Analysis of Unreachable Code using Static Code Analsysis

Lukas Braun

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Mai 2021

Advisor:

Dipl.-Ing. (FH) Dr. Josef Pichler

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, May 31, 2021

Lukas Braun

Contents

Preface

Abstract

1 Site

Kurzfassung

1 Site

Chapter 1

Introduction

Unreachable Code Detection nowadays is integrated in almost every available static code analysis software as well as Integrated Developments Environments and to some extent even compilers. Unreachable Code is defined as Code that can never be reached, because either the program flow ended prematurely or due to unsatisfiable path conditions. These types do not only increase the size of the program, but also increase the overall complexity of the sourcecode or may even lead to unwanted behavior and errors (e.g. goto-fail bug) [3].

1.1 Motivation

Using Static code analysis to find errors before compiling, building or rolling out software is a very essential part of the software development process. These tools are able to identify a wide range of Errors or make suggestions for adhering to a better style. Finding errors in sourcecode as it is written will not only make the program more robust, but also makes it more comprehensible, which leads to less time and energy needed during maintenence. Especially *Unreachable -, Unneccessary - and Dead Code* are a main source of incomprehensible sourcecode [7]. In some instances *Unreachable Code* may be intended due to extensibility of the program [5] and may even not be clear if it is intended or not. Before the break-through of the Object Oriented Paradigm many Programs were developed in an imperative, procedural way. Some Styleguides of that time suggested splitting files into so many smaller Files containing few Functions and/or procedures [8]. Interestingly the emerge of the Object Oriented Paradigm and its languages to the de-facto standard increased the precentage of *Unreachable - and Dead Code* [8]. In extreme cases Dead Code may even lead to the so called Anti-pattern Lava-Flow [7], which typically occurs when Dead Code will not be removed and has to be maintained, even tough it does not do much or even anything at all. Systems that undergo constant evolution, like Web Systems, are also prone to Lava-flow and Unreachable Code [2].

```
1     IF s_operationHour <> m_operationHour THEN
2         m_operationHour := s_operationHour;
3         IF s_operationHour = 0 THEN
4             RESET_ALARM(Name := er_service, SubID1 := m_iNumber);
5         END_IF;
6         RETURN;
7     END_IF;
8     // s_operationHour must be equal to m_operationHour
9     IF s_operationHour = 0 THEN
10        // unnecessary - already zero
11        m_operationHour := 0;
12        RESET_ALARM(Name := er_service, SubID1 := m_iNumber);
13    ELSE
14        // unreachable
15        IF s_operationHour <> m_operationHour THEN
16            s_operationHour := m_operationHour;
17        END_IF;
18    END_IF;
19
```

Figure 1.1: A minimal example containing unreachable code due to unsatisfiable conditions. The condition in line 15 is never reachable, since this case was already handeld in line 1 and the state of that variable did not change.

1.2 Technical foundation

Unreachable Code is often confused with similar defects and different definitions are used. It is often confused with *Dead Code*, but there are also other types besides these two.

- *Dead Code* is code that may be reachable, but has no effect on the result. An example of this would be unused variables whose value is the result of a calculation (also known as *Dead Variables*).
- *Unreachable Code* is code that will never be, executed because the flow of the program was interrupted. This either happens due to statements that interrupt the flow unconditionally (using break, exit, return, goto, et cetera) or conditionally (using if, while, for, et cetera).
- *Unnecessary Code* is code that may be deleted. *Unnecessary Code* may still be reachable, but serves no semantic purpose.

Analysing these kind if defects requires a representation called *Controlflowgraph* 1.2, which is a directed graph. This Graph consists of Blocks, which contain the statements. Blocks may contain more (or less) than one statement and usually end with either a condition or mark the end of the scope. Blocks are connected via edges. These blocks have a maximum of two Edges pointing away, when the last statement contains a condition, since a condition is either true or false. The first and last block are begin- and endblocks, which contain no statements and their only purpose is to mark the beginning or end of the graph.

It is worth mentioning that *Controlflow graphs* may be altered to make analysis

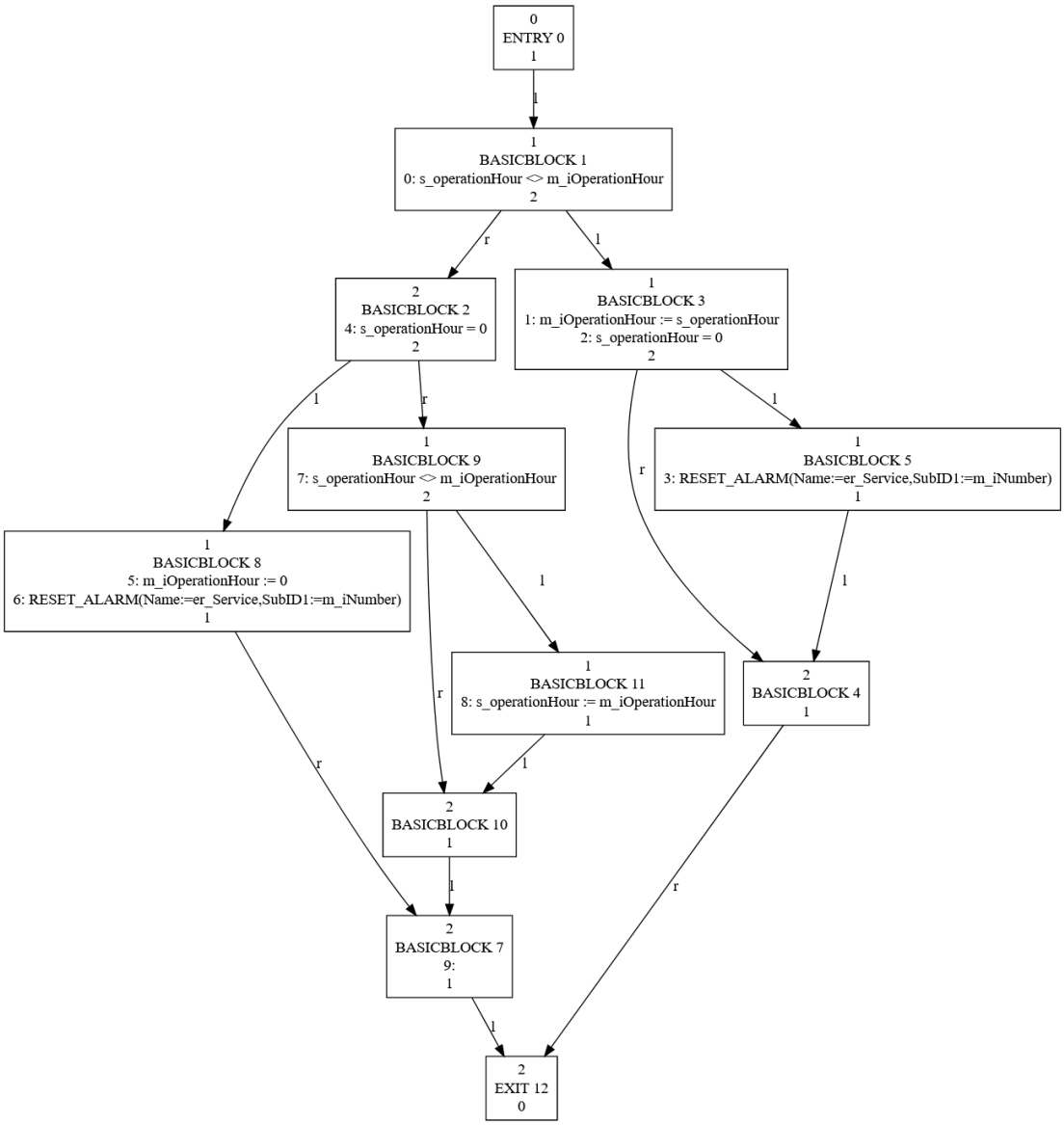


Figure 1.2: The constructed Controlflowgraph of 1.1. Edges are marked either with l - describing the path if the condition results true - or r. Loops (and sometimes gotos) may easily spotted since it is the only possibility for arrows pointing to former nodes.

easier, like *Single static Assignment*, which introduces many artifical variables to prevent redefinition of variables.

1.3 Project Overview

This thesis is written as a part of a research project between the *Software Competence Center Hagenberg GmbH* [11] and *Engel Austria GmbH* [9]. Engel produces injection molding machines, which are programmed using the *Kemro-IEC* language, a dialect

```
1     int x_0 ← 1;
2     do { x_1 ← φ(x_0, x_3);
3          b_0 ← (x_1 ≠ 1);
4          if( b_0 )
5              x_2 ← 2;
6          x_3 ← φ(x_1, x_2);
7     } while ( pred() )
8     return x_3;
9
```

Figure 1.3: This Problem [4] is the transformation to Single Static Assignment form, which does not check conditions and inserts a Phi-Function at line 6, even tough the condition in line 4 will never be satisfiable!

of the *IEC 61131-3* standard language. The project began as a joint work between the *Software Competence Center Hagenberg* and the *Johannes Kepler University Linz* [10; 6] within the competence centers programme COMET of the Austian Research Promotion Agency (FFG). Over the years multiple rules identifying a wide range of errors were implemented, for example detecting unguarded division where the divisor could potentially be zero, finding unused variables, highlighting high complexitiy of expressions, detecting Dead Code and other defects.

1.4 Objective of this thesis

Currently a rule for detecting unreachable Code is already in place, but is incomplete. The only kind of *Unreachable Code* that is detected when Statements stop the *Controlflow* unconditionally (e.g. break, return, exit, goto, etc.). The goal is to identify as much *Unreachable Code* as possible correctly while minimizing the number of reported false positives. A popular approch to finding *Unreachable Code* is converting a *Controlflow Graph* into *Single Static Assignment Form* and using *Constant Propagation* [4]. Afterwards Condtions may be evaluated if they only consist of constants. The result (either true or false) determines if a given part of sourcecode is always or never will be reachable (as mentioned before, only Conditions with constants only will be evaluated). This method is efficient and yields good results, but is not able to detect every error. Finding these kind of Errors 1.3 requires another approach, which does not rely on the transformation to the Single Static Assignment Form. Part of this thesis is about using another method for *Unreachable Code Detection*, which is able to find even more instances, while maintaining the same low reporting of false positives. The developed Method will make usage of a *Satisfiable modulo theory prover* (short: *SMT-Solver*) and *Symbolic Execution* [1]. The basis of this method will be - again - a *Controlflow Graph*.

At the end the advantages and disadvantages of both algorithms will be discussed and the results will be compared against each other in terms of accuracy and runtime.

Chapter 2

Terminology

2-3 Sites

2.1 Unreachable Code

2.2 Dead Code

2.3 Unused Code

2.4 Unnecessary Code

Chapter 3

Theoretical Foundation

7-10 Sites

3.1 Static Code Analysis

3.2 Abstract Syntax Tree

3.3 Controlflow Graph

3.4 Single Static Assignment Form

3.5 Constant Propagation

3.6 Satisfiable Modulo Theory

3.7 Kemro-IEC 61131-3

Chapter 4

Finding Unreachable Code using Single Static Assignment Form

10 Sites

4.1 Approach

4.1.1 Transform into Single Static Assignment Form

4.1.2 Apply Constant Propagation

4.1.3 Evaluation

4.2 Tests

Chapter 5

Finding unreachable code using a SMT-Solver

10 Sites

5.1 Approach

5.1.1 Transformation into SMTLib

5.1.2 Evaluation

5.2 Tests

Chapter 6

Conclusion

10 Sites

6.1 Comparison

6.1.1 Accuracy

6.1.2 Runtime

References

Literature

[1] Stephan Arlt, Zhiming Liu, and Martin Schäf. "Reconstructing Paths for Reachable Code". *Lecture Notes in Computer Science* (2013), pp. 431–446. URL: http://dx.doi.org/10.1007/978-3-642-41202-8_28 (cit. on p. 4).

[2] Hidde Boomsma, B. V. Hostnet, and Hans-Gerhard Gross. "Dead code elimination for web systems written in PHP: Lessons learned from an industry case". *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (Sept. 2012). URL: http://dx.doi.org/10.1109/icsm.2012.6405314 (cit. on p. 1).

[3] H.A. Boyes et al. "Trustworthy Software: lessons from 'goto fail' Heartbleed bugs". *9th IET International Conference on System Safety and Cyber Security (2014)* (2014). URL: http://dx.doi.org/10.1049/cp.2014.0970 (cit. on p. 1).

[4] Cliff Click and Keith D. Cooper. "Combining analyses, combining optimizations". *ACM Transactions on Programming Languages and Systems* 17.2 (Mar. 1995), pp. 181–196. URL: http://dx.doi.org/10.1145/201059.201061 (cit. on p. 4).

[5] Roman Haas et al. "Is Static Analysis Able to Identify Unnecessary Source Code?" *ACM Transactions on Software Engineering and Methodology* 29.1 (Feb. 2020), pp. 1–23. URL: http://dx.doi.org/10.1145/3368267 (cit. on p. 1).

[6] Herbert Prahofer et al. "Opportunities and challenges of static code analysis of IEC 61131-3 programs". *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)* (Sept. 2012). URL: http://dx.doi.org/10.1109/etfa.2012.6489535 (cit. on p. 4).

[7] Simone Romano et al. "A Multi-Study Investigation into Dead Code". *IEEE Transactions on Software Engineering* 46.1 (Jan. 2020), pp. 71–99. URL: http://dx.doi.org/10.1109/tse.2018.2842781 (cit. on p. 1).

[8] Amitabh Srivastava. "Unreachable procedures in object-oriented programming". *ACM Letters on Programming Languages and Systems* 1.4 (Dec. 1992), pp. 355–364. URL: http://dx.doi.org/10.1145/161494.161517 (cit. on p. 1).

Online sources

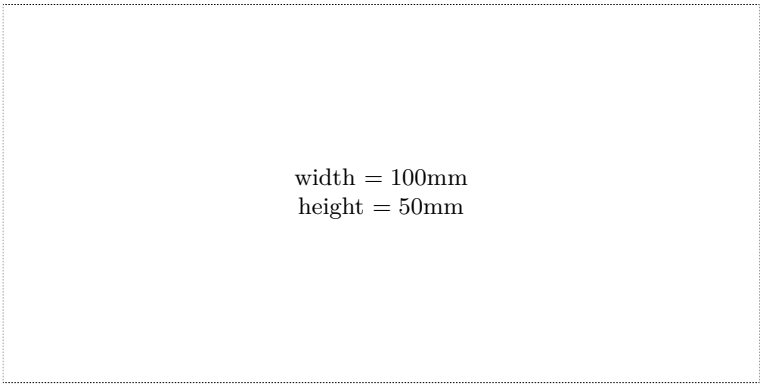
[9] *Engel Austria GmbH.* URL: https://www.engelglobal.com/de/at/unternehmen/daten-fakten.html (cit. on p. 3).

[10] *Johannes Kepler Universität*. URL: https://www.jku.at/en/masthead/ (cit. on p. 4).

[11] *Software Competence Center Hagenberg GmbH*. URL: https://scch.at/en/imprint (cit. on p. 3).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —