at limiting the mutation operators to be considered in order to avoid deriving mutants that do not optimize the source code. Indeed, executing all mutants against the set of inputs *M* may take a very long time. However, we note that even if the optimization process takes more time than executing the original program once, the time investment can be worthy for widespread programs which may be executed in millions of devices, or systems for which critical components are executed millions of times. Furthermore, selecting the critical parts of the code to be optimized can aid to reduce the complexity of this approach.

We propose Algorithm 1 for source code optimization using equivalent mutants. Hereafter, *mutate* denotes a mutation function which takes the mutation operator and the source code to mutate as parameters, and produces a set of mutants of the corresponding type. The resulting optimizations depend on the set of inputs *M* on which the program is stimulated.

Algorithm 1 returns a source code which compiles to a program that is *M*-equivalent to the initial one. Therefore, for assuring the program equivalence, one can derive a set *M* of inputs as a complete/exhaustive test suite which guarantees that the original and optimized programs have the same behavior [6]. In fact, the more *precise* this set *M* is constructed, the higher is the guarantee of the equivalence between the optimized and the original programs.

3. Preliminary experimental results

As a simple case study, we chose the source code of an intricate Java function which given a binary string, returns its integer value. Note that in this source code there is no verification that the string is indeed binary, however, we do not focus on such enhancements. The source code is shown below.

```
static int b2tob10 (String binary) {
    String bin = new StringBuilder(binary).reverse().
        toString();
    int size = bin.length();

    if(bin.length() == 0)
        return 0;

    int pos = 1, i = 2, number = 0, count, aux;

    number += Integer.parseInt(bin.substring(0,1));

    while (i <= 1 << size - 1){
        aux = i;
        count = 0;
        while(aux > 0){
            count++;
            aux = aux & (aux - 1);
        }
        if (count > 1) {
            i +=2;
            continue;
        }

        number += Integer.parseInt(bin.substring(pos, ++pos
            )) * i;

        i+= 2;
    }

    return number;
}
```

Download English Version:

https://daneshyari.com/en/article/8953928

Download Persian Version:

https://daneshyari.com/article/8953928

Daneshyari.com