Quantitative Symbolic Non-Equivalence Analysis

Laboni Sarker
labonisarker@ucsb.edu
University of California, Santa Barbara

Tevfik Bultan
bultan@ucsb.edu
University of California, Santa Barbara

ABSTRACT

Equivalence analysis focuses on assessing whether different programs, or different versions of a program, exhibit identical behavior. While extensive research has been done on equivalence analysis, there is a lack of detailed and quantitative reasoning techniques for non-equivalence. In this paper we introduce quantitative symbolic non-equivalence analysis and evaluate its effectiveness on the EqBench [3] benchmark (the largest available benchmark for equivalence analysis), and demonstrate how it can be used for reasoning about the non-equivalence of different versions of C programs.

KEYWORDS

symbolic execution, quantitative analysis, non-equivalence analysis

ACM Reference Format:
Laboni Sarker and Tevfik Bultan. 2024. Quantitative Symbolic Non-Equivalence Analysis. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3691620.3695324

1 INTRODUCTION

The goal of code equivalence analysis is to identify whether two different programs or two different versions of the same program behave identically or not. Symbolic execution can capture behavioral characterization of a program and has been used for equivalence checking together with a variety of heuristics to optimize the process [1, 2, 7, 8, 10].

When we assert that two programs are functionally equivalent, we mean that for any identical input they will produce the same output [6, 8]. But, saying two programs are "non-equivalent" can correspond to different scenarios: (1) In the extreme case, two programs could yield differing outputs for all inputs, essentially rendering them non-equivalent across the entire input domain. (2) Alternatively, there might be specific inputs for which the programs produce dissimilar outputs, while for the rest of the inputs, their respective outputs could be identical.

Two programs are considered non-equivalent even when the non-equivalence arises for only one input from the whole domain. So, when we assert that two programs are non-equivalent, we are not providing an assessment of *how different* the two programs (or two different versions of one program) are. Non-equivalence can be seen as a wide spectrum that can not be comprehensively reasoned about by saying that two programs are "non-equivalent". This is because, unlike being "equivalent", being "non-equivalent" does not mean non-equivalence over the whole input domain. Even though there is a lot of prior work on equivalence analysis and quantitative software analysis [4, 5], there is not that much work on quantitative assessment and reasoning of non-equivalence [9].

For any application of program equivalence, techniques for refined non-equivalence analysis and quantitative equivalence analysis would provide additional insights for the cases where two programs are assessed to be non-equivalent. In this paper, we analyze non-equivalence of different versions of programs using the EqBench [3] dataset for C programs. We demonstrate that quantitative symbolic analysis can be used to provide valuable information about the non-equivalent cases in the EqBench benchmark.

2 OUR APPROACH

Figure 1 shows two different versions of a C program from EqBench [3]. The two programs are marked as semantically "equivalent" there, even though they are syntactically different. Note that integer overflow can lead to undefined behaviors in C programs. For the programs in Figure 1, client functions are syntactically similar and the lib functions differ on the first branch. The client function calls the lib function with different values by comparing variable `x`. So, the two programs will be equivalent if they never reach the first branch in lib function which is ensured while calling lib function from both branches of client function. But interestingly here, the multiplication of positive `x+1` with 5 in the second return can result in a negative value due to integer overflow and the same can happen in the first branch of client function as well. Due to these integer overflows, the programs will generate two very different outputs. Classifying these programs as either equivalent or not equivalent will not provide sufficient information about their behaviors. For example, we may want to know more about the particular set of values of `x` for which the programs will be equivalent, or the set of values for which their outputs will differ.

Using our quantitative symbolic non-equivalence analysis, we can infer that the two program versions are equivalent with respect to C semantics only when the input `x` satisfies the following constraint: $-429496729 \leq x \leq 429496728$ given that the input is a 32 bit integer. Our analysis can compute this range of equivalence and also compute the percentage of input values in the input domain for which the two versions of the program are equivalent. In this case, two versions are equivalent for 59% of the input values.

Another example shown in Figure 2 illustrates a non-equivalent case from the benchmark, and our analysis can capture that they behave equivalent when $x \geq 11$, which corresponds to 50% of the input domain.

This material is based on research supported by ONR Contract No. N6833523C0019, Oceanit Laboratories Award #SB230168, by NSF grants CCF-2008660, and CCF-1901098, and by DARPA grant N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10.
https://doi.org/10.1145/3691620.3695324

```
Version 1                          Version 2
int lib(int x) {                   int lib(int x) {
if(x < 5) return 5;                if(x < 0) return 0;
else      return x;}               else      return x;}

int client(int x){                 int client(int x){
if (x < 0)                         if (x < 0)
return -lib((-x)*5)/5;             return -lib((-x)*5)/5;
else                               else
return lib((x+1)*5)/5-1;}          return lib((x+1)*5)/5-1;}
```

Figure 1: Two versions of equivalent C programs of the CLEVER/ltfive dataset from EqBench [3] benchmark

```
Version 1                          Version 2
int lib(int x){                    int lib(int x){
if (x > 10) return 11;             if (x > 10) return 11;
else        return x;}             else        return x+1;}
```

Figure 2: Two versions of non-equivalent C programs of the CLEVER/oneN2 dataset from EqBench [3] benchmark

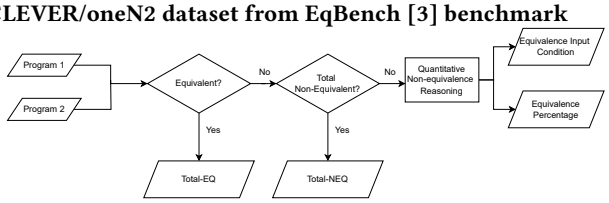

Figure 3: Workflow of non-equivalence analysis

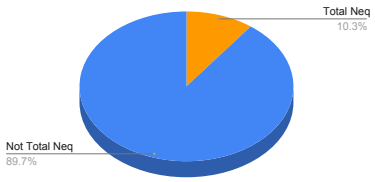

Figure 4: Total Neq Vs. Not Total Neq

The overall workflow of non-equivalence analysis approach we have been developing is shown in Figure 3. Given a pair of programs as inputs, we aim to determine whether they are equivalent using differential symbolic execution [8] by generating the program summaries. The symbolic summary for the programs shown in Figure 2, would be: $S_1 \equiv (x > 10 \wedge return = 11) \vee (x \leq 10 \wedge return = x)$ and $S_2 \equiv (x > 10 \wedge return = 11) \vee (x \leq 10 \wedge return = x + 1)$. We can check the equivalence by analyzing the equivalence constraint $S1 \Leftrightarrow S2$. If they are non-equivalent, we check whether the two programs are non-equivalent for the whole input domain, which we define as total non-equivalence. If they are totally non-equivalent, we do not need to go further for reasoning. Otherwise, we can further analyze the input domain for which the two programs are behaving differently. Quantitative reasoning provides us with the percentage of the input domain where the two programs behave equivalently, which can be done using projected model counting on input variables for the equivalence constraint.

3 RESULTS

To evaluate our technique for quantitative symbolic non-equivalence analysis, we used programs from the publicly available benchmark EqBench [3] for equivalence analysis. We have considered 76 C programs with the numeric domain, and 39 of them are non-equivalent. Of those 39, only 4 are totally non-equivalent, and the rest are not. Figure 4 shows the percentage of total non-equivalence versus the not total non-equivalence in the EqBench [3] benchmark.

Table 1: Quantitative Analysis for Non-Equivalence

Neq Program	Eq %	Eq Domain	Input Domain	# Param
loopmult20	99.999999906%	4294967292	4294967296	1
ltfive:Lib	49.999999883%	2147483643	4294967296	1
oneBound	49.999999720%	2147483636	4294967296	1
getSign2	99.999999976%	4294967295	4294967296	1
ltfive:Client	59.875488234%	2571632638	4294967296	1
oneN2	99.999999976%	4294967295	4294967296	1
dart	99.999999953%	1.84467E+19	1.84467E+19	2

Table 1 shows the total input domain and the percentage of input domain for which the programs are equivalent from the EqBench benchmark. The table shows that even though the programs are non-equivalent, they are not total non-equivalent. Rather, they show equivalence by different percentages. So, it is not enough just to deduce that two programs are non-equivalent, and we need to analyze input conditions for which they behave differently.

Table 2: Input Condition for Non-Equivalence

Neq Program	Input Condition of equivalence
loopmult20	x < 18 or x > 21
ltfive:lib	x > 4
ltfive:client	-429496729 <= x <= 429496728
oneN2	x!=2147483648
dart	(x <= 1290 and x >= -1290) or (y!=10 and y!=20)

Table 2 shows the input conditions of some programs from the EqBench for which they show non-equivalence. In this way, a quantitative symbolic non-equivalence analysis can help us provide refined results on the non-equivalence of the programs.

REFERENCES

[1] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries, Vol. 7976. https://doi.org/10.1007/978-3-642-39176-7_7

[2] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. 13–24. https://doi.org/10.1145/3368089.3409757

[3] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. 610–614. https://doi.org/10.1109/MSR52588.2021.00084

[4] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina Păsăreanu, and Willem Visser. 2014. Compositional Solution Space Quantification for Probabilistic Software Analysis. *ACM SIGPLAN Notices* 49 (06 2014). https://doi.org/10.1145/2594291.2594329

[5] Antonio Filieri, Corina Pasareanu, and Guowei Yang. 2015. Quantification of Software Changes through Probabilistic Symbolic Execution (N). 703–708. https://doi.org/10.1109/ASE.2015.78

[6] Benny Godlin and Ofer Strichman. 2010. Inference Rules for Proving the Equivalence of Recursive Procedures. *Acta Informatica* 45, 167–184. https://doi.org/10.1007/s00236-008-0075-2

[7] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. 441–451. https://doi.org/10.1145/3238147.3238178

[8] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pundefined-sundefinedreanu. 2008. Differential Symbolic Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 226–237. https://doi.org/10.1145/1453101.1453131

[9] Laboni Sarker. 2023. Quantitative Symbolic Similarity Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1549–1551. https://doi.org/10.1145/3597926.3605238

[10] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. 405–427. https://doi.org/10.1007/978-3-319-66706-5_20