Formal Verification of Industrial Software and Neural Networks

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Marko Kleine Büning

geboren am 05. Juni 1992 in Paderborn

Tag der mündlichen Prüfung: 29. April 2022

Referiert von:

1. Prof. Dr. rer. nat. Carsten Sinz
   Karlsruher Institut für Technologie (KIT)
   Deutschland

2. Prof. Dr. rer. nat. Bernhard Beckert
   Karlsruher Institut für Technologie (KIT)
   Deutschland

Acknowledgments

I want to thank Carsten Sinz for the opportunity to pursue a PhD in his research group and for his advice and guidance throughout my time as a PhD student. I also want to thank Bernhard Beckert for his review of my dissertation and for our collaborations.

I am grateful for my current and former colleagues Philipp Kern, Safa Omri, Felix Kutzner, David Farago, Michael Kirsten, Mihai Herda, Simone Meinhart, Jonas Klamroth, Markus Iser, Jonas Schiffl, Wolfram Pfeifer and Florian Lanzinger for great discussions and exchange of ideas. Furthermore, I thank former students, Samuel Teuber, Moirtz Laupichler, Johannes Meuer and Tobias Knorr, for their work that I had the pleasure to supervise.

I would like to thank my godparents and good friends Bärbel and Klaus Meerkötter for their unconditional support and advice. I thank my family, especially my father Hans, my aunt Xue and my little brother Thomas for their love and support during my PhD. I thank Freya, your love and strength was an inspiration throughout.

This dissertation is dedicated to the memory of my friend Klaus and my mother Jinfang.

Abstract

Software has become an integral part of today's society. As software is increasingly applied in safety-critical areas, we must be able to rely on its correct and safe execution. Especially embedded software, for example in medical devices, cars or airplanes, must be thoroughly and formally verified. The software of such embedded systems can be divided into two kind of components. Traditionally engineered control software and machine learning methods as applied for image recognition or collision avoidance.

This dissertations aims to improve the state-of-the-art for the verification of two main components of modern embedded systems: software written in C/C++ and neural networks. Verification for both components is formally defined and novel verification approaches are presented.

Software Verification. The first goal of this dissertation is to develop a method to automatically check (embedded) industrial software written in C/C++ for runtime errors. Embedded software is different compared to firmware because of programming standards such as MISRA-C and AUTOSAR that have to be respected by developers. These standards reduce the complexity of software to minimize potential errors. For example, they restrict the possible number of loop iterations or the use of dynamic memory allocation. Another difference of industrial software is its size. A modern car comprises about 100 million lines of code and in the near future this number is expected to triple [112].

In this dissertation, we present a *modular bounded model checking approach* for automatic verification of industrial software. In a first step, the software to be examined is built and critical program locations, which could potentially lead to runtime errors, are automatically detected and marked. Furthermore, the proposed verification method can be configured and allows for preprocessing, making it particularly fast and user-friendly. In a next step, our approach performs a fully automatic modularization of a program through novel structural abstractions. The modularization is able to abstract the environment of functions as well as function calls in programs and thus leads to independently verifiable modules. However, the abstractions can lead to a large number of false positives, i.e. error messages where there is no error.

Therefore, a third and last step refines the introduced abstractions. This happens on the one hand by including parent functions and thus by an enlargement and division (in case of several parents) of modules. On the other hand two procedures are presented, which can create preconditions for functions, based on a found error. Calls to the erroneous function can be substituted in higher level functions with the generated precondition, leading to a refined but still scalable analysis. An evaluation of our approach shows an improvement of the state-of-the-art in bounded model checking and advantages compared to commercially available tools.

Verification of Neural Networks. The second goal of this dissertation is the verification of neural networks (NNs). The success of machine learning and especially of neural networks in many areas such as image recognition and decision management systems has resulted in an increased application of neural networks in safety-critical domains. This development explains the increasing interest in the formal verification of neural networks. Among the properties to be verified are robustness [138], functional properties [85] and equivalence [126] of neural networks. The equivalence of neural networks is especially important in the context of embedded systems. Control elements of cars or airplanes, for example, have limited resources for the execution of neural networks. In order to deploy the neural network under such energy and memory constraints, compressed representations of these networks are being used. However, there are still few methods that can verify that a compressed network is similar or equivalent to the original network.

Therefore, the second goal of this dissertation is to develop approaches proving that two networks are equivalent. Due to the stochastic nature of neural network training, two networks are rarely exactly equivalent. Therefore, in a first step, we introduce novel and relaxed notions of equivalence and argue about the general complexity of equivalence verification. We then present two verification methods for neural networks. Our first method is based on the exact encoding of networks and properties as mixed-integer linear programming problems (MILP). By encoding networks and properties as the optimization problem maximizing the difference between two NNs, we can show equivalence of two networks over clustered input or find corresponding counterexamples. Our second approach aims to improve scalability through abstractions. Therefore, we first adjust the existing geometric path enumeration (GPE) approach from one to multiple networks. The GPE approach propagates sets through neural networks and applies predefined transformation steps. Abstractions and other optimization methods improve the scalability of the GPE approach for equivalence verification. Our two approaches, as well as the novel definitions, improve the state-of-the-art. Our evaluation shows that the approaches produce strong results compared to other tools, in particular for equivalence verification of structurally different networks.

Zusammenfassung

Software ist ein wichtiger Bestandteil unsere heutige Gesellschaft. Da Software vermehrt in sicherheitskritischen Bereichen angewandt wird, müssen wir uns auf eine korrekte und sichere Ausführung verlassen können. Besonders eingebettete Software, zum Beispiel in medizinischen Geräten, Autos oder Flugzeugen, muss gründlich und formal geprüft werden. Die Software solcher eingebetteten Systeme kann man in zwei Komponenten aufgeteilt. In klassische (deterministische) Steuerungssoftware und maschinelle Lernverfahren zum Beispiel für die Bilderkennung oder Kollisionsvermeidung angewandt werden.

Das Ziel dieser Dissertation ist es den Stand der Technik bei der Verifikation von zwei Hauptkomponenten moderner eingebetteter Systeme zu verbessern: in C/C++ geschriebene Software und neuronalen Netze. Für beide Komponenten wird das Verifikationsproblem formal definiert und neue Verifikationsansätze werden vorgestellt.

Software Verifikation. Das erste Ziel dieser Dissertation ist es ein Verfahren zu entwickeln um (eingebettete) industrielle Software geschrieben in C/C++ automatisch auf Laufzeitfehler zu überprüfen. In sicherheitskritischen Systemen eingebettete Software unterscheidet sich zu anderer Software durch bestehende Programmierstandards wie MISRA-C und AUTOSAR in der Automobilbranche. Solche Standards schränken die Komplexität von Software ein, um mögliche Fehler zu minimieren. So werden zum Beispiel die mögliche Anzahl von Schleifendurchläufen oder der Einsatz von dynamischen Speicher begrenzt. Ein weiterer Unterschied ist die Größe von industrieller Software. Die Software in einem heutigen Auto umfasst ungefähr 100 Millionen Zeilen von Code und in naher Zukunft soll sich diese Zahl verdreifachen [112].

In dieser Dissertation stellen wir einen *modularen Bounded Model Checking Ansatz* für die automatische Verifikation von industrieller Software vor. In einer ersten Phase wird die zu untersuchende Software gebaut und es werden automatisch kritische Programstellen, die zu Laufzeitfehlern führen können, erkannt und markiert. Weiterhin werden Konfigurationsmöglichkeiten und Vorverarbeitung für eine schnelle und einfache Verifikation vorgestellt. Daraufhin präsentiert diese Dissertation eine vollautomatische Modularisierung eines Programmes durch hier eingeführte strukturelle Abstraktionen.

Die Modularisierung ist in der Lage die Umgebung von Funktionen sowie Funktionsaufrufe in Programmen zu abstrahieren und führt somit zu unabhängig verifizierbaren Modulen. Die Abstraktionen können jedoch zu einer großen Anzahl an Falschmeldungen führen. Daher verfeinert ein dritter und letzter Schritt die eingeführten Abstraktionen. Dies passiert zum einen durch einbinden von Elternfunktionen und somit durch eine Vergrößerung und Aufteilung (bei mehreren Eltern) von Modulen.

Zum anderen werden zwei Verfahren vorgestellt, welche Vorbedingungen für Funktionen, basierend auf einem gefundenen Fehler, erstellen können. Aufrufe der fehlerhaften Funktion können in höher stellten Funktionen mit der erzeugten Vorbedingung substituiert werden und führen somit zu einer verfeinerten aber immer noch skalierbaren Analyse. Eine Evaluation unseres Ansatzes zeigt eine Verbesserung der Stand der Technik; auch im Vergleich zu kommerziell eingesetzten Werkzeugen.

Verifikation Neuronaler Netze. Das zweite Ziel dieser Dissertation ist die Verifikation von Neuronalen Netzwerken. Der Erfolg von maschinellen Lernverfahren und insbesondere von Neuronalen Netzen in vielen Bereichen wie der Sprach- und Bilderkennung sorgt dafür, dass Neuronale Netze vermehrt in sicherheitskritischen Bereichen zur Anwendung kommen. Diese Entwicklung begründet die zuletzt verstärkte Forschung im Bereich der formalen Verifikation von neuronalen Netzen.

Der Einsatz in eingebetteten Systemen wie Steuerelementen von Autos oder Flugzeugen, schränkt jedoch die für die Netzte zur Verfügung stehenden Ressourcen ein. Solche Energie- und Speichereinschränkungen haben daher Kompressionsverfahren für Neuronale Netze hervorgebracht. Es gibt bisher jedoch noch wenige Verfahren, welche zeigen können, dass die originalen und kompensierten Netze gleiche Eigenschaften besitzen.
Zu den zu überprüfenden Eigenschaften gehören Robustheit [138], funktionale Eigenschaften [85] und die Äquivalenz [126] von neuronalen Netzen. Die Äquivalenz neuronaler Netze ist vor allem im Zusammenhang mit eingebetteten Systemen wichtig. Steuerelemente von Autos oder Flugzeugen zum Beispiel haben begrenzte Ressourcen für die Ausführung neuronaler Netze. Solche Energie- und Speicherbeschränkungen haben daher zu Komprimierungsmethoden geführt. Es gibt jedoch noch wenige Verfahren, welche zeigen können, dass ein originalen und komprimierten Netze gleiche Eigenschaften besitzen.

Daher ist das zweite Ziel dieser Dissertation Ansätze zu entwickeln, mit dessen Hilfe die Äquivalenz zweier Netze bewiesen werden können. Durch die stochastische Natur des Trainings Neuronaler Netze, sind zwei Netze selten exakt äquivalent. Daher führen wir in einem ersten Schritt neue relaxierte Äquivalenzbegriffe ein und argumentieren über die allgemeine Komplexität von Äquivalenzverifikation. Daraufhin präsentieren wir zwei Verifikationsverfahren für Neuronale Netze. Unser erstes Verfahren basiert auf der exakten Kodierung von Netzen und Eigenschaften als mixed-integer linear programming Probleme (MILP). Durch die Kodierung von Netzen und Eigenschaften als Optimierungsproblem, welches die Unterschiede zweiter Netze maximiert, können wir über geclustertem Input die Äquivalenz von zwei Netzen zeigen oder Gegenbeispiele finden. Unser zweiter Ansatz strebt eine verbesserte Skalierbarkeit durch Abstraktionen an.

Daher erweitern wir zuerst den bestehenden geometric path enumeration (GPE) Ansatz von einem auf mehrere Netze. Der GPE Ansatz propagiert Mengen durch Neuronale Netze und wendet definierte Transformationsschritte an. Im letzten Abschnitt der Dissertationen werden Abstraktionen und andere Optimierungen vorgestellt, mit denen der Ansatz für die Äquivalenzverifikation skaliert wird. Unsere beiden Ansätze sowie die neuartigen Definitionen verbessern dabei den Stand der Technik und unsere Evaluation zeigt, dass die Ansätze im Vergleich zu anderen Werkzeugen im Besonderen für die Äquivalenzverifikation strukturell unterschiedlicher Netze starke Ergebnisse liefern.

Contents

Acknowledgments i

Abstract iii

Zusammenfassung v

Contents 1

I Introduction and Foundation 5

1 Introduction 7
- 1.1 Objective of Thesis . . . 8
- 1.2 Contribution and Publications . . . 9
  - 1.2.1 C1 Modular Bounded Model Checking . . . 10
  - 1.2.2 C2 Equivalence Verification for Neural Networks . . . 12
- 1.3 Structure of Thesis . . . 13
  - 1.3.1 Part I - Introduction and Foundation . . . 13
  - 1.3.2 Part II - Modular Bounded Model Checking . . . 14
  - 1.3.3 Part III - Equivalence Verification for Neural Networks . . . 14

2 Foundation 15
- 2.1 Software Verification . . . 15
  - 2.1.1 LLVM-Framework (LLVM) . . . 16
  - 2.1.2 Bounded Model Checking (BMC) . . . 18
  - 2.1.3 Low Level Bounded Model Checker (LLBMC) . . . 19
- 2.2 Neural Network Verification . . . 20
  - 2.2.1 Mixed-Integer Linear Programming (MILP) . . . 21
  - 2.2.2 Geometric Path Enumeration (GPE) . . . 21

II Modular Software Verification **25**

3 Modular Software Bounded Model Checking **27**

4 Applicable Bounded Model Checking **31**
4.1 Introduction . . . 31
4.2 Challenges and Requirements for the Verification of Industrial Embedded Software . . . 33
4.3 Architecture of `QPR Verify` . . . 34
4.4 Application of `QPR Verify` . . . 35
4.4.1 Setup and Preprocessing . . . 35
4.4.2 Solving Strategies and Abstractions . . . 39
4.4.3 Global Analysis. . . . 44
4.4.4 Error Traces and Result Display . . . 44
4.5 Conclusion . . . 45

5 Automatic Modularization Techniques **47**
5.1 Introduction . . . 47
5.2 Theoretical Foundations . . . 49
5.2.1 Program Semantics of LLVM . . . 49
5.2.2 Modularization . . . 51
5.3 Decomposition of Programs . . . 52
5.3.1 Structural Abstractions . . . 52
5.3.2 Properties of Modularization . . . 58
5.4 Conclusion and Future Work . . . 61

6 Refinement through Enumerative and Learned Preconditions **63**
6.1 Introduction . . . 63
6.2 Bird's Eye View of the Method . . . 64
6.3 Recapitulation of Modularization . . . 66
6.4 Module Extension by Caller Inclusion . . . 67
6.5 Refined Modularization through Preconditions . . . 68
6.5.1 Generation of Enumerative Preconditions based on BMC . . . 68
6.5.2 Learning General Preconditions from Data Points . . . 71
6.5.3 Subsumption of Preconditions . . . 75
6.6 Conclusion . . . 75

7 Tool Demonstration and Evaluation **77**
7.1 Tool Demonstration . . . 77
7.1.1 Set-Up . . . 78
7.1.2 Modularization . . . 79
7.1.3 Refinement . . . 80
7.1.4 Incremental Strategies . . . 82
7.2 Evaluation . . . 83
7.2.1 BMI160-Driver . . . 84

7.2.2 SQLite . . . 87
7.2.3 MNAV1.5 . . . 90
7.2.4 Summary . . . 92

8 Related Work **95**

9 Conclusion **99**
9.1 Summary . . . 99
9.2 Future Work . . . 101

III Verification of Neural Networks **103**

10 Equivalence Verification of Neural Networks **105**
10.1 Introduction . . . 105
10.2 Equivalence of Neural Networks . . . 106
10.3 NN Equivalence and NP . . . 109

11 Verification on Clustered Input through MILP Encoding **111**
11.1 Introduction . . . 111
11.2 Encoding as MILP . . . 112
11.2.1 Encoding of Neural Networks as MILP . . . 112
11.2.2 Encoding of Equivalence Properties in MILP . . . 113
11.3 Input Restriction . . . 116
11.3.1 Hierarchical Clustering . . . 117
11.3.2 Encoding of Clusters . . . 117
11.3.3 Searching for a Maximal Radius . . . 117
11.4 Application: Neural Network Compression . . . 118
11.5 Evaluation . . . 118
11.5.1 Equivalent Neural Networks . . . 120
11.5.2 Remarks . . . 122
11.6 Conclusion . . . 123

12 Adjusted Geometric Path Enumeration **125**
12.1 Introduction . . . 125
12.2 Extending GPE to multiple Networks . . . 126
12.2.1 Equivalence on Set Data Structures . . . 127
12.2.2 Challenges and Limitations of the approach . . . 128
12.3 Optimizing GPE for two NNs . . . 129
12.3.1 Zonotope Propagation . . . 129
12.3.2 Zonotope Over-Approximation . . . 129
12.3.3 LP approximation . . . 130
12.4 The Branch Tree Exploration Problem . . . 131
12.5 Experimental Evaluation . . . 134
12.5.1 Experimental Setup . . . 134

12.5.2 Comparison of NNEquiv versions 135
12.5.3 Comparison to previous work 136
12.5.4 Influence of ϵ-equivalence tightness 137
12.5.5 Finding Counterexamples 138
12.6 Conclusion and Future Work 139

13 Related Work **141**

14 Conclusion **145**
14.1 Summary 145
14.2 Future Work 146

Bibliography **147**

Own Publications **159**

15 Relevant Implementations **161**

Part I

Introduction and Foundation

Chapter 1

Introduction

In our modern society, software has become ubiquitous. We rely on its correct and safe execution, especially in safety-critical applications. Embedded software is for example deployed in automotive, aerospace and medical devices, which have to meet the highest quality standards, as errors can have catastrophic consequences. Dynamic testing of software covers only a limited number of program executions and thus provides no safety guarantees. Studies about the consequences of software errors like [153] show the necessity of precise and thorough verification and are backed up by catastrophic experiences from the past and present, such as the rocket crash of Ariane flight 501 [108] or the car crash of the Toyota Camry in 2005 [98].

Embedded industrial software, which can, for instance, be found in self-driving cars or modern airplanes, can be divided into two kind of components. In traditionally engineered software and in machine learning software, which includes neural networks. Neural Networks are increasingly applied to image recognition [100] or in advisory systems in airplanes [81]. This dissertation aims to improve the state-of-the-art for the verification of software written in C/C++ as well as neural networks. Verification for both components is formally defined and novel verification approaches are presented.

For traditional software, this dissertation concentrates on the existence of runtime errors leading to undefined behavior and software crashes. Such runtime errors encompass, among others, arithmetic overflows or divisions by zero. There exist several techniques of proving the absence of runtime errors in software written in C/C++. Approaches based on abstract interpretation [39] or bounded model checking [19] are often applied in science and industry for the verification of software projects. However, they often lack the scalability to automatically verify modern software projects consisting up to millions of lines of code.

Neural networks are currently among the most popular machine learning methods and are being applied in many safety-critical applications, such as image processing of stop signs for cars [84] or collision avoidance systems for airplanes [81]. The verification of neural networks is a relatively young research field that has mostly concentrated on the verification of adversarial robustness [138]: proving that a small deviation of inputs

does not lead to unintended change in output. Among other properties to be verified are functional properties [85] and the equivalence [126] of two neural networks. The equivalence of neural networks is especially important in the context of embedded systems. Control elements of cars or airplanes, for example, have limited resources for the execution of neural networks. Such energy and memory constraints have therefore given rise to compression methods. The safe application of such compressed networks necessitates the formal verification of equivalence between original and compressed networks.

In this thesis, we tackle both the scalability issue of traditional software verification and the novel challenge of equivalence verification for neural network. We advance the state-of-the art by proposing a modular bounded model checking approach that is able to verify large software projects through automatic modularization and refinement techniques. Moreover, we introduce novel notions of equivalence between two neural networks and derive a complexity bound for the equivalence verification of neural networks. Finally, we present two novel verification methods for neural network equivalence.

1.1 Objective of Thesis

The goal of this thesis is to improve the state-of-the-art for the verification of two main components of modern embedded systems: software written in C/C++ and neural networks.

Scalable Bounded Model Checking for Industrial Software Bounded Model Checking (BMC) inlines function calls and unrolls loops a finite number of times. Unrolling reduces the complexity of the problem to a feasible level making it decidable. The program can be encoded into a logical formulae and then solved by state-of-the-art SMT/SAT solvers. Due to an exact encoding of the program, BMC is very precise and therefore often applied for software verification. Due to continuous research both encodings and the underlying SMT solvers steadily improve. However, the size and complexity of industrial embedded software increases even faster. Modern cars are estimated to have around 300 millions lines of code (LoC) in the next few years with components of several million LoC. Even with a loop-bound of 1, BMC is not able to solve or even generate formulae for programs of that size with reasonable memory resources.

In this thesis, we aim towards a more scalable bounded model checking approach that can be applied to industrial software. Our goal is to find runtime errors or prove their absence. To be applicable in safety critical areas, the approach has to be complete, meaning that all critical program locations regarding a check category (like division by zero) have to be checked. Furthermore, our approach should be sound, in the sense that we never classify a property as safe when it is not. To be scalable for industrial software, we aim to develop a fully automatic approach with minimal user interaction. This thesis presents a modular bounded model checking approach optimized towards usability and scalability.

Establishing and Verifying Equivalence of Neural Networks. Verification of equivalence can be regarded as relational verification of neural networks and checks whether two NNs are equivalent to each other. The main application of equivalence verification, is in the context of NN compression. Compression of networks through retraining, pruning and other techniques [144, 6, 75] leads to smaller neural networks with "equivalent" behavior. However, most current research does not provide any sound proofs for equivalence but only statistical indications.

In this thesis, our seconds goal is to develop approaches capable of sound proofs for the equivalence of neural networks. Due to the novelty of equivalence verification of neural networks, we first have to define new notions of equivalence, which should represent a human understanding for neural networks. Furthermore, equivalence for regression and classification tasks have to differ due to different applications and thereby user demands. The second goal of this thesis are techniques that are able to verify the equivalence for neural networks and provide human-understandable counterexamples in case of network differences.

Overall, we regard two different kinds of programs under verification. Analysis of software written in C/C++ can rely on decades of research and there are several established techniques like Bounded Model Checking or Abstract Interpretation, which are steadily improving. A main challenge for software verification of industrial software lies in the usability of tools and the scalability of underlying verification techniques. Neural network verification is a much younger research field. Therefore, main challenges like problem definition and complexity proofs still remain. By considering both components to be verified, we also aim to transfer verification knowledge from the traditional software domain to neural network verification.

1.2 Contribution and Publications

This thesis presents two main contributions **C1** and **C2** in alignment with the two presented goals. An overview of the contributions is depicted in Figure 1.1. Publications of the author relevant for this dissertation are marked with "[KB]" to clearly distinguish them from other references.

The first contribution of this thesis, **C1**, is a modular bounded model checking approach consisting of three phases. First, all critical program locations are found and the program under verification is efficiently build, preprocessed and configured for verification. We call this phase *Set-up*, representing the contribution **C1.1**. To meet our scalability requirements, we then present a fully automatic modularization approach, such that smaller portions of the program can be verified individually. The modularization, represented by contribution **C1.2**, is based on structural abstractions of the program. Such abstractions lead to false positives and therefore, the third phase, presents refinements of these abstractions, representing contribution **C1.3**. Thus, we reach a user-friendly and configurable modular bounded model checking approach that balances scalability through abstraction and precision through refinement.

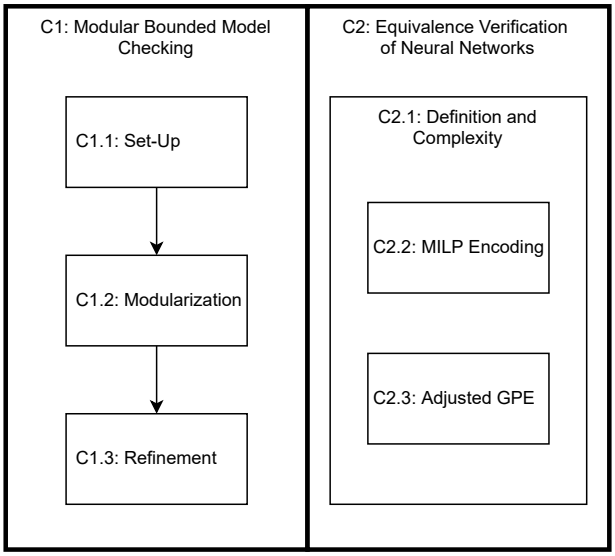


Figure 1.1: Overview of contributions of this thesis.

All approaches are evaluated on industrial software projects ranging from 2000 to 400k LoC plus external libraries and show the improvement of the modular bounded model checking approach for the state-of-the-art.

The second main contribution, **C2**, is a framework for the equivalence verification of neural networks. Due to the statistical training process of NNs, two networks are hardly ever exactly equivalent. The contribution **C2.1** presents different notions of equivalence and a complexity proof. These notions serve as foundations for two different approaches for equivalence verification. Contribution **C2.2** presents an approach based on exact MILP encodings of networks and properties. Contribution **C2.3** adjusts the existing geometric path enumeration (GPE) approach to equivalence verification. Through abstraction and refinement strategies we optimize the scalability of our adjusted GPE approach. The evaluation of both approaches shows the advancement of the state-of-the-art, especially for structurally different networks.

In the following, we will briefly describe each contribution and give insights into their publication.

1.2.1 C1 Modular Bounded Model Checking

C1.1 Set-Up. Verifying static analysis tools currently employed in industrial practice can generally be divided into two categories: tools based on Abstract Interpretation (AI) and tools based on Bounded Model Checking (BMC). Bounded Model Checking approaches using bitvector and array theories are generally more precise but lack scalability and usability, which are the main reasons why they are still not frequently applied

in industry. Therefore, this thesis presents the tool `QPR Verify` as an extension of the bounded model checker LLBMC with the aim to keep high precision while increasing scalability and usability for industry projects. The contribution **C1.1** consists of techniques implemented in `QPR Verify`. A user-friendly set-up of projects is realized through automatic generation of runtime checks and broad configuration possibilities. A lightweight preprocessing analysis and a slicing algorithm are introduced to minimize the number of checks and the source code to be analyzed, thus increasing scalability. Furthermore, verification results of a whole-program analysis can be shown as a detailed verification report including error traces.

The contribution **C1.1** is described in Chapter 4 and parts of this contribution were previously published in [KB5]. The initial design and implementation of main parts of `QPR Verify` has been done in the research group "Verification meets Algorithm Engineering" at KIT and in the startup QPR Technologies, with the author as a co-founder. Next to the author, main parts of `QPR Verify` were implemented by David Farago, Felix Kutzner, Robin Freyler, Florian Merz and Carsten Sinz.

C1.2 Modularization. This thesis presents an automatic modularization approach to improve the scalability of bounded model checking. Contribution **C1.2** first introduces definitions of program semantics and modules based on the LLVM framework. Then, a general model for program modularization is presented and mandatory and desirable properties are introduced improving the understanding of general possibilities and limitations of modularizations in the context of software verification. We present two concrete modularization techniques in the context of bounded model checking using two different types of abstraction. Given a module consisting of a set of functions, the first approach abstracts away calls to functions outside of the chosen module by over-approximating the function behavior. The second approach abstracts the call environment of the entry-point function of our module by setting input parameters, global variables and relevant memory locations to arbitrary values. The modularization is implemented into `QPR Verify` and allows a scalable verification approach for large software projects.

The contribution **C1.2** is described in Chapter 5 and main parts of this contribution were published in [KB4].

C1.3 Refinement. The abstractions introduced by our modularization result in increased scalability but can also produce false positives. To minimize such inaccuracies, contribution **C1.3** presents refinements of structural abstractions through three steps. By incrementally adding relevant parent functions to the modules, the call environment is refined and thereby the approach can incrementally increase the module size to the limits of the underlying bounded model checker. Furthermore, we refine abstractions through automatically generated preconditions. Based on an enhanced output of the bounded model checking approach, we generate enumerative preconditions that represent erroneous input for the entry-point function of a module. These possibly under-approximated preconditions are then generalized by a tree-based learning approach. Through feature extraction and an ID3-based [133] synthesizer, we are able to generate

complete preconditions. Function calls are then substituted with the according preconditions. Thus, the approach refines the introduced abstractions and minimizes false positives. The generated preconditions are human-readable and can be examined and adjusted by a user. The caller inclusion and enumerative precondition generation is implemented into `QPR Verify` and minimizes the number of false positives.

The contribution **C1.3** is described in Chapter 6 and is currently prepared for submission[1]. The tree-based learning approach was implemented during the "research laboratory" (Praxis der Forschung) by Johannes Meuer under the supervision of the author.

This thesis also presents an evaluation of all three verification phases individually and combined. The evaluation of implemented approaches on three industrial software projects ranging from 2000 to 400k LoC plus external libraries demonstrates the increase in usability, scalability and precision introduced by the three verification steps. A comparison to commercially available tools shows that the, in this thesis presented, modular bounded model checking approach significantly advances the state-of-the-art of software verification.

1.2.2 C2 Equivalence Verification for Neural Networks

C2.1 Definition and Proofs. We aim to verify the equivalence of neural networks, which is an important application with regards to compression techniques that minimize the size of networks. The training procedure of NNs is non-deterministic and can be performed on different training datasets. It is therefore unlikely for two NNs to be exactly equivalent. Therefore, contribution **C2.1** presents relaxed equivalence definitions to obtain more practical notions of equivalence. The three properties ϵ- , *top-1* and *top-k*-equivalence are defined and their merit over exact equivalence are described. Furthermore, a complexity proof showing that verifying ϵ-equivalence is coNP-complete is presented.

This contribution is described in Chapter 10 and main parts of it were previously published in [KB2] and [KB6].

C2.2 MILP Encoding. To verify this newly introduced equivalence properties, contribution **C2.2** presents an approach that encodes feed-forward neural networks with ReLU activation functions together with a given equivalence property as a mixed-integer linear program (MILP). The resulting optimization problem minimizes the distance between network outputs and is thereby able to verify the relaxed notions of equivalence. We employ the encoding in conjunction with restricting the input space for which the networks have to be equivalent by hierarchical clustering. The approach is able to verify the equivalence and, in case of a negative result, produces counterexamples. The evaluation of our approach using two existing reduction methods on a neural network for handwritten digit recognition shows the applicability and precision of our encoding.

[1] This will be changed at time of publication

This contribution is described in Chapter 11 and main parts of it were previously published in [KB2].

C2.3 Adjusted GPE. The presented MILP-encoding allows for precise verification but often lacks scalability to larger networks. Contribution **C2.3** therefore presents a modified version of the geometric path enumeration (GPE) algorithm and its application to the equivalence verification problem. GPE propagates a set of input values through a neural network and then evaluates whether the output set overlaps with a specified error region. When propagating sets through activation functions, the data structures have to be split by introduction of additional hyperplanes or new linear constraint predicates. We discuss several optimizations to increase the efficiency on practical problems by approximating data structures and introduction of heuristics minimizing the number of splits. Our evaluation shows an advance in scalability compared to our MILP approach and also the state-of-the-art.

This contribution is described in Chapter 12 and main parts of it were previously published in [KB6].

The first definitions of equivalence and the verification approach based on mixed-integer linear programs were pursued during the Master's Thesis of Philipp Kern [89] supervised by the author and lead to the publication [KB2]. The expanding work of proving coNP-completeness for equivalence verification and the modification and optimization of the geometric path enumeration approach were conducted in a "research laboratory" (Praxis der Forschung) project in which the student Samuel Teuber was also supervised by the author and lead to the publication [KB6].

1.3 Structure of Thesis

We now provide a brief outline for the rest of this thesis.

1.3.1 Part I - Introduction and Foundation

Next to this introduction Part I presents foundations in Chapter 2, which describes basic concepts and notions relevant for the understanding of this thesis.

Section 2.1 introduces concepts relevant to our modular bounded model checking approach. It therefore presents the LLVM-Framework (Subsection 2.1.1, Bounded Model Checking (Subsection 2.1.2) and the implementation of the bounded model checking approach LLBMC (Subsection 2.1.3).

Section 2.2 introduces neural networks with the ReLU activation function. It then presents two basic concepts later applied for NN verification. Mixed-integer problems are introduces in Subsection 2.2.1 and the geometric path enumeration approach in Subsection 2.2.2.

1.3.2 Part II - Modular Bounded Model Checking

Part II presents the modular bounded model checking approach and thereby contribution **C1**. The first chapter of the second part, Chapter 3, gives a more detailed overview of our approach and its contributions.

Then Chapter 4 presents the set-up phase and thereby contribution **C1.1**. Next to challenges and requirements of software verification presented in Section 4.2, we present features and implementation decisions of the tool `QPR Verify`. Chapter 5 presents the fully automatic modularization approach and thereby the contribution **C1.2**. In Section 5.2, we define program semantics and modularization concepts before describing specific modularization techniques in Section 5.3. The refinement of abstractions are described in Chapter 6. We present contribution **C1.3**. After a bird's eye view of our methodology in Section 6.2, we present refinement through caller inclusion in Section 6.4 and the generation and substitution of preconditions in Section 6.5. Chapter 7 presents a tool demonstration and shows evaluation results. In Section 7.1, we present step-wise instructions on how to configure and apply `QPR Verify` to industrial projects.

Afterwards, Section 7.2 presents evaluation results for the three industrial projects BMI160-Driver, SQLite and MNAV1.5. Related work is described in Chapter 8. A conclusion of Part II is then presented in Chapter 9 consisting of a summary in Section 9.1 and an outlook on future work in Section 9.2.

1.3.3 Part III - Equivalence Verification for Neural Networks

Part III presents the equivalence verification framework and thereby contribution **C2**.

We motivate and introduce the equivalence property for neural networks in Chapter 10, describing contribution **C2.1**. Chapter 11 then presents our first verification approach for neural networks and therefore contribution **C2.2**. Section 11.2 presents the encoding of NNs and introduced properties. In Section 11.3, we discuss input restrictions through hierarchical clustering and the application field of compression in Section 11.4. In Chapter 12, we present the adjusted geometric path enumeration and the contribution **C2.3**. The extension of the GPE algorithm is described in Section 12.2 and optimized in Section 12.3. An experimental evaluation of our approach and a comparison to our previous work as well as the state-of-the-art is described in Section 12.5. Related work is presented in Chapter 13 and the last chapter of this thesis, Chapter 14, concludes the third part of this thesis, giving a summary (Section 14.1) and an outlook into promising research directions for neural network verification (Section 14.2).

Chapter 2

Foundation

In this chapter, we describe and specify foundations regarding verification of software written in C/C++ and machine learning algorithms in the form of neural networks. For classical software, we introduce the LLVM-framework with its intermediate representation (LLVM IR). As our chosen technique to verify software is based on that representation, we present the bounded model checking (BMC) approach and introduce the tool LLBMC as an implementation of it. Furthermore, we briefly introduce neural networks and the ReLU activation function together with two basic techniques currently utilized in neural network verification: mixed-integer linear programming (MILP) and geometric path enumeration (GPE).

2.1 Software Verification

Verification and Validation (V&V) is an essential part of software engineering, which ensures the correct and safe performance of software. While validation of a product makes sure that the correct product is built, verification tackles the tasks of ensuring that the product is built right [151]. Thus verification of a software project can be understood as proving the absence of errors and undefined behavior of source code. In this thesis, we concentrate on runtime errors like division by zero or undefined overflows that occur during program execution. These are also referred to as safety-properties in contrast to liveliness-properties defining e.g. the termination of a program [2].

Compared to dynamic software testing, a formal verification of software does not only find such errors but aims to give a definite proof for the absence of such. Furthermore, we concentrate on automatic approaches that can be applied for larger scale projects. Interactive theorem proves like [12, 123] are based on powerful techniques and allow modular verification but they require manual labor. The number of lines of specification that has to be written for one line of source code varies depending on approach and application. Typical factors range between 2 for specialized [124], 5 for SMT-based [71] or up to 20 for interactive theorem prover approaches [83] and are thus not feasible for larger industrial projects.

Verifying static analysis approaches currently employed in industrial practice and scientific research can generally be divided into two categories: approaches based on Abstract Interpretation (AI) [39, 154, 43] and techniques based on Bounded Model Checking (BMC) [103, 115, 44, 67, 30]. Abstract interpretation tools are already established in application areas, because with a suitable AI domain (often the interval domain) they scale quite well. Bounded model checking approaches using bitvector and array theories are generally more precise but lack scalability, which is the main reason why they are still not frequently applied for larger projects. The aim of this thesis is to further develop the BMC approach to make it applicable and scalable for real-world applications. Therefore, we first introduce the LLVM framework which is often used to simplify and translate code and then describe BMC and one implementation of it in more detail.

2.1.1 LLVM-Framework (LLVM)

Main parts of this subsection including Table 2.1 were previously published as preliminaries in [KB4].

LLVM is an open source compiler framework that consists of a "collection of modular and reusable compiler and tool-chain technologies" [107]. It supports compilation for a wide range of languages and is known for its research friendliness and good documentation. It is very complex for verification approaches to work directly on C-code and it is extremely difficult to support all language features.

LLVM provides an intermediate representation for a number of high-level languages, including C. LLVM's intermediate representation is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers. IR programs are always kept in static single assignment (SSA) form, meaning that each register is assigned exactly once, and every variable is defined before it is used. This helps to close the gap between C-code written by developers and logical formulas solved by SAT-solvers during the BMC approach.

A program in LLVM-IR consists of a set of global variable declarations G, and a set of functions F. In LLVM terms such a program $P = (F, G)$ is often denoted as a module. To prevent confusion with terms introduced in later modularization approaches, we denote such programs P simply as *programs* and not modules. Each function is represented by a graph of basic blocks, while each basic block in turn is a linear sequence of instructions having one entry and one exit point. Instructions are the smallest executable unit in LLVM IR and the last instruction of every basic block is called terminator.

The instruction set, as of interest in this thesis, can broadly be split into four types (see also Table 2.1):

- Memory-related instructions such as `load`, `store`, stack allocation (`alloca`) and address calculation via base pointer and offsets (`getelptr`)[1];

[1]For brevity, we use `getelptr` instead of LLVM's name `getelementptr`.

- Three-address-code (TAC) instructions working on registers or constants, mainly for arithmetical and logical operations.
- Bit-level conversion instructions like extensions, truncations, and type casts.
- Control-flow related instructions for conditional and unconditional branching, the `phi` instruction (which is typical for SSA form) to conditionally select a value, as well as function-`call` and return (`ret`) instructions.

For the exposition of our approach, we have extended the IR language by two verification-related instructions (in the implementation these are modeled as intrinsic functions instead of instructions), one for checking assertions and another one to set a variable to a non-deterministic value.

All (conditional and unconditional) branch instructions are only allowed as a terminator instruction of a basic block. The branch instructions between basic blocks induce a basic block graph (a.k.a. control-flow graph), in which edges are annotated with the condition under which the transition between the two basic blocks is taken.

MEMORY OPERATIONS:	
p = `alloca` t	allocate stack memory for type t
q = `getelptr` $p, o_1, \ldots, o_n$	address calc. (base pointer, offsets)
x = `load` p	load from memory address p
`store` x, p	store x at address p
ARITHMETICAL / LOGICAL OPERATIONS:	
$z = x$ `<op>` y	where `<op>` $\in \{+, -, *, \ldots, \|\|, \&\&, \ldots\}$
$c = x$ `<op>` y	where `<op>` $\in \{<, =, >, \leq, \ldots\}$
CONVERSION OPERATIONS:	
y = `sext/zext` x `to` t	sign/zero extend x to width of type t
y = `trunc` x `to` t	truncate x to width of type t
y = `ptrtoint` p `to` t_i	convert a ptr. value p to integer type t_i
p = `inttoptr` x `to` t_p	convert an int. value x to pointer type t_p
CONTROL FLOW:	
`br` bb	unconditional branch to basic block bb
`br` c, bb_1, bb_2	conditionally branch to bb_1 or bb_2
`call` $f(x_1, \ldots, x_n)$	call (void) func. f with par. $x_1, \ldots$
y = `call` $f(x_1, \ldots, x_n)$	call func. f returning y
`ret` y / `ret void`	return value y / nothing
y = `phi` $[x_1, bb_1], \ldots, [x_n, bb_n]$	conditional selection of value x_i
VERIFICATION EXTENSIONS:	
`assert` c	assert that condition c is true
x = `nondet` t	set x to a non-determ. value of type t

Table 2.1: LLVM IR insructions

2.1.2 Bounded Model Checking (BMC)

In this thesis, we focus on the application and extension of the bounded model checking approach. We therefore do not introduce underlying concepts as the Boolean satisfiability problem (SAT) or Satisfiability Modulo Theories (SMT). We assume the reader to be familiar with propositional logic and first-order-logic.

The desire to mathematically prove the correctness of hardware and later on software produced a number of formal verification approaches over the years. Model checking as one of the first automatic techniques, proves that a system or model meets a given specification. Explicit state model checking, introduced in [35], applies Kripke structures as models and checks properties utilizing temporal logic formulae. There exist several decision procedures to verify properties given in temporal logic [51, 48]. The tool SPIN [13] is an implementation of such an approach and is able to successfully verify embedded C-code. Yet, the main challenge of explicit state model checking is the exponential growth of the checked state space, and is called *state space explosion*. Symbolic model checking relies on binary decision diagrams (BDDs) and allows for a larger number of states to be checked. Yet BDDs are not always efficient because for specific types there exists no efficient ordering and encoding. For software verification such as for example for the multiplication of integers.

Therefore, bounded model checking verifies bounded traces through a program by SAT-solvers. Biere et al. [19] introduced the first BMC approach applied to hardware systems and notably Clarke et al. [34] with their tool CBMC further developed the technique to be applied to software systems. For a program, loops and function calls are unrolled and inlined to a given bound b. The resulting program is then encoded into a SMT and then SAT formula together with a negated property to be checked. The verification problem V with bound b, can be represented by Equation 2.1:

$$V = I(s_0) \wedge \bigwedge_{i=0}^{b} T(s_i, s_{i+1}) \wedge \neg P\,, \tag{2.1}$$

where $I(s_0)$ represents the initial states of the program, $T(s_i, s_{i+1})$ the bounded transition system and P the set of properties to be checked. This notion for bounded model checking is typical formulation introduced for hardware verification.

Such a formula can then be verified by any state-of-the-art SAT-solver. If a satisfiable model is found, it can be regarded as a counterexample for the verified property. If there is no model within bound b, the approach increases b until a counterexample is found or the bound is sufficiently high. Biere et al. [19] developed several approaches that give upper bounds for b that are necessary for a sound verification.

2.1.3 Low Level Bounded Model Checker (LLBMC)

The tool LLBMC was originally developed at the Karlsruhe Institute of Technology by Merz et al. [115]. It is an implementation of the BMC approach utilizing the LLVM IR as an input language for its verification procedure. The high-level model checking algorithm, described in [113], is presented in Algorithm 1. As input the tool takes a program p, an entry-point function e, a bound for loop-unrolling b_l and a bound for function inlining b_c. The function $OPTIMIZE$ runs transformation passes implemented in LLVM and LLBMC to increase performance and extend the language support by simplifying structures e.g. switch structures and memory allocations. Afterwards, the function $UNROLL$ unrolls loops of program p to the given bound b_l. To inline function calls until bound b_c and handle function pointers through a type-based over-approximation [113], $CALLGRAPH$ creates a call-site sensitive call-graph starting at the entry-point function. The function $ENCODE(p, g)$ takes the unrolled program in LLVM IR and the bounded call-graph and creates a formulae that is the function $SIMPLIFY$ simplifies utilizing an existing term rewriting system. The resulting formulae is given to an SMT solver. The $SOLVE$ function returns a Boolean result r together with a model m. If the return value is *true* an error has been found and the function $COUNTEREXAMPLE$ can utilize the module m to transfer variable assignments from the SMT level to LLVM IR and the user. If the result is *false* or *unknown* LLBMC returns null, implying that no errors have been found.

Algorithm 1 ModelCheck(p,e,b_l,b_c)

$p \leftarrow OPTIMIZE(p)$
$p \leftarrow UNROLL(p, b_l)$
$g \leftarrow CALLGRAPH(p, e, b_c)$
$\sigma \leftarrow ENCODE(p, g)$
$\sigma \leftarrow SIMPLIFY(\sigma)$
$r, m \leftarrow SOLVE(\sigma)$
if r **then**
  $c \leftarrow COUNTEREXAMPLE(p, \sigma, m)$
  return r
else
  return null
end if

In this thesis, we utilize and extend LLBMC as our bounded model checking approach of choice. Our presented techniques do not rely on a specific BMC architecture, but the transformation into LLVM IR allows for modifications of the encoding procedure without interaction with C-code. Regarding performance LLBMC is comparable to state-of-the-art solvers like CBMC [34], as can be seen by software verification competition results from 2014-2017 [18, 17].

2.2 Neural Network Verification

Neural networks (NNs) have become popular methods for tackling various machine learning tasks and are increasingly applied in safety-critical systems. We introduce feed-forward neural networks with ReLU activation functions, which are widely applied for smaller and larger scale projects. We aim to verify such networks with two different approaches in later Chapters 11 and 12. Therefore, we present basic notions of mixed integer linear programming (MILP) and geometric path enumeration (GPE). Parts of this section are part of preliminaries published in [KB2] and [KB6].

Feed-Forward NNs. NNs consist of interconnected units called neurons. A neuron j computes a non-linear function of its input values $x_1, \ldots, x_n$ according to $y_j = \sigma(\sum_{i=1}^{n} w_{ij} x_i + b_j)$ where σ is called the activation function and w_{ij} are the weights, b_j is commonly referred to as bias of neuron j. Sometimes the formula is simplified by defining the bias b_j not separately but as the first weight w_{0j}. The output of a neuron j is then defined as $\sigma(\sum_{i=0}^{n} w_{ij} x_i)$.

In this thesis, we focus on the *rectified linear unit* (ReLU) activation function, $\mathrm{ReLU}(x) = \max(0, x)$, which is one of the most commonly used activation function in modern NNs [63].

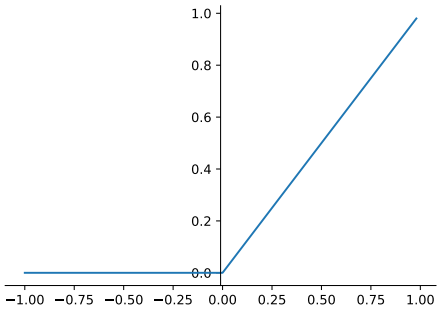


Figure 2.1: ReLU activation function.

Outputs of neurons are connected as input to other neurons, resulting in a directed graph. We focus on feed-forward NNs, where the underlying graph is acyclic. Neurons are organized in layers, where neurons in layer l take inputs only from the directly preceding layer $l-1$. The first layer, called input layer, is just a place holder for the inputs to be fed into the NN, the subsequent layers are called hidden layers, while the last layer, the output layer, holds the function value computed by the NN. We refer to the input space dimension as $I \in \mathbb{N}$ and to the output dimension as $O \in \mathbb{N}$.

For a classification task, the output y_i of neuron i represents the probability of the NN's input belonging to class i. To ensure that the resulting distribution over the outputs is normalized, each output neuron i uses the softmax activation function

$$\mathrm{softmax}(\vec{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad .$$

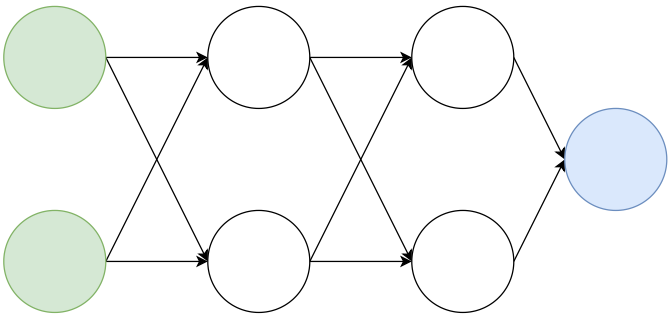


Figure 2.2: Exemplary neural network with one input layer consisting of two input neurons, two hidden layers, each with two neurons and one output layer consisting of a single neuron.

2.2.1 Mixed-Integer Linear Programming (MILP)

A MILP problem is an optimization problem for a linear objective function under additional linear constraints. Some variables are constrained to be integers, while others range over $\mathbb{R}$.

Definition 1. *A* mixed-integer linear programming problem *consists of*

1. *a linear* objective function $f(x_1, ..., x_k) = \sum_{i=1}^{k} c_i x_i$ *over decision variables* x_i *that is to be minimized or maximized,*
2. *a set of linear* constraints $\sum_{i=1}^{k} a_{ij} x_i \bowtie b_j$, $\bowtie \in \{\leq, =, \geq\}$, *where* a_{ij} *and* b_j *are constants,*
3. *and an integrality constraint* $x_i \in \mathbb{Z}$ *for some variables.*

The set S of feasible solutions to a mixed-integer linear program is often called a mixed-integer linear set. Solving MILP problems is in general NP-hard. Algorithms for solving them include branch and bound [106], cutting planes [61], or methods based on relaxations [110]. We later employ the tool Gurobi [69] to solve MILP problems representing equivalence verification tasks for neural networks. Gurobi implements a wide range of different algorithmic techniques and a short evaluation showed that, for our problem class, it marginally outperforms more general SMT solvers like Z3 [119].

2.2.2 Geometric Path Enumeration (GPE)

GPE is a methodology originally proposed by Tran et al. [150] for verifying safety properties in NNs. Given a NN $\mathcal{N}$, the input space I and the output space O, we denote the corresponding mathematical function of $\mathcal{N}$ as $g_{\mathcal{N}} : \mathbb{R}^I \to \mathbb{R}^O$. For a set of input instances $\mathcal{I} \subseteq \mathbb{R}^I$ and a specification of unsafe or unwanted output $\mathcal{U} \subseteq \mathbb{R}^O$ defined as a set of linear constraints, verification then is concerned with the question whether there exist any instances $i \in \mathcal{I}$ such that $g_{\mathcal{N}}(i) \in \mathcal{U}$.

Instead of pushing single data points through the NN and checking whether they satisfy the required safety property, GPE feeds an entire set into the NN. Through propagation and splitting of sets the approach is able to evaluate whether any parts of the output sets lie inside unwanted output $\mathcal{U}$.

Algorithm 2 High-level path enumeration algorithm for neural networks as described in [10]

Input: Input Set $\mathcal{I}$, Unsafe Set $\mathcal{U}$
Output: Verification result (`safe` or `unsafe`)

```
s ← ⟨layer : 0, neuron : None, Θ : convert(I)⟩
W ← List() {List of set datastructures to process}
W.put(s)
result ← safe
while result=safe && ¬W.empty() do
  s ← W.pop()
  result ← step(s, W, U) {This call may modify W, see Algorithm 3}
end while
return result
```

The general idea is depicted in Algorithm 2. The algorithm's input consists of an input set to evaluate and an unsafe set of states that should not be reached from the input set under consideration. We then convert the input set into an internal data structure storing both a representation of the set and how far the set has been pushed through the network so far. We add this data structure to a list `W` of sets to consider. We then continuously push the data structures inside `W` through the neural network using the `step` function, given in Algorithm 3, until there are either no further sets to consider or an unsafe state has been observed in the network's final layer. Each call of the step function either computes an affine transformation or the result of one ReLU node.

The Algorithm 2 is still very general and there remain questions both on what set representation to use and how to push this set representation through the neural network, i.e. how to calculate the affine transformations and the results of ReLU nodes. It is important to choose a set data structure which admits the affine-transformations and non-linear functions introduced by the neural network's hidden layers.

The sets can be represented through generalized star sets, which we define below:

Definition 2 (Generalized Star Set [150])**.** *A generalized star set Θ is a tuple $\langle c, G, P\rangle$ where $c \in \mathbb{R}^n$ is the center, $G \in \mathbb{R}^{n\times m}$ is the generator matrix, and $P \subseteq \mathbb{R}^m$ is a set defined through a conjunction of linear constraints (i.e. a polytope). The set represented by Θ is then defined as:*

$$[\![\Theta]\!] = \{x \in \mathbb{R}^n \mid \exists \alpha \in P : x = c + G\alpha\}$$

Algorithm 3 Step algorithm as described in [10], with abbreviated safety check

Input: $s = \langle \texttt{layer}, \texttt{neuron}, \Theta \rangle$, Waiting List $\texttt{W}$, Unsafe Set $\mathcal{U}$
Output: Safe so far? (`safe` or `unsafe`)

```
if s.neuron = None then
  if s.layer = L then
    return check_safety(s, U, W)
  else
    s.layer ← s.layer + 1
    s.Θ.affine_transformation(W^(s.layer), b^(s.layer))
    s.neuron ← 1
  end if
end if
n ← s.neuron
s.neuron ← n + 1
if s.neuron > n^(s.layer) then
  s.neuron ← None
end if
if get_sign(s, n) = neg then
  s.Θ.project_to_zero(n)
end if
if get_sign(s, n) = posneg then
  t ← deep_copy(s)
  s.Θ.add_constraint(n, ≥, 0) {positive case}
  t.Θ.add_constraint(n, ≤, 0) {negative case}
  t.Θ.project_to_zero(n)
  W.put(t)
end if
W.put(s)
return safe
```

Bak et al. [10] showed that the functions `get_sign` and `check_safety` can be efficiently calculated through linear programming. Furthermore, they presented efficient implementations for the `affine_transformation`, `add_constraint` and `project_to_zero` functions for star sets. For non-fixed ReLU nodes, i.e. ReLU nodes that can have negative and positive inputs, the star set has to be split by introduction of an additional hyperplane to the linear constraint predicate. This splitting leads to a growth in the number of star sets during execution, which can be exponential in the number of ReLU nodes in the worst case. We denote the GPE approach based on star set *exact GPE*.

Alternatively to star sets, it is possible to use zonotopes as a set data structure:

Definition 3 (Zonotopes [10])**.** *A zonotope Ψ is a generalized star set $\langle c, G, P \rangle$ with the further restriction that P may only be defined through interval constraints (i.e. P only enforces a lower and upper bound for each dimension).*

Utilizing zonotopes as data structures leads to faster runtimes because the ReLU function can be over-approximated, as shown for example in [9, 138]. We denote the GPE approach based on zonotype abstractions as *approximate GPE*.

Approaches and optimizations for the GPE approach that balance the precision of star sets and the runtime advantages of zonotopes are discussed in the context of equivalence verification of neural networks in Chapter 12 of this thesis.

Part II

Modular Software Verification

Chapter 3

Modular Software Bounded Model Checking

The second part of this thesis presents the contributions **C1** with its subsequent contributions **C1.1**, **C1.2** and **C1.3**, which are all directed towards a modular bounded model checking approach. Based on state-of-the-art bounded model checking, we introduce challenges that are hindering the real-world application of BMC, especially for embedded programs written in C/C++. We present solutions for those challenges and describe the implementation of our tool `QPR Verify`. Afterwards, we present a solution approach to the scalability issue of model checking industrial applications by introducing program partitioning or, as we call it, a modularization approach based on *structural abstractions*. Balancing out precision and scalability, we then refine the abstractions by generating *enumerative* and *learned preconditions*. Finally, we give some implementation details of our modularization and evaluate all approaches on a number of real-world applications ranging from two thousand to around half a million lines of code.

An overview of our modular bounded model checking approach is depicted in Figure 3.1.

Set-Up: Applicable Bounded Model Checking. The first challenge verifying real-world applications, is a user-friendly and automatic set-up of the program under verification and the associated verification task. This contains, among other things, the compilation, automatic check insertion, configuration and preprocessing of the given program P to be verified.

In Chapter 4, we present the tool `QPR Verify`, which was first published in [KB5]. `QPR Verify` is built as an extension of the bounded model checker LLBMC with the aim to keep high precision while increasing scalability and performance to be usable for industry sized projects. The main contributions are: (1) User-friendly setup for larger projects and encompassing automatic generation of runtime checks on the source code level. (2) Comprehensive configuration framework allowing, among others, different perspectives (grouped, bounded, conditional) on checks to be verified and configurable precision levels (3) Preprocessing to increase scalability and fast feedback through a lightweight analysis.

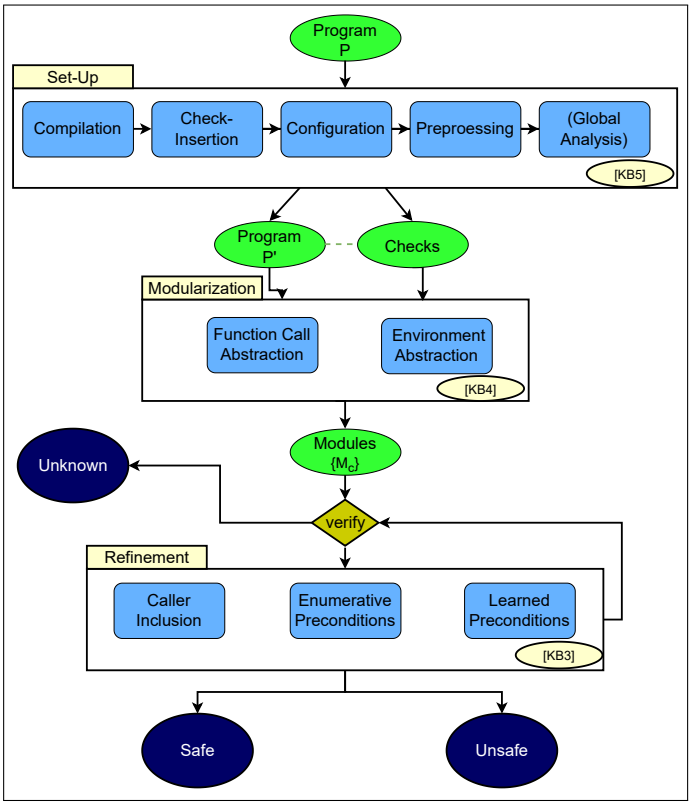


Figure 3.1: Modular software verification approach. Given a program, we set up the project and perform preprocessing, modularization by abstractions, followed by refinement steps through preconditions.

(4) a whole program (global) analysis of the program and (5) detailed verification reports, including exact error traces in C.

The first verification step (set-up) of our framework, can produce `safe` and `unsafe` results for easy checks resolved by preprocessing steps or, if the program is small enough, solved by the global analysis. For larger, thus for this thesis relevant, programs, it creates a program P' in an intermediate and optimized language representation together with a set of checks.

Modularization: Automatic Modularization through Abstractions. In this thesis, one of the main topics is an automatic modularization to improve the scalability of bounded model checking. The sheer size of modern software projects necessitates some form of abstraction. While the partitioning of problems to increase scalability is a well-known approach, we present a novel fully automatic modularization of programs for software verification.

Our modularization based on structural abstractions is described in Chapter 5 and was published in [KB4]. Given the compiled program P' together with a set of checks to

be verified, the modularization generates a set of modules. To soundly generate such a modularization, we first introduce definitions of program semantics and modules based on the LLVM framework. Then, we define a general model for program modularization and establish mandatory and desirable properties to help understand the possibilities and limitations of general modularization in the context of software verification. We present two concrete modularization techniques *Function Call Abstraction* and *Environment Abstraction*, in the context of bounded model checking and present refinement ideas. Verification of these modules produces `safe` or `unsafe` results for a large portion of generated checks. Yet over-approximation of program traces can lead to false positives, and thus error messages where there are no errors. In such cases a refinement of the abstraction is necessary.

Refinement: Refined Modularization through Preconditions. In Chapter 6, we present a refinement for our modularization based on automatically generated preconditions. The results are published in [KB3] and present three refinement steps that can be run successively to combine advantages. Based on modules, which consist of subsets of a program's functions, the first refinement strategy, called *Caller Inclusion*, extends the module size by including increasingly larger calling contexts of the module. The second refinement step, called *Enumerative Preconditions*, refines the abstraction of call environments by generating potentially under-approximated preconditions. Through the enumeration of relevant information generated by the bounded model checker about input assignments leading to errors, the refinement step is able to produce preconditions representing inputs leading to errors. Yet those preconditions do not always encompass all such inputs. Therefore, preconditions are then extended through a tree-based learning approach that generalizes from safe and unsafe program traces, in the third refinement approach called *Learned Preconditions*. The refinement and elimination of false positives is archived through the substitution of function calls by preconditions that are iteratively pushed through the program until the checks are `safe`, `unsafe` or `unknown` for checks that can not be verified.

Implementation and Evaluation. In Chapter 7, we present implementation details and demonstrate the application of `QPR Verify`. Furthermore, we show an overall evaluation of all presented approaches based on three real-world software projects ranging from 2000 to 500k lines of code. The evaluation illustrates the applicability of our approach to industrial software projects through increased scalability. The abstractions introduced by the fully automatic modularization can be refined through precondition substitution leading to precise verification results for programs with hundreds of thousands lines of code. A comparison with commercially available tools demonstrates advantages of our modular bounded model checking approach compared to the state-of-the-art.

Chapter 4

Applicable Bounded Model Checking

4.1 Introduction

This chapter develops requirements and necessary development steps for the bounded model checking approach to be applicable for the verification of industrial software projects. We present an implementation meeting the formulated requirements in `QPR Verify`, which is an extension of the bounded model checker LLBMC. The content of this chapter presents the set-up phase, depicted in Figure 4.1, of our software verification process. Furthermore, it presents contribution **C1.1** and our publication [KB5].

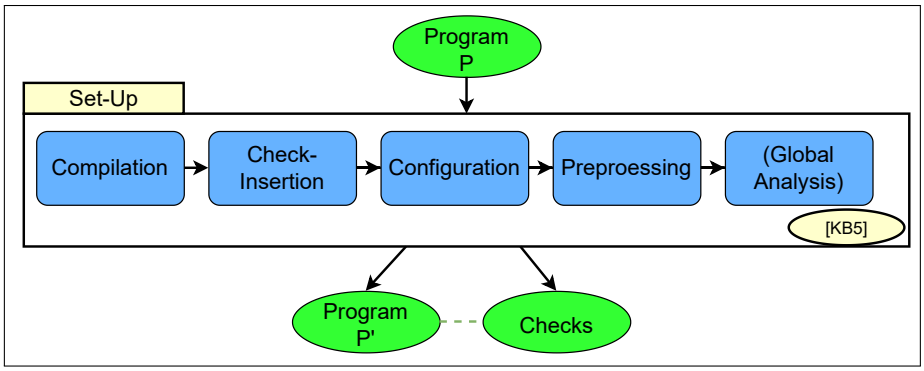


Figure 4.1: Set-up of our modular software verification approach.

Bounded model checking as introduced in Section 2.1.2, is a powerful technique to verify smaller to medium sized software projects. Yet, most current tools like CBMC [103] or LLBMC [115] lack support for user-friendly and large scale analysis. Therefore, our aim was to extend LLBMC to be user-friendly and applicable to real-world embedded C-code. As opposed to open-source projects, industrial software often needs a more elaborate approach, with partly code generation and environment specifications, following the AUTOSAR (AUTomotive Open System ARchitecture) [3] architecture and standards like ISO 26262 [142] and MISRA ((Motor Industry Software Reliability As-

sociation) [4]. Furthermore, the process of setting up the verification process as well as presenting and understanding the verification result needs more attention for industrial projects then most scientific approaches provide.

Tools based on abstract interpretation techniques are most dominant for the industrial verification of safety-critical embedded software [38]. Abstract interpretation approaches gain their scalability mostly through the abstraction of value domains but therefore lose the information about specific values that lead to errors. This can potentially produce a large amount of false positives. Bounded Model Checking approaches are more precise and the underlying SAT/SMT solver produces exact variable values for error traces and a lower amount of false positives. As trade-off, they often lack the scalability to verify large programs. Current tools based on the BMC approach like CBMC and ESBMC [56] are very precise but often not tuned towards the application of large industry projects. For example, it is hard to control and see which checks are actually performed at each program location, and sometimes code modifications are needed on the input to make it parsable by the tool.

To be compatible with industrial requirements, we implemented the tool `QPR Verify`. The initial design and implementation of `QPR Verify` has been done in the research group "Verification meets Algorithm Engineering" at KIT and in the startup QPR Technologies, with the author as a co-founder. Next to the author, parts of `QPR Verify` were implemented by David Farago, Felix Kutz, Robin Freyler, Florian Merz and Carsten Sinz.

Contribution C1.1. The main contributions presented in this chapter are: The investigation and formulation of challenges and requirements for the verification of industrial embedded software, which guided the development and implementation of `QPR Verify`. Design and implementation of a user-friendly setup for larger projects encompassing an automatic generation of runtime checks on the source code level, as opposed to the LLVM intermediate representation (LLVM-IR) level, as is done in many tools. A preprocessing analysis to increase scalability and fast feedback through a lightweight preprocessing analysis and a slicing algorithm to minimize the source code to be checked, increasing scalability. In this chapter, we present a configurable whole-program analysis implemented in `QPR Verify`. We are able to generate detailed verification reports, including exact error traces in C (and partially C++) to be shown to a user.

Structure. Section 4.2 presents the challenges and requirements of industrial software verification. Based on these challenges, Section 4.3 gives an overview of our chosen architecture of the verification process, while Section 4.4 gives deeper insights of features pushing `QPR Verify` to meet the established requirements. Section 4.5 concludes the chapter and transitions to the scalability challenge addressed in the next chapters.

4.2 Challenges and Requirements for the Verification of Industrial Embedded Software

Generally, every verification tool has to balance the competing criteria of scalability versus precision and completeness. This challenge grew further over the last years because the size of safety-critical embedded software increased dramatically. For example, the software running on board a Boeing 787 encompasses roughly 8 millions lines of code, in a Chevrolet Volt in 2017 the size is around 40 millions [143] and modern cars hava around 100 million lines of code, with more and more safety-critical features. Software verification can hardly keep up with this trend. Yet there are more challenges than just pure scalability that arise for industrial embedded code. Through a market research with 30 companies [8] and several proof-of-concepts on industrial projects, we derived the following criteria to rate the industrial applicability of an analysis tool [KB5]. Bessey et al. [15] came to similar conclusions in 2010.

REQ-SetUp How laborious is the setup of the verification tool for the software project to be analyzed?

REQ-Precision Which language features are covered, with how much precision?

REQ-Assertion Are user assertions supported?

REQ-Enviroment Can the (hardware and software) environment of the program be simulated?

REQ-Specificity How many incorrect analysis results (false positives and false negatives) occur?

REQ-Information How much information is supplied about detected faults?

REQ-Scalability Does the tool scale in the size and complexity of the checked code, i.e. does it have sufficiently low runtime and space requirements?

REQ-Incrementally Can analysis be performed incrementally, i.e. in iterations of increased precision or successive code changes?

REQ-Languages Are multiple programming languages supported?

We will refer to those requirements when describing the tool's architecture and its features in more detail. Furthermore, we compare `QPR Verify` to other state-of-the-art tools based on these requirements while discussing related work in Chapter 8.

Embedded software follows additional rules that distinguish industrial code developed for safety critical applications from e.g. web-services or other desktop programs. As mentioned earlier, developers working on safety critical software have to follow guidelines like MISRA [4] and in the automotive sector AUTOSAR [3]. Therefore, such embedded software often contains loops with a fixed number of iterations to guarantee termination

and restrict complexity of programs. We make use of this by computing a suitable unroll bound for each such loop. Furthermore, often no dynamic memory allocation is allowed (e.g., enforced by the MISRA-C standard) to ensure deterministic runtimes of programs. This influenced the selection of checks implemented in `QPR Verify`.

4.3 Architecture of `QPR Verify`

The architecture of `QPR Verify` is intended to support the criteria from Section 4.2 and to offer an automatic approach ranging over multiple stages: from source code compilation over preprocessing by lightweight static analysis, encoding and solving, to the user friendly display of verification results. In the following, we sometimes abbreviate `QPR Verify` as QPR.

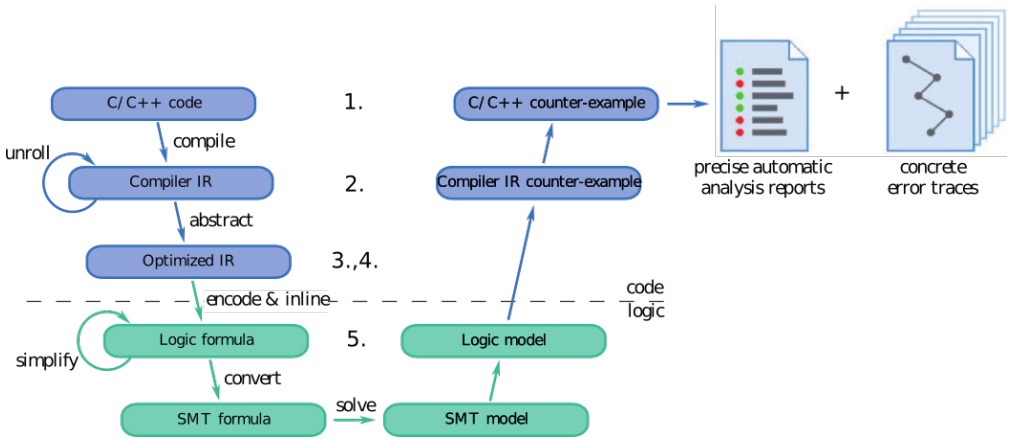


Figure 4.2: Architectural layers of `QPR Verify`.

Verification Process. `QPR Verify` analyses the given source code in the following stages, depicted on the left part of Fig. 4.2, along the downward arrows:

1. `QPR Verify` **automatically detects properties to be checked** by traversing the abstract syntax tree (AST) of a given C (or C++) program. Currently, this stage (called *Murphy*) marks program locations where the program's behavior might be undefined and thus run-time errors can occur, as well as all user-defined assertions (see REQ-Assertion). The tuples of location and properties results in a list of *checks*. The term check and its categories currently supported by `QPR Verify` are presented in Sec. 4.4.1 in more detail.

2. The program under analysis is then **compiled to LLVM-IR**. For this, our extension of *Clang* additionally emits an IR-level assertion for each check established in the first step. Operating on LLVM-IR, other languages besides C and C++ that can be translated to LLVM-IR (e.g. Rust) could easily be covered by LLBMC, too (see REQ-Languages).[1]

[1] We also extended LLBMC to handle rarely-supported constructs such as variadic functions.

3. To reduce the verification workload, **lightweight preprocessing analysis** is applied to the inserted checks (see REQ-Scalability). It performs less precise, but more scalable checks during compile time to supersede the subsequent BMC checks (e.g., based on bit-width arguments).

4. In the preparation step, the **problem is abstracted and partitioned** according to user-provided settings. A slicing algorithm also incorporating memory abstraction removes unnecessary instructions. To handle the trade-off between REQ-Precision and REQ-Scalability, `QPR Verify` offers different modes of verification, where some partition the program using structural abstractions presented in Chapter 5. The user chooses the set of checks to be analyzed, and may opt to enable IR-level optimizations to increase scalability, see Sec 4.4.2.

5. The code produced in step 3 is **analyzed using the bounded model checking** technique implemented in LLBMC. This step may be performed an arbitrary number of times with different configurations and abstractions, enabling incremental analysis to increase precision (see REQ-Incremental).

On the right part of Fig. 4.2, the results produced in the previous steps are translated back to the C (or C++) level and compiled into a **verification report** consisting of a list of check results and an interactive graphical presentation of the error traces produced by LLBMC (see REQ-Information). The error traces are presented as traces within the source code of the analyzed program, thus hiding all aspects from lower layers of Fig. 4.2 (esp. LLVM-IR) from the user.

4.4 Application of `QPR Verify`

Besides a GUI-based report, `QPR Verify` is implemented as a suite of command-line utilities, facilitating the integration of static code analysis in automated development workflows. Furthermore, we reduced the effort required to set up the analysis by aiming to provide a "push button" solution requiring minimal user intervention while still providing advanced options for experienced users. We will demonstrate the workflow of `QPR Verify` step-wise from a user perspective and give insights into development decisions and features tuned towards industrial application.

4.4.1 Setup and Preprocessing

We describe steps the first three steps from Section 4.3 in more detail. The first, and often underestimated, step for verification is the set-up of the tool and software project to be verified.

To meet REQ-SetUp, we implemented an out-of-source configuration framework, where the compiled program, the configuration and results are saved in machine readable format into one directory. Thereby, all relevant information is saved at one location and the user can interact (incrementally) with the solving process, such matching (REQ-Information) and (REQ-Incrementally). A number of verification approaches, especially

bounded model checkers like CBMC, regard the whole verification process in-scope and restrict results to terminal output. File handling is time consuming but provides a more flexible and incremental form of analysis.

Therefore, the first step of `QPR Verify` is to create an empty folder and given the root directory of the software project under verification, initialize the folder for the verification with `QPR Verify`. Given this directory or a list of files, `QPR Verify` then finds all necessary source and header files to compile the program. We additionally included a set of standard libraries in `QPR` that can be referenced when the software under test is written for a system that substantially deviates from the system `QPR Verify` is running on. `QPR` will use its own implementation of the standard library headers, if the minimal implementation of such libraries in `QPR`is better suited.

In `QPR Verify`, external functions can be either treated as uninterpreted functions, replaced by a `no-op` instruction removing the function call or are reported as an error when called. For uninterpreted functions, we assume that the function has no side effects and returns a non-deterministic value. Assuming no side effects leads to an potential under-approximation of program traces. Thus, for checks that lie after such an approximation, a warning is appended to the check result. The user can then verify, if an implementation for the uninterpreted function has to be provided for a precise verification. If another behavior is intended, data range specifications (DRS) [111] or stubs can be added manually. Data range specification are an implemented mechanism to specify bounds on memory changes and return values produced by functions. These turned out to be sufficient for many environment and library modeling. Such data range specifications can be inserted into the software code and are read-in during encoding.

Afterwards, the step we denoted as *murphy* automatically detects locations and properties which can lead to undefined behavior and thus run-time errors. In the context of `QPR Verify`, we denote the tuple of program location together with the property that needs to be verified a *check*. Checks are implemented as an assert intrinsic on the LLVM level. For a division instruction an intrinsic function call with three parameters is inserted into the llvm bitcode. The first parameter is an incremented identifier of the check and the second and third parameter represent the nominator and denominator of the division. For a division of the constant 5 by a calculated 32 bit integer, here denoted by "%6", the following intrinsic would be inserted before the division instruction.

```
call void @qpr.divbyzero.i32(i32 2, i32 5, i32 %6) .
```

Detecting possible vulnerabilities is done by building and traversing the abstract syntax tree (AST) of the given program. We implemented the AST traversal as an extension of the corresponding functionality of the Clang compiler encompassed with the LLVM framework. Clang's AST is well suited to detect program locations that should be annotated with checks, because very few instructions and arithmetic operations are summarized or optimized. Thus, Clang's AST closely resembles the original source code, while being better suited for traversal.

Currently, *murphy* marks program locations where the program's behavior might be undefined as specified in the C language standard [78], as well as all user-defined assertions. The *check categories* currently supported by `QPR Verify` are:

- **Arithmetic overflows:** Overflow of signed arithmetic operations such as addition, multiplication, division, or pre- and post-decrement and -increment.
- **Illegal shift operations:** Left and right shifts of negative values, shifts by an amount that is larger than the bit-width of the first operand, and shift overflows (cf. C99 standard, Sec. 6.5.7).
- **Division by zero** for the division and modulo operations.
- **Type casts:** A type cast of a signed integer to a smaller bit-width is implementation-defined behavior.
- **Non-initialized local variables:** Reading uninitialized memory locations is undefined behavior. (Limited support)
- **Array index out of bounds:** For arrays, where the bound can be determined from the declared type, we check for out-of-bound indices.

These check categories match those of the underlying bounded model checker LLBMC and a more detailed description including an encoding of the check categories can be found in [113]. The selection of checks is based on feedback from embedded systems practitioners and the market research mentioned in Section 4.2. For example, the index for dynamic arrays (an information not provided by C) are not tracked because most arrays in embedded software are fixed size. Memory is modeled according to the model of the used bounded model checker, in our case LLBMC [140].

Fundamentally, a check result can be `safe`, `unsafe` or `indeterminate` and may be further qualified depending on the mode of analysis. Check results that have a potential to be a false positive (due to abstractions) resp. false negative (due to the unroll limit) are marked with a *conditional* flag, i.e. `cond.safe` and resp. `cond.unsafe`.

In many cases, such as analyzing undefined behavior in integer-arithmetic expressions involving only constants, checks can be analyzed efficiently in a small portion of the syntax tree. This gives rise to our *preprocessing analysis*. In our implementation of preprocessing analysis, a check C pertaining to an operation O is said to be `safe` rsp. `unsafe` if a check result for C can be established by obtaining the operands of O via compile-time evaluation and directly showing the safety of C rsp. a violation of C's condition.

Preprocessing analysis, besides performing a compile-time evaluation of constant expressions, also computes so-called *effective bit-widths* of expressions and decides checks based on this information. The effective bit-width is an upper bound on the number of bits that are relevant in an expression, and is especially helpful in connection with the C standard's arithmetic conversion and integer promotion rules. So, e.g., in the two statements

```
char a, b; short z = a+b;
```

a signed extension to 32 bit is performed on a and b (on a 32/64 bit Intel architecture) before the addition is executed and the result is cast back to 16 bit. In LLVM, the following code sequence is generated:

```
%tmp = sext i8 %a to i32
%tmp2 = sext i8 %b to i32
%tmp3 = add nsw i32 %tmp, %tmp2
%z = trunc i32 %tmp3 to i16
```

The truncation on the last line would induce a check to be added in our verifier, whether an overflow can occur on the downcast or not. Obviously, there is no overflow in this case, which can be proven by pure bit-width arguments.

We thus define the effective bit-width, ebw, for integer expressions e as follows:

$$ebw(e) = \begin{cases} bw(c) & \text{if } e \text{ is a constant } c \\ bw(x) & \text{if } e \text{ is a variable } x \\ 1 & \text{if } e \text{ is a comparison} \\ \max(ebw(e_1), ebw(e_2)) + 1 & \text{if } e = e_1 \pm e_2 \\ ebw(e_1) + ebw(e_2) & \text{if } e = e_1 * e_2 \\ ebw(e_1) + 1 & \text{if } e = e_1 / e_2 \\ \min(ebw(e_1), ebw(e_2)) & \text{if } e = e_1 \% e_2 \\ ebw(e_1) + 1 & \text{if } e = -e_1 \\ \max(ebw(e_1), ebw(e_2)) & \text{if } e = t \ ? \ e_1 : e_2 \\ \min(bw(T), ebw(e_1)) & \text{if } e = (T)e_1 \end{cases}$$

Here, $bw(c)$, $bw(x)$, and $bw(T)$ are the LLVM-provided bit-widths for constant c, variable x or type T, respectively. Note, that the ebw-computation is performed on the AST-level, thus the scope of the analysis is restricted to source code expressions with no information about global variables or memory allocations and assignments.

While our implementation of prepossessing analysis does not improve scalability for hard verification problems, it does significantly reduce the workload put on the model checker and yields results much faster. In our experience, more than two thirds of the checks in industrial code can be analyzed conclusively using our light-weight analysis. Evaluation results can be found in Chapter 7.

The check insertion, compilation, and preprocessing analysis are all performed by issuing the two commands `qpr murphy` and `qpr compile` on the command line. The result is a list of checks, where a portion of checks is already verified, leaving checks in the categories `safe`, `unsafe` and `indeterminate`. These indeterminate checks, representing harder verification tasks, can then be solved sequentially or in parallel with different solving strategies.[2]

[2]We have implemented an experimental parallel version of `QPR Verify` based on MPI; first experiments show promising results of increased scalability.

4.4.2 Solving Strategies and Abstractions

We now describe step 4. and 5. of QPR's architecture in more detail, presenting different configuration and solving strategies to verify the checks remaining after preprocessing. All strategies use LLBMC but they differ in entry points, memory assumptions, encoding optimizations and check activation. In the following, we present the configuration options with the largest impact on the verification approach.

Loop-Unroll-Bound. The unroll bound k is passed through to the bounded model checker as the maximal limit for how often loops are unrolled. If the compiler Clang can infer the maximum amount of iterations a loop is executed, e.g. if the loop bound is given by a constant or trivial arithmetic calculation resulting in a constant, the loop is unrolled sufficiently often disregarding the loop bound. For all other cases, especially if the loop bound is given as a function parameter, read from memory or for infinite while-loops, the loop is unrolled to the given bound k. As for all approaches based on bounded model checking, this parameter strongly influences the precision and scalability of the verification approach. For industrial projects, loops are often either bounded by a small sized constant or are infinite while-loops for programs that should run endlessly. While small loop iterations fit the bounded model checking approach perfectly, infinite loops are difficult to model. In practice, verification of infinite loops is done by tools checking for liveliness properties [90], while run-time-analysis is performed on a single or sometimes two loop iterations. For checks that are called inside or after a loop, we track whether the loop bound is sufficiently large or introduces an under-approximation of program traces. Thus, if checks are resolved as `safe` wrt. the loop bound, we mark them as `cond.safe` or more precise `loop-bound safe`, such that the user can incrementally increase the loop bound or judge manually, if the loop bound was sufficient to prove the safety of the property.

AnalyzeChecksIndividually. This Boolean option classifies whether checks are verified in groups or individually. Proving that a program P does not contain undefined behavior at runtime necessitates that all critical program locations, (e.g. additions, division, etc.) are safe. Therefore, a program aiming at disproving program correctness can simply search for one unsafe location and report on it. Yet in practice this does not match the developers workflow, because the developer or verification expert does not want to rerun the verification tool after every bug fix. To circumvent the termination of `QPR Verify` at every unsafe check location, we inserted two alternatives. First, our tool can still regard all checks in P but continue the analysis upon finding errors. A user can activate such behavior by setting *AnalyzeChecksIndividually=false*.

The analysis will continue after an unsafe operation and the assertion for the specific program location, or multiple program locations if multiple checks fail in a single verification run, is deactivated. After deactivation of each check `QPR Verify` saves the check result for the deactivated check and continues verification. If the configuration *FixPoint* is set to *false*, the analysis of program traces is terminated at a program error, and subsequent errors on that trace are not reported.

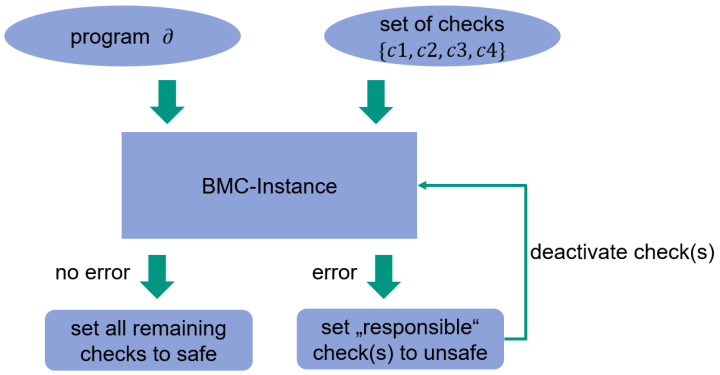


Figure 4.3: All checks of our program are grouped together and solved incrementally.

While this improves performance, it typically requires another analysis run, as soon as a program error has been fixed. If *FixPoint* is set to *true*, the program trace leading to the error is truncated. For arithmetic overflows, illegal shift operations and type casts, we assume wraparounds for otherwise illegal inputs leading to an error. For division by zero checks, we assume that the result of the division by zero can produce an arbitrary result. We also assume arbitrary values for non-initialized local variables and for array accesses exceeding the array bound. Therefore, for all check categories supported by `QPR Verify` over-approximations are defined that simulate a safe execution of prior operations.

The verification terminates if the bounded model checker does not find any violations and returns UNSAT. The final verification result for P is then comprised by the union of prior `unsafe` checks and the remaining checks which are resolved as `safe`. The described process is shown in Figure 4.3.

Alternatively, the user can set *AnalyzeChecksIndividually=true*, leading to an individual verification task for every check. `QPR` will create an independent verification task for every property to be checked. For large functions or hard verification tasks, the model checker may time out, if all checks in a function are encoded into a single formulae. In such cases, processing checks individually will typically result in a larger number of successful proofs. However, the run-time may be increased due to repetitive work. Yet independent verification task can be solved in parallel without any communication thus counterbalancing additional runtimes when enough computing resources are available.

During the encoding of individual verification tasks, we assume that all operations prior to the regarded check are safe and apply the truncation of program traces described above. Therefore, every check can be analyzed individually without being influenced by prior checks. However, these approximations can potentially lead to false positives. The described process is depicted in Figure 4.3 for four checks.

In the following, we present a number of configurations and strategies improving scalability, that rely on this individual check handling.

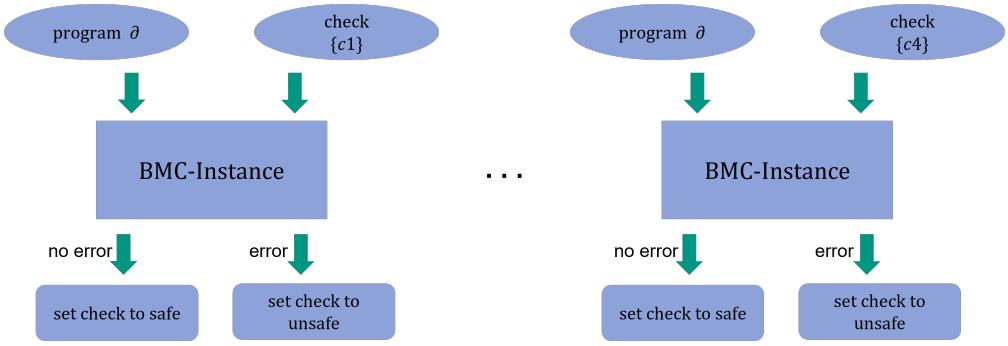


Figure 4.4: We create an individual BMC instance for every check. Such instances solved independently while only the results are merged.

EnableSlicing. Given an individual check, `QPR Verify` can compute a static backwards slice with the target being the single assertion representing the check. In theory, this can also be done for multiple checks, yet the benefits would strongly decrease for multiple checks, because this would often result in a slice encompassing the full program. Our slicing inlines function calls and is thus interprocedural. Yet, our approach shown below often abstracts function call and thus the property needs to be softened. We name our level of slicing that is in between intraprocedural and interprocedural as intermediateprocedural slicing. The computation of this splice is accomplished via a rule system which recursively marks dependencies (instructions, basic blocks, and functions) of a slicing target.

In the following, we present the rules listed in Fig. 4.5. Note that in LLVM-IR, instructions are identified with their result values (this is possible due to the *single static assignment* form of LLVM-IR code). We also assume this behavior here, such that, e.g. in rule *(Op)*, the instructions computing $y_1, \ldots, y_n$ are marked as dependencies, when $x = op(y_1, \ldots, y_n)$ is dependent on the target.

- The *(Target)* rule marks the instruction computing the Boolean argument of the assertion as a dependency.
- The *(Op)* rule marks all instructions computing an argument of *op* as a dependency.
- Consistent with memory abstraction, the *(Ld)* rule marks no dependencies for `load` instructions.
- The *(Ca)* rule, for calls of non-void functions, marks the called function as a dependency, as well as the return instruction of the called function.
- The *(Cp)* rule handles passing dependencies from a called function's arguments back to the caller. If in the called function f one of f's (formal) parameters p_i is marked as dependent, then so is the corresponding argument a_i in the associated function call. Parameters of pointer type are ignored and do not result in a dependency of the respective a_i.

- Rule *(Vc)* is responsible for handling calls of functions without return values. As memory accesses are ignored due to abstraction and thus no relevant state change can occur in a function, a call to such a function can be ignored and results in no further dependencies.
- The *(Phi)* rule handles *phi* instructions and marks the non-basic-block arguments.
- In rule *(Rt)*, return instructions are handled. If a return instruction is a dependency, so is the returned value (resp. the instruction computing this value).
- Rule *(Bx)* marks basic blocks as dependency whenever an instruction within the basic block is a dependency.
- Rule *(Fb)* similarly marks functions as dependencies, if any of its basic blocks are dependencies.
- Rule *(Bp)* handles control dependencies and makes use of the control dependence graph: if a basic block is a dependency, so are all its predecessors in the Control Dependence Graph (CDG).
- The last two rules, *(CBr)* and *(UBr)*, take care of conditional and unconditional branches. For conditional branches, the condition (resp. the instruction computing the condition's Boolean value) is marked as a dependency. For unconditional branches no further dependencies are needed. In both cases, the control dependency is handled by rule (Bp), thus no additional dependencies on the branch targets are needed.

Applying the rules of the system can be handled in a bottom-up fashion following control and data dependencies backwards. A little care has to be taken for function calls, but, e.g., giving priority to dependencies of called functions (to compute dependent a_i's) is a sound strategy.

The dependencies and use-defs chains are extracted from the LLVM IR but are not complete for memory dependencies. Modeling main memory and accesses to it comes with a considerable overhead in solving verification problems. A simple way to over-approximate memory accesses is to assume that each `load` instruction returns an arbitrary, non-deterministic value. Thus `store`s as well as address calculations become irrelevant and can be ignored. While this is a rather coarse over-approximation, it turns out that it is still sufficient for proving many program properties that do not depend on memory.

StopEncodingAfterCheck. Another merit of regarding only a single check inside a program is that we can stop the encoding after reaching the regarded check. This, seemingly simple, optimization is not implemented in most solvers but can offer notable scalability improvements, if the check is located in the earlier part of the program under verification. Stopping the encoding after a regarded check is not always trivial, if the check is located inside a loop or inside repeatedly called functions.

$$\frac{T(\texttt{assert } x)}{D(x)}\ (Target) \qquad \frac{D(\texttt{ret } r)}{D(r)}\ (Rt)$$

$$\frac{D(x \texttt{ = } op(y_1, \ldots, y_n))}{D(y_1) \quad \cdots \quad D(y_n)}\ (Op) \qquad \frac{D(\mathrm{x})}{D(\texttt{BB}(\mathrm{x}))}\ (Bx)$$

$$\frac{D(x \texttt{ = load } p)}{}\ (Ld) \qquad \frac{D(\mathrm{bb})}{D(\texttt{F}(\mathrm{bb}))}\ (Fb)$$

$$\frac{D(x \texttt{ = call } f(a_1, \ldots, a_n))}{D(f) \quad D(\texttt{ret } r)}\ (Ca) \qquad \frac{D(bb)}{D(pred_{CDG}(bb))}\ (Bp)$$

$$\frac{D(p_i) \quad D(x \texttt{ = call } f(a_1, \ldots, a_n))}{D(a_i)}\ (Cp) \qquad \frac{D(\texttt{br } c\texttt{, } bb1\texttt{, } bb2)}{D(\mathrm{c})}\ (CBr)$$

$$\frac{D(\texttt{call } f(a_1, \ldots, a_n))}{}\ (Vc) \qquad \frac{D(\texttt{br } bb)}{}\ (UBr)$$

$$\frac{D(y = \texttt{phi } [x_1, bb_1], \ldots, [x_n, bb_n])}{D(x_1) \quad \cdots \quad D(x_n)}\ (Phi)$$

Figure 4.5: Rules for computing dependencies for slicing. $T(x)$ means that the object x is a slicing target, while $D(x)$ means that the object x is a dependency.

For such cases, `QPR Verify` tracks the start and end of function calls and loop iterations even after inlining and unrolling, similar to what a compiler does e.g. for valid bracket pairs in a program. The encoding is terminated after the last occurrence of the assertion and we assume again that the earlier checks do not fail. Stopping the encoding after the last occurrence of the check is sound, because no later instructions can influence the verification result due to the SSA form of the LLVM IR on which we encode the given program.

Optimize-Bitcode. Finally, the user can choose to run a number of optimizations on the analyzed bitcode. Such optimizations can simplify the bitcode and thus produce an easier formulae during encoding. We support several LLVM and LLBMC transformation phases that are targeted specifically at bitcode simplifications listed at [113]. The most notable of those is called *mem2reg*, which converts memory references to register. By rewriting allocas, loads and stores referencing local memory, it creates a "pruned" SSA form that is less memory depended. Additionally, in our evaluation, we often run the *instNamer* optimization, assigning names to LLVM variables for easier to read preconditions or intermediate results not leveled to C-code.

In later chapters, we present more novel abstraction and refinement configurations and strategies specific to the aim of according contributions (C1.2) and (C1.3). In our evaluation in Chapter 7, we demonstrate the relation and experimental advantages of configuration possibilities.

4.4.3 Global Analysis.

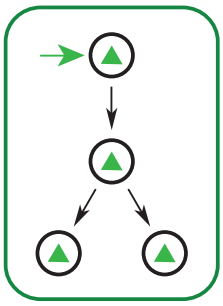


Figure 4.6: Global Analysis

The basic mode of analysis with `QPR Verify` is called *global analysis*. It performs the standard bounded model checking approach, where the program is encoded as a whole.

Let P be the program under analysis and C be a check or group of checks in P. With global analysis, `QPR Verify` checks whether C holds for all execution paths starting at the start of the main function. If there is an execution path violating C, the check result is set to `unsafe`. If C is shown to hold for all execution paths starting at main, the check result is set to `(globally) safe`. However, if some execution path was truncated during analysis, e.g. due to a loop bound having been reached, the check result for C is further qualified corresponding to the reason for the path truncation. The analysis can be performed by setting the entry point (default value is the function "main") and then starting the analysis.

4.4.4 Error Traces and Result Display

Simplified verification results are presented to the user as a tuple that assigns every check an unambiguous check result introduced in Section 4.4.1. Checks that are verified as `unsafe` or in case of abstractions as `cond.unsafe` have to be analyzed and in general fixed by the developer. Therefore, it is important to provide the user with enough information to identify and understand the reported error. One advantage of bounded model checking compared to other approaches like abstract interpretation is that they provide exact variable assignments through their underlying SAT model. LLBMC is capable of lifting such SAT model to the LLVM IR. While the LLVM IR can be read and understood by experienced user, it does not provide a direct translation back to the original C-code. Thus, `QPR Verify` transforms error traces on the IR level to C-code, such that a developer can analyze the error on the code-basis he or she is working on.

An error trace produced by LLBMC can be described as a list of LLVM instructions L_t. To transform a list of LLVM instructions into a program trace in the original C-code, information about relations between LLVM instructions and C-code fragments is needed. Branching decisions and function calls, can be derived through variable assignments. Therefore, the core information for every instruction is (1) the location of included variables in the original software code and (2) values assigned of those variables. To extract such information, we utilized LLVM debug information. If debug information are activated, information about the original program is maintained in LLVM metadata. Next to top-level information, various debug information is inserted into the LLVM IR as intrinsic function calls [55]. For a local variable, the intrinsic `void @llvm.dbg.declare(metadata, metadata, metadata)` provides the address containing the original column and row of declaration of the variable. `void @llvm.dbg.value(metadata, metadata, metadata)` then provides information about changes like new value assignments to that variable.

Computing such debug values is sufficient to calculate a mapping from LLVM variables to C-variables and thereby calculate an error trace containing variable assignments and branching decisions representing a single program trace leading to a failed check. Given such information the developer can marginally faster understand (and hopefully fix) the error. With the support of the student assistent Calvin Urankar, we developed an interactive graphical report consisting of general source code information, chosen configuration options during verification, check results and an interactive representation of error traces on the C-code level. The user interface allows to step forwards and backwards through the trace, similar to a source level debugger. This client can be found at `https://github.com/MarkoKleineBuening/DissertationTools` and is depicted in Figure 4.7.

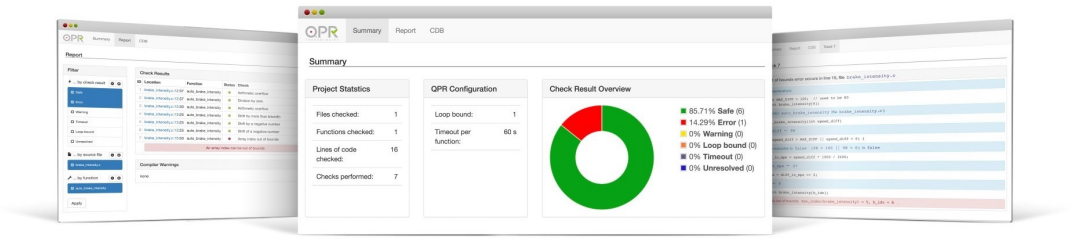


Figure 4.7: QPR client providing a project overview (center), check information (left) and detailed error traces (right).

4.5 Conclusion

In this chapter we presented `QPR Verify`, a static analysis tool based on software bounded model checking with a focus on industrial application. We devised requirements for the verification of industrial embedded software and implemented `QPR Verify` accordingly. Based on the bounded model checking approach, the tool produces precise verification results, while providing a range of configurations, abstractions and solving strategies for a scalable analysis. New features like a fast preprocessing analysis, solving strategies and the support of a wide range of C features have been implemented to fulfill industrial requirements. Providing additional information for checks, like error traces, in an interactive GUI should make the tool applicable for a wide user base.

To be applicable to industrial projects, the greatest remaining challenge is the size of software and thus the scalability of our approach. Therefore, the next chapter introduces a modularization to further scale our here presented approach.

Chapter 5

Automatic Modularization Techniques

5.1 Introduction

After introducing `QPR Verify` and its user-friendly and configurable design to set-up and verify larger projects, this chapter specifically addresses the scalability issue of software verification by bounded model checking. We present a modularization approach partitioning a program into smaller modules that can be verified individually. Our fully automatic modularization is based on newly developed structural abstractions and scales the approach to large software projects. The content of this chapter presents contribution **C1.2** and is based on research published in [KB4].

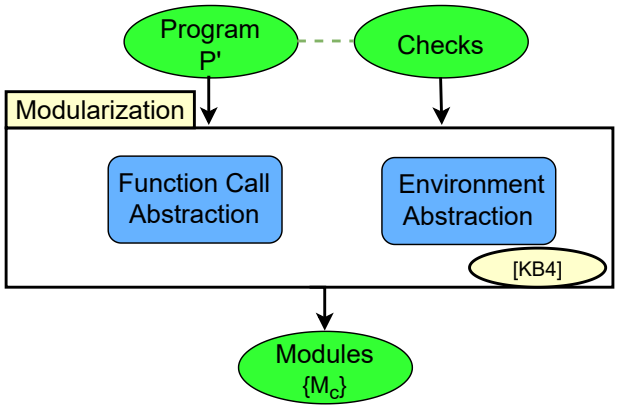


Figure 5.1: Modularization for our modular software verification approach.

Software verification with techniques like bounded model checking are making continuous progress, but at the same time the size of systems embedded in aircrafts, cars, or mobile phones grow even faster. Modern cars are currently at around 100 MLoC and

are estimated to go up to a total of 300 MLoC in the next years. Even current audio control software in a car can have several millions LoC and is thereby hardly verifiable by most if not all approaches. For bounded model checking, a program under verification has to be encoded into a logical formula. Even when ignoring time constraints, the memory requirements to encode millions of lines of code is not attainable by state-of-the-art systems. A well-known approach to increase scalability of software verification is to partition the program into smaller modules that can then be solved individually. Such modularization typically requires formalization of interfaces and dependencies between modules. Under the headline of *compositional verification* or *assume-guarantee reasoning* several approaches for modular verification have been proposed in the past [36, 66, 73]. Such work, however, generally does not cover the aspect of how to generate modules; instead it relies on manual approaches for partitioning. There exist frameworks that automate part of the modularization task, e.g., by creating necessary preconditions automatically through an incremental learning algorithm [37], or by deducting modules from program design [60]. However, these approaches do not provide a framework for fully automatic verification of large systems. The same applies to modular interactive approaches like [12, 123], where the user has to manually write interface specifications. The number of lines of specification that has to be written for one line of source code varies depending on approach and application. Typical factors range between 2 for specialized [124], 5 for SMT-based [71] or up to 20 for interactive theorem prover approaches [83]. It is apparent that this is not feasible for projects with millions of lines of code.

Contribution C1.2. To automatically verify large projects, an automatic modularization is needed. Therefore, this chapter presents a formal definition of program semantics for LLVM and general modularization. We develop two main modularization approaches in the context of bounded model checking using two different types of abstraction. First optimizations and refinements of these abstraction are also introduced. To argue over the quality of a modularization, we define mandatory and desirable properties for an automatic modularization in the context of software verification. The modularization approaches are implemented into `QPR Verify` and the evaluation in Chapter 7 shows a significant increase in scalability through the presented abstractions.

Structure. We first introduce definitions of program semantics and modules in Section 5.2 to then describe automatic modularization approaches based on abstractions in Section 5.3. In Subsection 5.3.1, we define a general model for program modularization, followed by four concrete modularization techniques in the context of bounded model checking using two different types of abstractions and two refinement possibilities that are discussed in Chapter 6 in more detail. We then define mandatory and desirable properties of an automatic modularization procedure for software verification in Section 5.3.2. Finally, we give a short conclusion in Section 5.4.

5.2 Theoretical Foundations

We base our modularization on the in Section 2.1.1 introduced LLVM IR. In the following, we define a program trace semantics for LLVM IR and thereupon a modularization of programs in LLVM IR for software verification. This allows a sound definition of modules and a clear overview of state space over-approximations. The introduced definitions and their description in this section correspond to our definitions published in [KB4].

5.2.1 Program Semantics of LLVM

We define the semantics of an LLVM IR program as a set of program *traces*. A trace T is a (possibly infinite) sequence of program states $T = (s_0, s_1, \dots, s_n, \dots)$, and the trace semantics of a program encompasses all traces the program can take. We denote the set of all such traces by $\mathcal{T}_P$. The set S of states is defined as

$$S = (Var \to Val) \times (Adr \to Val) \times Loc^* \quad .$$

A state $s = (v, m, l)$ is thus a triple consisting of a variable-value-map v, a representation m of the memory content (including stack variables generated by LLVM's `alloca`), and a representation l for a program location, which is a sequence of triples $t = (f, b, i)$ encoding the call stack. Each triple consists of a function f, a basic block b and an instruction number i, consecutively numbering the instructions within basic block b, starting at 0. The first element in sequence l is the stack top, which we also denote by $l_{\texttt{top}}$, or $l_{\texttt{top}}(s)$, if we want to denote the topmost frame in the location stack of state s.

We assume that $Var = GVar \cup LVar$ is the set of program variables, split into local and global variables; $LVar = Loc^* \times Name$ characterizes a local variable consisting of a call stack and a name; $GVar = Name$ denotes a global variable (which, in LLVM, is always a pointer variable); $Val = Int \cup Adr$ is the set of variable values, consisting of integer variables and pointer variables[1]. To simplify access to both local and global variables by name n in a given call stack l, we define a variable's stack-related name n_l by

$$n^l = \begin{cases} n & \text{if } n \in GVar \\ (l, n) & \text{if } n \in LVar \end{cases}$$

Each trace has to start in an initial state $s_0 \in I$, and the effects of LLVM operations is defined via transition relations $\tau : S \to \mathbb{P}(S)$. We define transition relations for instructions and functions. As the transition relation may be non-deterministic, each state can have multiple successors (next-states).

[1]For simplicity, we assume that integer and pointer variables have the same bit-width, and that all program variables are of type integer. We also identify pointer values with integers, such that $Val = Adr$. A more refined model would differentiate between different data types stored in memory (including floating-point). In practice, a byte-oriented memory model is often used [114].

For an instruction I and a state $s_i = (v, m, l)$ we have, e.g.,

$$\tau_{x=\texttt{load}\,p}(v,m,l) = \{(v[x^l \leftarrow m(v(p^l))], m, next(l))\}$$
$$\tau_{\texttt{store}\,x,p}(v,m,l) = \{(v, m[v(p^l) \leftarrow v(x^l)], next(l))\}$$
$$\tau_{z=x\ \texttt{<op>}\ y}(v,m,l) = \{v[z^l \leftarrow v(x^l)\ \texttt{<op>}\ v(y^l)], m, next(l))\}$$
$$\tau_{\texttt{br}\,c,bb_1,bb_2}(v,m,(f,b,i):l) = \begin{cases}\{(v,m,(f,bb_1,0):l)\} & \text{if } v(c^l) \neq 0\\ \{(v,m,(f,bb_2,0):l)\} & \text{if } v(c^l) = 0\end{cases}$$

$$\tau_{y=\texttt{call}\,g(x_1,\ldots,x_n)}(v,m,l) = \{(v',m,l^*) \mid v' \in V\}$$
where $l^* = ((g, bb_{\texttt{Entry}}, 0) : l)$,
$v^* = v[p_1^{l^*} \leftarrow v(x_1^l)]\cdots[p_n^{l^*} \leftarrow v(x_n^l)]$,
$V = \{v^*$ updated with local variables set to arbitrary values in the topmost stack frame$\}$,
and p_i are the actual parameters of the called function g

$$\tau_{\texttt{ret}\,y}(v,m,t:l) = \{(v[ret(y) \leftarrow v(y^{(t:l)})], m, next(l))\}$$
where $ret(y)$ = the return var. in the call instr. at loc. t

$$\tau_{x=\texttt{nondet}\,t}(v,m,l) = \{(v[x \leftarrow i], m, next(l)) \mid i \in Val\}$$

Here, $f[x \leftarrow y]$ stands for updating the function f at location x to a new value y; $next : Loc^* \to Loc^* : ((f,b,i):l) \mapsto ((f,b,i+1):l)$ computes the next location within the top-most basic block of the call stack (":" shall denote the list constructor) for a non-terminator instruction.

We define the set of initial states I by

$I = \{(v,m,l) \mid$
$v(g)$ = address of global variable g for all globals
$v(x^l)$ = arbitrary value for local variable x
m : any function $Adr \to Val$, respecting initializers for globals
$l = (\texttt{main}, bb_{\texttt{Entry}}, 0)$ $\}$

A trace T for an LLVM program P is then defined as a sequence $s_0, s_1, \ldots$ of states with $s_0 \in I$ and $s_{i+1} \in \tau_{Inst}(s_i)$, where $Inst$ is the instruction at $l_{\texttt{top}}(s_i)$, i.e. the program location of the top-most stack frame. The semantics of program P is the set of all such traces.

In our modularization approach, we also want to define traces that start at the entry of a particular function up to the execution of the last instruction in this function.

We thus define trace sets $\mathcal{T}_f$ for functions f in a program P:

$$\begin{aligned}\mathcal{T}_f = \{(s_i, \ldots, s_j) \mid\ & (s_0, \ldots, s_n, \ldots) \in \mathcal{T}_P \text{ and} \\ & s_i = (v, m, (f, bb_{\texttt{Entry}}, 0) : l) \text{ and} \\ & s_j = (v', m', (f, bb_{\texttt{ret}}, k_{\texttt{ret}}) : l) \\ & j > i \text{ is the smallest index such that } (f, bb_{\texttt{ret}}, k_{\texttt{ret}}) \\ & \text{is the location of a } \texttt{ret} \text{ instruction} \\ & \text{for some } v, v', m, m', l, bb_{\texttt{ret}} \text{ and } k_{\texttt{ret}} \quad \}\end{aligned}$$

5.2.2 Modularization

There are several possible views on what a module in a program is. We thus want to give, in a first step, a very general definition of a module. Later, we will identify properties that are mandatory in the context of software verification and present desirable properties of a modularization suiting verification techniques leading to accordingly refined definitions.

The uniform definition will serve as the foundation of properties, statements and approaches for modularization. The static analysis of software and the definition of programming errors are always closely related to the compiler framework the program is based on. In the context of bounded model checking, we relate closely to the described compiler framework LLVM and developed the definitions in close relation with the additional advantage that we do not have to discuss the diversity of language features of programming languages like C and C++.

We define a module as a fragment of code and thus create a definition of a program parallel to the trace semantic on source code level. Let a function and global variables be defined as by standard notions in LLVM.

Definition 4 (Program)**.** *A (LLVM) program $P = (\mathcal{F}, \mathcal{G})$ is a tupel of a non-empty set of functions $\mathcal{F} = \{f_1, ..., f_n\}$ $(n \geq 1)$ and a set $\mathcal{G} = \{g_1, ..., g_m\}$ $(m \geq 0)$ of global variables that may be referenced in the functions f_i.*

We do not demand that there is a unique entry point in the program (a `main` function) nor that the program is "closed" in the sense that all functions called in $\mathcal{F}$ are contained in $\mathcal{F}$.

A module is then just a subset of the functions and global variables in a program.

Definition 5 (Module)**.** *Given a program $P = (\mathcal{F}, \mathcal{G})$ and sets $\mathcal{F}', \mathcal{G}'$ with $\varnothing \subset \mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{G}' \subseteq \mathcal{G}$, $M_P = (\mathcal{F}', \mathcal{G}')$ is a module for program P.*

Note that a module is itself a program according to our definition. Such a broad definition allows for a number of different modularization approaches and is thus beneficial for the exploration of diverse strategies. After introducing decomposition approaches, we will restrict modularizations by introducing properties in Section 5.3.2. We assume that program properties that are supposed to be verified are included in the program in the form of `assert` instructions as described in chapter 4. Thus, a module "inherits" the *checks* from the program that it is a part of via assertions present in $\mathcal{F}'$.

In some decompositions that we present in the following, we do not require that all checks are inherited from the original program. Instead we sometimes will allow that only a subset of the inherited assertions are present in a module, i.e. we only inherit checks for a subset of the functions $\mathcal{F}'$.

5.3 Decomposition of Programs

We will introduce two modularization approaches which partition a program P into a set of modules $M_P^1, \ldots, M_P^n$ such that a bounded model checker can derive verification results about program P by verifying each M_P individually. Dropping parts of the program in a module of course loses information. In our modularization approaches we do not require to add specifications about missing parts of the program. Instead, we want to make sure that a module represents an over-approximation (i.e. abstraction) of the original program in the sense that the set of program traces that we check for a module are a superset of the traces present in the original, complete program.

In this case, it is to note that the modularization also has to decide where to insert checks. The approach can only activate checks in certain modules or part of modules as long as all program checks are verified by the sum of verifying all modules. We base our modularization on abstractions that are necessary to decompose the program into verifiable subsets.

5.3.1 Structural Abstractions

Abstractions are an important technique to simplify verification tasks. Most often abstractions are over-approximations of variable values (such as in abstract interpretation [40]). The abstractions that we are interested in are different and of a "structural" kind. We abstract function calls and replace them by over-approximations of the function behavior, or we ignore the calling context of a function in a larger program. In applying these structural abstractions, we distinguish between abstracting the program "bottom up", where we abstract away called functions, and "top down", where we abstract away the calling context.
We will now describe our two abstraction approaches in detail and present first refinement ideas.

5.3.1.1 Function Call Abstraction

The first approach abstracts away calls to functions outside of the chosen module M_P. This process is also called *havocing* function calls. At first, assume that M_P only contains one function f and all global variables that are either read or written in f: $M_P = (\{f\}, \mathcal{G}')$. We will later explain how to meaningful extend the initial module size of M_P to more then one function f. To verify M_P, we only keep checks (assertions) in function f, and abstract away all functions calls in f. When abstracting a function call without any further knowledge, an over-approximation of its behavior has to be assumed. Next to the return value (if existent), memory content (including global variables) can be altered by the called function, and thus have to be assumed to be arbitrary.

Therefore, to abstract away a function call in the context of LLVM means to set the return value and the memory content to nondeterministic values (`nondet`). Referencing the trace semantics of a program, the abstraction updates all transition relations $\tau_{y=\texttt{call}\, f^*(\dots)}$ where f^* is not part of the function set of M_P. The transition relation for such a call to f^* is replaced by the following:

$$\tau_{y=\texttt{call}\, f^*(x_1,\dots,x_n)}(v,m,l) = \{(v[y^l \leftarrow i], m', next(l)) \mid i \in Val, m' \in (Adr \rightarrow Val)\}\,.$$

I.e., the variable that takes the value returned by f^* can be an arbitrary value and the memory in the follow-up state can be an arbitrary function $Adr \rightarrow Val$.

The updated transition relation τ allows a higher number of possible program traces and is thus a clear over-approximation of the program semantics. Therefore, updating τ guarantees soundness of the approach. The cost of such an over approximation is the possibility of false positives – error reports where there actually is no error. The amount of possible false positives positively correlates to the number of additionally allowed program traces through undefined values. Calculating the exact number of additional traces when abstracting function calls is not efficiently computable. Therefore, when refining such abstractions, we assume that all abstracted function calls are similar in reference to additional program traces.

Given the module containing only function f, we thus abstract all function calls from f. Yet, to verify our module M_P, we have to model the calling context of f. To precisely include the calling context of f, the approach includes any function with a (transitive) call to function f into the module M_P, together with accessed global variables. Given the call graph of P, $F \rightarrow G$ indicates that there is a call from function F to function G (in program P). Therefore, our module has to include all calling functions of the module's function, i.e., if $G \in M$ and $F \rightarrow G$, then $F \in M$. We call such a module *caller-closed*. Modules generated by function call abstraction are caller-closed and thus contain the `main` function or a top-level API function, which is used as an entry point for analysis.

Our slicing algorithm presented in Section 4.4.1, can be utilized to improve the creation of M_P by removing function calls that do not contribute to a precise calling context of f and are thereby not needed to verify the checks in f. Such the slicing algorithm has the potential to minimize the module size.

Figure 5.2 shows the modularization of a simple program with four functions applying the described abstraction. The green arrows are representing the entry point for verification. The triangles are representing the checks that are verified and the boxes represent the modules M_P. The dotted boxes are parts of the program that are likely to be removed by our slicing algorithm dependent on the implementation.

Summarizing the first abstraction, we see that verifying a module created in this way encompasses fewer functions and thus increases scalability compared to a global analysis. Yet, for functions deep in the call graph the module size can still be too large and the abstraction can lead to false positives. In particular when checking for memory properties, the complete havoc of the heap at each call to a function outside of the module can lead to false error reports.

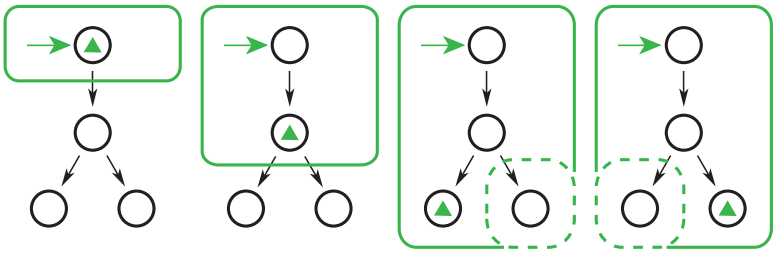


Figure 5.2: Modularization into four independent modules based on abstraction of called functions. The entry point for every module is the *main* function and the abstraction of called functions starts depending on the location of the assertions.

The memory content does not need to be fully abstracted at every function call. Larger projects often contain logging functions or output functions that print the status of the modeled state machine. Such functions do not have a relevant return value or affect the memory state of the program. Utilizing the LLVM framework, we can check whether a function is clearly side-effect free. If a function does not write in memory, we do not need to abstract the full memory assignments but can just remove this function call from our program model.

To further refine the abstraction, we do not have to abstract function calls in f but can abstract function calls deeper in the call graph of f. Given our function f, our initial approach abstracts away every function call in f. Given the call graph of P every function $g \in G$ with $f \to G$ is abstracted. We write $F \to_i G$, with $i \in \mathbb{N}_+$, iff there is a path from F to G in the call graph of length i. Our initial function call abstraction havocs all function with depth $i = 1$. We can increase this *havoc-limit* to meaningfully increase our module size. Setting the havoc-limit to 2, our approach does not abstract function calls in f but only function calls in the callees of f. We are thereby able to increase the module size and refine the abstraction of our modularization. Setting the havoc-limit to the call-depth of the program leads to a whole-program analysis. The havoc-limit can be seen as a trade-off parameter between precision and scalability. Our evaluation showed that increasing the havoc-limit increased precision but produced too large modules fast, because of the often exponential growth of the module for each added function depth.

5.3.1.2 Use Postconditions of Abstracted Functions

As a refinement that does not increase the size of modules but refines the formulae representing the possible program traces, we propose to insert postconditions for abstracted functions. One can create postconditions of called functions and replace the call of a function outside a module with the function's postcondition instead of fully abstracting the function. While general postconditions about function behavior might be used, they are computationally costly to calculate and often need manual input to fully represent the analyzed function. Therefore, we focus here on memory-related postconditions that result from an automatic analysis of memory locations that are written in functions abstracted away. The modularization itself stays the same as in the earlier approach, only

the abstraction is refined and thus the transition relation update of τ to minimize the number of possible false positives.

As a first refinement step, we analyze memory accesses in the called function f^* with the aim of reducing memory locations we have to set to `nondet`. A rather simple analysis over the LLVM IR gives us all accesses of pointer and global variables in f^* and further called functions. After gaining all relevant memory changes, we have to obtain points-to information so that we can havoc only those memory locations written to by possible executions of f^*. This points-to information can be gained through a scalable and flow-sensitive alias analysis like e.g. described in [70]. The alias analysis has to be scalable to be run on large programs without negating the scalability benefit. Furthermore, a flow-sensitive approach takes the program flow into account and ignores the later-on called functions providing the necessary level of precision for our postconditions.

We denote the set of memory locations that have to be abstracted by $AbsM$. We then update the transition relation for a call instruction to a function f^* outside of the module $\tau_{y=\texttt{call}\, f^*(\dots)}$ to

$$\begin{aligned}\tau_{y=\texttt{call}\, f^*(x_1,\dots,x_n)}(v,m,l) = \{(v[y^l \leftarrow i], m', next(l)) \\ \mid i \in Val, m'(a) = j \text{ with } j \in Val, \text{ if } a \in AbsM, \\ \text{and } m'(a) = m(a) \text{ for all } a \notin AbsM\}\end{aligned}$$

Again, it is clear that the update of τ leads to more traces of P and is thus an over-approximation guaranteeing soundness.

Alias-based postconditions were generated and inserted into `QPR Verify` during a student research project with Moritz Laupichler under the authors supervision in the context of the research seminar "Praxis der Forschung" at the KIT.

Choosing a fitting alias analysis approach meeting scalability and precision requirements was the first challenge. The aim of our modularization was to scale the bounded model checking approach to millions of lines of code, thus the alias analysis needs to be scalable. Furthermore, the algorithm needs to be struct- and array-sensitive to prevent falling back to over-approximation when faced with arrays and structs that often appear in embedded C-code.

Through a survey of state-of-the-art alias analysis approach, we utilized and adjusted SeaDsa [68], which presents a scalable, context-sensitive approach computing detailed, struct- and array-sensitive points-to graphs for functions. The alias analysis of SeaDsa was integrated into `QPR Verify` and experiments were conducted to evaluate the precision gain and access scalability. The experiments showed that the integrated alias analysis was not scalable beyond around 200 KLoC and did not bring any precision mertis for smaller projects. Because of the scalability issues of tested alias analysis approaches and no clear advantages for smaller projects, postconditions as a refinement step will not be further regarded in this thesis and our evaluation in Chapter 7. Yet together with a scalable alias-analysis postconditions are a promising refinement step and will be discussed in future work in Chapter 9.

5.3.1.3 Call Environment Abstraction.

The first approach abstracted the program bottom up by regarding function calls. The next approach address the problem by abstracting the caller of module M_P. We again start with the assumption that the function set of M_P consists of one function f and the global variable set G' is created accordingly. We again insert checks only into f. In contrast to the earlier approach, we do not abstract the transition relation τ of instructions, but the initial states I of the analysis. Thus we abstract the call context and the input parameters of f. We thereby do not have to include all functions of the call graph prior to f and can thus modularize the problem. The abstraction of the initial states I' for f is done by setting

$$
\begin{aligned}
I' = \{(v, m, l) \mid \\
& v(g) = \text{address of global variable } g \text{ for all globals} \\
& v(x^l) = \text{arbitrary value for local variable } x \text{ for all local vars.} \\
& v(p_k^l) = \text{arbitrary value for parameter } k \text{ of function } f \\
& m : \text{any function } Adr \rightarrow Val \\
& l = (f, bb_{\texttt{Entry}}, 0) \quad \}
\end{aligned}
$$

and considering function f as the start of the program. Note that we do not initialize global variables here, as their values may have changed before entering function f.

The new set of initial states is a superset of possible states that would be calculated during a normal program execution. Thus, the abstraction is again an over-approximation guaranteeing the soundness of the approach. For M_P to be verifiable, the approach has to include all called functions in M_P because no abstraction is defined and the transition relation τ needs the exact function semantics. The approach iteratively adds called functions while also adding all global variables that are needed. We call such a module M *callee-closed* if all call instructions in M invoke functions that are already part of M, i.e. if $F \in M$ and $F \rightarrow G$, then $G \in M$. The modularization is demonstrated on the same abstract program in Figure 5.3. The notations are equal to the figure above.

Compared to the over-approximation of function calls, which can happen any number of times inside a module, the abstraction of the initial function call over-approximates the state only once. Furthermore, such an analysis can match user concepts. If a function is proven correct using this abstraction, the function is safe from error in every call environment. Such statements are recommended for library functions or functions that are accessible throughout the system. We later refer to the verification of such modules as *call-environment analysis* or *local analysis*.

In reality developers can argue against such an abstraction by stating that the input necessary for the error can never occur in the current program setting. Thus, we present an refinement idea collecting more information and restrictions on the abstracted input.

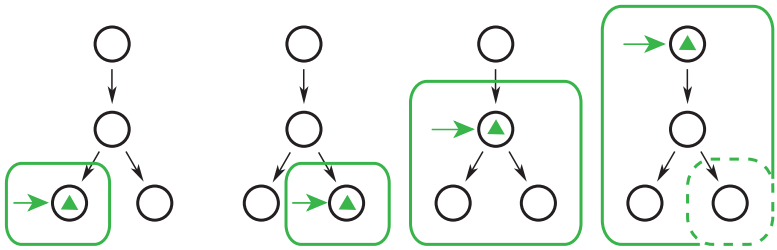


Figure 5.3: Modularization into four independent modules based on abstraction of call environment. The entry point for every module is the function containing the assertion. Through the abstraction of the call environment no prior functions have to be included.

5.3.1.4 Generate Preconditions for Entry Function.

Similar to the previous refinement, we refine the abstraction using additional conditions that hold when the entry function is called. We create preconditions to restrict possible inputs to minimize the amount of false positives. The overall concept and idea is presented here before two formulated and implemented precondition approaches are shown in more detail in Chapter 6.

Generating exhaustive preconditions for a given function is a research field of its own. The automatic generation of precise preconditions for large programs is currently not feasible in reasonable time. Preconditions that represent all possible call environments would have to encapsulate the complete prior program execution and are thus very costly. Nevertheless an automatic generation of such preconditions is possible using coarse approaches like abstract interpretation with an interval domain to generate value ranges for possible inputs.

Nevertheless, the idea we chose is to create preconditions not based on the prior program execution but based on the erroneous checks of our function f. First, we perform an analysis without preconditions possibly resulting in failed assertions of two kinds: `unsafe` and `cond. unsafe`. Globally unsafe checks are such checks that will fail independent of the input, a simplified example would be the statement (`x = y/0`). The `unsafe` status is given to checks for which an error is found that dependents on input values of f (parameters and memory state). The precondition generation only regards `cond. unsafe` checks. For every check, we create a precondition representing the input for which the error arises.

Bounded model checking can create an exact error trace for a failed assertion in the abstracted program. Using a symbolic execution approach can generate preconditions by following constraints backwards from the location of the failed assertion up to the start of function f. The transition relation τ is therefore inverted and symbolically executed. The symbolic execution is built upon the earlier executed bounded model checking attempt. The program is already inlined and the loops are unrolled up to a given bound. Furthermore, the exact error trace gives restrictions on branching possibilities.

After the creation of such a (partial) precondition for a trace, the function has to be re-verified and the procedure to be repeated until all traces that lead to a failed assertion are covered. The amount of traces are assumed to be small because simple errors that occur on all traces are found earlier and marked globally unsafe while only `cond. unsafe` locations, which only appear for a subset of traces, are checked for false positives. The conjunction of the resulting constraints is negated to form the precondition for f and thereby represents all input values for which there is no error in the module M_P. After generating such an over-approximating precondition for a check, the precondition is inserted into the initial state formula for module M_P. The approach iteratively chooses all modules containing the precondition and verifies the module while deactivating all internal checks. If the precondition holds, we have proven that the check is `safe`. If the precondition does not hold, the process is repeated iteratively until we reached the main function or abord due to time or memory constraints.

Such a refinement of the call environment abstraction is discussed in Chapter 6 in detail. There, we develop two kind of preconditions representing module inputs leading to failed checks.

5.3.1.5 Extension and combination of abstractions.

The two modularization approaches and their refinement ideas were described by starting with a single function in which checks are inserted. As mentioned earlier, the approach works the same way when starting with modules of bigger size. These enlargements of modules reduce the amount of abstractions and thereby the amount of false positives generated. The cost of such larger modules is the scalability of the approach. For the function call abstraction modules can be meaningfully extended through a deeper abstraction of function calls. An extension method for the call environment abstraction will be presented in Chapter 6.

Furthermore, a combination of the above abstractions is possible to improve scalability or to refine the verification. For programs with a deep call graph the inclusion of either the functions calling the module or all the called functions can still lead to formulas which are too large to be handled by an SMT solver. Thus, we can separate the program into three parts based on the call graph to further improve scalability: Top level modules are verified using the postcondition abstractions and bottom level modules are solved using the precondition abstraction. For modules found in the middle part of the call graph both pre- and post-conditions are necessary. Another possibility is to run the different approaches one after each other to refine the analysis result step wise. For every analysis only the checks which are marked as `cond. unsafe` or `undetermined` are rechecked using a different abstraction.

5.3.2 Properties of Modularization

We want to define properties that every module and the total modularization should strive towards. We divide them into *mandatory properties* that are necessary to guaran-

tee the soundness of the verification approach and *success properties* that every module should strive towards for a high probability of optimal modularization for verification. These success properties can often be represented by optimization problems that are again NP-hard and thus for efficient applications approximations will be necessary. A shorter description of these properties can also be found at [KB4].

5.3.2.1 Mandatory Properties:

Given a program P and a modularization $\mathcal{M}_P$. Following properties have to hold for every valid modularization.

Total-Coverage: The union of all modules has to cover the whole program, and each check has to be included at least once in every function. Every function has to appear in at least one module and thus the union of all functions included in modules represent the complete function set of P. The same is not required for the set of global variable symbols, because of, e.g., unused symbols that do not influence the program. These are not advisable, because, if included, they are encoded and they optimized out of the program leading to unnecessary overhead. Yet they do not influence the soundness of verification tasks. When verifying a program it is easy to see that all functions and all checks must be at least once verified and that not only from the perspective of one module (one entry point) but from every possible input call. Later on it is shown that this is given by the total coverage property together with the third property, the information principle.

Single-Entry: Every module $M_i \in \mathcal{M}_P$ should have one single entry point from which the verification starts. For verification methods like bounded model checking the encoding of the program has to start at one entry point. When verifying a program with multiple entry functions, for example a library with a number of API functions, several verification jobs have to be run. These jobs can be run independently and also in parallel. In our case, we regard them as disjoint verification jobs and thus choose to enforce the single-entry property. Otherwise every smart verification approach would solve the module in two separate and independent runs which would present an inner modularization which we want to make explicit. To make the modules larger and to simplify the human understanding of the modularization, one could summarize modules with more then one entry point. Consider, e.g., a SAT solver that has been integrated into a larger program. The API has three interface methods of $addClause()$, $removeClause()$ and $solveFormula()$ and one can argue that a manually created module would be the complete SAT solver. However, the verification approach would analyze every interface call, including deeper function calls, according to the call site in the larger program as single modules. Thus, such a summary would benefit the human understanding, but not the performance of verification tasks.

We assume that each module possesses a unique function F_e, the *entry point function*, which has no callers in the module.

Information-Principle: All information that is needed for the sound verification of the module is included in the module itself. Meaning that all functions and global variables that are written to or read from are included in the verification task. Furthermore, the input of the entry point function or an abstraction of it has to be included.

A module that is callee-closed and caller-closed would result in the whole program P. Therefore, over-approximations of the program behavior is needed to generate a meaningful modularization. Over-approximation in contrast to under-approximations are needed to guarantee soundness.

Computable: The modularization should be computable in polynomial time with respect to the size of the input program. The separation of graphs into a fixed number of partitions that have minimal amount of edges between them is closely related to our partitioning. Edges in this case can be regarded as call or data dependencies. The so-called k-partitioning problem itself is NP-hard and thus one can assume that also precise algorithms for the efficient modularization of a program will have a similar complexity. Most likely, as in the case of the k-partitioning problem, we have to use abstractions and approximations of an ideal modularization that are computable in reasonable time. Otherwise the modularization negates the scalability advantage for the aspired verification.

5.3.2.2 Success Properties:

Given a program P and a modularization $\mathcal{M}_P$. The following properties should be striven towards by every *efficient* modularization.

Solvable: The size and complexity of a module should be manageable by the chosen verification approach, in our case bounded model checking. The module size that is manageable by a given approach depends on the programming style and the design of single functions. The scalability of bounded model checking approaches limits at program sizes of about 10-100 KLoC of C code represented by at maximum several hundred functions. On the other hand there are examples where a single function containing only a few lines of code is not manageable in reasonable time [11].

Minimal Dependencies: The second success property addresses the amount of dependencies between modules and thus the quantity of pre-/post-conditions or `nondet`-abstractions generated. We distinguish between call and data dependencies based on a graph structure. Let there be a node for every function in P, and let edges describe either call or data dependencies, then a directed edge in graph G_P from function f_1 to function f_2 represents one of the following: (1) function call from f_1 to f_2, (2) memory read in f_2 after a memory write in f_1 at the same location. The modularization of a program summarizes nodes and thereby also incoming and outgoing edges into modules. The minimal dependencies property states that the overall number of edges between modules should be as low as possible.

It should not be the aim of any modularization to minimize the dependencies for large programs. For a modularization $\mathcal{M}_P = \{P\}$, there would be no dependencies, but $\mathcal{M}_P$ would not be solvable in reasonable time. With equal intention one should be careful optimizing only scalability by analyzing every function by itself, which would lead to the maximal number of dependencies between the modules. One has to find a balance between these two properties. Current practical implementations for modularization have a tendency towards regarding every function by itself. Furthermore, while the sizes of modules can vary considerably, so can the complexity of the included functions. Finding heuristics for optimal module sizes considering both properties is part of future work.

5.4 Conclusion and Future Work

In this chapter, we defined a denotational program semantics for LLVM as well as notions for modularization of LLVM programs. Based on these notions, we developed four fully automatic modularization approaches. The discussion of mandatory and success properties for a modularization in the context of software verification is a foundation for further future modularization approaches. The implementation and evaluation of our presented modularization will be presented in Chapter 7 and shows the scalability improvements of our introduced abstraction approaches.

Through call environment and function call abstraction, our modularization can in theory scale to programs of arbitrary size. Yet introducing to many abstraction with modules including only precise information about a single function leads to a high number of false positives. To reduce the number of possible false positives, the next chapter will present a refinement of the call environment abstraction introduced in Section 5.3 through three refinements steps with a focus on automatically generated preconditions.

Chapter 6

Refinement through Enumerative and Learned Preconditions

6.1 Introduction

This chapter extends on Chapter 5 by refining the abstractions introduced by the automatic modularization. We present contribution **C1.3**, which is based on work currently in preparation for publication [KB3].

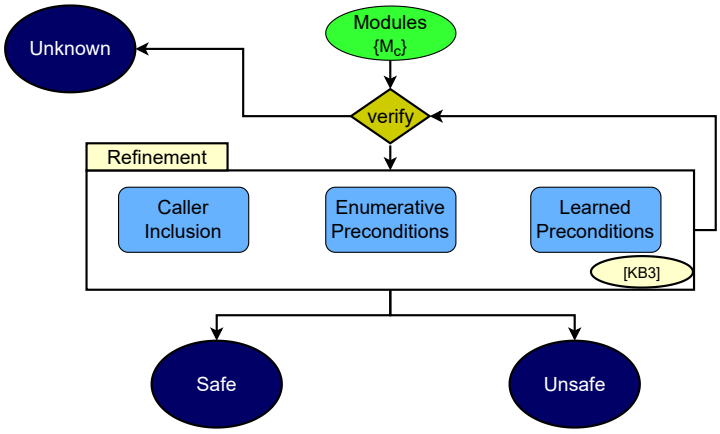


Figure 6.1: Modularization for our modular software verification approach.

Fully automatic modularization as presented before utilizes structural abstractions of program behavior and through over-approximations can divide the program with quite general module specifications. This leads to a fast and modular analysis but is prone to false positives (error messages where there are no errors). We concentrate on the call environment abstraction, which fully abstracts the input parameter and memory assignments before the call of our entry function. Structural abstractions are novel in the context of whole-program verification, thus there are currently no refinement strategies.

Yet, preconditions are often utilized to model program or function inputs. E.g. Design by contract [116] systematically requires users to provide preconditions, postconditions, and (loop) invariants. In Section 5.1, we established that manual specifications are not practical when verifying large projects and thus need an automatic approach. In contrast to current approaches, where preconditions are aiming at specifying the complete function behavior, our goal is to automatically generate preconditions that given a check to be verified specify the input leading to undefined behavior and thus an error. Trough this description of erroneous inputs, we can refine our modularization and reduce the amount of false positives introduced by our modularization without counterbalancing scalability.

Contribution C1.3. In this chapter, we present an approach that refines structural abstractions for C-programs. We incrementally increase the size of modules to exhaust the limits of bounded model checkers. Furthermore, we refine abstractions through automatically generated preconditions. Based on an enhanced output of the bounded model checking approach, we generate enumerative preconditions that represent erroneous input (including memory) for the entry function of a module. These possibly under-approximated preconditions are then generalized by a tree-based learning approach. By substitution of function calls with preconditions, the approach can refine the verification task and thus minimize false positives. The approaches are implemented and evaluated on real-world software projects, including SQLite with nearly half a million lines of code. The generated preconditions are human-readable and can thus be examined and adjusted.

Structure. In Section 6.2, we give an overview of our approach showing three refinement steps and their application on a small example. The first refinement step, which extends modules through the inclusion of callers is described in Section 6.4. Afterwards, Section 6.5 present two refinement steps based on preconditions. Summarizing enumerative counterexamples from the bounded model checker, leads to enumeratrive preconditions in Section 6.5.1.2. While 6.5.2, generalizes the bounded model checking output leading to learning based precondition. Finally, we give a conclusion of our refinement approaches in Section 6.6.

6.2 Bird's Eye View of the Method

We present an overview of the method on a simple example given in Fig. 6.2. Function `top` calls functions `mid1` and `mid2`, which in turn call function `bot`. Modularization through call environment abstraction, as presented in Section 5.3, will produce four modules $M_1 = \{\texttt{bot}\}$, with checks $C(\texttt{bot})$, $M_2 = \{\texttt{mid1}, \texttt{bot}\}$, with $C(\texttt{mid1})$, $M_3 = \{\texttt{mid2}, \texttt{bot}\}$, with $C(\texttt{mid2})$ and $M_4 = \{\texttt{top}, \texttt{mid1}, \texttt{mid2}, \texttt{bot}\}$, with $C(\texttt{top})$. Our goal is to verify that the addition $x = x + 2$ in function `bot` does not overflow (assuming 32-bit integers). In our modular approach, we start by verifying M_1. The bounded model checker returns `unsafe`, because both x and y are unrestricted inputs assumed to take arbitrary values.

```
void bot(int x, int y){
  if(y >= 0 && y < 100){
    //assert(x<2147483646)
    x = x + 2;
  }
}

void mid2(int x, int y){
  if(y > 100){
    bot(x,y);
  }
}
```

```
void mid1(int x, int y){
  if(x > 0){
    x = x - 1;
  bot(x,y);
  }
}

void top(int x, int y){
  if(x > 0){
    x = x - 1;
    mid1(x,y);
    mid2(x,y);
  }
}
```

Figure 6.2: Example for bird's eye view of the method.

By **inclusion of callers**, we can refine these arbitrary values by taking the calling functions of `bot` into account. Our approach creates new modules $M_1^{mid1} = \{\texttt{mid1}, \texttt{bot}\}$ and $M_1^{mid2} = \{\texttt{mid2}, \texttt{bot}\}$, checking $C(\texttt{bot})$ with entry point functions `mid1` and `mid2`, respectively. The bounded model checker recognizes that M_1^{mid2} is `safe`, because of the restriction on y, but M_1^{mid1} is still `unsafe`. Thus, a third new module is created $M_1^{top} = \{\texttt{top}, \texttt{mid1}, \texttt{mid2}, \texttt{bot}\}$, checking $C(\texttt{bot})$ with entry function `top`. Note that `mid1` has to be included, for M_1^{top} to be verifiable. The bounded model checker can prove the safety of the arithmetic operation in `bot` but had to include all functions of P in module M_1^{top}. It is more efficient and more scalable to not include functions but preconditions representing the input space for which the check is `unsafe` to the module.

Enumerative preconditions are based on the counterexamples generated by the bounded model checker. For function `bot`, the approach would generate negative data points like $(x = 2147483647 \wedge y = 1)$. In a refinement step, this input assignment is negated and added as an assumption to the SMT formulae that is rechecked through incremental SMT-solving. Such, the algorithm can enumerate all input assignments leading to an error. In our example, 200 (2 values for x times 100 values for y) data points would be generated, which combined represent the enumerative precondition. While enumerating 200 input assignments takes under a second for such a small example, it is often not feasible for larger modules with more parameters. Therefore, we enumerate a smaller amount of assignments producing a possibly under-approximated precondition.

Learned preconditions generalize the under-approximated precondition. In addition to the enumerated error data points, we generate inputs that are guaranteed to be safe by negating all assumptions and assertions in the program to be checked by the bounded model checker. Then, through a featurizer and synthesizer based on the ID3-algorithm [134], we iteratively generalize the data points leading to a complete (or over-approximated) precondition. For our example, the learning process produces the complete precondition $((x \neq 2147483647 \wedge x \neq 2147483646) \vee y > 99 \vee y < 0)$.

Through **substitution** of function calls of `bot` with the generated precondition, we can refine the verification without enlarging the module size. The algorithm including callers is reused, but this time with function bodies substituted by preconditions. Therefore, $M_1^{mid1} = \{\texttt{mid1}, pre(\texttt{bot})\}$ and $M_1^{mid2} = \{\texttt{mid2}, pre(\texttt{bot})\}$, checking $C(\texttt{bot})$ are not containing the function `bot` but the much smaller representation of erroneous inputs. While M_1^{mid2} would again be verified as safe, the approach would generate a new precondition for M_1^{mid1} because of the new restriction on x. This precondition is then substituted in function `top` leading to $M_1^{top} = \{\texttt{top}, pre(\texttt{mid1})\}$. For the final verification of $C(\texttt{bot})$, the function call of `mid2` can be ignored, because it was already classified as `safe` and has no relevant return value or memory writes influencing the function call of `mid1`.

6.3 Recapitulation of Modularization

This section briefly summarizes the introduced notations and statements about modules, functions, and checks to facilitate understanding of the refinements presented in the next sections.

We consider programs in a procedural programming language, i.e. the program consists (mainly) of a set of functions, and functions may call each other. To automatically check for errors, program statements (instructions) may be annotated with correctness conditions, which are called *checks*. More formally, we can write $P = \{F_1, \ldots, F_n\}$ for a program P consisting of functions F_i. The set of checks C in a program P can be partitioned according to the functions F_i due to the unambiguous assignment of instructions to functions. We write $C(F)$ to denote the checks annotated to instructions in function F. Thus $C = \dot{\bigcup}_{F \in P} C(F)$.

In order to check the properties of a program modularly, we will, on one hand, partition the set of checks C. On the other hand, we try to prove checks not on the complete program P, but only on a fraction $M \subseteq P$ of it. By ensuring that the fractions correspond to over-approximations of the program behavior, we can guarantee that a "no error" answer on a fraction also holds wrt. the complete program. The converse does not hold, however, such that in case of an "error" answer a refinement step is needed. We will also say *modules* instead of program fractions and denote the functions and checks of a module M by $F(M)$ and $C(M)$, respectively. Typically, we use as checks C in a module M either (a) all checks occurring in M, i.e., $C = \dot{\bigcup}_{F \in M} C(F)$; (b) exactly those checks of one particular function $F^* \in M$, i.e. $C = C(F^*)$, or (c) exactly one check c,

i.e. $C = \{c\}$. The functions of a module should not be arbitrarily selected, as that would make it hard to guarantee the over-approximation property. We, therefore, make use of the call graph to specify closure-properties of a module. Consider the function call relation $\rightarrow_P$ of a program P. (We might drop the index P if it is clear from the context.) $F \rightarrow G$ indicates that there is a call from function F to function G (in program P). By $\rightarrow^+$, $\rightarrow^*$ and $\leftrightarrow$ we denote the transitive, reflexive-transitive and symmetric closure of $\rightarrow$, respectively. For a module M, we now demand that (a) if $F_1, F_2 \in M, G \in P$ and $F_1 \rightarrow^* G \rightarrow^* F_2$ then $G \in M$; and (b) that the vertex-induced subgraph of a module M is weakly connected, i.e. $F_1 \leftrightarrow^* F_2$ for all $F_1, F_2 \in M$. Moreover, we call a module M *callee-closed* if all call instructions in M invoke functions that are already part of M, i.e. if $F \in M$ and $F \rightarrow G$, then $G \in M$. We call a module *caller-closed* if all calling functions of a module's function are already included in the module, i.e., if $G \in M$ and $F \rightarrow G$, then $F \in M$. We assume that each module possesses a unique function F_e, the *entry point function*, which has no callers in the module.

In this chapter, we will mainly deal with callee-closed modules. These form over-approximations of the program behavior, as they are contiguous fractions of programs, where simply the calling context of the entry point function F_e is dropped - denoted by us as *call-environment abstraction.*

6.4 Module Extension by Caller Inclusion

We define a verification task as a triple $vt = (f_e, m, r)$, f_e being the unique entry point function, m being the module containing the properties to be checked and r representing the verification result, with $r \in \{$`safe`, `cond.safe`, `cond.unsafe`, `unsafe`, `unknown`$\}$. Potential false positives are summarized as `cond.unsafe`, they are mostly generated by over-approximations of program behavior, e.g., by abstracting the call environment. Results that so far have been shown to be safe, but are potentially false negatives, are considered `cond.safe`; they are often produced by under-approximations like the loop-bound restriction of BMC.

For simplicity of exposition, we from now on assume that every module contains exactly one check to be analyzed. Conceptually, if there are several checks in one function, we can create copies of the module with separate checks.

As the starting point of our verification, we assume a set of callee-closed modules and a modularization as shown in the example in Fig. 5.3. We generate a set of verification tasks, $VT = \{vt_1, ..., vt_m\}$, one for each module, where the result is set to `unknown`. We then run our BMC on each verification task, updating the results. Those tasks resulting in `safe` or `unsafe` are completed and need not be considered further. For the remaining tasks VT' (with results `unknown`, `cond.safe` and `cond.unsafe`) a refinement step is needed. In both cases we refine the abstraction introduced by modularization and add callers of the respective module's entry point function. Note that for `cond.safe` results

we could also choose to increase the loop bound, but extending the call environment is also a viable alternative, as it often introduces additional constraints to derive smaller loop bounds. Thus, for such $vt_i = (f_e, m, \texttt{cond.unsafe})$, we include functions that are calling f_e, denoting the set by $Callers(f_e) = \{f_1, ..., f_n\}$. In order to maintain the single-entry-point property of a module, we then define a set of n new refined verification tasks, taking each caller of f_e in turn into account. To ensure that the module is $callee-closed$, all called functions of the caller f_i are added to the module. For these new verification tasks we have $m_i = \{m \cup \{f_i\} \cup callees(f_i)\}$, the entry point function is unchanged.

After verification of the n extended modules (for each vt_i) by the bounded model checker, these n results, say r_j for $1 \leq j \leq n$, have to be summarized. If there is one $r_j = \texttt{unsafe}$, the property is violated and the overall check is `unsafe`. Otherwise, as long as there is a sub-result marked as abstracted (`cond.*`), we again search for all callers and reiterate the process until (1) there is a `unsafe` result, (2) all results are `safe` or (3) the limitations of the underlying BMC is reached and the verification runs into a time- or memory-out and returns `unknown`.

The presented approach exhausts the limitations of the given encoding and solving process implemented in the bounded model checking approach. For programs with a relatively small call depth like APIs or libraries, the approach can reach top-level functions and thus prove the overall safety of checks. Furthermore, the extension can remove false positives and false negatives by refining the call environment of functions containing the check. While this approach incrementally increases modules to the solver's limitations, it does not optimize the maximal problem size the underlying bounded model checking approach is able to verify. Therefore, we have to minimize the module to be encoded.

6.5 Refined Modularization through Preconditions

Through preconditions, we are able to include callers of entry-point functions in our analysis without steadily increasing the size of the module. After creation of a precondition that represents the input space at f_e leading to an error, we can substitute function calls of f_e with the generated precondition. Generally, encoding the precondition simplifies the problem, because it represents only constraints over input parameters in reference to a specific check, instead of the whole function. We present a novel approach that first creates enumerative (sometimes under-approximated) preconditions based on failed proof attempts (counterexamples) Afterward, we employ a tree-based learning approach to generalize the precondition to a complete (or over-approximated) precondition based on the enumerated data points.

6.5.1 Generation of Enumerative Preconditions based on BMC

We introduce our approach of generating enumerative preconditions by example:

Example 1. *The module contains a single function* `example1`*. The input space is defined as* $I = \{x, y, *c, *g\}$.

```
char *g = nondet_ptr();
void example1(int x, char y, char *c) {
  if(c[1] == 'z' && *g == 'a') {
    x = x + 2; // possible arithmetic overflow
  }
}
```

If the `if`-statement is `true`, the addition $x = x + 2$ leads to an undefined overflow for `INT_MAX` and `INT_MAX` $- 1$. Verifying `example1` results in a task vt with $vt.r$ = `cond.unsafe`, so, for refinement, we generate counterexamples, in turn on the SMT, LLVM IR, and, finally, the C-code level. Both during encoding and solving of the formulae the underlying BMC-approaches optimize and eliminate variables. We have to track these optimizations but can also benefit from them. If an input parameter is not part of the formula or the SAT-model, it is not relevant to the erroneous execution and can thus be removed from the input space. In our example, variable y would be eliminated. We obtain a counterexample consisting of two functions $A : (Var \rightarrow Val)$ returning a value (either constant or pointer) for each program variable and $Mem : (Loc \rightarrow Val)$ returning the value saved at a given address location in memory.[1] Special handling is needed for pointer values during refinement, as we want to exclude values stored in memory rather than particular addresses used for storing them. We thus refine only pointer goals (for primitive, non-pointer values), but not memory addresses.

6.5.1.1 Excluding Values

For refining the counterexample we gather relevant elements of the input space I_m of module m and its entry-point function f_e, which are: parameters of f_e, global variables and memory locations accessed in m. Excluding the complete counterexample would lead to a massive under-approximation. Finding a minimal conjunction of values to exclude could be computed by following def-use chains backwards from the check's variables, including memory-based dependencies (via LLVM IR's `load`s and `store`s), but this would be extremely expensive. We therefore over-approximate the use-def chain leading to under-approximated preconditions. Our approach distinguishes between non-pointer and pointer parameters. For non-pointer parameters, the approach excludes the value assigned by the counterexample, for all parameter but those which have been optimized out by the solver. For pointer parameters, we first list all loaded addresses in m. We minimize this set of loaded locations by including only locations that are loaded prior to the validation check, by excluding reoccurring load instructions and concentrating on the first occurrence of a memory location. For direct loads, like char or integer pointers, we are able to enquire first the address and then the value stored at that address from the module created by the SMT solver.

[1] We assume memory accesses occur only to allocated memory.

For indirect loads to struct or array members, we check additional parameters like index and type of the pointer, such that we are able to exclude values for struct and array members instead of the address of the overall struct. Given the set of relevant elements, we divide them into non-pointer (denoted by $e^n \in I_m^n$) and pointer (denoted by $e^p \in I_m^p$). Due to single error points given by the counterexample, we only use the $=$ *or* $\neq$ relations in our preconditions. Given functions A and Mem, as introduced in the previous subsection, we generate the following precondition:

$$pre(I_m) = \neg \left(\bigwedge_{e_n \in I_m^n} e^n = A(e^n) \wedge \bigwedge_{e_p \in I_m^p} e^p = A(Mem(e^p)) \right).$$

For our Example 1, the relevant input space is $I_m = \{x, c[1], *g\}$ removing y and handling the array access for $*c$. A preliminary precondition would then be:

$$pre(I_m) = \{\neg\,(x = 2147483647 \wedge {*g} = \text{'a'} \wedge c[1] = \text{'z'})\}\,.$$

6.5.1.2 Enumerative Preconditions

A single bounded model checking run creates a partial precondition (also called single error data point). By adding the partial precondition to the SMT-formula of the module, we can exclude this particular assignment and generate more counterexamples. The effort for re-checking can be minimized by utilizing incremental SAT/SMT-solving. As long as the BMC approach is able to generate more counterexamples, the precondition is not complete, thus we refine the under-approximation of the precondition iteratively until the precondition is complete or a given bound is met. In every iteration, we exclude single values that are guaranteed to generate an error and can thus never over-approximate the precondition.

Definition 6 (Enumerative Precondition)**.** *Given a bound k for the number of iterations, the enumerative precondition is created by:*

$$pre(I_m) = \bigwedge_{i=1}^{k} pre_i(I_m)\,.$$

For our Example 1, the approach generates the following complete precondition after two iterations:

$$\begin{aligned} pre(I_m) = \{&\neg\,(x = 2147483647 \wedge {*g} = \text{'a'} \wedge c[1] = \text{'z'}) \wedge \\ &\neg\,(x = 2147483646 \wedge {*g} = \text{'a'} \wedge c[1] = \text{'z'})\}\,. \end{aligned}$$

Disregarding time and memory limitations, enumerative preconditions guarantee complete preconditions as long as the bound k is chosen big enough. In practice, choosing a large enough k might be prohibitive, especially if the number of variables in I_m is large. Thus the enumerative approach might not be sufficient to generate a complete precondition. Nevertheless, enumerative preconditions can be deployed to find bugs and can give a human user an idea of which inputs lead to an error.

Refinement of loop bounds for bounded model checking. Bounded model checking is safe wrt. to a given loop bound. Most model checker calculate loop bounds and unroll the loop accordingly. For cases where the bound can not be calculated they unroll the program to a user-given bound b. Model checker can recognize cases where the loop bound was not sufficient and thus a check is not marked as safe but as `cond.safe`. These conditions can be refined by preconditions over loop constrains and bounds. If the loop bound could not be calculated it is often given by function parameter or read from memory. Stating that the parameter or value in memory has be less then the given loop bound of the program creates a precondition that can be checked and refine the potential false positive.

6.5.2 Learning General Preconditions from Data Points

For some examples, the enumerative approach would need billions of iterations for a complete precondition. Therefore, we implemented a learning approach that can generalize from data points to the relevant input space that has to be excluded. The approach generalizes preconditions by learning from data points generated by both erroneous and error-free verification runs for the module. In our implementation, the preconditions are human-readable and are parsed directly into C-code.

Our technique is based on a white-box teacher black-box learner approach, as for example presented in [58], the main difference being that the teacher will be handled by our static-analysis tool LLBMC. Compared to dynamic testing approaches this brings three advantages: (1) Modules can be verified without needing to be executable. (2) A static-analysis technique like LLBMC typically finds more error points than testing. (3) LLBMC can serve as a data-set generator, reliably providing inputs for both successful and erroneous executions.

From this data set, a `featurizer` extracts features, classifying sets of data-points with characteristics. The `synthesizer` refines these features to a Boolean function, resulting in the minimal number of features that suffice to represent the data set. The result is the precondition that applies to the enumerated data points. This cycle continues, until the bounded model checker is not able to find any erroneous data points, implying completeness, as well as data points resulting in a successful execution, implying maximality.

6.5.2.1 Enumerating Data Points with BMC

Additionally to the presented approach to generate enumerative preconditions (called negative data points here), the learning approach needs positive data points representing the input space for error-free executions of the module. Otherwise it would overapproximate the precondition such that `assume(false)` would always be a valid option. To generate positive executions, we negate the disjunction of checks and the conjunction of assumptions and cover special cases, where the assertions are in branches. If there exists an assertion in one side of a branch, we have to generate a failed assertion in the other side of that branch and thereby offering the solver to find a program execution where the assertion holds.

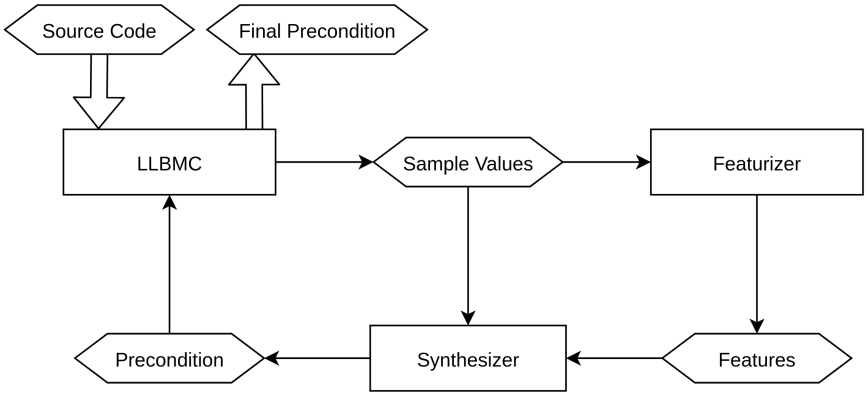


Figure 6.3: Architecture of the static precondition learner

This assertion is then added to the conjunction of negated assertions. Through this extension, LLBMC is able to generate both positive and negative data points for the learning approach.

6.5.2.2 Featurizer

The task of the featurizer is the extraction of characteristics from the data-set provided by LLBMC. All features combined should be able to represent the relevant properties for preconditions. Figure 6.4 presents this technique.

The approach of feature extraction is based on the algorithm implemented in the tool PIE by Padhi et al. [127]. Initially the algorithm considers a set of random sample values that are classified as program failures or successes. Then a conflict group is defined as a set of differently classified data-points which can not be distinguished because they are represented by the same valuation of current features. As long as such a conflict group exists, the current features do not sufficiently describe the data-set and a new feature has to be determined and added. Such a feature is determined by iterating over existing features from a predefined *feature pool*.

We enhance [127] by dynamically expanding the feature pool on demand. The feature pool is defined as a set containing Boolean combinations of relational formulae. Every atomic formula consists of variables, constants, comparison operands $\{<, >, \leq, \geq, =, \neq\}$ and arithmetic operations $\{+, -, *, \div\}$. The feature pool is initially considered empty. If no conflict resolving feature exists in the current feature pool, it is extended by more complex features, continuing until a conflict resolving feature has been found. We define the complexity of a feature by two different parameters: The term-depth and the function-depth. Term-depth defines the number of arithmetic operations in a feature. The feature $x > 0$, for example, has a term-depth of zero, $x > 2 * y$ a depth of one and $x > 2 * y + z$ a depth of two. Function-depth is defined as the number of atomic formulae in a feature.

The feature $x > 0$ for example is composed of only one atomic formula, thus its function-depth is one. The feature $x > 0 \wedge x < 10$, consists of two atomic formula, concluding a function-depth of two.

Example 2. *We consider function* `divide` *from* `stdlib.h`*. Division by zero and division of* `INT_MIN` *by* -1 *lead to undefined behavior and are treated as errors.*

```
#include <stdlib.h>
div_t divide(int numer, int denom) {
  div_t result;
  result.quot = numer / denom;
  result.rem = numer % denom;
  return result;
}
```

The approach generates the feature `denom = 0` *in the first iteration. After running the synthesizer and realizing that the feature is not strong enough to generate a fitting precondition, it adds the feature* `denom = -1` *and then in the third iteration the feature* `numer = -2147483648`*.*

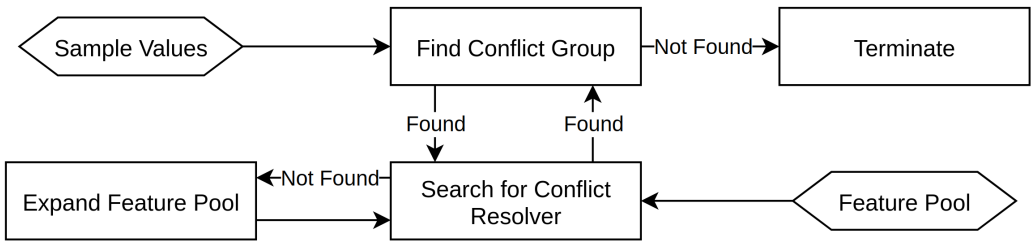


Figure 6.4: Architecture of the featurizer

In principle, such featurizers are able to generate preconditions when provided with a complete data set. However, the featurizer alone is much less scalable than an approach in which the precondition generation is extended by a Boolean learning processes, the synthesizer. The featurizer is restricted by a given parameter limiting the size of regarded conflict groups.

6.5.2.3 Synthesizer

After resolving all conflicts from the data-set, the selected features suffice to separate the data-set into correct classes. But the feature set is not necessarily minimal. Imagine the featurizer learned the two features $x < 5$ and $x < 6$. Here, $x < 5$ implies $x < 6$, making the second feature redundant.

We use the synthesizer to create a decision tree on the features in order to represent their structure and relations. As many previous precondition learning techniques [5, 135] did, we apply the ID3-classification algorithm [134] to learn decision trees.

The ID3-algorithm is an entropy-based classifier. First, the entropy H of the complete data-set S is calculated. Then, the information gain of each feature regarding the data set is computed in order to determine the feature with the highest information gain.

This feature is selected as the root node of the decision tree, and the algorithm is applied recursively, considering only the leftover features and dividing the data-set into two subsets: Subset 1 contains all data-points for which the feature with the highest information gain is true, and Subset 2 all data-points for which this feature is false. The recursion terminates, when the decision tree is able to classify every data-point from the data-set correctly. We can transform this decision tree into a precondition by traversing every path from the root to a leaf. For each path, we build a conjunction of the nodes' features where a feature is negated if its node is left by a *No* edge. The disjunction of all paths leading to a *success* leaf then represents the precondition guaranteeing a successful program execution.

For Example 2, our approach generates the decision tree and the precondition in three iterations as shown in Figure 6.5.

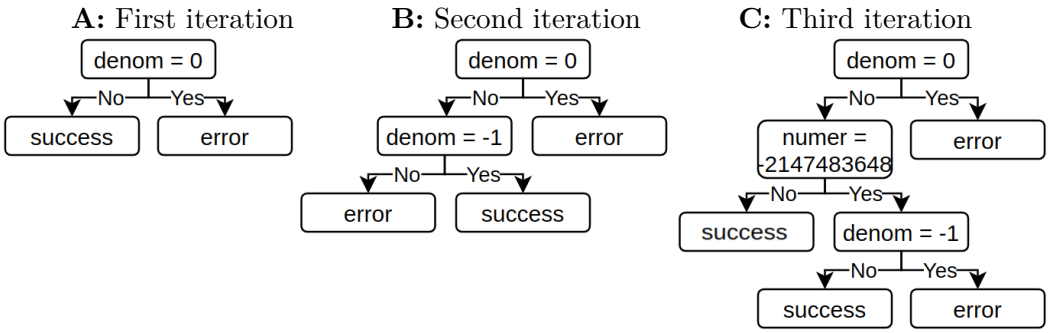


Figure 6.5: Decision trees created by the ID3-classification algorithm

$$(\texttt{denom} \neq 0 \wedge \texttt{numer} \neq -2147483648) \vee$$
$$(\texttt{denom} \neq 0 \wedge \texttt{numer} = -2147483648 \wedge \texttt{denom} \neq -1)$$

The precondition excludes all divisions by zero as well as integer overflows resulting from dividing `INT_MIN` with -1.

Such preconditions are inserted directly into C-code via `assume` statements that are placed at the beginning of the regarded function. The precondition can still be incomplete, if the sampling does not represent all relevant inputs. Thus, the precondition is strengthened by another run of the bounded model checker LLBMC generating more data points. Generally, the approach as shown in Figure 6.3 is able to iteratively refine the precondition with new data points until it's complete. If time-limitations are reached without concluding a complete precondition, the algorithm returns the latest over-approximated precondition.

The performance of the approach depends on the quality of data points generated by LLBMC as well as the order and relevance in which features are chosen to represent the data. Given a formula containing the program containing the current precondition, the underlying SAT-solver can produce data points with varying entropy. For an error space of $0 < x < 100$, the solver can return $x = 1$, $x = 2$, $x = 3$ or $x = 1$, $x = 99$, $x = 15$, as data points for three consecutive runs. In our experience, the solver samples values quite randomly from the error space. The influence of activated heuristics and specific decisions of the SAT-solver on the sampling quality is part of future work.

6.5.3 Subsumption of Preconditions

The enumerative and the generalized preconditions can be applied to refine the modularization without massively increasing the size of verified modules. Similar to the caller inclusion, we extend the module. However, instead of including the original entry-point function f_e, we substitute the function call of f_e with a generated precondition, thereby omitting the encoding of f_e and all successively called functions. The precondition is inserted as an intrinisc function call (similar to inserted checks). The parameters of the function call are equivalently kept as input parameters for the new intrinsic call. If there are multiple function calls of f_e, each call is substituted. If f_e has a return value it is generally ignored and assumed to be an undefined value. This leads to an over-approximation in case the return value influences input parameters of later function calls of f_e and thereby the preconditions. This case only occurs if there are multiple calls of f_e or the function is called inside a loop. In the current implementation such cases are excluded and no precondition substitution takes place. While merging results of different callers, the algorithm tracks if the precondition is under-approximated (enumerative), thus regarding only result transitions to unsafe, over-approximated (over-generalized), regarding only safe checks or complete.

6.6 Conclusion

We presented a refined modularization approach based on bounded model checking and precondition learning. Given a modularization that divides a program into smaller modules by fully abstracting the call environment of functions containing checks, we refined this abstraction by first extending the module size through caller inclusion before generating preconditions representing input space that leads to erroneous execution of the module. Based on counterexamples from the BMC approach, we generated (under-approximated) enumerative preconditions and then generalized these preconditions utilizing an ID3-based learning approach. The preconditions can be successively pushed through the program leading to a more precise verification. The preconditions are user-readable and can be examined and extended by the user. The evaluation shows the potential to verify large projects and to reduce both false positives and false negatives significantly through caller inclusion and preconditions.

Currently, the refinement leads to significant overhead in verification time. Including callers, splitting modules, gathering enumerative preconditions and the ID3 algorithm for learning preconditions take additional time. Writing and reading preconditions to and from memory as well as module splitting has the largest influence on runtime. This can be minimized through a parallel verification of modules and a database supporting effective handling of thousands of checks. Furthermore, we aim to investigate the impact of precondition simplifications and better integrate the precondition-learner, function-pointer handling and function call abstraction into `QPR Verify`.

Chapter 7

Tool Demonstration and Evaluation

This chapter illustrates the application of our modular bounded model checking approach with `QPR Verify` for the verification of real-world applications. Furthermore, we present an extensive evaluation based on three industrial software projects. We will demonstrate the usability and scalability of our approach on programs between 2000 to 400k lines of code not including external libraries.

7.1 Tool Demonstration

The application of `QPR Verify` was first described in [KB5]. As mentioned in Section 1.2, the implementation of `QPR Verify` has not been done by the author alone. Specifically, procedures described in the *set-up* phase have been previously implemented but were not published. The tool including documentation are installed on a virtual machine [1] additional instructions and information can be found at `https://github.com/MarkoKleineBuening/DissertationTools`.

We will follow Figure 3.1 (Chapter 4) of our verification approach and distinguished between *Set-Up*, *Modularization* and *Refinement*. Additionally, we shorty describe the global analysis and the possibility of incremental analysis with `QPR Verify`. Design and development of features and functionality of `QPR Verify` were described throughout chapters 3, 4, 5 and 6. Here, we will give the reader an idea how to practically apply modular bounded model checking with `QPR Verify` to real-world applications. This thesis does not aim to present implementation details, especially those in regard to C++ and LLVM features.

`QPR Verify` is implemented in C++ and has to be applied as a command line tool. It supports operating systems based on UNIX like Linux and mac-OS as well as Microsoft Windows.

[1] Link to VM: `https://baldur.iti.kit.edu/qpr/QPR-Artefact-2021.ova`

7.1.1 Set-Up

The first step of every verification process is to initialize an empty folder for the verification with `QPR Verify`. In this initial folder all further information, configuration decisions and results will be placed in machine and user readable format. Given the root directory of the project to be verified, a verification task is set up with the command

```
qpr init <root-directory>.
```

After initialization, we have to add those program files to the verification project that should be verified. We can add all program files that can be found in the root-directory of our project with the command

```
qpr find-source-files.
```

Alternatively, the user can list all program files that he wants to include and add them by

```
qpr add-source-files <files>.
```

For larger projects compile commands and options need to be specified. Most common options are include directories or values for constants describing the machine environment. These options can either directly be given in a *compilecommands.xml* file or added via the command

```
qpr add-compiler-option <option>.
```

Given the files under verification, together with potential compile options, `QPR Verify` can perform check insertion, compilation and the preprocessing analysis by issuing the two commands

```
qpr murphy
qpr compile.
```

At this point, `QPR Verify` created the bitcode file *bitcode.bc*, of the code under verification and inserted checks. Furthermore, it created a list of all checks in the file *checks.xml*. Some of which are already verified by the preprocessing analysis and marked with a check result.

To further verify the program, there are a number of different configuration and solving strategies available. Next to the configurations discussed in Section 4.4.2, options like `TimeOut:<seconds>` and the output format of program traces (`XMLTraces<bool>`) can be set. Configuration options apply for the whole project and are set by

```
qpr configure-project <name>:<value>.
```

As a final step before applying solving strategies, the bitcode can be optimized by a number of transformation passes with the command

```
qpr optimize-bitcode <list of transformation pass names>.
```

At this point of the analysis, the software project under verification is set-up. The configurations can be later adjusted according to the solving strategies.
Global Analysis As mentioned in Section 4.4.3, the whole-program analysis of `QPR Verify` is called global-analysis. Verification of the whole program necessitates a single entry-point function for encoding. The global-analysis can be executed by setting the entry-point and then starting the analysis by executing the following commands:

```
qpr configure-project EntryPoint:<functionName>
qpr analyze-globally.
```

The global-analysis assign a verification result to every check created by the `murphy` command. In an optimal setting every check would be either `safe` or `unsafe`. In case of under-approximations produced by truncating program traces through the given loop-bound, the check result `loop-bound safe` is chosen. If the program is too large to be encoded or the underlying SMT solver does not comes to a conclusion the check is marked as `unknown`.

The check results are stored into the file *checks.xml*. This file can be read-in to create a GUI representation or accessed by a subsequent analysis run.

7.1.2 Modularization

As presented in Chapter 5, our approach partitions the program to increase scalability. All checks that are nor definite (`safe`, `unsafe`) can be rechecked utilizing different abstraction techniques.

7.1.2.1 Call-Environment Abstraction.

The abstraction of the call environment presented in Section 5.3 allows `QPR Verify` to partition the program into smaller modules. In `QPR` this analysis is called *local-analysis*, because every check is verified in it's local environment.

Using local analysis greatly reduces the verification's computational effort. However, since the semantics of function calls are taken into account (but not their correctness properties, i.e. their checks), local analysis is still an interprocedural analysis and may require considerable resources for large programs. The analysis is simply started by a single command and no entry points have to be specified:

```
qpr check-all-functions-locally.
```

The approach iterates through all functions in P and starts the analysis with a function F as the entry point. The call-environment (input parameter and memory) of F are over-approximated by setting them to undetermined values. The analysis only verifies checks that are located within F, but still takes called functions within F into account.

The check results are assigned analogously to the global analysis. However, for checks that are verified as *unsafe*, we have to regard the over-approximation of the environment and thereby the possibility of false positives. Therefore *unsafe* checks are classified as `cond. unsafe`.

7.1.2.2 Function-Call Abstraction.

If the calling context should remain precise and the abstraction of function calls is preferred one can choose the *call abstraction analysis* of `QPR Verify`. The call abstraction analysis abstracts away calls to functions outside of the chosen entry point function f. As described in Section 5.3, the analysis only keeps checks (assertions) in function f, and abstract away all function calls that exceed a user defined call depth (called *havoc-limit*). When abstracting a function call without any further knowledge, an over-approximation of its behavior has to be assumed. Next to the return value (if existent), memory content (including global variables) can be altered by the called function, and thus are assumed to be arbitrary. The function call abstraction analysis can be started by configuring the havoc-limit and then starting an analysis.

```
qpr configure-project HavocLimit:<positive number>,
    qpr check-all-functions-with-call-abstraction.
```

Due to the over-approximation of function calls, the check results are assigned similar to the local analysis; unsafe checks are classified as `cond. unsafe`.

7.1.2.3 Combined Abstraction.

As described in Section 5.3, both abstraction techniques can be activated at the same time. Applying both, function call and function environment abstraction, leads to a partition of the program into modules consisting of a single function. Given the program $P = (F, G)$, the approach generates an individual module for every function $f \in F$ and over-approximated both the environment the function is called in and every function call in f. This combined analysis is the most scalable approach but leads to a high number of false positives. It can be seen as a first analysis that successfully verifies checks that can be proven as `safe` in a small function context.

The combined abstraction analysis can be started by performing a local-analysis with an active havoc-limit set to 1.

```
qpr configure-project HavocLimit:1
  qpr check-all-functions-locally.
```

The verification of modules containing a single function is currently the most scalable approach implemented into `QPR Verify`. Still there are checks that can not be verified. For large or complex functions the SMT solver can time-out. In practice these functions have to be manually partitioned or rewritten for verification.

At this point, checks are classified into the five categories `unsafe`, `safe`, `cond. unsafe`, `loop-bound safe` and `unknown`.

7.1.3 Refinement

While the modularization verifies most checks, there is a high chance that additional program traces introduced by the abstraction are not reachable in the original source code. Potential false positives are represented by the check category `cond. unsafe`.

In Chapter 6, we introduced three main refinement steps to minimize false positives. The refinements assume a prior local analysis. The introduced call environment abstraction by the local analysis is incrementally refined.

The first refinement incrementally increases the module size by adding callers of the entry-point function to the module. As it is typical for any verification tool, our underlying bounded model checker LLBMC builds a call graph to traverse function calls during encoding. To efficiently find callers, we added backwards edges creating a bidirectional graph. Therefore, callers can be automatically and efficiently found. If there is a single caller, the caller function is added to the module and if there are multiple caller functions, the approach splits the module for each additional caller. The first refinement step can be started by the following command:

```
qpr check-all-jobs-by-callers.
```

The command can be called multiple times to incrementally include additional callers higher up the reversed call graph. If only a single verification task, containing for example only a single check, should be refined, the user can issue the following command:

```
qpr check-job-by-callers job:<id>.
```

The second refinement step creates enumerative preconditions based on failed checks in a module. The approach can be run with grouped or individual check handling. Yet preconditions generated for multiple checks are often more difficult to generate. The number of enumerated data points representing an input leading to an error can be given by the parameter `precondition-number:<uint>`. If this number is greater zero, `QPR Verify` creates enumerative preconditions during every local analysis. To refine a prior analysis, one has to rerun the analysis with a given precondition number. During precondition generation the enumerative preconditions are classified as either *complete* or *under-approximated*. `QPR Verify` is able to distinguish between these precision levels by checking if an additional bounded model checker run would still produce an `cond. unsafe` result or if the generated preconditions completely encompass the erroneous inputs. As long as preconditions are *under-approximated* no `cond. unsafe` can be classified as `safe` due to potentially missing inputs leading to an error that are not included in the generated precondition.

Setting the entry-point function to the callers of our initial function and substituting the function call can be done by the command `qpr check-all-jobs-by-preconditions`. So to refine existing checks, the user can issue the following commands:

```
qpr configure-project precondition-number:<uint>
qpr check-all-functions-locally
 or qpr check-all-jobs-by-callers
qpr check-all-jobs-by-preconditions,.
```

Again single jobs can be refined through preconditions by manually choosing a job id for a given check and running the command:

```
qpr check-job-by-precondition job:<id>.
```

The generalization of preconditions through a tree-based learner that is described in Section 6.5.2 is not directly implemented into `QPR Verify`. During the Praxis der Forschung project with Johannes Meuer, the student wrote a separated tool that has not yet been integrated into `QPR Verify`. Currently, only single verification jobs can be refined through a precondition learner. The user has to navigate into the targeted job folder. There he can start the precondition learner that takes the compiled program *bitcode.bc* as an input. To run the precondition-learner the number of positive (`pos=<uint>`) and negative (`neg=<uint>` data-points have to be specified as well as the size of the conflict group (`conflict=<uint>`. The following command creates the generalized precondition that can again be read in by `QPR Verify`.

```
precgen bitcode.bc function-name=<name>
        neg=<uint> pos=<uint> conflict=<uint>
```

`QPR Verify` automatically detects generated preconditions and does not distinguish between enumerative and learned preconditions. The substitution is again performed by:

```
qpr check-all-jobs-by-preconditions,.
```

Again preconditions can incrementally be parsed through the reversed call graph by issuing the command multiple times. A step-wise refinement was implemented because the parsing of preconditions directly to the highest function (often main), is often not achievable in acceptable time and thus intermediate results are favorable.

An additional refinement step that should be regarded when performing bounded model checking is an increased loop-bound. The user can set the loop-bound by

```
qpr configure-project LoopBound:<value>.
```

Afterwards, any analysis can be rerun and results that are not already verified as `safe` or `unsafe` will be rechecked with the increased loop-bound.

Check results at this point have been checked through modularization and refinement and represent the best of both abstraction and refinement techniques implemented into `QPR Verify`.

7.1.4 Incremental Strategies

We presented a number of different configuration and analysis modes. Given a fixed loop-bound and time-out, there are three main configuration options with Boolean values: `optimize-bitcode`, `analyzeChecksIndividually` and `enableSlicing`. Slicing

can only be enabled with individual check handling. Additionally, there are four modularization options, (1) no abstraction, (2) *call-environment abstraction*, (3) *function-call abstraction* and (4) both abstractions. Finally, there are three refinement strategies *caller-inclusion*, *enumerative preconditions* and *learned preconditions* that can be run multiple times with the call depth of the program as an upper bound. They can also be run in arbitrary order. Therefore, given a fixed time-out and loop-bound and a call depth of the program under verification d, there are $(2^3-3)\times 4\times d^3$ possible configuration and verification runs. If the refinement steps are only performed once ($d = 1$), there are 60 different configuration and solving approaches the user can choose.

Additionally, the verification approaches of `QPR Verify` can be run subsequently to increase precision and scalability. Every new run of `QPR Verify` rechecks checks with results `cond. unsafe`, `loop-bound safe` and `unknown` with a new configuration or solving approach. Therefore, every verification run increases precision.

The optimal order in which to run different configuration and solving approaches depends on the program to be verified. In general faster verification runs with different modularization techniques should be run first. Afterwards, refinement strategies can be utilizes to minimize potential false positives. The evaluation in Section 7.2 shows how to run different strategies for real-world applications.

7.2 Evaluation

The techniques and approaches presented in the second part of this thesis aim towards a scalable and modular verification framework for industrial software projects written in C/C++. As described above, all approaches were implemented into the tool `QPR Verify`. Previous evaluation results of `QPR Verify` on different benchmarks can be found at [KB5, KB4, KB3]. For this dissertation, the evaluation was extended and uniformly rerun. All approaches were evaluated as sequential programs on a single CPU on a Gigabyte R282-Z93 machine with 3200MHz and 1024GB RAM.

We evaluate `QPR Verify` on three industrial projects and compare our approach to three state-of-the-art tools. The presented benchmarks represent different industrial software applications and highlight varying challenges and features of our approach. For every benchmark, we first show the results of the set-up phase applying only the preprocessing analysis. Afterwards, we show `QPR Verify` verification runs with different optimizations and abstractions. We then compare our modular bounded model checking approach with three different techniques and tools. First, the standard bounded model checking approach represented by the global analysis with LLBMC. Second, we apply the commercial tools Coverity [1], which focuses on bug finding. Finally, we compare our approach with the abstract interpretation tool Polyspace [45], which is, together with Astree [14], the most applied static analysis tools for software written in C/C++ for German industry companies like Bosch and Daimler.

All evaluation results of `QPR Verify` and Polyspace as well as links to results of Coverity can be found at `https://github.com/MarkoKleineBuening/DissertationTools`.

A verification run V of `QPR Verify`, as well as other tools, can be evaluated by the following four evaluation criteria:

(E1) V should aim to minimize the number of checks for which no verification result can be produces - meaning minimizing `unknown` checks.

(E2) V should aim to maximize the number of definite verification results - meaning maximizing `safe` and `unsafe` checks.

(E3) V should aim to minimize the number of potential false positives of false negatives, where a developer has to manually check the verification results - meaning minimizing `loop-bound safe` and `cond. unsafe` checks.

(E4) V should aim to minimize verification runtime.

7.2.1 BMI160-Driver

The open-source library BMI160-Driver [21] is a low power control unit driver designed for mobile applications. It is implemented by the Robert Bosch GmbH with around 2000 lines of code plus libraries.[2] BMI160 represents a small scale embedded software project, implemented under program paradigms as stated in MISRA. The driver provides an API for sensor controls. It does not contain any main function, which is a problem of many approaches and tools. The bug-finder Coverity needs the program under verification to be build utilizing their build tool *cov-build* and can therefore not verify the BMI160-Driver. Other approaches, like the bounded model checker LLBMC or CBMC also require a distinct entry-point function and can therefore also not be applied. The modularization techniques implemented in `QPR Verify` on the contrary allow for a modular verification of the program.

During set-up of the project, `QPR Verify` automatically collects all source and header files. During the compilation of the project, 2593 checks are detected. The preprocessing analysis described in Section 4.4.1 is then able to verify 2019 (78%) of these checks as `safe`. Thus, there are 574 hard verification tasks left. The project was configured with a loop-bound of 12, a timeout of 300 seconds and with the option `optimize-bitcode` as set to true.

Table 7.1: Results set-up and preprocessing analysis of `QPR Verify` for BMI160

	`safe`	`unsafe`	`unknown`	Total	Time
BMI160	2019	0	574	2593	1 sec

We now only focus on the hard verification tasks left after preprocessing. Table 7.2 shows the results of different verification runs of `QPR Verify`. The whole-program analysis is denoted as **Global**. The different check handling options are then appended: **I** represents individual checks and **C** represents combined check handling.

[2] We slightly adapted the source code to overcome current technical limits of our tool, e.g. by removing irrelevant function pointers in `structs`.

A verification run with an active slicing algorithm is marked with **S**. The modularization approaches have the prefix **Mod**. The call environment abstraction is denoted as **EA** and the function call abstraction as **FA**. If refinement strategies were applied runs are marked with the prefix **Ref** and the addition of **CI(x)** for caller inclusion of depth x, **EP(x)** for enumerative preconditions with x enumerations of errors. Learned preconditions were not evaluated on larger projects due to the missing integration into `QPR Verify`. The percentages are always in reference to hard verification checks and not the overall number of checks before preanalysis. We sorted the approaches according to the evaluation criterion solving time (E4) (right most column).

Table 7.2: Results of individual solving approaches for BMI160

BMI160-Driver	`safe`	`loop-bound safe`	`cond. unsafe`	`unsafe`	`unknown`	Time(sec)
Global	**not applicable due to missing main function**					
I-S-Mod-EA	445 (78%)	9(2%)	120 (21%)	0(0%)	0 (0%)	112
C-Mod-EA-FA	431 (75%)	22(4%)	100 (17%)	0(0%)	21(4%)	268
C-Mod-EA	349 (61%)	141(25%)	84 (15%)	0(0%)	0 (0%)	360
I-Mod-EA-FA	441 (77%)	22(4%)	109 (19%)	0(0%)	2 (0%)	702
I-Mod-EA	349 (61%)	140(24%)	84 (15%)	0(0%)	1 (0%)	798
I-Mod-EA-Ref-EP(5)	386 (67%)	134(23%)	49 (9%)	5(1%)	0 (0%)	899
I-Mod-EA-Ref-CI(1)	378 (66%)	136(24%)	55 (10%)	5(1%)	0 (0%)	8791
I-Mod-EA-Ref-CI(5)	497 (87%)	32(6%)	5 (1%)	40(7%)	0 (0%)	16485
I-Mod-EA-Ref-CI(5)-EP(5)	501 (87%)	20(3%)	13 (2%)	40(7%)	0 (0%)	24114

The fastest analysis (E4) verifies every check individually (I) with active slicing (S) with the environment abstraction (Mod-EA) in 112 seconds. 445 checks (78%) can be verified as `safe` and only 9 checks (2%) are `loop-bound safe`. The slicing algorithm abstracts large parts of memory and therefore introduces the highest number of possible false positives – 120 checks (21%) are therefore classified as `cond. unsafe`.

The analysis with no `unknown` results (E1), the highest amount of `safe` and `unsafe` checks (E2), as well as the least amount of potential false positives or false negatives (E3) is the analysis I-Mod-EA-Ref-CI(5)-EP(5) applying call environment analysis on individual checks with caller inclusion to depth 5 and successively enumerative and learned preconditions. This analysis leaves only 20 checks (3%) that are `loop-bound safe` and 13 checks (2%) that are `cond. unsafe`. The analysis needs 24114 seconds, meaning roughly 214 times longer than the fastest verification run.

In general, we can see that a high level of abstraction correlates with short runtimes and a higher number of `cond. unsafe` results. Therefore, the solving strategy balances scalability with precision and should be customized to meet user needs. A typical application could for example be a first verification run leading to initial results that can then be refined with longer verification runs overnight. It is to note that applying only environment abstraction with enumerative preconditions on individual checks (row 7) leads to only 9% of `cond. unsafe` checks in 899 seconds and can be seen as a good compromise between fast runtime and precise results. While it produces a relatively high amount of `loop-bound safe` checks, it is the only approach that is able to find definite errors (5 `unsafe` checks) in relatively short time. Including callers (CA) leads has a large impact on the runtime. Verification time roughly increases tenfold,

due to the increasing size and splitting of modules in case of multiple callers of an entry point function. It therefore is the most precise refinement and at some point leads to modules too large to be efficiently verified.

Table 7.3: Results of verifying BMI160 with Polyspace

Polyspace	`green`	`orange`	`red`	`grey`	Total	Time
BMI160	45 (19%)	142 (61%)	0 (0%)	46 (20%)	233	988 sec

We ran Polyspace version 2017b[3] with default configurations on BMI160. We chose a generic compile with an i368 processor style and left the precision and verification level at default value 2. Polyspace first verifies the program by propagating sets represented as static integer domains and afterwards applies a polyhedron model to refine the verification. A description of Polyspace and related work can be found in Chapter 8.

It is difficult to compare different static analysis tools on industrial software, because there is no distinct definition of properties that are verified. Every tool has it's own definition of errors and critical program locations. While we regard every operation (assignment, addition, division, shift, etc.) as potentially critical and verify it's behavior, tools like Polyspace have their own understanding of critical program locations. Furthermore, the C standard [79] sometimes leaves room for interpretation of operation behavior in edge cases and thus operations are sometimes modeled differently in different tools. There are initiatives working towards exchangeable and comparable static analysis results. The Static Analysis Results Interchange Format (SARIF) [65] developed by Microsoft or the Tool Common Configuration Format (ASC3F) [88] developed by the ASSUME CONSORTIUM (`http://assume-project.eu`) are two recent attempts and unifying tool output but are currently not established in research or most commercial static analysis tools. For our verification, we thus concentrate on percentages of solved properties and manually compare individual checks.

Polyspace took 988 seconds on 4 cores to verify the BMI160-Driver. It generated and verified 233 checks. The number of inserted checks by Polyspace is much lower than the 2593 checks of `QPR Verify`. Even only regarding the 574 hard verification tasks left after preprocessing, Polyspace inserts only half as many checks as `QPR Verify`. We assume that Polyspace disregards most of the critical program locations containing constant operations. We therefore compare the 233 checks only with the 574 hard verification tasks not the 2593 overall checks.

Of those 233 properties, 45 checks are classified as `green`, which is equivalent to the check result `safe` of `QPR Verify`. 142 checks have the status `orange` representing potential false positives, which is equivalent to the `cond. unsafe` check result. Whether to classify `loop-bound safe` checks as `green` or as `orange` is an open discussion. Applying the *small-scope hypothesis*, such checks could be regarded as `safe`. However, there is no complete proof for the safety of such checks and in comparison, we therefore equate `loop-bound safe` checks in `QPR Verify` with `orange` warnings in Polyspace.

[3]Due to licensing, 2017b was the latest available version of Polyspace for evaluation.

For BMI160, there are no checks marked as `red`, which is equivalent to `unsafe` checks in `QPR`. The 46 checks marked as `grey` represent checks that are not reachable from any entry-point function. Our approach looks for a program trace from a given entry-point function to those checks. For dead code, our approach does not find any such critical program trace and thus verifies those checks as `safe`.

Summarizing the results, Polyspace classifies 31% (19% green +20% grey) of checks as `safe` and 61% of checks as `cond. unsafe` in 988 seconds. Looking only at hard verification tasks, `QPR Verify` in comparison verifies 78% checks as `safe` 2% as `loop-bound safe` and 21% as `cond. unsafe`) in just 112 seconds. Given longer verification time, `QPR Verify` is also able to verify 94% (87% (safe) +7% (unsafe))of checks as `safe` or `unsafe` and only 2% as `loop-bound safe` and 3% as `cond. unsafe`. We therefore, outperform Polyspace both in time of analysis and precision producing between 38% to 56% less `orange` warnings.

A comparison with Coverity or the standard bounded model checking approach implemented in LLBMC or CBMC was not possible because of the missing main function. Coverity needs the user to build the program using their build tool. This is not possible for API programs without main function. Applying standard bounded model checking with LLBMC or CBMC is equivalent to the global analysis with `QPR Verify` and was thus also not possible due to no unique entry-point function.

7.2.2 SQLite

SQLite is a C-language library that implements one of the most used SQL database engines in the world [141]. It contains around 400K LOC without included libraries. It represents the main target for our modular verification approach. Programs of such size were prior to our work not verifiable by the bounded model checking approach. Even tools utilizing abstract interpretation are not able to verify such large programs as a whole but rely on manual partitioning and specification of modules.

The build process of SQLite produces two source files *sqlite3.c* and *shell.c* containing all functionality. During set-up of the project, `QPR Verify` added those two files and searched for needed libraries and include files. SQLite relies on a number of included libraries that aggravate internal building processes. To simply the building process, we therefore set the `QPR` configuration `Hosted` to true, choosing libraries implemented by us over libraries provided by the actual system SQLite is built upon. During the compilation of the project, 27,236 checks were detected. The preprocessing analysis was able to verify 18,685 (69%) of these checks as `safe` and 93 (0%) as `unsafe`. Thus, there are 8,371 (31%) hard verification tasks left. The project was configured with a loop-bound of 1, a timeout of 600 seconds and with the option `optimize-bitcode` set to true.

The names of approaches are equivalent to the notions described for the BMI160-Driver. We again show the results of different verification runs, first the global analysis and then modularization approaches sorted by time.

Table 7.4: Results set-up and preprocessing analysis of `QPR Verify` for SQLite.

Set-Up	`safe`	`unsafe`	`unknown`	Total	Time
SQLite	18,685 (69%)	93 (0%)	8,371 (31%)	27,149	15 sec

Table 7.5: Results of individual solving approaches for SQLite.

Approach/Results	`safe`	`loop-bound safe`	`cond. unsafe`	`unsafe`	`unknown`	Time(sec)
Global	0 (0%)	0(0%)	0 (0%)	0(0%)	8371 (100%)	TO
I-S-Mod-EA	698 (8%)	2095 (25%)	3138 (37%)	0 (0%)	2426 (29%)	38'893
I-S-Mod-EA-FC	684 (8%)	2091 (25%)	3157 (37%)	0 (0%)	2439 (29%)	43'865
C-Mod-EA	42 (1%)	92 (1%)	42 (1%)	0 (0%)	8195 (98%)	529'222
I-S-Mod-EA-Ref-EP(1)	1965 (23%)	791(9%)	3135 (37%)	54 (1%)	2426 (29%)	445'147 (6d)
I-S-Mod-EA-Ref-CI(1)	698 (8%)	2096 (25%)	2769 (33%)	369 (4%)	2439 (29%)	639'563 (8d)

Verification of such a large code basis with the standard bounded model checking approach is not feasible. For the sake of thoroughness, we have started a global analysis with `QPR Verify`. The underlying bounded model checker was not able to transform the program into a valid SMT formulae. Therefore, none of the hard verification tasks could be verified.

We therefore ran different modularization and refinement approaches on SQLite. The fastest analysis (E4), that also produced the least amount of `unknown` checks (E1), verifies every check individually (I) with slicing (S) and the environment abstraction activated. The analysis needs 38893 seconds (nearly 11 hours) for 8371 checks, meaning on average under 5 seconds per check. The analysis is that fast because due to the slicing algorithm abstracting memory and function calls and the local analysis abstracting the call environment, we gain modules consisting of only a single function. The abstractions lead to 698 (8%) `safe` checks, 2095 (25%) `loop-bound safe` checks, 3138 (37%) `cond. unsafe` checks and 2426 (29%) checks that our solver was not able to verify. 2426 checks could not be verified even with modules containing only a single function. Such cases arise, when the function relies strongly on memory states and has a large number of function calls. Abstracting such calls, can lead to large search spaces where the underlying SMT solver is not able to generate a proof in the given time of 600 seconds. For such a large projects, we see 30% of hard verification checks or 10% of the 27149 overall checks as a huge improvement to the state-of-the-art, where often no checks could be verified at all.

The best result (E2, E3) with `QPR Verify` was archived through the refinement of the fastest analysis with enumerative preconditions (I-S-Mod-EA-Ref-EP). Our approach is able to refine the analysis and classifies 23% of checks as `safe`, 9% as `loop-bound safe`, 37% as `cond. unsafe`, 1% as `unsafe`. The verification tasks that could not be verified through the fast analysis are not refined through the precondition analysis, thus 29% of checks remain unsolved. The 37% of `cond. unsafe` checks contain potential false positives that could further be refined through additional callers or learned preconditions. Refining the analysis with enumerative preconditions took about 8 days. The long verification time arises from the splitting of modules because of multiple callers.

However, the modules are independent from each other and could in future work be parallelized.

Note, that splitting of modules through preconditions lead to better verification results than the splitting through caller inclusion. This arises due to complex and large modules that are harder to verify if the complete caller is included into the module instead of a substitution of a given function call with the generated precondition. Time saved by substituting preconditions more then compensates for the time spend on precondition generation.

SQLite with around 400k lines of code is to large to be verified by any standard bounded model checking tool. Similar to the BMI160 project, we wanted to compare `QPR Verify` with Polyspace, implementing the scalable abstract interpretation approach. We set precision levels to minimal values and started an analysis on the whole program without any verification timeout. After 24h Polyspace terminated the analysis because it was not able to translate the program into its internal intermediate representation. If the translation of the given program into their IR, Polyspace assumes that no verification will be possible. The failed transformation into an IR is similar to the failed translation of LLBMC and `QPR Verify` into our IR, because of the sheer size of the program.

Polyspace 2017b has the option to create a modularization of the program. For SQLite this modularization produced a single module containing the whole program. We assume that for a more meaningful partition an expert user would have to specify module limits including specifications. Therefore, Polyspace is not able to produce any verification results for SQLite.

Approaches that do not verify the whole program but concentrate on bug finding are in general more scalable then throurough verificiation approaches. The tool Coverity is thus able to verify SQLite. Coverity offers an online verification. We submitted a build of SQLite with their build tool Cov-Build and verification results were made available 24 hours later on their website [4]. Coverity analyzes 19,3254 lines of code and finds 115 defects. Of those 115 checks it categorizes 21 with an high impact, 88 with a medium impact and 3 with low impact. The checks are similar to `QPR Verify` and Polyspace but additionally include extensive memory checks and resource leaks. With `QPR Verify` we are able to find 369 `unsafe` checks in about 8 days with our analysis *I-S-Mod-EA-Ref-CI* and 54 `unsafe` checks in about 6 days. Therefore, regarding bug finding, Coverity is more efficient but with more time, we are able to find more defects. Note, that the check categories and their definitions do not perfectly align and therefore the comparison of total numbers should be seen as general estimations of verification prowess of `QPR Verify` and Coverity. Additionally to the bug detection implemented in Coverity, our approach is able to verify all critical program locations wrt. our check categories and proves safety for 23% of hard verification tasks and 67% (18,685+1,965 of 27,149) of overall generated verification tasks.

[4]Results can be found at: `https://scan.coverity.com/projects/sqlite-verification-details?tab=overview`

The verification of SQLite marks, to the authors knowledge, the first fully automatic whole-program verification of a software project of that size. Through our modular software bounded model checking approach, we are able to verify around 69% of the 8371 hard verification tasks. Our abstractions still introduce a possibly large number of false positives that can be further refined given more time or in future work more efficient (potentially parallel) module splitting and refinement.

7.2.3 MNAV1.5

MNAV [117] is an open-source control software for fixed-wing aircrafts provided by [80] and first verified by Gurfinkel et. al in [67], who concentrated on the verification of buffer overflows and managed a 4% warning rate for 815 buffer overflow checks. MNAV contains 2190 lines of original source code, including external header files totaling in 160K LOC [67]. After presenting two benchmarks representing the strength of our approach for a driver function BMI160 and a large scale program of about 400K LoC excluding libraries, we present limitations and analyze the the number of false positives introduced by our abstractions. MNAV has a main function and is of medium size and thus verifiable through a global analysis.

We again set the `QPR` configuration `Hosted` to true, leading to a error free building process of the software. During compilation of the project, 1253 checks were detected. The preprocessing analysis was able to verify 929 (74%) of these checks as `safe` and 0 (0%) as `unsafe`. Thus, there are 324 (26%) hard verification tasks left. The project was configured with a loop-bound of 1 a timeout of 300 seconds and with the option `optimize-bitcode` as set to true.

Table 7.6: Results set-up and preprocessing analysis of `QPR Verify` for MNAV1.5.

Set-Up	`safe`	`unsafe`	`todo`	Total	Time
MNAV	929 (74%)	0	324 (36%)	1253	6 sec

The names of approaches are equivalent to the notions described for the BMI160-Driver and SQLite. We first show two results applying global analysis representing optimal results. Afterwards, we again display results of different modularized verification runs sorted by time.

MNAV is an control software that is nested in an infinite loop. It therefore shows the limitation of Bounded Model Checking, with which we can only prove safety for a fixed loop-bound. We show verification results of analyzing MNAV with a loop-bound of 1. Verification of MNAV with a loop-bound of 3 showed that while runtime increases the results do not differ greatly. For the global analysis the runtime is increased by around 54% from 130 to 200 seconds and produces the same results. For the slowest refinement analysis with 5 iterations, the analysis takes 115% times longer, while results are still equivalent. We therefore assume the small scope hypothesis and argue that if the program contains errors they are detected during the first few iterations of the infinite while loop.

Table 7.7: Results for `QPR Verify` solving approaches for MNAV1.5 with a loop-bound of 3.

Approach/Results	`safe`	`loop-bound safe`	`cond. unsafe`	`unsafe`	`unknown`	Time(sec)
Global Analysis	0 (0%)	324(100%)	0 (0%)	0(0%)	0 (0%)	375
I-S-Mod-EA	70 (22%)	13(4%)	79 (24%)	0(0%)	162(50%)	49
C-Mod-EA	0 (0%)	228(70%)	77 (24%)	0(0%)	19 (6%)	5921
I-Mod-EA-Ref-EP	0 (0%)	301(93%)	19 (6%)	0(0%)	4 (1%)	8973
I-Mod-EA-FA	0 (0%)	234(72%)	79 (24%)	0(0%)	11 (3%)	10851
I-Mod-EA	0 (0%)	230(71%)	76 (23%)	0(0%)	28 (9%)	15484
I-Mod-EA-Ref-CI(1)	0 (0%)	298(92%)	26 (8%)	0(0%)	0 (0%)	39491
I-Mod-EA-Ref-CI(5)	0 (0%)	310(96%)	14 (4%)	0(0%)	0 (0%)	56882

However, errors that arise from thousands or even millions of iterations over a long period of time can not be detected. Such errors can occur, if a counter is infinity increased or information is written to memory until an overflow happens.

We will now describe the results of the different analysis runs on the basis of Table 7.7 with a loop-bound of 3. The global analysis verifies all 321 hard verification tasks as `loop-bound safe` and will serve as an optimal solution with which we compare the number of false positives introduced by our structural abstractions.

The fastest analysis (E4) is again our environment abstraction (EA) with active Slicing (S) for individual checks (I). Due to call environment abstraction and specially due to memory and function behavior abstractions of our slicing algorithm, 70 (22%) of checks can be classified as `safe`. Therefore, through abstractions, the bounded model checking approach can prove safety even regarding checks lying inside an infinite loop. Contrary to this positive effect are 162 (50%) of checks not verifiable with an active slicing algorithm, because the verified function has on a LLVM bitcode level multiple return blocks. Currently, our slicing algorithm depends on a single return instruction and therefore single return block in LLVM. It is part of future work to either implement an bitcode transformation path generating a single return block or extending our algorithm to handle multiple return blocks.

The call environment analysis (Mod-EA) with grouped checks (C) produces 228 (70%) `loop-bound safe` checks, 77 (24%) `cond. unsafe` and 19 (6%) `unknown` checks in 5921 seconds. The environment abstraction therefore leads to 30% false positives. Furthermore, the overhead produced by our modularization increases the runtime by a factor of 15 compared to the global analysis. However, the later verification runs of `QPR Verify` show the effectiveness of our refinement strategies. Through preconditions alone (I-Mod-EA-Ref-EP), we can reduce the number of false positives to 7%. If we accept longer runtimes activating caller inclusion (I-Mod-EA-Ref-CI(5)), we can even reduce the number of false positives to 4%.

Overall, we demonstrate a program that is verifiable by our global analysis. We show that the whole program analysis is able to effectively very such smaller programs. Our modularization is aimed towards larger software projects or API programs without a main function.

The verification of MNAV showed that structual abstractions introduced by our modularization produce about 30% false positives but also that the number of false positives can be reduced to 6% through preconditions and 4% by caller inclusion.

We again compare our evaluation results with the two commercial tools Polyspace and Coverity. For Polyspace, we adopted the same configurations as for the BMI160-Driver. The results can are presented in Table 7.8. Polyspace finds an illegally dereferenced pointer (`red` check) in the main function. For overflows one can activate wraparounds in Polyspace, such that the tool is able to verify checks located after an unsafe location. For illegally dereferenced pointer, Polyspace did not implement any such abstraction semantics and therefore does not verify any source code after the illegally deferenced pointer. It therefore additionally gives the warning that MNAV contains 90 unused variables. Concentrating only on the verified properties, we can see that Polyspace is able to prove safety for 87% of checks lying inside an infinite loop. Abstract interpretation approaches calculate fix-points and are thereby able to argue over infinite loop iterations. They therefore produce 9% orange warnings representing potential false positives. If we assume the small-scope hypothesis and see `loop-bound safe` checks as `safe` or `green`, this is higher then the 0% false positives of our global analysis or the 4-6% of our refined modularization approaches.

Table 7.8: Results of verifying MNAV1.5 with Polyspace

Polyspace	`green`	`orange`	`red`	`grey`	Total	Time
MNAV1.5	110 (87%)	12 (9%)	1 (1%)	4 (3%)	123	154 sec

Coverity analyzes 14206 lines of code and finds 19 defects. It finds 7 resource leaks and 12 additional errors regarding API, pointer and error handling. Such properties are not supported by `QPR Verify` and therefore a comparison is not possible. However, it should be noted, that Coverity does not find arithmetical, type cast or any other by us supported errors thus implicating that the 324 `loop-bound safe` checks by `QPR Verify` are indeed `safe`.

7.2.4 Summary

We verified three different real-world applications with sizes between 2000 and 400k lines of code plus external libraries. The BMI160-Driver showed the applicability of our approach for API programs without a main function. The database SQLite demonstrated that through our modularization, we are able to meaningfully verify modern sized programs. Through the autopilot software MNAV1.5, we got a sense of the number of false positives introduced by our structural abstractions and quantized the precision increase of our refinement strategies through a reduced number of false positives.

We summarize the main insights and improvements of our modular bounded model checking approach. The results are differentiated by the three verification phases *set-up*, *modularization* and *refinement*. The main improvements of our approach are compared to the state-of-the-art are:

Set-up 1 Detection of a higher number of checks compared to current verification approaches like Polyspace and Coverity.

Set-up 2 Fast preprocessing analysis that is able to verify between 61% and 74% of checks in seconds.

Set-up 3 Slicing algorithm that leads to very fast results but produces a relatively high number of false positives.

Modularization 1 Verification of API programs without a main function, which was previously not possible through bounded model checking.

Modularization 2 Increase in scalability to verify 400k lines of code software project for which state-of-the-art bounded model checker (CBMC) and an abstract interpretation implementation (Polyspace) timeout.

Refinement 1 Reduction of false positives through caller inclusion and automatically generated preconditions and therefore fewer `"orange"` warnings compared to Polyspace.

Refinement 2 Precondition generation and substitution produces precise results in relatively short time compared to no refinements with call environment abstraction or increased module sizes through caller inclusion.

Overall, we presented different approaches that balance between precision and scalability (represented by runtime). In particular, our refined modularization advances the state-of-the-art of static analysis for large scale software projects compared both to standard bounded model checking as implemented in LLBMC or CBMC and abstract interpretation approaches like Polyspace.

Chapter 8

Related Work

We developed a modular software bounded model checking approach to verify run-time errors in industrial (embedded) software. In this chapter, we present related work in the context of static analysis of software written in C or C++. There are many different techniques and tools performing static analysis.

Tools based on **Abstract Interpretation** techniques are most dominant for the industrial application of safety-critical software. Polyspace [154], Astrée [43], IKOS [22], C Global Surveyor [23] and CRAB [57] are widely used implementations. In general, the theory of Abstract Interpretation by Cousot and Cousot [39], envelops approaches approximating system semantics. The mentioned implementations of this theory gain scalability mostly through the abstraction of value domains.

Polyspace is one of the first static analyzer based on Abstract Interpretation. As value domains, it implements static intervals, polyhedron models and a mixed approach consisting of integer lattices and complex polyhedrons. Which domain is used can be specified through a precision level. Domains that are scalable often lose information about specific values that lead to errors. For Abstract Interpretation tools, this can potentially produce a large amount of false positives.

Our evaluation showed that even with the most scalable interval domain, abstract interpretation approaches are not able to verify modern projects as a whole. Tools applied in industry therefore offer a modular verification. However, the modularization, meaning program partitioning and specification of the program has to be done manually. Additionally to the manual labor needed, this requires expert knowledge of the program.

Bounded Model Checking approaches are very precise and the underlying SAT and SMT solvers produce exact error values and a lower amount of false positives. As trade-off, they often lack the scalability to verify large programs. Current tools based on the BMC approach like CBMC [103] and ESBMC [56] are very precise but often not tuned towards the application of large industry projects. For example, it is hard to control and see which checks are actually performed at each program location, and sometimes code modifications are needed on the input to make it parsable by the tool.

Thus, the support for complete C-features, easy set-up and abstraction methodologies are limited for those approaches. There are frameworks trying to combine different techniques: The tool Sea-Horn [67] scales up its analysis using a combination of Abstract Interpretation and property directed reachability (PDR) to generate loop invariants. Frama-C [44] and VeriAbs [30] also combine different solving techniques as portfolio solvers. They provide a range of functionalities, but do not explicitly tackle the scalability of BMC. Approaches taking this direction typically try to minimize the formulas for BMC, e.g. lifting the assertions closer to the entry point of analysis [105] or abstracting loops [31].

Manual modularization typically requires formalization of interfaces and dependencies between modules. Under the headline of **compositional verification** or **assume-guarantee reasoning** several approaches for modular verification have been proposed in the past [36, 66, 73]. This work, however, generally does not cover the aspect of how to generate modules; instead it relies on manual approaches for partitioning. There exist frameworks that automate part of the modularization task, e.g., by creating necessary preconditions automatically through an incremental learning algorithm [37], or by deducting modules from program design [60]. In this thesis, we have taken an another approach. We abstract the problem on a structural level and our implementation `QPR Verify` offers a fully automatic modularization. Dependencies do not have to be specified manually but are abstracted and then refined through preconditions. Tools using separation logic for modularization are among others ETH's Viper [123] and Facebook's INFER [26], which are loosely related to our approach but concentrate either on manual specification or limited memory properties.

The two approaches most closely related to verification through **precondition generation** are the automatic precondition interference by Cousot et al. [42, 41] and the compositional verification implemented in the tool BLITZ [32].

The tool BLITZ implements a composition verification approach for a bounded model checking based verification. They create under-approximated preconditions through the extension of information gained by SAT-proofs leading to a property violation. While the bounded model checker can extract and exploit this information, they are not human-readable. Furthermore, they are only applied in the context of bug finding and not whole program verification. The general techniques of function and loop summaries [102] serve a similar purpose. Cousot et al. introduced a framework for automatic inference of under-approximated (they call it necessary) preconditions based on Abstract Interpretation techniques. They argue that developers will object to over-approximated preconditions due to the occurrence of false positives.

We contrary argue that the context of the verification tasks can necessitate stronger preconditions. When verifying safety critical software, it is the aim to guarantee and prove an error free execution. While finding errors is an important step, software is only accepted in safety critical environments when error-free execution is guaranteed. We first create enumerative/under-approximated preconditions on a user-readable level and then generalize them through a learning approach.

Over-approximated (or sufficient) preconditions can be generated by different techniques. In [25], Hoare triples are established to produce preconditions for shape analysis. [120] presents modular assertion checking through a mixture of Abstract Interpretation and weakest precondition calculus and [130] produces over-approximated preconditions for heaps. Generally, such approaches do not substitute functions at their call-sites with generated preconditions and do not consider whole program verification of larger programs. This also applies to learning algorithms presented in tools like PIE [127] and others [5, 127, 52]. Furthermore, such approaches rely on data generated by test-runs, which can never guarantee the completeness of preconditions.

Snugglebug [27] and others [54, 131] use symbolic backwards execution together with dynamic analysis to find bugs. Again these approaches are not able to prove the safety of programs. Yet Snugglebug as well as Cousot et al. showed the strength of precondition simplifications and are a strong motivation to include presented simplifications schemes into our approach.

We presented approaches aiming to verify the safety of a given program, thus implying that all bugs and errors will be found. There exist tools that do not claim to find all errors but a relevant subset of them. In our evaluation, we compared our implementation `QPR Verify` to Coverity [1] as a **bug-finding tool**. Tools like Coverity or Klocwork [77] sacrifice finding all bugs for the benefit of fewer false positives. They do not model all possible program traces but rely on local information and pruned execution paths.

Chapter 9

Conclusion

9.1 Summary

We introduced novel approaches aimed at advancing software verification based on bounded model checking for large software projects written in C. Our overall approach presented in Chapter 2.1 verifies a program in three phases set-up, modularization and refinement.

In the first phase, we established challenges for industrial software verification and improved the set-up phase of the verification process (Chapter 4). Checks are automatically inserted at critical program locations and the source code together will all configuration information is automatically compiled into a single bitcode file. Through an extensive configuration framework, the user can choose different options, e.g. for bitcode optimization, check handling, and a novel slicing algorithm based in memory abstraction. A fast preprocessing analysis utilizing only local information available through the LLVM framework, classifies a majority of checks as `safe` or `unsafe`. Hard verification tasks are left and, at this point, can be verified through a whole program analysis performed with the state-of-the-art bounded model checker LLBMC.

The second phase, presented in Chapter 5, focused on the scalability issue of bounded model checking. Encoding the complete program into a single formulae is not feasible regarding the size of modern software projects. Contrary to manual program partitioning or abstraction of value ranges, we introduced structural abstractions based on the call graph of the program under verification. First, we defined program semantics for LLVM and thereupon introduced notions and properties of modularization in the context of software verification. We presented and implemented two structural abstractions. Given an entry-point function containing checks, we can abstract the environment in which the function is called and can thus create smaller modules. Furthermore, we implemented a function call abstraction that over-approximates the return value and memory changes of function calls. Through these abstractions, a given program can be partitioned into smaller modules. The approach is able to prove statements over programs marginally larger then the current capabilities of standard bounded model checking tools.

In the third phase, we tackled the drawback of over-approximations, namely false positives. Through over-approximation of program traces, the modularization produces error messages where there are no errors. In Chapter 5, the refinement of function calls through postconditions were discussed and due to limitations of current alias analysis methods not prioritized. Therefore, the third phase presented refined abstractions for a given modularization abstracting the call environment of functions containing checks. Next to a call graph method that iteratively included callers into the module, the generation of enumerative and learned precondition was presented. Such preconditions were pushed bottom-up through the program by substituting function calls with preconditions.

Our final evaluation on three different real-world applications with sizes between 2,000 and 400k lines of code plus external libraries showed the advantages of our modular bounded model checking approach compared to the state-of-the-art. The analysis of the BMI160-Drived presented a small-sized use-case of our call environment abstraction and showed that global approaches implemented in standard bounded model checker like LLBMC or tools like Coverity can not be applied to API programs. Furthermore, `QPR Verify` outperformed the commercial abstract interpretation tool Polyspace both in runtime and solved checks.

The increased scalability of the modular bounded model checking approach was demonstrated on the SQLite database. `QPR Verify` was able to classify 71% of all hard verification tasks while tools like LLBMC and Polyspace are not able to even encode the program. For both benchmarks, potential false positives were refined through caller inclusions and preconditions. The comparison of the global analysis with modularization and refinement showed that the modularization alone introduces around 30% false positives but also that the false positive rate can be reduced to 4% through our refinement steps. The best results for all benchmarks balancing time consumption and precision was a call environment abstraction with refinement through precondition generation and substitution. Our evaluation presented different approaches that balance between precision and scalability showing an improved state-of-the-art for the static analysis of large scale software projects compared both to standard bounded model checking as implemented in LLBMC and abstract interpretation approaches like Polyspace.

Overall, our approach advanced the state-of-the-art of bounded model checking both in usability and most important scalability. Through the introduced modularization techniques, we are able to verify projects modern projects with millions of lines of code. The number of potential false positives is the main challenge given such a modularization. Introduced refinement strategies minimize such unnecessary error reports.

9.2 Future Work

We present some ideas how to further advance our modular verification approach towards a scalable and precise analysis of modern sized software projects.

The modularization approach generates modules, where only a single function containing a check is encoded into a SMT formulae. While this is the most scalable approach regarding functions as the smallest verifiable unit, it can be more efficient to initialize the verification on larger modules. As described in Section 5.3.2, an optimal modularization should have minimal dependencies to other modules. A modularization aimed towards such modules could adjust existing k-partitioning algorithms for call graphs and create modules of size k. It would be interesting to explore, how the increase in precision and scalability would justify the overhead for such an partitioning.

Gaining scalability through abstractions potentially leads to unnecessary error messages. This is true for structural abstractions as introduced in this thesis, as well as for value abstractions as utilized in various Abstract Interpretation approaches. In Section 5.3, we briefly discussed postconditions to refine our function call abstractions. As future work, an evaluation with a more scalable alias analysis may produce better results. Otherwise, there is potential for customized, maybe partial alias analysis refining data dependencies.

The automatic generation of module dependencies, namely pre- and postconditions, based on erroneous checks is, in the authors view, the most promising approach tackling the scalability issue of large scale verification. Preconditions should be generated based on information already available during the solving process. For bounded model checking, error traces can be utilized through symbolic backwards execution of checks to generate better preconditions. Furthermore, the impact of precondition simplifications and a better integration of the precondition-learner and function-pointer handling into `QPR Verify` worthy to further investigate. Currently, the refinement steps through caller inclusion and precondition lead to significant overhead in verification time. This can be minimized through a parallel verification of independent modules and a database supporting effective handling of thousands of checks.

Evaluation of software verification approaches on larger software projects is still a difficult venture. The Competition on Software Verification (SV-COMP) [16] provides source code of problems containing a handful of functions. To improve the comparability of software verification approaches larger projects would have to be uniformly annotated with (thousands of) different checks and added to the benchmark set of the SV-COMP. Alternatively, projects like SAFIR and ASSUME aiming towards a unified verification format have to be adapted and more intensively applied in research and industry.

Overall, the automatic verification of industry-sized applications is still a huge challenge in research and industry. We see our modular software bounded model checking approach as a promising technique to verify large-sized programs and further improve the state-of-the-art of software verification.

Part III

Verification of Neural Networks

Chapter 10

Equivalence Verification of Neural Networks

The third part of this thesis presents the contribution **C2** and therefore a framework for the verification of equivalence for neural networks. In this chapter, we will present contribution **C2.1**. First, we introduce and motivate the relatively young research field of neural network verification. Afterwards, we present exact and relaxed notions of equivalence for regression and classification tasks. We prove that the ϵ-equivalence problem for neural networks is coNP-complete. In the following chapters, we then present the contributions **C2.2** presenting a verification through mixed-integer linear programming based on our publication [KB2] and **C2.3** presenting an adjusted geometric path enumeration approach based on our publication [KB6].

Parts of the work on defining and verifying equivalence verification of neural networks were conducted in student projects. The first definition of equivalence and the verification approach based on mixed-integer linear programming was pursued during the Master's Thesis of Philipp Kern [89] closely supervised by the author. The expanding work of proving coNP-completeness for equivalence verification and the modification and optimization of the geometric path enumeration approach were conducted in a "research laboratory" (Praxis der Forschung) project in which the student Samuel Teuber was also supervised by the author.

10.1 Introduction

The popularity of neural networks (NNs) for solving machine learning tasks has strongly increased over the last years. Nowadays, NNs are considered state of the art solutions for many machine learning tasks, including machine translation [7], image processing [101] or playing games like Go and Chess [136]. However, for non-trivial networks, the complex structure of neurons, layers and weights, often renders them incomprehensible to humans. While this does not have serious consequences when it comes to playing games, it can have a severe impact when neural networks are applied to safety-critical systems such as airborne collision avoidance [81] or self-driving cars [20]. Validation and verification procedures are thus needed to provide safety guarantees.

The verification of NNs is a relatively young research field. Among the first papers published is the work by Pulina and Tacchella [132], where the authors checked bounds on the output of multilayer perceptrons. Since then, an increasing size of work in the field of neural network verification has been published.

The literature can broadly be classified into *adversarial robustness* verification (e.g. the work by Singh et al. [138]), functional property verification (e.g. the work by Katz et al. [85] on the verification of ACAS Xu) and equivalence verification (e.g. Narodytska et al. [126] and Paulsen et al. [129]).

The first category of verifying adversarial robustness is concerned with a problem specific to NNs, i.e. the question whether there are seemingly "normal" inputs which produce unexpected outputs. Or more specifically, checking whether a network assigns the label of a known reference input-point to all points in a small region around it. The second category checks whether a given NN fulfills a functional property. Functional verification is particularly relevant to ensure that a NN is capable of handling a certain safety critical task.

The third category of equivalence verification can be regarded as relational verification of neural networks and checks whether two NNs are equivalent to each other. The main application of equivalence verification, is in the context of NN compression. As NNs grow ever larger and computing becomes ever more ubiquitous, resource restrictions require to compress large NNs into smaller models. Cheng et al. [29] give an extensive survey of such compression techniques. Furthermore, equivalence verification can be deployed to examine the influence of certain NN-based pre-processing steps [126] or in cases where performing multiple verification tasks on a large NN would be too expensive [KB2].

The work of Narodytska et al. [126] marked the first time that equivalence was defined and a verification approach for binarized neural networks was presented. Binarized Neural Networks are (deep) feed-forward neural networks, where network parameters like weights and biases have only binary values, thus allowing a SAT encoding. The next publications verifying equivalence of neural networks were published around the same time by Kleine Büning et al. in [KB2, KB6] and Paulsen et al. in [128, 129].

In this chapter, we present novel notions of equivalence first presented in [KB2] and prove the equivalence verification to be coNP-complete. The next two chapters, present two approaches verifying equivalence based on an exact MILP encoding (Chapter 11) and an abstract interpretation approach (Chapter 12).

10.2 Equivalence of Neural Networks

There has recently been a line of work which proposes various compression techniques for NNs (for a full review see Cheng et al. [29]). While such techniques have been shown to be useful in practice, most lack a formal proof of correctness and only rely on empirical evidence. The usage of such techniques thus raises the question of how to prove that two NNs $\mathcal{R}$ (reference) and $\mathcal{T}$ (test) and their corresponding mathematical functions $g_{\mathcal{R}} : \mathbb{R}^I \to \mathbb{R}^O$, $g_{\mathcal{T}} : \mathbb{R}^I \to \mathbb{R}^O$ are equivalent, i.e. that they produce the same results.

Narodytska et al. [125] consider two feed-forward NNs equivalent if, for all valid[1] inputs of the input domain, the NNs produce the same output labels. They present an technique verifying, among other properties, equivalence for the specialized class of binarized NNs, which allows them to produce a SAT formula representing the equivalence of two NNs. However, they do not experimentally evaluate their encoding of the equivalence property. Kumar et al. [104] denote this equivalence property by local equivalence over a domain, and use the term equivalence for NNs that give the same output for all inputs of that domain. Based on these notions they collapse layers of a given NN while guaranteeing the equivalence to the original NN.

Given a reference network $\mathcal{R}$ and a test network $\mathcal{T}$, with their corresponding mathematical functions $g_{\mathcal{R}} : \mathbb{R}^I \to \mathbb{R}^O$, $g_{\mathcal{T}} : \mathbb{R}^I \to \mathbb{R}^O$. We assume that the inputs and outputs of the neural networks have the same cardinality and ordering. All networks with varying input or output cardinality are classified as not equivalent. In practice, one can append additional neurons or an intermediate layer representing a mapping from the cardinality of the input and output neurons of $\mathcal{R}$ to the cardinality of $\mathcal{T}$ for input and output neurons.

To simplify notation, we now denote the input space of the two neural networks under verification as X. Proving exact equivalence of the test NN $\mathcal{T}$ and the reference NN $\mathcal{R}$ would then mean to ascertain that

$$\forall \vec{x} \in X : f_{\mathcal{R}}(\vec{x}) = f_{\mathcal{T}}(\vec{x}) \ . \tag{10.1}$$

However, the training procedure of NNs is highly non-deterministic and training could be on different datasets, thus leading to differences in the learned weights, even if the NNs share the same number of layers and neurons. It is therefore unlikely for two NNs to fulfill the exact equivalence property stated above. Hence, we need to relax it to obtain a more practical notion of equivalence. In general, this can either be achieved by (1) relaxing the exact equality of the function values in Equation 10.1 through a less strict relation $\simeq$, or (2) restricting the domain of the inputs to the NNs to smaller subsets, for which equality is more likely. The first approach is described below, while we discuss the input restriction in Chapter 11.

Relaxed Equivalence Properties. The definition of exact equivalence in Equation 10.1 can be written as a difference: $\forall \vec{x} \in X : g_{\mathcal{R}}(\vec{x}) - g_{\mathcal{T}}(\vec{x}) = 0$. An relaxation would be to consider two functions equivalent, if their difference is at least close to zero within some threshold ϵ.

Definition 7 (ϵ-Equivalence)**.** *Two NNs $\mathcal{R}$ and $\mathcal{T}$ are ϵ-equivalent with respect to a norm $\|\cdot\|$, if $\|g_{\mathcal{R}}(\vec{x}) - g_{\mathcal{T}}(\vec{x})\| < \epsilon$ for all $\vec{x} \in X$*

This notion of equivalence for neural networks was first established in [KB5] and denoted at about the same time by Paulsen et al. [128] as *Differential Equivalence*.

[1] Validity just ascertains that inputs are suitably bounded, e.g. to a range of $[0, 255]$ for greyscale pixels.

While this is a valid relaxation in the context of regression NNs, the functions $g_{\mathcal{R}}$ and $g_{\mathcal{T}}$ compute class probability distributions when it comes to classification problems. In most of these cases, one is not interested in the full class probability distribution, but only in the classification result, the class assigned the highest probability by the NN. In that case, we can obtain a relaxed equivalence property by comparing only the classification results.

Definition 8 (Top-1 Equivalence)**.** *We call $\mathcal{R}$ and $\mathcal{T}$* top-1 equivalent*, if $g_{\mathcal{R}}(\vec{x}) = \vec{r}$ and $g_{\mathcal{T}}(\vec{x}) = \vec{t}$ satisfy* $\arg\max_i r_i = \arg\max_i t_i$ *for all $\vec{x} \in X$.*

There exist different definitions of the arg max function in literature. Here, we determine that the function returns a single index. In case of multiple elements with the same maximal value, arg max returns the first index regarding the ordering of the input vector. In our Definition 8, the vectors $\vec{r}$ and $\vec{t}$ are equivalently ordered and therefore, *top-1* equivalence holds for the case that two networks produce the exact same probabilities for two or more classes.

This notion of equivalence for classification can also be denoted as one-hot equivalence in reference to the one-hot vector for classification NNs. We note that this definition is closely related to the property of adversarial robustness [147], however we compare the classification results of two NNs instead of comparing the classification result of one NN with the ground-truth.

The notion of *top-1* equivalence can be relaxed even further when we consider not only the most likely class, which is the classification result, but instead take the k most likely classes into account (the definition is motivated by a similar idea in [28]).

Definition 9 (Top-k Equivalence)**.** *A test NN $\mathcal{T}$ is* top-k*-equivalent to a reference NN $\mathcal{R}$, if $g_{\mathcal{R}}(\vec{x}) = \vec{r}$ and $g_{\mathcal{T}}(\vec{x}) = \vec{t}$ satisfy*

$$\arg\max_i r_i = j \implies \mathrm{pos}(t_j, \vec{t}) \leq k \ ,$$

where $\mathrm{pos}(w_j, \vec{w})$ *returns i, if w_j is the i-th largest value in vector $\vec{w}$, and r_j is the unique maximum component of vector $\vec{r}$.*

Here, we require r_j to be the *unique* maximum component to prevent verification results that are conditional to the ordering of two equivalent probabilities. This is also discussed during the encoding of properties into MILP in Section 11.2.2.

Informally, a testing NN $\mathcal{T}$ is *top-k* equivalent to a reference NN $\mathcal{R}$, if the classification result of $\mathcal{R}$ is amongst the top k largest results of $\mathcal{T}$. This can be interpreted in a way, such that the NN, even if it differs from the classification result of the original NN, at least only makes sensible errors. The *top-1* equivalence can also be seen as a special case of *top-k*-equivalence for $k = 1$.

Note that, while exact equality and *top-1* equality are equivalence relations in the mathematical sense, neither ε-equivalence, nor *top-k* equivalence for $k > 1$ meet that criterion, as both are not transitive and the latter additionally is not symmetric. Nevertheless they are useful relations as they still express guarantees about the similarity of the outputs of two NNs.

10.3 NN Equivalence and NP

After establishing notions for equivalence verification for neural networks, we analyze the complexity of proving ϵ-equivalence. We assume that the complexity of proving *top-1* equivalence and *top-k* equivalence are similarly complex.

Katz et al. [85] have previously shown that the satisfiability problem for linear input and output constraints of a single NN with ReLU nodes is NP-complete. We refer to this decision problem as Net-Verify. An instance of Net-Verify is given by a conjunction of linear constraints $\psi_1(x)$ on the input, a conjunction of linear constraints $\psi_2(y)$ on the output and a NN N. $\psi(x, y) = \psi_1(x) \wedge \psi_2(y)$ is said to be satisfiable if there is an x such that the network N outputs y for input x and $\psi(x, y)$ is satisfied.

In this section, we show that the ϵ-equivalence problem for NNs is coNP-complete. Since disproving ϵ-equivalence is NP-complete, the task of proving ϵ-equivalence is coNP-complete.

Theorem 1 (ϵ-Net-Equiv is NP-complete)**.** *Let $\mathcal{R}, \mathcal{T}$ be two arbitrary NNs with* ReLU *activation functions and let $\mathcal{I}$ be some common input space of the two NNs. Determining whether $\exists x \in \mathcal{I} : \|g_{\mathcal{R}}(x) - g_{\mathcal{T}}(x)\|_p \geq \epsilon$ is NP-complete for any p-norm $\|\cdot\|_p$.*

Proof idea. In essence, the proof consists of a reduction from Net-Verify to ϵ-Net-Equiv. In order to reduce a Net-Verify instance consisting of a NN $\mathcal{N}$ and a linear constraint specification ψ, we encode it as follows: The first NN $\mathcal{R}$ only consists of $\mathcal{N}$. The second NN $\mathcal{T}$ consists of $\mathcal{N}$ and a suitable encoding of the linear constraints ψ. We show that we only can disprove ϵ-equivalence iff $\mathcal{N}$ satisfies the given specification ψ.

Proof. We begin by showing that the problem is in NP. Assuming a witness x returned by some algorithm for a given instance of ϵ-Net-Equiv, we can easily check whether the witness is indeed violating the ϵ equivalence property by computing $\|g_{\mathcal{R}}(x) - g_{\mathcal{T}}(x)\|$.

We now need to establish that ϵ-Net-Equiv is NP-hard. To this end we demonstrate a reduction from Net-Verify to ϵ-Net-Equiv. Note that we can represent ψ_1 and ψ_2 as vectors of linear constraints of the form

$$C_1 x \leq b_1 \qquad\qquad C_2 y \leq b_2 \tag{10.2}$$

Reusing N and denoting $N(x)$ as the output of N for input x we can then construct a network with the following outputs:

$$\max(0, C_1 x - b_1 + \epsilon) \tag{10.3}$$
$$\max(0, C_2 N(x) - b_2 + \epsilon) \tag{10.4}$$
$$N(x) \tag{10.5}$$

The vectors in Equation 10.3 has as many dimensions as ψ_1 has constraints and the vectors in Equation 10.4 has as many dimensions as ψ_2 has constraints. Note that outputs of both (10.3) and (10.4) are only larger than 0 if a constraint is violated or an assignment is closer than ϵ to the bound imposed by the constraint.

We now make use of an additional ReLU gadget which computes the non-negative maximum of two values:

$$\text{relu}\max(a, b) = \max(0, a + \max(0, b - a))$$

We can now calculate the maximum deviation between Equation 10.3 and 10.4 by taking values in pairs of two and giving them as input into relu max. We therefore construct a kind of pyramid of relu max gadgets on top of (10.3) and (10.4) which finally outputs the maximum deviation d_{max}. Note, that this pyramid of maxima is of course polynomially bounded in the problem size. $d_{max} > \epsilon$ iff there exists a constraint on in- or output which is violated. This follows from the observation that $d_{max} > \epsilon$ iff the maximum value of (10.3) and (10.4) is larger than ϵ. (10.3) and (10.4) contain a value larger than ϵ iff there is some constraint (c, b) and some input $i \in \{x, y\}$ such that:

$$ci - b + \epsilon > \epsilon \iff ci > b$$

Conversely, $d_{max} < \epsilon$ iff all constraints in Equations 10.2 are satisfied. We now define $d^* = \max(0, 2\epsilon - d_{max})$ and define our network's final output as

$$N(x) + d^* \tag{10.6}$$

with $N(x)$ being the output vector of our designed neural network.

By checking ϵ-equivalence on N and the output defined in Equation 10.6 we can then solve our original Net-Verify instance:

$$\begin{aligned}
&\|(N(x) + d^*) - N(x)\| > \epsilon \\
\iff &\|d^*\| > \epsilon \\
\iff &2\epsilon - d_{max} > \epsilon \\
\iff &\epsilon > d_{max} \\
\iff &\text{all constraints are satisfied}
\end{aligned}$$

I.e. any input which violates ϵ-equivalence, is a solution to the Net-Verify instance. It follows that ϵ-Net-Equiv is NP-complete and therefore that proving ϵ-equivalence is coNP-complete. □

Chapter 11

Verification on Clustered Input through MILP Encoding

11.1 Introduction

In the previous chapter, we established the importance, notions and the complexity of equivalence verification for neural networks. To prove properties of neural networks different solving approaches have been proposed. On the one hand there are techniques based on the encoding of neural networks into formulas, such as SAT, SMT or MILP. On the other hand, there are approaches that are related to abstract interpretation. Domains such as zonotopes or star sets are pushed through networks and properties can be deducted by analyzing the relation of input and output sets.

This chapter presents contribution **C2.2**, which is based on our publication [KB2]. We choose an exact approach and aim at an MILP encoding of networks and properties. We concentrate on feed-forward neural networks with ReLU activation functions, as they are currently the most common and utilized form of networks [62]. Encoding the reference and test network together with one of the three equivalence properties into a mixed-integer linear programming problem, allows us to argue over neural network equivalence. We discuss over which regions of the input space, the networks should be equivalent and developed an method that maximizes the radii around a chosen input point, for which two networks are equivalent.

Contribution C2.2. We present an encoding for the developed equivalence properties in MILP. Additionally, we show an encoding for the verification of equivalence, as well as maximizing the size of equivalent regions, when the input domain is restricted to radii around a point in input space. We evaluate our approach using the constraint solver Gurobi [69] and a NN trained on the Optical Recognition of Handwritten Digits dataset [46]. The paper [KB2], was as far as we know, the first paper that evaluated NN compression methods by verification methods with respect to generating equivalent NNs. We name our approach **MilpEquiv**.

Structure. In Section 11.2, we present the encoding of two neural networks into a MILP formula as well as the encoding of our three presented equivalence properties into MILP. The next section 11.3 argues over the relevant input under which the networks should be equivalent. Compression of neural networks is shortly introduced in Section 11.4 as an application of our equivalence verification. Section 11.5 then marks the evaluation of our MILP encodings on the optical recognition of handwritten digits dataset. This chapter is finalized by a conclusion in Section 11.6.

11.2 Encoding as MILP

We present the encoding of neural networks and the properties ϵ-equivalence, $top-1$-equivalence and $top-k$-equivalence into MILP. Next to the encoding, we shortly discuss distance norms and interval arithmetic optimizations.

11.2.1 Encoding of Neural Networks as MILP

To argue over properties of NNs, we encode them in MILP utilizing the big-M encoding presented in [28]. By means of transformation one can see that our encoding is similar to the ReLU-encoding of [50].

For a NN to be encoded, we first have to encode a **single neuron**. A neuron j applies a non-linear activation function σ to a linear combination $s_j = \sum_{i=0}^{n} w_{ij}x_i$ of its inputs $x_0, ..., x_n$. Given fixed weights w_{ij}, this equation can be directly encoded in a mixed integer linear program. The non-linear ReLU activation function $y_j = max(0, s_j)$ can be encoded using given bounds $m \leq s_j \leq M$, which can be calculated knowing the bounds for the inputs x_i and weights w_{ij} of the NN. The ReLU function can be encoded using a new zero-one variable $\delta \in \{0, 1\}$, with $\delta = 0$ representing the case $(s_j \leq 0 \wedge y_j = 0)$ and $\delta = 1$ representing $(s_j \geq 0 \wedge y_j = s_j)$. The ReLU function is then encoded by the following set of linear inequalities:

$$
\begin{aligned}
y_j &\geq 0 & \qquad s_j &\geq m(1-\delta) \\
y_j &\geq s_j & \qquad y_j - s_j &\leq -m(1-\delta) \\
s_j &\leq M\delta & \qquad y_j &\leq M\delta\,.
\end{aligned}
$$

Given tight bounds $m \leq s_j \leq M$, the encoding can be further simplified [28]. If $M \leq 0$, we can directly encode the ReLU function as $y_j = 0$, and if $m > 0$, we encode the output of the activation function as $y_j = s_j$. These reductions in complexity are particularly valuable as they do not use any integer variables, on whom we might have to branch when solving the resultant mixed-integer linear program. Therefore, we employ the approach of [28] to generate tighter bounds by means of interval arithmetic and also solve for bounds on intermediate variables by maximizing or minimizing their values using smaller mixed-integer linear programs only covering a low number of layers of the NN at a time.

Based on the encoding of a single neuron, we can encode a **whole NN**. Each input of the NN is represented by a variable x_i with associated bounds $l_i \leq x_i \leq u_i$. These l_i and u_i can be set according to physical limitations or might be obtained from the respective training dataset and can be used for the calculation of subsequent bounds in the encoding. The neurons in the first layer are then encoded according to the previous description. The same procedure is applied to the neurons of the next layers with the outputs y_i of the neurons of the previous layer taking the role of the inputs above, until the output layer is reached. In classification NNs, the neurons in the output layer use the softmax activation function to normalize the classification distribution (see Foundations, Section 2.2.) Due to its exponential functions, an exact encoding of the softmax function in MILP is impossible. However, since the softmax function is monotonic, we are able to reason about the order of the outputs by encoding the linear combination of the input values for the neurons of the output layer in a classification NN. Even though we lose the ability to reason about the exact values of the outputs, the encoding is exact in the context of our introduced equivalence properties.

11.2.2 Encoding of Equivalence Properties in MILP

In the context of adversarial robustness, properties are often encoded as MILP problems [24, 86], a formalism we also employ for our equivalence properties. Searching for an input that maximizes the violation of these equivalence constraints has the advantage that we get information about the extent to which the corresponding NNs are not equivalent. With an encoding as a satisfiability problem, we would only get a single and possibly very small violation. In general, we encode equivalence of two NNs $\mathcal{R}$ and $\mathcal{T}$ as the following mixed-integer linear program

$$\begin{aligned} \max \quad & d && (11.1)\\ s.t.\ & \vec{r} = \mathrm{enc}_{\mathcal{R}}(\vec{i}) && (11.2)\\ & \vec{t} = \mathrm{enc}_{\mathcal{T}}(\vec{i}) && (11.3)\\ & d = f(\vec{r}, \vec{t}) && (11.4) \end{aligned}$$

where Equations 11.2 and 11.3 encode a reference NN $\mathcal{R}$ and the testing NN $\mathcal{T}$ on the common inputs $\vec{i}$ as described in Section 11.2, yielding the respective outputs of the NNs $\vec{r}$ and $\vec{t}$. The linear function f represents the encoding of an equivalence property yielding a scalar violation score d, that is then maximized to yield the maximum possible violation. The variables x_j include $\vec{i}, \vec{a}, \vec{b}$ and d as well as the intermediate variables introduced by the encoding of the neural networks and the chosen equivalence property. As we are dealing with MILP, some of these variables can be real numbers, while others are restricted to be integers. Below we are going to discuss the encoding of the function f, which calculates the scalar violation score d for a given equivalence property, for $top-k$ and then for ε equivalence.

Top-k Equivalence. We can encode the violation score of the *top-k* equivalence $\mathcal{R} \simeq \mathcal{T}$ (or $\mathcal{T}$ is equivalent to $\mathcal{R}$) as a simple difference

$$d = \hat{t}_k - t_j \ , \tag{11.5}$$

where $\arg\max_i r_i = j$. The variable $\hat{t}_k$ denotes the k-th largest component of $\vec{t}$. If $d = \hat{t}_k - t_j \leq 0$, then we have $\hat{t}_k \leq t_j$, meaning that the output of $\mathcal{T}$ corresponding to the classification result of $\mathcal{R}$ is larger or equal to the k-th largest output of $\mathcal{T}$. Therefore, t_j would be amongst the k largest outputs of $\mathcal{T}$ and thus satisfy $top-k$-equivalence. The main difficulty in encoding $top-k$-equivalence lies in encoding the sorting of the outputs of the NNs according to their activation value. We can encode the calculation of the descendingly sorted vector $\hat{\vec{x}} = \Pi\vec{x}$ of a NN's output values $\vec{x}$ by using a permutation matrix $\Pi = (\pi_{ij})_{i,j=1}^{n}$ similar to [82] and then adding the necessary ordering constraints (Constraint 11.9):

$$\hat{x}_i = \sum_j \pi_{ij} x_j \ \ \forall i \leq n \tag{11.6}$$

$$\sum_i \pi_{ij} = 1 \ \ \forall j \leq n \tag{11.7}$$

$$\sum_j \pi_{ij} = 1 \ \ \forall i \leq n \tag{11.8}$$

$$\hat{x}_i \geq \hat{x}_{i+1} \ \ \forall i \leq n-1 \tag{11.9}$$

$$\pi_{ij} \in \{0,1\} \ \ \forall i,j \leq n \ , \tag{11.10}$$

where the Constraints 11.7 and 11.8 together with the binary constraint on the π_{ij}, ensuring that each column and each row only contain one 1 and only 0 elsewhere, are sufficient to characterize Π as a permutation matrix. The constraints in 11.9 ensure that $\hat{\vec{x}}$ is sorted in descending order.

While multiplications of two variables are in general non-linear, we can utilize that the π_{ij} are binary variables, to encode the products $\pi_{ij}x_j$ in the above formulation. Binary multiplications $\delta x = y$, where $\delta \in \{0,1\}$, can be linearized by encoding the implications $\delta = 0 \rightarrow y = 0$ and $\delta = 1 \rightarrow y = x$ as linear inequalities.

Using the above information, we can retrieve sorted vectors $\hat{\vec{r}}, \hat{\vec{t}} \in \mathbb{R}^n$ of the outputs of two NNs. To find the component of t_j of $\vec{t}$ that corresponds to the largest component of $\vec{r}$, one can apply the permutation matrix to calculate $\hat{\vec{r}}$ to $\vec{t}$ and extract its first component. However, we don't need to generate two full permutation matrices. Realizing that we are only interested in the largest value of $\vec{r}$ and the k largest values of $\vec{t}$, it is sufficient to encode the first row of the permutation matrix for $\vec{r}$ and the first k rows of the permutation matrix for $\vec{t}$, thus reducing the number of binary variables. When multiple outputs r_i of NN $\mathcal{R}$ share the highest activation value, valid permutations could be obtained, such that in one of them component r_j and in the other $r_{j'}$ is the top component in $\hat{\vec{r}}$. Assume that we compare a reference NN $\mathcal{R}$ to a testing NN $\mathcal{T}$, that assigns the highest activation only to t_j, when given the same input as $\mathcal{R}$. Then, we would use this input as a counterexample to their equivalence, since we maximize the violation of the equivalence property and the solver would chose the permutation of

$\mathcal{R}$'s outputs, that assigned $r_{j'}$ as the top component. The classification results of $\mathcal{R}$ and $\mathcal{T}$ however could still be the same. Therefore, we require $\mathcal{R}$ to have a unique highest output activation. Since we are not allowed to use strict inequalities in MILP, we use an $\varepsilon > 0$ to ensure a unique greatest output activation. We then arrive at the final encoding of $top-k$-equivalence. First, we obtain $\mathcal{R}$'s unique top output $\hat{r}_1$:

$$\hat{r}_1 = \sum_i \rho_i r_i \tag{11.11}$$

$$\hat{r}_1 \geq r_i \ \forall i \leq n \tag{11.12}$$

$$\rho_i = 0 \rightarrow \hat{r}_1 \geq r_i + \varepsilon \ \forall i \leq n \tag{11.13}$$

$$\sum_i \rho_i = 1 \text{ and } \rho_i \in \{0,1\} \ \forall i \leq n \ , \tag{11.14}$$

where $\vec{\rho} = (\rho_1, ..., \rho_n)^T$ is used just as the first row of a permutation matrix. Then, we can solve for $\mathcal{T}$'s activation t_r for the component of $\mathcal{R}$'s largest output, by applying $\vec{\rho}$ to the output of $\mathcal{T}$.

$$t_r = \sum_i \rho_i t_i \tag{11.15}$$

The k greatest outputs of $\mathcal{T}$ are computed as follows:

$$\hat{t}_i = \sum_j \pi_{ij} t_j \ \forall i \leq k \qquad \hat{t}_i \geq \hat{t}_{i+1} \ \forall i \leq k-1$$

$$z_j = \sum_i \pi_{ij} \leq 1 \ \forall j \leq n \qquad z_j = 0 \rightarrow t_j \leq \hat{t}_k \ \forall j \leq n$$

$$\sum_j \pi_{ij} = 1 \ \forall i \leq k \qquad \pi_{ij} \in \{0,1\} \forall i \leq k, j \leq n \ ,$$

where the $(\pi_{ij})_{i,j=(1,1)}^{(k,n)}$ form the first k rows of the permutation matrix for $\mathcal{T}$'s outputs and z_i indicates, whether t_i is amongst $\mathcal{T}$'s k largest outputs. Finally, we can compute the violation of the $top-k$ equivalence property as the difference

$$d = \hat{t}_k - t_r \ , \tag{11.16}$$

which is then maximised to find the counterexample resulting in the largest possible violation of the equivalence property.

Interval Arithmetic. We assume that lower and upper bounds are given for the input variables and use existing interval extensions for the sum and multiplication to generate bounds on the linear combinations of inputs. The ReLU function is then applied to these bounds to generate bounds on the output of the neuron. This process is repeated throughout the network. Naively applying this kind of interval arithmetic to the equations defining $\hat{r}_1, t_r$ and the $\hat{t}_i$ respectively would produce large overestimates. In Equation 11.11 for example, the upper bound on $\hat{r}_1$ would be the sum instead of the maximum of the upper bounds of the r_i (only one entry is equal to 1 in a row of the permutation matrix). Therefore, we use context groups to compute tighter bounds for these variables. Assume, we are choosing a variable x from a set $X = \{x_1, ..., x_n\}$, where $x_i \in [l_i, u_i]$. Let $\hat{\vec{l}}$ and $\hat{\vec{u}}$ denote the vectors containing the lower, respectively upper bounds sorted in decreasing order.

If we choose x to be the k-th largest variable out of X, we can combine x in a *top-k-group* and assign tighter lower and upper bounds for x according to: $x \in [\hat{l}_k, \hat{u}_k]$.

ε-Equivalence. We encode ε-equivalence and exact equivalence as maximizing

$$d = \|\vec{r} - \vec{t}\| \ . \tag{11.17}$$

The equivalence property is satisfied, if $\max d \leq \varepsilon$ for ε-equivalence. The value of ε has to be chosen according to the dataset. The "optimal" value can be determined by incrementally looking at counterexamples for the equivalence and deciding if, from the user-perspective, the outputs are equivalent. For exact equivalence $\varepsilon = 0$ is required. In order to use Equation (11.17) in MILP, we need to encode the non-linear $\|\cdot\|$ operator. We restricted ourselves to the Manhattan $\|\cdot\|_1$ and the Chebyshev norm $\|\cdot\|_\infty$ defined as

$$\text{Manhattan:} \quad \|\vec{x}\|_1 = \sum_i |x_i|, \qquad \text{Chebyshev:} \quad \|\vec{x}\|_\infty = \max_i |x_i| \ , \tag{11.18}$$

because they are piecewise linear functions and can thus be encoded in MILP.

Just as we have done earlier, $y = |x|$ can also be expressed as cases $x \leq 0 \wedge y = -x$ and $x \geq 0 \wedge y = x$, that can be encoded as linear inequalities by introducing a binary variable. If the bounds $l_x \leq x \leq u_x$ indicate, that the domain of x contains only positive ($l_x \geq 0$) or only negative ($u_x \leq 0$) values, we can just set $y = x$ or $y = -x$, respectively.

In case of the Manhattan norm, we just sum over the absolute values of the components. The maximum operator used in the Chebyshev norm can be represented in the same way, as we have done to obtain the output with the highest activation for a NN in equations (11.11) - (11.14) in the previous section.

However, a unique largest value is not required in this case, so Equation (11.13) is not needed in this encoding.

11.3 Input Restriction

As mentioned in Section 10.2, exact equivalence can also be relaxed by restricting the input domain, for which the equivalence property has to hold. In practice, it is especially useful to restrict the input domain to values that are covered by the training dataset of the respective NNs. Differences in the output of NNs in the neighborhood of their training samples are far more meaningful than differences in regions, where they would not have been applied anyway. Furthermore, restricting the input values allows for the calculation of tighter bounds.

Below, we give a short overview of the hierarchical clustering approach of [64] we used for restricting the input space to regions within a radius around cluster-centers of training data. Subsequently, we show MILP encodings for proving equivalence of two NNs for the restricted input regions. We also show, how this process can be modified for maximizing the radius around a point, such that the violation of a chosen equivalence property is smaller than a specified threshold.

11.3.1 Hierarchical Clustering

The hierarchical clustering method of [64] starts by clustering a set of labelled data-points $\{(x_i, y_i)\}_{i=1}^{n}$, with k distinct labels into k clusters. If a cluster contains input points of different labels, the method is recursively applied to that cluster, until all clusters only contain inputs of a common label. Every cluster is then characterized by its cluster center and its radius, which denotes the maximum distance from the cluster center to its input points. The underlying assumption for this clustering is that all points, not just the training data-points, in a dense cluster should be assigned the same label. As points close to a cluster boundary might lie on a real decision boundary between two classes, [64] set the radius r_c of a cluster to the average distance of the input-points to the cluster center. Note, that the above assumption only holds for clusters of high density n/r_c, where n is the number of training data-points in the respective cluster.

11.3.2 Encoding of Clusters

As each cluster is characterized by its center $\vec{c}$ and radius r_c, one can place a norm restriction $\|\vec{i} - \vec{c}\| \leq r_c$ on the vector of inputs $\vec{i}$, to reduce the domain of the verification procedure to only inputs from that cluster. Thus, we can encode the input restriction by extending the encoding of NN equivalence given in Section 11.2 through adding the norm restriction to Equations (11.2)-(11.4).

Again, we restrict ourselves to encoding the Manhattan and Chebyshev norms (Equation 11.18). The encoding of the Chebyshev norm for input restriction is less complicated than its encoding needed for ε-equivalence, as we do not need to actually calculate the value of the norm, but just ensure, that all input values are within a set distance of the cluster center. Which leads to a box constraint on the input variables $\vec{i}$. Therefore the lower and upper bounds l_j and u_j of variable i_j can be updated to $l'_j = \max(c_j - r_c, l_j)$ and $u'_j = \min(c_j + r_c, u_j)$ respectively. The Manhattan norm $\|\cdot\|_1$ however has to be encoded just as in Section 11.2.1. Nonetheless, we can use the fact that $\|\vec{x}\|_1 \geq \|\vec{x}\|_\infty \ \forall \vec{x}$, to achieve faster tightening of the variable bounds by adding the bounds calculated in the encoding of the Chebyshev norm.

11.3.3 Searching for a Maximal Radius

In order to find the largest radius around a center $\vec{c}$ in input-space, where NNs $\mathcal{R}$ and $\mathcal{T}$ are equivalent, it is not possible to just use the equivalence encoding adding the presented norm and set r_c to be maximized. Since the solver finds an assignment to the input variables $\vec{i}$, such that the objective, in that case the radius, is maximized, the NNs are still equivalent and $\|\vec{i} - \vec{c}\| \leq r_c$. In that situation the solver could choose $\vec{i} = \vec{c}$. Therefore, the equivalence constraint would be met, if the NNs are equivalent on the center, and the maximum of the radius would be arbitrarily large. Hence, we search for the smallest radius r_v, for which a counterexample to the equivalence of $\mathcal{T}$ and $\mathcal{R}$ can be found. This optimization problem is similar to the one proposed in [145] for finding adversarial examples for a single NN close to training inputs, which they approximately solve using gradient based methods.

Our MILP formulation reads:

$$\begin{aligned} \min \quad & r_v && (11.19)\\ s.t.\ & \vec{r} = \mathrm{enc}_{\mathcal{R}}(\vec{i}) && (11.20)\\ & \vec{t} = \mathrm{enc}_{\mathcal{T}}(\vec{i}) && (11.21)\\ & f(\vec{r}, \vec{t}) \geq \varepsilon_v && (11.22)\\ & \|\vec{i} - \vec{c}\| \leq r_v\,. && (11.23) \end{aligned}$$

Note that we used a small threshold value of $\varepsilon_v > 0$ for the violation.

If r^* is the optimal solution for the above minimization problem, the two NNs are not equivalent for radii $r' \geq r*$, as the solver could generate a counterexample for r^*. But we cannot guarantee that the NNs are equivalent for $r' < r^*$ because of the use of the threshold value. For small values of ε_v, the NNs are likely to be equivalent for radii $r' \leq r^* - \varepsilon_r$ for small ε_r. This can then be verified using the methods for fixed radii described in Section 11.3.2. If verification tasks for fixed radii have been carried out beforehand, the largest (smallest) radius, for which the NNs were (not) equivalent can be used as a lower (upper) bound on r_v in the radius-minimization problem.

11.4 Application: Neural Network Compression

The huge number of parameters in modern NNs lead to large amounts of memory – AlexNet [101], for example, uses 200MB. Hence, it is desirable to reduce the number of parameters of a NN, without compromising its performance on the task it is designed to solve. Our approach for verifying equivalence properties of NNs in combination with the presented input restrictions could be used to verify the equivalence, or at least quantify the similarity, of the original NN and the smaller NN, which is the result of the reduction in parameters. This reduction is usually done by pruning unimportant weights - setting their value to zero - essentially removing insignificant connections between neurons. In the context of *magnitude based pruning*, weights of small absolute value are considered negligible [144]. The NN may be retrained after pruning, to correct for the missing connections [144]. During this step and the following iterations of pruning and retraining, the weights of the pruned connections are fixed at zero. Another way to reduce the number of parameters, applicable only to classification tasks, is to directly train a smaller NN on the outputs of a well performing large NN, which is called *student-teacher training* [6, 75].

11.5 Evaluation

The approach has been implemented into the tool MilpEquiv. The tool is written in Python 3 and is able to automatically generate MILP encodings together with input restrictions. Our program is able to read in NNs exported by Keras [33] and uses version 8.1.1 of Gurobi [69] to solve the generated instances.

As mentioned in the Introduction (Section 1.2), MILPEQUIV was mainly implemented during the master thesis of Philipp Kern [89]. We used this implementation to analyze the equivalence between compressed and original NNs, as well as between compressed NNs. The main findings of the following evaluation are:

- Our technique of encoding NNs and properties into MILP is successful in proving or disproving equivalence for structurally identical and structurally different networks.
- All three equivalence properties can be verified by our approach, where $top - 1$ equivalence is faster then ϵ and $top - k$ equivalence for MILP approaches.
- The hierarchical clustering approach leads to meaningful input restrictions on human understandable counterexamples.
- We were able to calculate maximal equivalent radii around cluster centers for pruning and student-teacher training. The radii are relatively stable up to 50% pruned weights and highly depend on the structure of student networks.
- The scalability for our approach is currently limited to the Optical Recognition of Handwritten Digits Dataset with NN consisting of under 100 neurons.

Neural Networks. Our original NN consists of an input layer, hidden layers of 32 and 16 ReLU units and an output layer of size 10 (denoted: 32-16-10). It was trained using the *Optical Recognition of Handwritten Digits Dataset* [46].

The dataset consists of 8x8 pixel labeled images of handwritten digits, giving us 64 input variables, whose values are in the closed interval $[0, 16]$, which can be used as naive bounds on the input variables. We implemented bounds tightening and interval arithmetic to increase scalability.

Reduced size NNs were obtained by pruning and retraining the original NN in 10% increments. Additionally, NNs with less ReLU units were learned using student-teacher training. All NNs were trained using the Keras machine learning framework [33]. The achieved accuracy values on the training and testing datasets are shown in Figure 11.1 for the pruned NNs, as well as for different structures of student NNs. After the training process, we removed the softmax activation function in the output layer of the NNs to allow for their encoding in MILP.

Experiments. Verification tasks for top-k-equivalence were conducted with and without input restricted around the cluster centers shown in Figure 11.2. These clusters were the five most dense clusters obtained by hierarchical clustering, when applied to the *Optical Recognition of Handwritten Digits* dataset using Manhattan distance. Each cluster contains between 66 and 91 training images.

Experiments were conducted for $k \in \{1, 2, 3\}$ without input restriction and for fixed radii, while the experiments for searching maximal equivalence radii were conducted for $k \in \{1, 2\}$. Due to space limitations, we present the experimental results for searching maximal radii. The tool and instructions how to produce the results can be found under `https://github.com/phK3/NNEquivalence`.

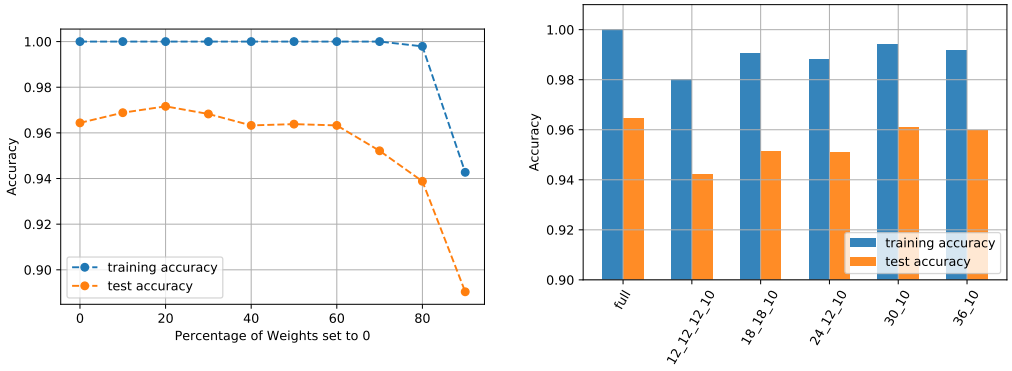
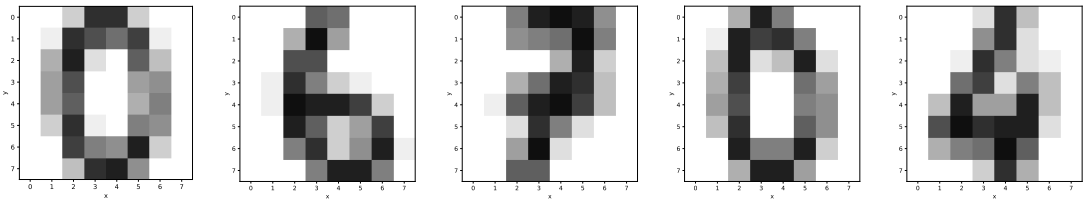


Figure 11.1: Accuracy values for original NN, low magnitude weight pruning NNs (left) NNs trained by student-teacher training (right).



cluster (a) cluster (b) cluster (c) cluster (d) cluster (e)

Figure 11.2: The cluster-centers of the five most dense clusters obtained from hierarchical clustering of the *Optical Recognition of Handwritten Digits* dataset.

Before the problem encoding was passed to Gurobi, bounds tightening was performed using interval arithmetic and optimization of two-layer subproblems for each linear combination of ReLU inputs. Each subproblem solution-process was stopped after a maximum of 20 seconds. All experiments were conducted on a computer with an Intel Core i5-3317U 1.70GHz processor, which has 2 physical and 4 logical cores, and 8GB of RAM running an x64 version of Windows 10.

11.5.1 Equivalent Neural Networks

We want to verify the equivalence of compressed NNs, by calculating the maximal equivalence radii for the chosen input clusters. Figure 11.3 shows the development of maximal radii for top-1 equivalence for both compression methods. The individual radius depends on training data and is reflected by the total number for the maximal radius for all reduction methods.

When pruning a larger percentage of parameters, the equivalence radius fluctuates around a constant level for each cluster up until the 50% reduced NN. If too many parameters were set to zero, the pruned NNs are no longer equivalent to the original NN and the equivalence radius deteriorates, as can be seen for the 80% and 90% pruned NNs.

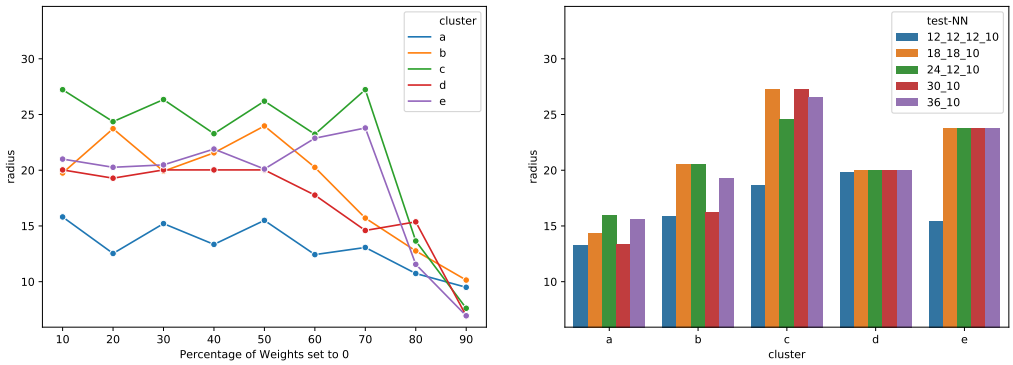


Figure 11.3: Maximal equivalence radii for top-1 equivalence to the full NN for weight pruned NNs (left) and student-teacher trained NNs (right)

For the 60% and 70% pruned NNs however, one notices, that the equivalence radii for clusters b, d drop as expected. Radii for the clusters a, c and e, on the other hand, either stay on the same level or even increase. An explanation for the observed behavior could be, that about 50% of the original NN's parameters are sufficient to capture the underlying knowledge in the data for the tested clusters. If more parameters are pruned, the reduced NNs focus on the obviously classifiable clusters to still achieve a low training error. When the NNs are pruned even further, their capacity is clearly too low.

Examining the student-teacher trained NNs, we notice, that the equivalence radii not only depend on the number of ReLU nodes, but also the structure of the NNs. While the 12-12-12-10 student has more ReLU units than the 30-10 student and the same number as all other student NNs, it exhibits sometimes significantly smaller equivalence radii on all clusters. Among the student NNs, it also exhibited the lowest accuracy on the training and testing datasets, indicating that 12 neurons per layer are not best suited for this classification task. The 18-18-10 student and the 36-10 student show however, that good accuracy and large equivalence radii can be obtained for this number of ReLU nodes. Comparing the different compression algorithms for top-1 equivalence, we notice, that most student NNs achieve similar radii as the up to 50% pruned NNs on clusters a, b, c and d and radii as large as that of the 70% pruned NN on cluster e. For the 12-12-12-10 student on clusters b and c and additionally the 30-10 student on cluster b, significantly smaller radii indicate a lack of capacity for the student NNs, although this effect is less severe than for the the 70%, respectively 80% pruned NNs.

Figure 11.4 represents the same data for the top-2 equivalence. The verification of top-2 equivalence is harder, thus our approach only returns upper (dotted lines) and lower (normal lines) bounds for the given timeout. In general, the maximal equivalence radii are, as expected, larger then for the top-1 equivalence. This indicates that the NNs still assign large probabilities to the correct classification result for the cluster regions. It is also possible to observe, that for example the 12-12-12-10 student network does not lag as far behind the other student NNs as before, indicating, that it at least captured a rough understanding of the data.

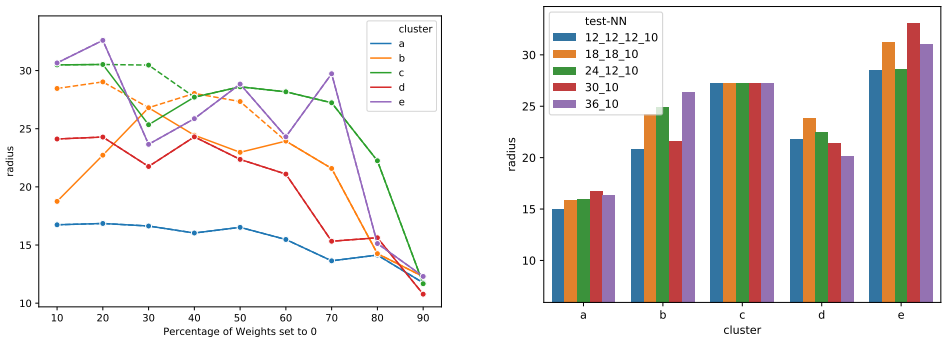
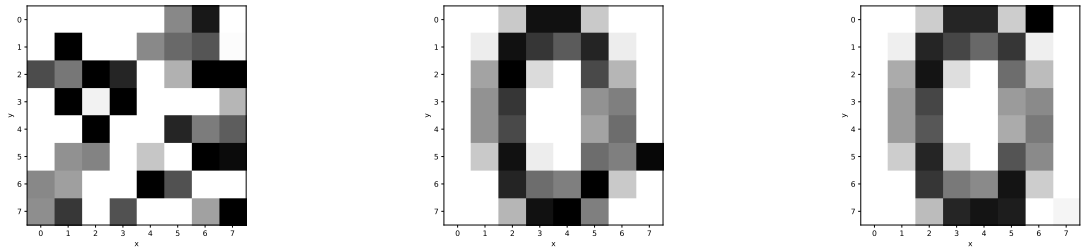


Figure 11.4: Maximal equivalence radii for top-2 equivalence to the full NN for weight pruned NNs (left) and student-teacher trained NNs (right)

11.5.2 Remarks

The runtime of the verification procedure depends on the complexity of the MILP encoding, where the number of integer variables seemed to have the largest effect. For our experiments runtime fluctuated between a minute and 30 minutes for top-1 equivalence. For top-2 equivalence, we set a timeout of 3 hours. Verification of equivalence for student-teacher trained NNs with fewer ReLU nodes was in most cases faster, than for the pruned NNs, as fewer integer variables had to be introduced. Only considering pruned NNs, sparser NNs proved to be verified faster than their less sparse counterparts. Overall, verification of equivalence for small fixed radii is faster, than for larger radii, as tighter bounds for all variables in the encoding can be obtained via bounds tightening. For very large radii, however, some NNs seem to be obviously not equivalent and large counterexamples are quickly found by the solver. In the extreme case without input restrictions, counterexamples to equivalence were all found within a minute.

The presented approach is able to search for a maximal radius for which NNs are equivalent and returns an input at the edge of the radius for which the networks are not. We denote this input as a counterexample, which can be analyzed by a potential user. He then has to decide, whether the counterexample should be classified as an valid input. If it is valid, the maximal radius is too small and the NNs are not equivalent, otherwise the NNs are equivalent w.r.t. the cluster. Three kinds of counterexample are shown in Figure 11.5. The leftmost picture shows an input picture for unrestricted input. This kind of counterexample is negligible in practice and should not be seen as valid input. It demonstrates the necessity for input restrictions when verifying NNs. The counterexample in the middle shows a picture of a (in our opinion) zero which is misclassified by the 20% pruned NN. Such a result could indicate that the pruned NN does not fit the wanted equivalence criterion. The left picture on the other hand, shows a digit that is misclassified by the original NN, which could indicate that the original NN should be retrained with the given counterexample.



full NN: 5
50% pruned NN: 6

full NN: 0
20% pruned NN: 6

full NN: 5
50% pruned NN: 0

Figure 11.5: Counterexamples to *top-1* equivalence: Without input restriction (left), input within a 13.6 (middle) and 27.2 (right) radius in Manhattan norm around the center of cluster (a).

11.6 Conclusion

We presented encodings of equivalence properties and neural networks with ReLU activation functions in MILP. Despite relaxed equivalence properties, NNs rarely meet equivalence over the whole input space, as their training only encourages them to agree on areas close to training data. Therefore we used the restriction of inputs to regions around clusters of training data, as proposed in [64]. We then developed MILP formulations, of equivalence for inputs within a fixed radius around obtained cluster-centers, as well as maximizing that radius, such that the NNs are still equivalent. Experiments with a NN trained on the *Optical Recognition of Handwritten Digits Dataset* [46] and its downsized counterparts obtained by student-teacher training or weight pruning showed the validity of our approach. As compression algorithms for NNs are typically only evaluated empirically by measuring the performance of the resultant NNs on a test dataset, this marks the first verification based examination of such methods. The notion of verified equivalence in a given cluster radius can be used to give guarantees for smaller networks. Furthermore, it can be utilized for finding meaningful counterexamples for the pruned and original network which can than be used for further training.

Our approach could also be applied, when numerous verification tasks have to be carried out for a large NN. In this case, a smaller NN could be obtained by compression algorithms. We could then prove its equivalence to the large NN within the input space of interest and subsequently perform the initial verification tasks on the smaller NN, requiring less computation time. For this scenario further improvements in scalability are needed. A first step could be using dedicated solvers for piecewise linear NNs like *Reluplex* [86] or the assistance of approximate methods [49].

Chapter 12

Adjusted Geometric Path Enumeration

12.1 Introduction

This chapter addresses the equivalence verification challenge for neural networks through a different method. Contrary to our earlier approach, we do not encode the neural network but extend on the geometric path enumeration (GPE) approach. We present contribution **C2.3**, which was previously published at [KB6].

GPE [150, 10] has previously been shown to yield good results for adversarial and functional verification on NNs. However, the algorithm was initially devised as an approach operating on a single NN. In this work, we extend GPE to a setting with multiple NNs and implement its extension for the problem of equivalence verification. We explore which (sometimes previously used) optimizations yield good results when applied to the equivalence problem. While our work in this chapter is specific to the problem of equivalence verification, the extended GPE algorithm can also be used for other verification tasks involving more then two NNs. We focus on the problem of equivalence verification for (potentially) structurally differing NNs.

Contribution C2.3. We extend the GPE algorithm (Tran et al.[150]) to a setting with multiple NNs and apply it to the equivalence verification problem. Afterwards, we evaluate several optimizations for this setting which increase efficiency on practical problems. Showing that our novel approach outperforms the state-of-the-art, we perform a comparative evaluation of our algorithm (on ACAS Xu and modified MNIST benchmarks). We name our approach **NNEquiv**.

Structure. We explain the idea behind our extension of GPE to multiple NNs in Section 12.2. Starting from a naive algorithm, we then evaluate optimizations to enable efficient equivalence verification using GPE in Section 12.3. We further explore these optimizations, in particular the question of good refinement heuristics, in Section 12.4.

In Section 12.5, we evaluate our algorithm and show advantages and disadvantages to the current state of the art represented by MILPEQUIV presented in previous Chapter 11. We give a final conclusion in Section 12.6.

12.2 Extending GPE to multiple Networks

The existing GPE approach was introduced as a foundation in Section 2.2.2. The general idea of GPE is to propagate *data structures* through a neural network. As possible data structures, we introduced *star sets* and *zonotopes*.

The most trivial approach to extend GPE to multiple NNs would be to *stitch* multiple NNs into a single composite NN and then execute regular GPE on this composite NN. However, the composite NN's weight matrices would be considerably larger which would increase the computational load. Furthermore, NNs with a different number of layers would have to be padded for this approach. This would nullify any performance gains which could otherwise be achieved through the reduced NN size. Instead, we propose to propagate star sets through both NNs sequentially. The approach and optimization methods correspond to techniques published in [KB6].

By carefully selecting the constraint sets of the propagated sets, we can ensure that there remains a point-wise correspondence between the output data structures of GPE for the two (or more) NNs considered. To make our approach clear, we introduce *transfer functions* as a way of reasoning about exact propagation of set data structures.

Definition 10 (Transfer Function)**.** *Let $\mathcal{N}$ be a NN. A transfer function $F_{\mathcal{N}}$ is a function which, given an input data structure $\Theta = \langle c, G, P \rangle$, produces a set of output data structures s.t.*

$$\forall \langle c', G', P' \rangle \in F_{\mathcal{N}}(\Theta) \ . \ \forall \alpha \in P' \ . \ g_{\mathcal{N}}(c + G\alpha) = c' + G'\alpha$$

and that the union of all P' within $F_{\mathcal{N}}(\Theta)$ equals P.

Using these transfer functions, we show that there is a correspondence between the output sets of two NNs in GPE:

Theorem 2 (NN Output Correspondence)**.** *Let $\mathcal{R}, \mathcal{T}$ be two NNs with their corresponding transfer functions $F_{\mathcal{R}}$, $F_{\mathcal{T}}$ and let $\Theta = \langle c, G, P \rangle$ be some input data structure. For any $\Theta_{\mathcal{R}} = \langle c_{\mathcal{R}}, G_{\mathcal{R}}, P_{\mathcal{R}} \rangle \in F_{\mathcal{R}}(\Theta)$ and $\Theta_{\mathcal{T}} = \langle c_{\mathcal{T}}, G_{\mathcal{T}}, P_{\mathcal{T}} \rangle \in F_{\mathcal{T}}(\langle c, G, P_{\mathcal{R}} \rangle)$:*

$$\{(g_{\mathcal{R}}(x), g_{\mathcal{T}}(x)) \mid x \in [\![\langle c, G, P_{\mathcal{T}} \rangle]\!]\} \quad = \quad \{(c_{\mathcal{R}} + G_{\mathcal{R}}\alpha, c_{\mathcal{T}} + G_{\mathcal{T}}\alpha) \mid \exists \alpha \in P_{\mathcal{T}}\} \ .$$

An over-approximation would produce additional, spurious points in the output of $T(\Theta)$ and may therefore produce spurious output tuples. In this case the right side of Theorem 2 becomes a superset. This in turn gives rise to the modified GPE algorithm outlined in Algorithm 4. We begin by feeding our input data structure $\langle c, G, P \rangle$ into the first NN. The propagation step function for the data structures (`step`) is the same as in the single NN GPE algorithm (see Foundations Section 2.2.2 for Algorithm).

For every output star set $\langle c_{\mathcal{R}}, G_{\mathcal{R}}, P_{\mathcal{R}} \rangle$, we restrict the input data structure according to the predicate of the output of the first NN, i.e. $\langle c, G, P_{\mathcal{R}} \rangle$. Then we feed this data structure into the second NN to obtain $\langle c_{\mathcal{T}}, G_{\mathcal{T}}, P_{\mathcal{T}} \rangle$. In the end, we can compare the two output tuples $\langle c_{\mathcal{R}}, G_{\mathcal{R}} \rangle$ and $\langle c_{\mathcal{T}}, G_{\mathcal{T}} \rangle$ constrained by the predicate $P_{\mathcal{T}}$.

Note that both considered output sets are therefore constrained by $P_{\mathcal{T}}$ (not $P_{\mathcal{R}}$). This is the essential insight, that allows our approach to produce point-wise correspondences between the outputs of the two NNs.

Algorithm 4 High-level path enumeration algorithm for equivalence checking. ^{LP} indicates the step uses LP solving.

Input: Input $\Theta = \langle c, G, P \rangle$, NNs $\langle \mathcal{R}, \mathcal{T} \rangle$
Output: Verification result (`equiv` or `nonequiv`)

```
s ← ⟨nn : R, layer : 0, neuron : None, Θ : Θ, Θ_R : ⊥, Θ_T : ⊥⟩
i ← Θ
W ← List() {Working list of set data structures}
W.put(s)
while ¬W.empty() do
  s ← W.pop()
  step_{s.nn}(s, W)^LP {Propagate s.Θ by one neuron}
  if s finished network R then
    s.Θ_R ← s.Θ {Store output from R for comparison}
    s.nn ← T
    s.Θ ← ⟨i.c, i.G, s.Θ.P⟩
    s.layer, s.neuron ← 0, None
  end if
  if s finished network T then
    s.Θ_R.P ← s.Θ.P {Output of R}
    s.Θ_T ← s.Θ {Output of T}
    if ¬is_equiv(s.Θ_R, s.Θ_T)^LP then
      return not equivalent
    end if
  else
    W.push(s)
  end if
end while
return equivalent
```

12.2.1 Equivalence on Set Data Structures

For our equivalence verification approach it is necessary to define an equivalence check `is_equiv` which verifies whether two set data structures $\Theta_{\mathcal{R}}, \Theta_{\mathcal{T}}$ satisfy ϵ-equivalence or top-1 equivalence.

First, we present how ϵ-equivalence with Chebyshev norm $\|\cdot\|_\infty$ can be proven for zonotopes. Afterwards, we show how Star Sets can be used to prove ϵ-equivalence and top-1 equivalence.

ϵ-Equivalence In order to prove ϵ-equivalence with the Chebyshev norm, we need to bound the maximum deviation between the two NN outputs by ϵ. That is, given the two output zonotopes $\Psi_\mathcal{R} = \langle c_\mathcal{R}, G_\mathcal{R}, P_\mathcal{T} \rangle$ and $\Psi_\mathcal{T} = \langle c_\mathcal{T}, G_\mathcal{T}, P_\mathcal{T} \rangle$ we want to find the maximal deviation:

$$\begin{aligned} &\max_{\alpha \in P_\mathcal{T}} \| (c_\mathcal{R} + G_\mathcal{T} \alpha) - (c_\mathcal{T} + G_\mathcal{T} \alpha) \|_\infty \\ =& \max_i \max_{\alpha \in P_\mathcal{T}} \left| (c_\mathcal{R} - c_\mathcal{T})_i + (G_\mathcal{R} - G_\mathcal{T})_i \, \alpha \right|. \end{aligned}$$

As can be seen by the reformulation above, we can find the maximal deviation over the output by solving optimization problems for each dimension of the *differential zonotope*

$$\partial\Psi = \langle (c_\mathcal{R} - c_\mathcal{T}), (G_\mathcal{R} - G_\mathcal{T}), P_\mathcal{T} \rangle).$$

Recalling that zonotopes can be optimized with a closed form solution, this enables a quick check for the adherence of the desired ϵ-equivalence property. However, since zonotopes only approximate the output set, one may need to fall back to the use of Star Sets if equivalence cannot be established using zonotopes. In this case, we can reuse the same formula from above to obtain a *differential star set* $\partial\Theta$ which is then optimized using LP solving.

Top-1 Equivalence For top-1 equivalence there are two possible approaches which both rely on propagated star sets. We can reuse the MILPEQUIV encoding, presented in Chapter 11 and employ a MILP solver. Alternatively, we can use a simplex (LP) solver. In the latter case we split up the output star set $\Theta_\mathcal{T}$:

For every output dimension $1 \leq j \leq O$ we generate a polytope P_j. Additional constraints $r_j \geq r_i \ \forall i \neq j$ ensure that output r_j is the maximum among the outputs of $\mathcal{R}$ in $\langle c_\mathcal{R}, G_\mathcal{R}, P_\mathcal{T} \cap P_j \rangle$. Note that the union of P_1 to P_O covers all of $P_\mathcal{T}$. We then examine the outputs of $\Theta_j = \langle c_\mathcal{T}, G_\mathcal{T}, P_\mathcal{T} \cap P_j \rangle$ for every $1 \leq j \leq O$. Since j is always the maximum of $\mathcal{R}$ for this part of the output space, we want to ensure that j is also always the maximum of $\mathcal{T}$. Therefore, we compute the maximal difference between output dimension j and the other dimensions in Θ_j. If all of these differences are below 0, we can guarantee top-1 equivalence. This procedure produces $\mathcal{O}(O)$ star sets and $\mathcal{O}\left(O^2\right)$ optimization operations in total.

12.2.2 Challenges and Limitations of the approach

While the techniques outlined above permit a straightforward extension of GPE to multiple NNs, and thus allow achieving equivalence verification, the approach comes with a number of pitfalls which should be avoided. The most obvious is probably the possibility of exponential growth in the number of star sets.

As previously noted, the exact GPE approach based on star sets splits the star sets on ReLU nodes. Tran et al. [150] rightly note that the observed growth usually drastically falls behind the worst case, however the increase in ReLU nodes through the processing of two NNs at once certainly leads to an increase in necessary splits. This is particularly the case for ReLU nodes which cut off very similar hyperplanes (such as the two ReLU nodes in a NN at the same position with truncated weights). This can not only double the work, but it may also lead to precision problems with LP solvers which tend to show problematic behavior when encountering a problem which has a very small feasible set[1]. To avoid such numerical problems we thus use 64-bit floats by default and always ensure that feasibility is checked at least once by an exact (i.e. rational) LP solver before a branch is declared infeasible. While this can mitigate most numerical problems, the approach is weaker than the approaches by Paulsen et al. ReluDiff [128] and NeuroDiff [129] for the specific use case of weight truncation for structurally similar NNs (e.g. truncation from 32-bit to 16-bit floats). Although these initial improvements help in making GPE for equivalence *possible*, this approach is not yet *scalable*. Hence, we devote the next section to various optimizations.

12.3 Optimizing GPE for two NNs

The approach presented above is not yet scalable. In particular, we identify two bottlenecks: The number of splits and the time taken for LP optimization. Therefore, we consider a number of optimizations, some of which have previously been applied by Bak [9].

12.3.1 Zonotope Propagation

As an initial optimization we reused the zonotope propagation technique presented by Bak et al. [10], which reduces the number of LP optimizations necessary through a zonotope based bounds computation. We refer to this first version of the algorithm as NNEquiv-E (for exact). As can be seen in Figure 12.3 later on, this approach produces a total runtime of 54,390s on our 9 benchmark instances.

12.3.2 Zonotope Over-Approximation

To further optimize the algorithm we can either reduce the time spent per zonotope or we can try to reduce the number of zonotopes which have to be considered. In order to achieve the second objective, we can over-approximate certain ReLU splits through a methodology first presented by Singh et al. [138] and later reused by Bak [9]: The idea is to introduce an additional dimension to the zonotope and use it to over-approximate the ReLU node by a parallelogram. Over-approximation errors accumulate across layers (Bak [9] refer to this as *error snowball*). To make the parallelogram as tight as possible and minimize the over-approximation error, we use the bounds computed through LP

[1] In one case the solver would return drastically differing maximum values for the same optimization problem depending on the previous requests or would suddenly deem the problem infeasible.

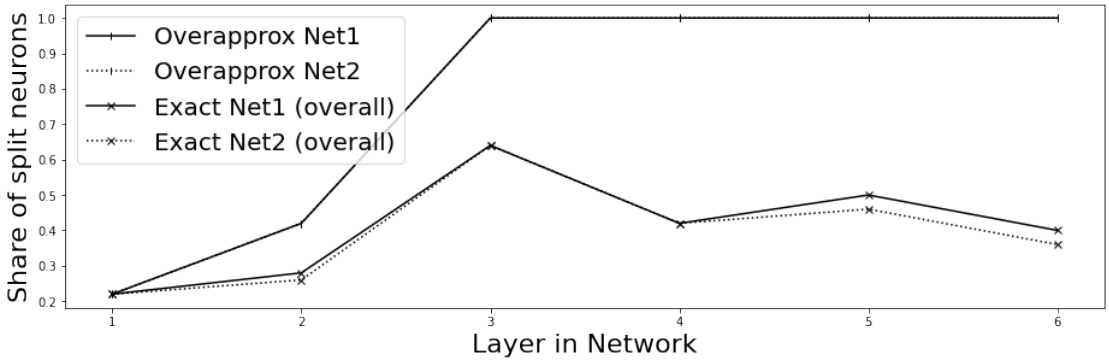


Figure 12.1: Comparison of exact star set propagation (Exact) and propagation of an over-approximating zonotope (Overapprox) for ϵ-equivalence over ACAS_1_1: Net1 corresponds to $\mathcal{R}$ and Net2 corresponds to $\mathcal{T}$ (Overapprox Net2 is hidden behind Overapprox Net1)

solving (instead of the looser zonotope bounds) if there were any exact splits beforehand. In an abstraction-refinement setting, we would start by propagating over-approximating zonotopes through both NNs and then check, whether the equivalence property can be established. If the property does not hold, we refine one over-approximated ReLU node by splitting the zonotope and propagating the split zonotopes further through the NNs.

In Figure 12.1 we compare the share of nodes per layer whose bounds contain the value 0 for an exact approach in comparison to the propagation of an over-approximating zonotope. Any such node can be considered a split candidate which could be used for refinement. Each refinement can then help in reducing the over-approximation error and in establishing the desired property. As can clearly be seen in the plot, the over-approximation approach produces a lot more split candidates than the exact approach. Not all of the splits candidates encountered for the over-approximation would actually have to be refined in the worst case. This is, because many of the split candidates are only artifacts of previous over-approximations. We refer to these split candidates as *ghost splits*. These *ghost splits* cannot be easily distinguished from actual, necessary splits. The only guaranteed non-ghost split, is the first split candidate encountered, while all later split candidates might be artifacts of over-approximation.

Thus, the simplest refinement strategy would be to refine only this node. We refer to this strategy as NNEquiv-F (First), and it reduces the runtime on our benchmark set to 2,489s (Figure 12.3). However, this approach still leaves room for improvement.

12.3.3 LP approximation

The zonotope approximation introduced in the last subsection over-approximated the ReLU node by a parallelogram. The constant offset between the lower and upper bound of the parallelogram introduced an additional dimension to our problem. Therefore, splitting hyperplanes is no longer dependent on the input variables *only*, but also depend on the dimensions introduced through the over-approximation.

This raises the question how to handle the additional dimension in the propagated star set: Since equally increasing the dimensionality of the LP problem leads to increased solver runtimes, we instead opted to over-approximate the LP problem. Classically, for an m dimensional zonotope with initial input dimensionality I we observe a hyperplane cut of the following form:

$$\sum_{i=1}^{I} g_i \alpha_i + \sum_{i=I+1}^{m} g_i \alpha_i \leq c \tag{12.1}$$

We can now over-approximate this inequality by computing $\mu = \min_\alpha \sum_{i=I+1}^{m} g_i \alpha_i$ through zonotope optimization and constraining the LP problem with the following inequality:

$$\sum_{i=1}^{I} g_i \alpha_i \leq c - \mu \tag{12.2}$$

Since any solution for Equation 12.1 implies a solution for Equation 12.2 the second inequality is an over-approximation and can be used to reduce the number of dimensions the LP solver has to handle despite the over-approximation of the zonotope.Note that we need to take this over-approximation into account for minimization/maximization tasks. Since the LP solver only optimizes the first I dimensions, we need to add the optimization result of the over-approximating zonotope for the remaining dimensions. We refer to this version as NNEquiv-A (for approximate LP). Figure 12.3 shows that this approach reduces the runtime to 1,631s.

12.4 The Branch Tree Exploration Problem

Given the introduced over-approximations over ReLU splits, it becomes necessary to define a strategy that decides which over-approximations are refined if it turns out that the property cannot be established with the current over-approximation. The problem of refinement heuristics has previously been studied for single NNs by Bak [9] who experimentally showed that a classic refinement loop approach which over-approximates everything and step by step refines over-approximations starting at the beginning of the NN (i.e. NNEquiv-F/A) sometimes performs worse than exact analysis. While we were able to reproduce this problem for some benchmark instances, we observed an improvement for others. It seems like a good approach to begin propagation with an exact strategy which splits on every encountered neuron, which, however, eventually transitions into over-approximation.

We proceed with a formal analysis on different strategies and their (dis)advantages. For this we consider binary trees that are implicitly explored by a GPE algorithm: For given NNs and input space $\mathcal{I}$, the implicit tree explored by GPE consists of vertices $V = N \uplus L$ where N are the inner nodes of the tree representing ReLU splits and L are the leafs of the tree representing the output set data structures. The execution of an exact GPE algorithm implicitly produces a set of paths of the form $p \in N^* \times L$ that are (for now) explored sequentially. We denote this set of paths as P.

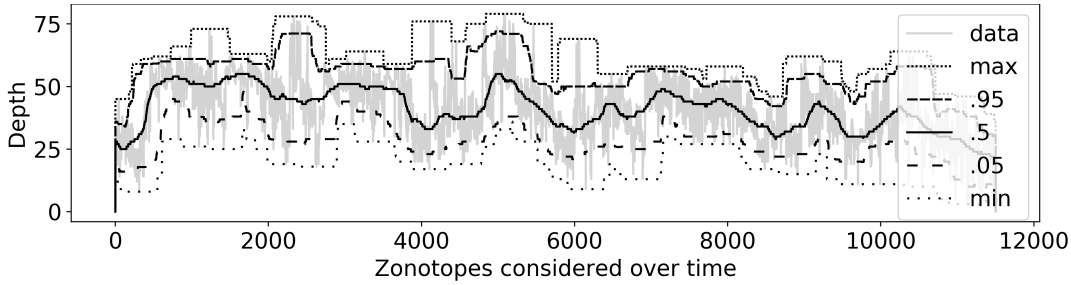


Figure 12.2: ACAS_1_1-retrain: Descend depth at point of successful equivalence proof for $\epsilon = 0.05$ (Running percentile window width: 479)

For the exact case, the number of explored paths is fixed to the number of leafs. Since GPE produces a partitioning of the input space $\mathcal{I}$, we can associate a part of the input space to every leaf and to every inner node.

For its execution GPE needs to `descend` into each leaf and execute a `check` function on each leaf to prove equivalence. `descend` refers to the operations necessary to process a star set up to its next ReLU split. `check` refers to the operations necessary to prove equivalence on an output star set. Since the `descend` function is executed once for each of the $2|P| - 2$ edges of the tree and the `check` function is executed once for each of the $|P|$ leaves, the execution time of NNEquiv-E is bounded by $\mathcal{O}(|P| * (t_{\texttt{check}} + 2t_{\texttt{descend}}))$.

Omitting the option of reordering the inner nodes and thus producing a smaller tree, we must either reduce $t_{\texttt{check}}$ and $t_{\texttt{descend}}$ or $|P|$ to reduce solving times. In many cases, the considered property $\mathcal{E}$ cannot only be proven on part of the input space associated to the leaf, but there also exists some inner node $n \in N$ with an associated part of the input space which is already sufficiently partitioned to show $\mathcal{E}$ using over-approximation. For a given equivalence property $\mathcal{E}$ we can define a function $\min_{\mathcal{E}} : N^* \times L \to \mathbb{N}$ which returns the number of necessary steps in the given path p for the property $\mathcal{E}$ to be verifiable on the input space part associated to element $\min_{\mathcal{E}}(p)$ of path p. The exploration of the tree induced by the set of paths

$$P^* = \left\{ p' \in N^* \cup N^* \times L \mid \exists p \in P : p' = p_{1:\min_{\mathcal{E}}(p)} \right\}$$

would then be sufficient to prove equivalence ($p_{1:\min_{\mathcal{E}}(p)}$ denotes the prefix path of p from step 1 to step $\min_{\mathcal{E}}(p)$).

NNEquiv-F/A manages to obtain this minimal number of paths – however at the cost of a much higher time spent on each path. In particular, `check` is not only executed for each leaf, but also for each inner node. Ignoring the over-approximation costs, this produces the following lower bound for the cost of NNEquiv-F/A:

$$\Omega\left(|P^*| * (2t_{\texttt{check}} + 2t_{\texttt{descend}})\right) .$$

Even when assuming the omitted over-approximation steps to be free, NNEquiv-F/A

becomes less effective than NNEquiv-E if asymptotically

$$|P| < |P^*| \frac{2\,(t_{\texttt{check}} + t_{descend})}{t_{\texttt{check}} + 2t_{\texttt{descend}}}\,,$$

i.e. if the reduction of paths in P^* is insignificant in comparison to the check time for the additional paths. While there are cases where NNEquiv-F/A is effective, this is not guaranteed to be the case – especially for larger NNs with higher values for $\min_{\mathcal{E}}$ (which increases $|P^*|$) and expensive check functions.

However, this formal framework allows us to define the (virtual) optimal run which takes the minimal amount of work for a given tree: An algorithm which has an oracle for $\min_{\mathcal{E}}$ and always over-approximates at the right node. This approach has a runtime of $\mathcal{O}\left(|P^*| * (t_{\texttt{check}} + 2t_{\texttt{descend}})\right)$.

Since $|P^*| \leq |P|$ and the omitted over-approximation time tends to be smaller than the `descend` time, this approach can provide the optimum achievable through heuristics for $\min_{\mathcal{E}}$. In fact, we simulated such virtual runs using a pre-computed oracle by computing $\min_{\mathcal{E}}$ using NNEquiv-A and descending only the minimum necessary number of steps for each path. In our evaluation we refer to this approach as NNEquiv-O. As expected, NNEquiv-O produced the best results of all variants considered in our work running only 635s on our benchmark set. This is not a practical algorithm, but provides a lower bound for the time taken using $\min_{\mathcal{E}}$ heuristics.

It is thus important to find a good heuristic which estimates $\min_{\mathcal{E}}$. These heuristics are much more difficult to analyze theoretically because they are particularly dependent on the distribution of the encountered paths. Therefore, we only explore two heuristics experimentally which show that heuristics have a significant impact on the runtime.

Figure 12.2 plots the depth at which GPE was successful in proving equivalence for a path in an ACAS Xu NN (i.e. the values of $\min_{\mathcal{E}}$). Besides the data in grey, we plotted a number of running percentiles over the depth values.

A strategy which we have found to be inefficient is the use of a running maximum over the number of refinements needed by previous paths. This strategy is referred to as NNEquiv-M (for maximum) and drastically increases runtime to 19,191s, presumably by over-estimating the number of refinements, thus increasing the number of paths considered.

Since Figure 12.2 suggest that there are *phases* in which the NN needs deeper or less deep refinement depths, we considered a heuristic which predicts a refinement depth equal to the depth of the previous path minus 1. This accounts for the possible phases of the depth and also ensures that the algorithm is *optimistic* in the sense that it always tries to reduce the number of refinement steps.This can then reduce the number of considered paths. We refer to this heuristic as NNEquiv-L which reduces runtime on the benchmark set by another 5% to 1,553s. While the methodology of over-approximation using Zonotopes is the same for NNEquiv-A and NNEquiv-O/L/M the approaches differ in the strategy deciding where the over-approximation is refined.

12.5 Experimental Evaluation

The GPE based equivalence verification technique was implemented using parts of a pre-existing (single NN) GPE implementation by Bak et al. [10] in Python. We will refer to our implementation as NNEquiv[2]. As mentioned in the Introduction (Section 1.2), NNEquiv was mainly implemented during the "research laboratory" (Praxis der Forschung) project in which the student Samuel Teuber was supervised by the author of this thesis.

Our evaluation aims at answering the following questions:

(E1) Do the proposed optimizations make the algorithm more efficient?

(E2) How does NNEquiv compare to previous work such as MilpEquiv [93] and ReluDiff [128]?[3]

(E3) How does the tightness of the ϵ-equivalence constraint influence solving behavior?

12.5.1 Experimental Setup

The benchmark landscape for the task of equivalence verification is still very limited. Paulsen et al. [128] proposed a number of benchmark NNs consisting of pairs of NNs differing only in the bit-width of the weights (32 bit vs. 16 bit). As discussed before, we see this as a restricted use case and are more interested in generic NNs with varying structures and weights. This is why we omit a comparison on these NNs where the approach by Paulsen et al. [129] is clearly faster and more precise. Structurally differing NNs have been previously proposed by Kleine Büning et al. [93] who examined 3 NNs of differing layer depths for digit classification on an MNIST [109] data set with reduced resolution [47] (8x8 pixels).

In order to evaluate and compare the approaches we thus proceeded as follows: First, we decided to look at two types of NNs: Image classification on the 8x8 pixel MNIST data set and NNs used in control systems in the context of an Airborne Collision Avoidance System (ACAS Xu [81]). Then, based on the original ACAS Xu NNs 1_1 and 1_2, we contructed a total of 4 mirror NNs through retraining (ACAS_1_1-retrain, ACAS_1_2-retrain) and student teacher training [74] for smaller NNs (ACAS_1_1-student, ACAS_1_2-student). In additon to the smallest and largest MNIST 8x8 NNs considered in previous work (MNIST_small-top, MNIST_medium-top), we constructed two larger MNIST models using student teacher training (MNIST_large-top, MNIST_larger-top). Moreover, we constructed a second version of MNIST_large-top for ϵ-equivalence verification (MNIST_large-epsilon). All NNs were trained using variants of student teacher training and were trained in such a way that they were likely to be top-1 or ϵ-equivalent in some parts of the input space. More details on the properties of the 9 considered benchmark NNs are available online.[4]

[2]The implementation is available on GitHub: `https://github.com/samysweb/nnequiv`

[3]Unfortunately, there is no artifact for NeuroDiff [129] which we could have evaluated.

[4]An overview table of all benchmarks is available at `https://github.com/samysweb/experiments/blob/main/benchmarks.md`

Name	Optimization	Described in Section
NNEquiv	no optimization	Section 12.2
NNEquiv-E (for Exact)	zonotope propagation	Section 12.3.1
NNEquiv-A (for LP Approximation)	LA approximation	Section 12.3.3
NNEquiv-O (for Oracle)	heuristic: optimal depth	Section 12.4
NNEquiv-M (for Maximum)	heuristic: maximal refinements	Section 12.4
NNEquiv-L (for Less)	heuristic: less refinements	Section 12.4

Table 12.1: List of configurations of our NNEquiv tool based on optimizations and heuristics.

The input space considered for verification is a sensitive choice, as it can have significant and varying impact on the performance of different verification techniques. For the case of GPE, the algorithm's performance tends to degrade with increasing input space size due to the growth in necessary splits. Therefore, for each individual benchmark, we decided to look at an input size which was hard to handle for NNEquiv-E. This has two reasons. First, it allows us to evaluate the impact of the optimizations presented above in their ability to decrease runtimes. Secondly, it permits to compare the performance of NNEquiv to the performance of MilpEquiv on instances which are difficult *for our* approach. The entire experimental setup can be found online.[5]

We used a machine with 4 AMD EPYC 7281 16-Core processors (i.e. 64 cores in total) and a total of 64GBs of RAM. All experiments were run with a single thread, a memory limit of 512MB[6], and a timeout of 3 hours. The experiments were run in parallel, up to 24 processes at once. All times given in the subsequent sections are the median of 3 runs.

12.5.2 Comparison of NNEquiv versions

In a first step, we evaluated the impact of the previously outlined optimizations on the runtime of our algorithm. Table 12.1 summarizes the optimizations and heuristics introduced in the previous sections for a better overview. Figure 12.3 shows that the proposed optimizations help in reducing the runtime of the algorithm (note that the upper half of the y-axis has a logarithmic scale for improved visibility of the results). On the one hand, we can observe, that heuristics for $\min_{\mathcal{E}}$ can, in principle, improve and worsen the result of the approach (as seen with NNEquiv-L and NNEquiv-M). On the other hand, we see that there is still significant room for improvement through the development of better refinement heuristics – this optimization would be supplementary to further optimizations which could be developed.

[5]On GitHub: `https://github.com/samysweb/nnequiv-experiments`

[6]The memory limit was irrelevant in practice, as no experiment hit this limit.

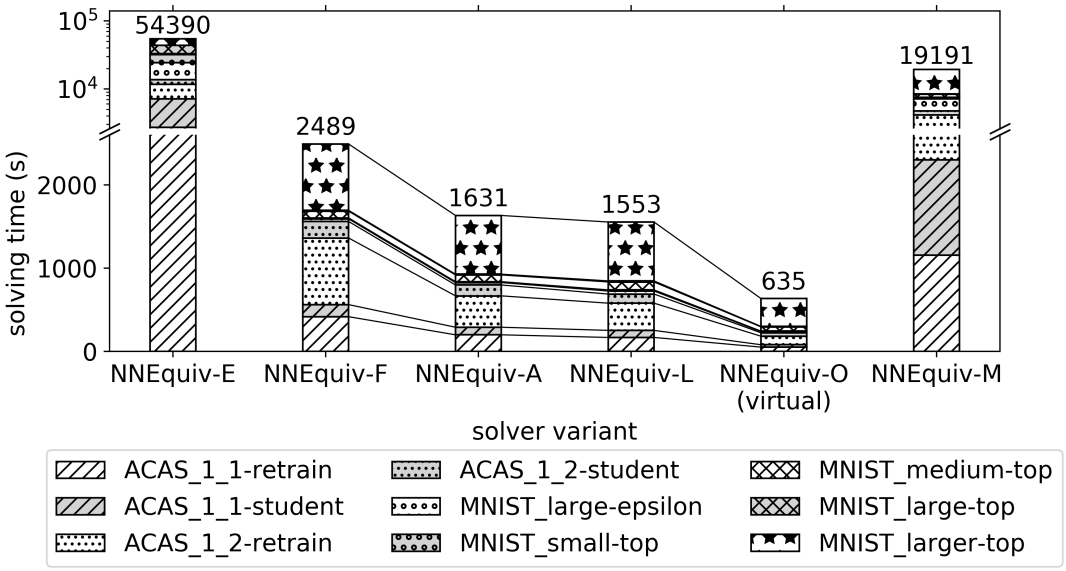


Figure 12.3: Time in seconds taken for our equivalent benchmarks per version. Note that the upper half of the y-axis has a logarithmic scale for improved visibility of the results.

12.5.3 Comparison to previous work

The comparison to MILPEQUIV is shown in Table 12.2 . NNEQUIV outperforms MILPEQUIV on the ACAS instances, where MILPEQUIV even runs into a timeout for three of the four verification tasks. In particular, this seems to be the case for larger NNs with low-dimensional inputs. The superior performance of MILPEQUIV for the case of MNIST_large-epsilon seems to be caused by the LP solver in NNEQUIV which is a magnitude slower for solving optimizations tasks for MNIST in comparison to ACAS Xu. As this cannot be explained by the number of constraints, we suspect this is a problem related to the larger input dimensionality for the MNIST case (64 inputs in comparison to 5 inputs for ACAS Xu). The ACAS-retrain NNs have the same structure as the original ACAS NN, allowing us to compare NNEQUIV to RELUDIFF. While RELUDIFF was able to quickly verify equivalence for the truncated NNs, where the mean absolute weight difference was $\approx 9.37 \cdot 10^{-5}$, it was significantly slower than our approach on the retrain instances, with a mean weight difference of ≈ 0.48, for $\epsilon \leq 5$ and even timed out for smaller values of ϵ. This suggests that the applicability of RELUDIFF is not only restricted to structurally similar NNs, but that its performance also heavily depends on small weight differences.

Regarding question (E2) we note that our approach is applicable to a broader class of NNs than RELUDIFF and solved instances where both other approaches timed out.

Benchmark	Property	NNEquiv-L	MilpEquiv
ACAS_1_1-retrain	$\epsilon = 0.05$	**167.45**	TO
ACAS_1_1-student	$\epsilon = 0.05$	**84.85**	TO
ACAS_1_2-retrain	$\epsilon = 0.05$	**326.59**	TO
ACAS_1_2-student	$\epsilon = 0.05$	**109.46**	320.07
MNIST_large-epsilon	$\epsilon = 15$	35.90	**19.97**
MNIST_small-top	top-1	14.39	**3.51**
MNIST_medium-top	top-1	94.51	**3.85**
MNIST_large-top	top-1	**13.02**	25.85
MNIST_larger-top	top-1	706.56	**386.04**

Table 12.2: Runtime comparison (in seconds) for NNEquiv-L and MilpEquiv

12.5.4 Influence of ϵ-equivalence tightness

Concerning question (E3) we evaluated the performance of the approaches as we vary the tightness of ϵ-equivalence for $\epsilon \in [0.005, 500]$. We varied ϵ between 0.005 and 500 on 4 benchmarks and observed the changes in running time for the two approaches. Note that we did not prove equivalence for $\epsilon = 0.005$ for ACAS_1_2-student as we found this NN not to be 0.005-equivalent to the original NN. Intuitively, a proof for a tighter ϵ bound will require more work as the approach either needs to refine more over-approximations (in the case of NNEquiv-L) or do further branch-and-bound operations (in the case of MILPEquiv). We can observe this behavior in Figure 12.4 which plots the runtime of MILPEquiv and NNEquiv-L as we tighten the ϵ bound. Taking into account the log scales on both axes, we can observe that NNEquiv-L is at least one magnitude faster in proving equivalence for ACAS Xu NNs for $\epsilon \leq 0.05$. In particular, MILPEquiv produces time-outs for 3 of the 4 considered NNs once $\epsilon \leq 0.05$. We therefore suspect that our approach is better at handling very tight ϵ constraints in large NNs with low dimensional input. This could potentially be due to the fact that GPE can use additional NN information (layer structure etc.) for its refinement decisions which is not readily available in the branch-and-bound algorithm in the backend of MILPEquiv.

For comparison, we plotted the performance of ReluDiff on NNs for truncated weights and retrained NNs. As can be seen in Figure 12.4 the approach by Paulsen et al. [128] behaves similarly with respect to ϵ tightness. However, the approach is less efficient for retrained NNs where the equivalence for $\epsilon \leq 0.05$ cannot be established. The differences between single neurons is relatively small for the networks with truncated weights *ACAS_1_1-trunc* and *ACAS_1_2-trunc*. ReluDiff propagates differences of corresponding neurons and is therefore optimized for such networks. This can be seen observing the two dotted lines with lower runtimes then our approaches. For networks *ACAS_1_1-retrain* and *ACAS_1_2-retrain* weights were retrained and even though the networks are structurally equivalent, the weights of corresponding neurons can have larger differences. For such networks our NNEquiv approach outperforms ReluDiff significantly and can prove equivalence for smaller ϵ.

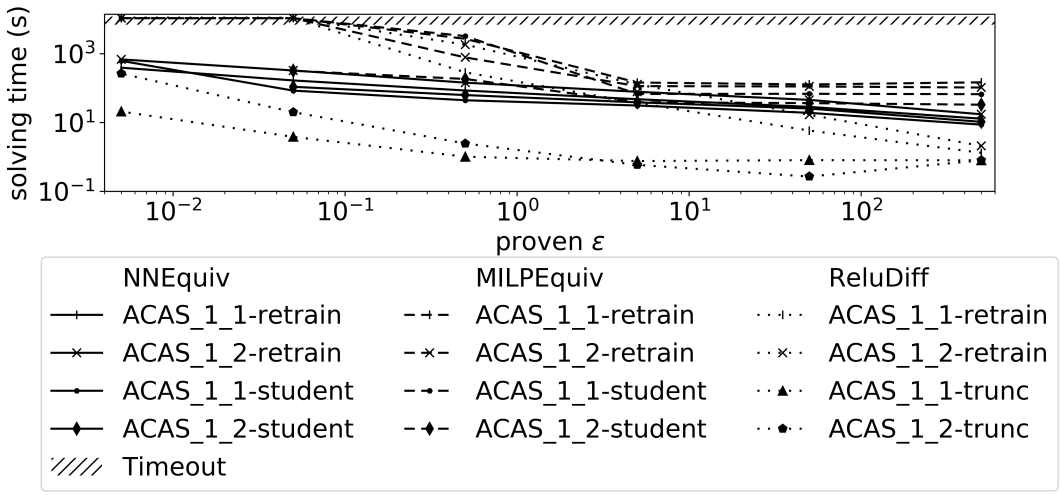


Figure 12.4: Solving time in seconds for ϵ-equivalence with varying ϵ: Solving times increase with tighter properties, however NNEquiv-L outperforms MILPEquiv. ReluDiff is outperformed by NNEquiv-L for retrained NNs.

12.5.5 Finding Counterexamples

Our technique can also be used to find counterexamples, showing that two NNs are not equivalent at a certain point. This information can be interesting to further train NNs after a failed equivalence proof. To this end, we compared the capabilities of NNEquiv-L in counterexample finding with the capabilities of MILPEquiv. To account for possibly easy instances, we looked at a large number of non-equivalent input spaces for each of our benchmark NNs which we know to be equivalent on other parts of the input space. To generate counterexamples, we randomly sampled input points and selected the points with differing NN outputs. Using this technique, we produced 100 distinct non-equivalent input points for each of our 9 benchmarks. These points were used as center for L_∞-balls which represented our input spaces.

We then evaluated NNEquiv-L and MILPEquiv on the same input space radii as in Subsection 12.5.2 with the objective of finding counterexamples. Since counterexample extraction is much faster than equivalence proofs, we set a timeout of 2 minutes. We found that NNEquiv-L was significantly faster and extracted a counterexample for 890 of the 900 considered benchmarks, while even a version without expensive ReLU bounds computation[7] executed in the initialization (named MILPEquiv-B) was only able to find 315 counterexamples. Also, the time per counterexample was significantly lower for NNEquiv-L, making this approach an interesting technique for retraining NNs via counterexamples. Looking at the behavior of MILPEquiv's solver backend, it seems that the reason for our superior performance lies in the time needed by MILPEquiv to find an initial feasible solution.

[7]While this optimization step improves performance for equivalence proofs, it may degrade performance for counterexample finding.

Table 12.3: Comparison of counterexample finding capabilities of NNEquiv-L, MilpEquiv and MilpEquiv-B (no ReLU node bounds)

	NNEquiv-L	MilpEquiv	MilpEquiv-B
#Solved	**890**	305	315
Time (incl.TO)	**3,597s**	75,989s	72,515s
Time/Solved (excl.TO)	**2.69s**	14.91s	7.21s

MilpEquiv first needs to resolve the integer based ReLU node encodings, which are automatically resolved by NNEquiv-L through the propagation of sets. NNEquiv-L has the potential to extract polytopes of non-equivalent input space subsets which could allow for even more efficient sampling.

12.6 Conclusion and Future Work

We proposed an approach extending Geometric Path Enumeration [150] to multiple NNs. Employing this method, we presented an equivalence verification algorithm which was optimized by four techniques: Zonotope propagation, zonotope over-approximation, LP approximation, and refinement heuristics. Our evaluation shows that the optimizations increase the approach's efficiency and that it can verify equivalence of NNs which were not verifiable by our previouse approach MilpEquiv or the tool ReluDiff [128]. Our approach significantly outperforms the state-of-the-art in counterexample finding by solving 890 instances in comparison to 315 instances solved by MilpEquiv. In addition, we presented a formal way of reasoning about refinement heuristics in the context of GPE.

In terms of efficiency, one could further explore possible refinement heuristics and consider parallelized (possibly GPU based) implementations. Moreover, while GPE can increase the confidence in NNs, the role of numerical stability for the verification approach has to be further investigated. Furthermore, an integration of MILP constraints into GPE propagation could be explored resulting in an algorithm inbetween NNEquiv and MilpEquiv. Additionally, we see a need for a larger body of equivalence benchmarks which allows the conclusive evaluation of equivalence verification algorithms.

Chapter 13

Related Work

The work of Pulina and Tacchella [132] is seen as the first significant research contribution in the field of formal verification of neural networks. The authors analyzed feed-forward neural networks known as Multi-Layer Perceptron (MLP) and verified specific bounds on the output given every possible input value. Research in the area of neural network verification has increased greatly since then. A survey paper by Narodytska et al. [126] in 2018 collected current research and divided it by verified properties. Under the term *invariance* they collected works verifying that given constraints on the input specific constraints hold for the output. The second property *invertibility* summarizes works that prove the inverse, given conditions on the output specific conditions hold on the input. As a third property they present the term *equivalence* of two neural networks.

We present related work, based on those three properties and different basic techniques. Most research focuses on checking invariance of neural networks and a variety of automated reasoning techniques have been introduced. Approaches encoding neural networks and properties into SMT [50, 76] or MILP [85, 87] improved upon the early work of Pulina and Tacchella [132].

In [86], Katz et al. extended a standard simplex algorithm for solving linear programs to support ReLU constraints. They introduced the Acas Xu networks and showed through their implementation Reluplex, that SMT based techniques are able to achieve sound and complete proofs for networks with multiple layers and hundreds of neurons.

The tool Marabou [87] by Katz et al. improved upon Reluplex by supporting different activation functions and inclusion of additional features like MILP-based bound tightening [148]. Similar to the software verification community, tools based on the abstract interpretation theory [122, 121, 137, 139, 149, 155] are popular and aim towards a scalable analysis. The tool ERAN [122], leverages abstract interpretation together with MILP constraints on input and output. The tool DeepPoly [139] presents a novel abstract domain consisting of floating-point polyhedral with intervals allowing for efficient affine transformations and a range of activation functions. Additionally, the Neural Network Verification Tool (NNV) [149] focuses on cyber-physical systems and utilize geometric represents such as star sets as a domain.

Furthermore, there exits approaches based on symbolic interval propagation (SIP) [152, 139, 53, 59, 72], which can be viewed as a specific form of abstract interpretation. The state-of-the-art tool VeriNet [72] for example utilizes SIP to create a linear abstraction of feed-forward neural networks. The abstraction is then embedded in an LP-encoding and through branch and bound algorithms solved to argue over network robustness.

The geometric path enumeration approach by Bak et al. [10] is implemented in the tool nnenum and applies multiple levels of abstractions. They introduce zonotopes as well as different star sets that are propagated through the network to argue over relations between input and output again encoded as LP programs.

All these approaches aim at verifying the invariance of neural networks and were able to push the state-of-the-art of neural network verification. Approaches proving invertibility [50, 99] are less common and are based on similar techniques.

The verification of the third property, namely the equivalence of neural networks, were scarcely researched. In their survey paper of 2018, Narodytska et al. found a single verification approach. In [126], binarized neural networks are investigated and, among other properties, the exact equivalence is defined and analyzed. Binarized Neural Networks are deep feed-forward neural networks, where integers can be encoded with binary values. This simplification allows for an efficient SAT encoding and significantly reduces the complexity.

Since 2018, there has been further work in the area of equivalence verification of neural networks. In addition to our work [KB2, KB6], Paulsen et. al. [128] published an approach called differential verification of neuronal networks implemented in their tool ReluDiff. They define differential verification similar to our epsilon equivalence. Their verification approach implements symbolic interval propagation and propagates value differences of corresponding neurons in different networks. Through gradient differences calculated in a backwards pass, they refine their analysis and are able to verify equivalence. Since they consider the differences between single neurons, they can only analyze structurally identical neural networks. Structurally different networks, such as those for example produced through student-teacher training, can only be analyzed if the smaller network is adapted to the larger one and additional layers and neurons are introduced so that the structure is the same and the behavior of the network does not change. However, for networks with such large differences, our evaluation shows that Paulsen's differential verification approach does not scale. However, for very similar networks, where the weight only changes by decimal places, their approach scales very well. In [129] Paulsen et al. optimize their approach by introducing new convex approximations and symbolic variables. Their new tool NeuroDiff improves scalability but still relies on value differences between corresponding neurons and therefore structurally similar networks. Additionally, they investigate equivalence for recurrent neural networks (RNN) in [118]. The recursion and nonlinear functions introduced by RNNs are solved through bounding nonlinear activation functions with linear constraints and calculation of tight bounds on non-linear surfaces.

Overall, there have been significant improvements for neural network verification. Relational verification and specially equivalence verification as presented in Chapters 10 is still a novel research field. The works of Paulsen et al. [128, 129, 118] come closest to our approaches but focus on structurally similar networks with minimal differences in weights. Our two approaches 11 and 12 advance the state-of-the-art and are, to the best of our knowledge, the only published research tuned towards equivalence verification for structurally different neural networks.

Chapter 14

Conclusion

14.1 Summary

In the third part of this thesis, we defined novel equivalence properties for neural network verification and presented two approach that verify equivalence for feed forward neural networks with ReLU activation functions.

Given two neural networks, equivalence denotes same behavior of networks represented by same input-output relations. Due to the stochastic nature of the training process of neural networks, two networks are seldom exactly equivalent. Therefore, we introduced the term ϵ-equivalence that was simultaneously denoted as differential verification by Paulsen et al. [128]. For classification tasks, such ϵ differences are often not relevant as long as classification results are the same. In *top-1* and *top-k* equivalence, we presented two properties matching human expectations of two equivalent classification networks. We have further proven that proving ϵ equivalence is coNP-complete for neural networks.

Encoding of networks was one of the first approaches to argue over network properties. We adapted this technique and encoded two neural networks and the three equivalence properties into MILP. Neural networks are only trained on problem related input regions. To model such input space, we applied hirachical clustering and additionally encoded an optimization problem finding the maximal equivalent radius around the cluster center into MILP. We then presented the two compression methods student-teacher training and weight-pruning demonstrating application areas of equivalence verification. Finally, our evaluation on the hand-written-digit-recognition dataset showed that our encoding was able to prove equivalence or produce never seen before counterexamples.

To improve scalability through abstractions, we then investigated the geometric path enumeration approach (GPE) that was previously applied to single network verification. GPE could not naively be applied to network equivalence and was therefore adjusted to sequentially push sets through multiple networks. The sequential effort was balanced through omitted splitting of sets for similar networks. Yet to be applied to equivalence, we needed further optimizations utilizing zonotopes abstractions and refinement strategies.

Concluding equivalence verification, we compared both our approaches on the ACAS and MNIST datasets showing merits for both approaches and an advantage over ReluDiff [128] approaches either for structurally different networks or networks with a marginally difference in weights.

Overall the third part of this thesis, introduces the novel research field of equivalence verification for neural networks. Next to giving new definitions and relaxations of the problem, we presented two approaches that are capable of successfully verifying equivalence and outperform the state-of-the-art.

14.2 Future Work

We presented the equivalence verification property and two approaches, which are able to verify equivalence for neural networks consisting of a few hundred neurons. We have described approach-based future work in the respective chapters. Here, we will highlight more general research challenges in the field of neural network verification and specially equivalence verification.

Currently, there is no distinct definition over which inputs neural networks have to be verified. This applies regardless of the property to be verified. Many approaches verify their property for ϵ areas around chosen training data points. The size of ϵ hugely influences the scalability and effusiveness of verification methods. Furthermore, single training data points may not reflect the distribution or special cases of the network application. We conducted first theoretical work applying variational autoencoders [91] to generate a lower-dimensional input representation making it easier to extract relevant input. The smaller dimension can also improve scalability but the additional autoencoder adds uncertainty and there is much (promising) work to do in this direction.

Connecting multiple networks and proving equivalence leads to interesting challenges and insights into the safety and functionality of NNs. Next to equivalence there are a number of other *relational* properties to be examined. In practice, retraining a neural network or adding additional layer should not lead to an equivalent but *better* neural networks that is at least as good as the previous network. For such and similar applications, for example, it would be interesting to define and verify functional sub- or supersets.

Overall neural networks is a promising research field with many theoretical and practical challenges. Many lessons learned through decades of software verification can be transferred and adopted for neural networks. Neural networks are more narrowly defined than traditional software, which could benefit verification. However, the stochastic nature of the learning process will be a major challenge for future work.

Bibliography

[1] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. "Analysis tool evaluation: Coverity prevent". In: *Pittsburgh, PA: Carnegie Mellon University* (2006), pp. 7–11.

[2] Bowen Alpern and Fred B Schneider. "Defining liveness". In: *Information processing letters* 21.4 (1985), pp. 181–185.

[3] Automotive Open System Architecture. *AUTOSAR*. 2021. URL: www.autosar.org.

[4] Motor Industry Software Reliability Association. *MISRA C*. 2018. URL: https://www.misra.org.uk/.

[5] Angello Astorga, P Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. "Learning stateful preconditions modulo a test generator". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2019, pp. 775–787.

[6] Jimmy Ba and Rich Caruana. "Do Deep Nets Really Need to be Deep?" In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 2654–2662. URL: http://papers.nips.cc/paper/5484-do-deep-nets-really-need-to-be-deep.pdf.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. arXiv: 1409.0473 [cs.CL].

[8] Rüdiger Bährle. "Market analysis of safety-critical software in the automotive industry". MA thesis. Karlsruhe Institue of Technologie, 2016. URL: https://baldur.iti.kit.edu/qpr/Marktanalyse-baehrle.pdf.

[9] Stanley Bak. "nnenum: Verification of ReLU Neural Networks with Optimized Abstraction Refinement". In: *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*. Ed. by Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez. Vol. 12673. Springer, 2021, pp. 19–36. DOI: 10.1007/978-3-030-76384-8_2.

[10] Stanley Bak, Hoang-Dung Tran, Kerianne Hobbs, and Taylor T Johnson. "Improved Geometric Path Enumeration for Verifying ReLU Neural Networks". In: *International Conference on Computer Aided Verification*. Springer. 2020, pp. 66–96.

[11] Tomas Balyo, Marijn JH Heule, and Matti Järvisalo, eds. *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. University of Helsinki, Department of Computer Science, 2017.

[12] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[13] Mordechai Ben-Ari. *Principles of the Spin model checker*. Springer Science & Business Media, 2008.

[14] Julien Bertrane et al. "Static analysis and verification of aerospace software by abstract interpretation". In: *AIAA Infotech@ Aerospace 2010*. 2010, p. 3385.

[15] Al Bessey et al. "A few billion lines of code later: using static analysis to find bugs in the real world". In: *Communications of the ACM* 53.2 (2010).

[16] Dirk Beyer. "Competition on software verification". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2012, pp. 504–524.

[17] Dirk Beyer. "Software verification with validation of results". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 331–349.

[18] Dirk Beyer. "Status report on software verification". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 373–388.

[19] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. *Bounded model checking*. 2003.

[20] Mariusz Bojarski et al. *End to End Learning for Self-Driving Cars*. 2016. arXiv: 1604.07316 [cs.CV].

[21] Bosch. *Bosch Sensortec Sensor Driver*. 2020. URL: https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi160.html.

[22] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. "IKOS: A framework for static analysis based on abstract interpretation". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2014, pp. 271–277.

[23] Guillaume Brat and Arnaud Venet. "Precise and scalable static program analysis of NASA flight software". In: *2005 IEEE Aerospace Conference*. IEEE. 2005, pp. 1–10.

[24] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. *A Unified View of Piecewise Linear Neural Network Verification*. 2017. arXiv: 1711.00455 [cs.AI].

[25] Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. "Footprint analysis: A shape analysis that discovers preconditions". In: *International Static Analysis Symposium*. Springer. 2007, pp. 402–418.

[26] Cristiano Calcagno et al. "Moving Fast with Software Verification". In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 978-3-319-17524-9.

[27] Satish Chandra, Stephen J Fink, and Manu Sridharan. "Snugglebug: a powerful approach to weakest preconditions". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 363–374.

[28] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. "Maximum Resilience of Artificial Neural Networks". In: *Lecture Notes in Computer Science* (2017), pp. 251–268. ISSN: 1611-3349. DOI: 10.1007/978-3-319-68167-2_18. URL: http://dx.doi.org/10.1007/978-3-319-68167-2_18.

[29] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. *A Survey of Model Compression and Acceleration for Deep Neural Networks*. 2020.

[30] Bharti Chimdyalwar, Priyanka Darke, Avriti Chauhan, Punit Shah, Shrawan Kumar, and R Venkatesh. "VeriAbs: Verification by abstraction". In: *TACAS*. Springer. 2017.

[31] Bharti Chimdyalwar, Priyanka Darke, Anooj Chavda, Sagar Vaghani, and Avriti Chauhan. "Eliminating Static Analysis False Positives Using Loop Abstraction and Bounded Model Checking". In: *FM 2015: Formal Methods*. 2015, pp. 573–576.

[32] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. "Blitz: Compositional bounded model checking for real-world programs". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 136–146.

[33] François Chollet et al. *Keras*. https://keras.io. 2015.

[34] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pp. 168–176.

[35] Edmund M Clarke and E Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on logic of programs*. Springer. 1981, pp. 52–71.

[36] Edmund M Clarke, David E Long, and Kenneth L McMillan. "Compositional model checking". In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. IEEE. 1989, pp. 353–362.

[37] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. "Learning assumptions for compositional verification". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2003, pp. 331–346.

[38] Patrick Cousot. “Abstract interpretation”. In: *ACM Computing Surveys (CSUR)* 28.2 (1996), pp. 324–328.

[39] Patrick Cousot and Radhia Cousot. “Abstract interpretation frameworks”. In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[40] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*. 1977, pp. 238–252.

[41] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. “Automatic inference of necessary preconditions”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2013, pp. 128–148.

[42] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. “Contract precondition inference from intermittent assertions on collections”. In: *VMCAI*. 2011, pp. 150–168.

[43] Patrick Cousot et al. “The ASTRÉE analyzer”. In: *Euro. Symp. on Prog.* Springer. 2005, pp. 21–30.

[44] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. “Frama-C”. In: *Intl. Conf. on Software Engineering and Formal Methods*. Springer. 2012, pp. 233–247.

[45] Alain Deutsch. “Static verification of dynamic properties”. In: *ACM SIGAda 2003 Conference*. 2003.

[46] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml` (visited on 01/07/2020).

[47] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits`.

[48] Zhenhua Duan, Cong Tian, and Li Zhang. “A decision procedure for propositional projection temporal logic with infinite models”. In: *Acta Informatica* 45.1 (2008), pp. 43–78.

[49] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A. Mann, and Pushmeet Kohli. “A Dual Approach to Scalable Verification of Deep Networks”. In: *UAI*. 2018.

[50] Ruediger Ehlers. “Formal verification of piece-wise linear feed-forward neural networks”. In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2017, pp. 269–286.

[51] E Allen Emerson and Joseph Y Halpern. “Decision procedures and expressiveness in the temporal logic of branching time”. In: *Journal of computer and system sciences* 30.1 (1985), pp. 1–24.

[52] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. "Dynamically discovering likely program invariants to support program evolution". In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[53] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. "Dl2: Training and querying neural networks with logic". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 1931–1941.

[54] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. "Extended static checking for Java". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 234–245.

[55] LLVM: A Compiler Framework. *LLVM Debug Information*. 2021. URL: `https://llvm.org/docs/SourceLevelDebugging.html`.

[56] Mikhail R Gadelha, Felipe Monteiro, Lucas Cordeiro, and Denis Nicole. "ESBMC v6. 0: Verifying C Programs Using k-Induction and Invariant Inference". In: *TACAS*. Springer. 2019, pp. 209–213.

[57] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. "An abstract domain of uninterpreted functions". In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer. 2016, pp. 85–103.

[58] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. "ICE: A robust framework for learning invariants". In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 69–87.

[59] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. "Ai2: Safety and robustness certification of neural networks with abstract interpretation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 3–18.

[60] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. "Assume-guarantee verification of source code with design-level assumptions". In: *Proceedings. 26th International Conference on Software Engineering*. IEEE. 2004, pp. 211–220.

[61] Ralph E Gomory. "An algorithm for integer solutions to linear programs. Princeton IBM Mathematics Research Project". In: *Techn. Report,(1)* (1958).

[62] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[63] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[64] Divya Gopinath, Guy Katz, Corina S. Pasareanu, and Clark Barrett. *DeepSafe: A Data-driven Approach for Checking Adversarial Robustness in Neural Networks*. 2017. arXiv: `1710.00486 [cs.NE]`.

[65] GrammaTech. *tatic Analysis Results: A Format and a Protocol: SARIF and SASP.* 2018. URL: https://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp.

[66] Orna Grumberg and David E Long. "Model checking and modular verification". In: *International Conference on Concurrency Theory*. Springer. 1991, pp. 250–265.

[67] Arie Gurfinkel, Temesghen Kahsai, and Jorge A Navas. "SeaHorn: A framework for verifying C programs (competition contribution)". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2015, pp. 447–450.

[68] Arie Gurfinkel and Jorge A Navas. "A context-sensitive memory model for verification of C/C++ programs". In: *International Static Analysis Symposium*. Springer. 2017, pp. 148–168.

[69] Gurobi Optimization LLC. *Gurobi*. URL: https://www.gurobi.com/ (visited on 12/02/2019).

[70] Ben Hardekopf and Calvin Lin. "Flow-sensitive pointer analysis for millions of lines of code". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2011, pp. 289–298.

[71] Chris Hawblitzel et al. "Ironclad apps: End-to-end security via automated full-system verification". In: *11th Symposium on Operating Systems Design and Implementation*. 2014, pp. 165–181.

[72] Patrick Henriksen and Alessio Lomuscio. "Efficient neural network verification via adaptive refinement and adversarial search". In: *ECAI 2020*. IOS Press, 2020, pp. 2513–2520.

[73] Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. "You assume, we guarantee: Methodology and case studies". In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 440–451.

[74] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *arXiv preprint arXiv:1503.02531* (2015).

[75] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network". In: *NIPS Deep Learning and Representation Learning Workshop*. 2015. URL: http://arxiv.org/abs/1503.02531.

[76] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. "Safety verification of deep neural networks". In: *International conference on computer aided verification*. Springer. 2017, pp. 3–29.

[77] Klocwork Inc. *K7 product documentation*. 2005.

[78] ISO. *C99 Standard (ISO/IEC 9899:1999)*. Geneva, Switzerland: International Organization for Standardization, Dec. 1999, 683 (est.)

[79] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/. Sept. 1998, p. 732. URL: http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%2D1998;%20http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%2FIEC+14882%3A1998;%20http://www.iso.ch/cate/d25845.html;%20https://webstore.ansi.org/.

[80] Jung Soon Jang and Darren Liccardo. "Automation of small UAVs using a low cost MEMS sensor and embedded computing platform". In: *2006 ieee/aiaa 25TH Digital Avionics Systems Conference*. IEEE. 2006, pp. 1–9.

[81] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. "Policy compression for aircraft collision avoidance systems". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE. 2016, pp. 1–10.

[82] Derek Justice and Alfred Hero. "A binary linear programming formulation of the graph edit distance". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.8 (2006), pp. 1200–1214.

[83] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong logic for weak memory: Reasoning about release-acquire consistency in Iris". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[84] Stephen Karungaru, Hitoshi Nakano, and Minoru Fukumi. "Road traffic signs recognition using genetic algorithms and neural networks". In: *International Journal of Machine Learning and Computing* 3.3 (2013), p. 313.

[85] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 97–117.

[86] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *Lecture Notes in Computer Science* (2017), pp. 97–117. ISSN: 1611-3349. DOI: 10.1007/978-3-319-63387-9_5. URL: http://dx.doi.org/10.1007/978-3-319-63387-9_5.

[87] Guy Katz et al. "The marabou framework for verification and analysis of deep neural networks". In: *International Conference on Computer Aided Verification*. Springer. 2019, pp. 443–452.

[88] Matthias Kern et al. "Integrating static code analysis toolchains". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE. 2019, pp. 523–528.

[89] Philipp Kern. *Verifying Equivalence Properties of Neural Networks with ReLU Activation Functions*. 2020.

[90] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. “Life, death, and the critical transition: Finding liveness bugs in systems code”. In: NSDI. 2007.

[91] Diederik P Kingma and Max Welling. “An introduction to variational autoencoders”. In: *arXiv preprint arXiv:1906.02691* (2019).

[92] Marko Kleine Büning, Tomáš Balyo, and Carsten Sinz. “Using DimSpec for Bounded and Unbounded Software Model Checking”. In: *International Conference on Formal Engineering Methods (ICFEM)*. Springer. 2019, pp. 19–35.

[93] Marko Kleine Büning, Philipp Kern, and Carsten Sinz. “Verifying Equivalence Properties of Neural Networks with ReLU Activation Functions”. In: *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, Proceedings*. 2020.

[94] Marko Kleine Büning, Philipp Kern, and Carsten Sinz. “Verifying equivalence properties of neural networks with relu activation functions”. In: *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer. 2020, pp. 868–884.

[95] Marko Kleine Büning, Johannes Meuer, and Carsten Sinz. *Refined Modularization for Bounded Model Checking by Precondition Generation*. Manusscript in preparation for submission. 2022.

[96] Marko Kleine Büning and Carsten Sinz. “Automatic modularization of large programs for bounded model checking”. In: *International Conference on Formal Engineering Methods (ICFEM)*. Springer. 2019, pp. 186–202.

[97] Marko Kleine Büning, Carsten Sinz, and David Faragó. “QPR Verify: A Static Analysis Tool for Embedded Software Based on Bounded Model Checking”. In: *Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, 2020, pp. 21–32.

[98] Phil Koopman. “A case study of Toyota unintended acceleration and software safety”. In: *Carnegie Mellon University Presentation. Sept* (2014).

[99] Svyatoslav Korneev, Nina Narodytska, Luca Pulina, Armando Tacchella, Nikolaj Bjorner, and Mooly Sagiv. “Constrained image generation using binarized neural networks with decision procedures”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2018, pp. 438–449.

[100] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[101] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: http://doi.acm.org/10.1145/3065386.

[102] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. “Loop summarization using state and transition invariants”. In: *Formal Methods in System Design* 42.3 (2013), pp. 221–261.

[103] Daniel Kroening and Michael Tautschnig. “CBMC–C bounded model checker”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 389–391.

[104] Abhinav Kumar, Thiago Serra, and Srikumar Ramalingam. *Equivalent and Approximate Transformations of Deep Neural Networks*. 2019. arXiv: 1905.11428 [cs.LG].

[105] Akash Lai and Shaz Qadeer. “A program transformation for faster goal-directed search”. In: *2014 Formal Methods in Computer-Aided Design*. IEEE. 2014, pp. 147–154.

[106] Ailsa H Land and Alison G Doig. “An automatic method for solving discrete programming problems”. In: *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 105–132.

[107] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[108] Gérard Le Lann. “An analysis of the Ariane 5 flight 501 failure-a system engineering perspective”. In: *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on* (1997), pp. 339–346.

[109] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[110] Jeffrey T Linderoth and Ted K Ralphs. “Noncommercial software for mixed-integer linear programming”. In: *Integer programming: theory and practice* 3 (2005), pp. 253–303.

[111] MathWorks. “Polyspace Code Prover”. In: *Matlab & Simulink* (2017).

[112] McKinsey and Inc. Company. *Cybersecurity in automotive*. 2020. URL: https://www.gsaglobal.org/wp-content/uploads/2020/03/Cybersecurity-in-automotive-Mastering-the-challenge.pdf.

[113] Florian Merz. “Theory and Implementation of Software Bounded Model Checking”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2016.

[114] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR”. In: *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE ’12)*. Springer, 2012, pp. 146–161. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_12.

[115] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded model checking of C and C++ programs using a compiler IR”. In: *International Conference on Verified Software: Tools, Theories, Experiments*. Springer. 2012, pp. 146–161.

[116] Bertrand Meyer. “Applying ’design by contract’”. In: *Computer* 25.10 (1992), pp. 40–51.

[117] Micro NAV autopilot software. http://sourceforge.net/projects/micronav/. Accessed: 2021-10-14.

[118] Sara Mohammadinejad, Brandon Paulsen, Jyotirmoy V Deshmukh, and Chao Wang. "DiffRNN: Differential Verification of Recurrent Neural Networks". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2021, pp. 117–134.

[119] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[120] Yannick Moy. "Sufficient preconditions for modular assertion checking". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2008, pp. 188–202.

[121] Christoph Müller, François Serre, Gagandeep Singh, Markus Püschel, and Martin Vechev. "Scaling polyhedral neural network verification on GPUs". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 733–746.

[122] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. "Prima: Precise and general neural network certification via multi-neuron convex relaxations". In: *arXiv preprint arXiv:2103.03638* (2021).

[123] Peter Müller. *Modular specification and verification of object-oriented programs*. Springer-Verlag, 2002.

[124] Peter Müller. "The Binomial Heap Verification Challenge in Viper". In: *Principled Software Development*. Springer, 2018, pp. 203–219.

[125] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying properties of binarized deep neural networks". In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[126] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying properties of binarized deep neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[127] Saswat Padhi, Rahul Sharma, and Todd Millstein. "Data-driven precondition inference with learned features". In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 42–56.

[128] Brandon Paulsen, Jingbo Wang, and Chao Wang. "ReluDiff: differential verification of deep neural networks". In: *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ACM, 2020, pp. 714–726. DOI: 10.1145/3377811.3380337.

[129] Brandon Paulsen, Jingbo Wang, Jiawei Wang, and Chao Wang. "NEURODIFF: Scalable Differential Verification of Neural Networks using Fine-Grained Approximation". In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia*. IEEE, 2020, pp. 784–796. DOI: 10.1145/3324884.3416560.

[130] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. “Heap assumptions on demand”. In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 314–327.

[131] Corneliu Popeea and Wei-Ngan Chin. “Dual analysis for proving safety and finding bugs”. In: *Science of Computer Programming* 78.4 (2013), pp. 390–411.

[132] Luca Pulina and Armando Tacchella. “An Abstraction-Refinement Approach to Verification of Artificial Neural Networks”. In: *Computer Aided Verification*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–257. ISBN: 978-3-642-14295-6.

[133] J Ross Quinlan. “Learning efficient classification procedures and their application to chess end games”. In: *Machine learning*. Springer, 1983, pp. 463–482.

[134] J. Ross Quinlan. “Induction of decision trees”. In: *Machine learning* 1.1 (1986), pp. 81–106.

[135] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. “Dynamic inference of likely data preconditions over predicates by tree learning”. In: *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 295–306.

[136] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: https://science.sciencemag.org/content/362/6419/1140.

[137] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. “Beyond the single neuron convex barrier for neural network certification”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[138] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. “Fast and effective robustness certification”. In: *Advances in Neural Information Processing Systems* 2018-Decem.Nips (2018), pp. 10802–10813. ISSN: 10495258.

[139] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. “An abstract domain for certifying neural networks”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[140] Carsten Sinz, Stephan Falke, and Florian Merz. “A precise memory model for low-level bounded model checking”. In: *Proceedings of the 5th international conference on Systems software verification*. USENIX Association. 2010, pp. 7–7.

[141] SQLite. http://sqlite.org. Accessed: 2021-10-14.

[142] International Organization for Standardization. *ISO 26262-1:2018, Road vehicles - Functional safety*. 2018. URL: https://www.iso.org/standard/68383.html.

[143] Miroslaw Staron. *Automotive Software Architectures - An Introduction*. Springer, 2017.

[144] Nikko Ström. “Phoneme probability estimation with dynamic sparsely connected artificial neural networks”. In: *The Free Speech Journal* 5 (1997), pp. 1–41.

[145] Christian Szegedy et al. *Intriguing properties of neural networks*. 2013. arXiv: 1312.6199 [cs.CV].

[146] Samuel Teuber, Marko Kleine Büning, and Carsten Sinz. “Geometric Path Enumeration for Equivalence Verification of Neural Networks”. In: *International Conference on Tools with Artificial Intelligence (ICTAI)*. 2021.

[147] Vincent Tjeng, Kai Xiao, and Russ Tedrake. *Evaluating Robustness of Neural Networks with Mixed Integer Programming*. 2017. arXiv: 1711.07356 [cs.LG].

[148] Vincent Tjeng, Kai Xiao, and Russ Tedrake. “Evaluating robustness of neural networks with mixed integer programming”. In: *arXiv preprint arXiv:1711.07356* (2017).

[149] Hoang-Dung Tran et al. “NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems”. In: *International Conference on Computer Aided Verification*. Springer. 2020, pp. 3–17.

[150] Hoang-Dung Tran et al. “Star-based reachability analysis of deep neural networks”. In: *International Symposium on Formal Methods*. Springer. 2019, pp. 670–686.

[151] Dolores R Wallace and Roger U Fujii. “Software verification and validation: an overview”. In: *Ieee Software* 6.3 (1989), pp. 10–17.

[152] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. “Formal security analysis of neural networks using symbolic intervals”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1599–1614.

[153] J Christopher Westland. “The cost of errors in software development: evidence from industry”. In: *Journal of Systems and Software* 62.1 (2002), pp. 1–9.

[154] Klaus Wissing. “Static Analysis of Dynamic Properties – Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors”. In: *GI Jahrestagung* (2007).

[155] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. “Output reachable set estimation and verification for multilayer neural networks”. In: *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5777–5783.

Own Publications

[KB1] Marko Kleine Büning, Tomáš Balyo, and Carsten Sinz. "Using DimSpec for Bounded and Unbounded Software Model Checking". In: *International Conference on Formal Engineering Methods (ICFEM)*. Springer. 2019, pp. 19–35.

[KB2] Marko Kleine Büning, Philipp Kern, and Carsten Sinz. "Verifying equivalence properties of neural networks with relu activation functions". In: *International Conference on Principles and Practice of Constraint Programming (CP)*. Springer. 2020, pp. 868–884.

[KB3] Marko Kleine Büning, Johannes Meuer, and Carsten Sinz. *Refined Modularization for Bounded Model Checking by Precondition Generation*. Manusscript in preparation for submission. 2022.

[KB4] Marko Kleine Büning and Carsten Sinz. "Automatic modularization of large programs for bounded model checking". In: *International Conference on Formal Engineering Methods (ICFEM)*. Springer. 2019, pp. 186–202.

[KB5] Marko Kleine Büning, Carsten Sinz, and David Faragó. "QPR Verify: A Static Analysis Tool for Embedded Software Based on Bounded Model Checking". In: *Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer, 2020, pp. 21–32.

[KB6] Samuel Teuber, Marko Kleine Büning, and Carsten Sinz. "Geometric Path Enumeration for Equivalence Verification of Neural Networks". In: *International Conference on Tools with Artificial Intelligence (ICTAI)*. 2021.

Chapter 15

Relevant Implementations

We give an overview of tools implemented or adjusted in the course of this dissertation.

QPR-Verify: The modular bounded model checking approach presented in Chapters 4 to 9 is implemented into the tool `QPR Verify`. The initial design and implementation of main parts of `QPR Verify` has been done in the research group "Verification meets Algorithm Engineering" at KIT and in the startup QPR Technologies, with the author as a co-founder. Next to the author, main parts of `QPR Verify` were implemented by David Farago, Felix Kutzner, Robin Freyler, Florian Merz and Carsten Sinz. The author additionally implemented the modularization and refinement techniques.

`QPR Verify` is not openly available but a running version as well as instruction to run the tool are available in a prepared VM. Instruction can be found at: `https://github.com/MarkoKleineBuening/DissertationTools`.

LLBMC: The bounded model checker LLBMC was originally developed at the Karlsruhe Institute of Technology by Merz et al. [115]. It is an implementation of the BMC approach utilizing the LLVM IR as an input language for it's verification procedure. LLBMC had to be adjusted to meet requirements imposed by `QPR Verify` and the modular bounded model checking approach. The author implemented and adjusted among other things data structures, formula abstractions and precondition generation into LLBMC.

LLBMC is not openly available but a running version is available in a prepared VM: `https://github.com/MarkoKleineBuening/DissertationTools.git`

precondition-learner: The tool PRECONDITION-LEARNER can generate generalized preconditions based on enumerative data points produced by LLBMC. The tree-based learning approach presented in Chapter 6 was implemented during the "research laboratory" (Praxis der Forschung) by Johannes Meuer under the supervision of the author.

The tool PRECONDITION-LEARNER is not openly available but a running version as well as instructions to start and navigate the `https://github.com/MarkoKleineBuening/DissertationTools.git`

QPR-Report: The tool QPR-Report is an interactive graphical report consisting of general source code information, chosen configuration options during verification, check results and an interactive representation of error traces on the C-code level. It is mentioned in Section 4.4.4. The client was mainly implemented by the student assistent Calvin Urankar.

QPR-Report is not openly available but a running version as well as instruction to run the tool are available in a prepared VM: `https://github.com/MarkoKleineBuening/DissertationTools.git`

MilpEquiv: The tool MilpEquiv verifies the equivalence between neural networks through MILP encodings as presented in Chapter 11. It was mainly implemented during the master thesis of Philipp Kern [89].

The implementation of MilpEquiv is avalable on GitHub at: `https://github.com/phK3/NNEquivalence`.

NNEquiv: The tool NNEquiv implements the adjusted GPE approach for equivalence verification of neural networks presented in Chapter 12. It was mainly implemented during a "research laboratory" (Praxis der Forschung) project in which the student Samuel Teuber was supervised by the author.

The implementation of NNEquiv is available on GitHub at: `https://github.com/samysweb/nnequiv`.