MANCHESTER 1824
The University of Manchester

Interval Analysis and Methods in Software Analysis

A thesis submitted to the University of Manchester for the degree of
Doctor of Philosophy
in the Faculty of Science and Engineering

2024

Mohannad A. Aldughaim
School of Sciences and Engineering
Department of Computer Science

Contents

Word count: 27766

List of figures

List of tables

List of publications

Publications Directly Contributing to the Thesis

Published:

- [1] M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro, "Fusebmc_ia: Interval analysis and methods for test case generation," in *Fundamental Approaches to Software Engineering*, L. Lambers and S. Uchitel, Eds., Cham: Springer Nature Switzerland, 2023, pp. 324–329, ISBN: 978-3-031-30826-0

- [2] R. S. Menezes, M. Aldughaim, B. Farias, *et al.*, "Esbmc v7.4: Harnessing the power of intervals," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2024, pp. 376–380, ISBN: 978-3-031-57256-2

Submitted:

- [3] R. S. Menezes, E. Manino, F. Shmarov, M. Aldughaim, R. de Freitas, and L. C. Cordeiro, *Interval analysis in industrial-scale bmc software verifiers: A case study*, 2024. arXiv: 2406.15281 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2406.15281

- [4] J. Deveza, M. Aldughaim, R. S. Menezes, L. Cordeiro, and R. de Freitas, *Reducing the bmc formal verification state-space by a cp preprocessing with variable-domain interval contraction*, 2024

In-Progress:

- [5] M. Aldughaim, K. Alshmrany, and L. Cordeiro, "Working title: Interval analysis and methods in testcase generation," 2024

- [6] M. Aldughaim, K. Alshmrany, R. Menezes, L. Cordeiro, and A. Stancu, "Incremental symbolic bounded model checking of software using interval methods via contractors," *arXiv preprint arXiv:2012.11245*, 2020

Related Publications

- [2] R. S. Menezes, M. Aldughaim, B. Farias, *et al.*, "Esbmc v7.4: Harnessing the power of intervals," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2024, pp. 376–380, ISBN: 978-3-031-57256-2

- [7] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs," in *International Conference On Tests And Proofs*, Springer, 2021, pp. 85–105

- [8] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc v4: Smart seed generation for hybrid fuzzing:(competition contribution)," in *International Conference On Fundamental Approaches To Software Engineering*, Springer International Publishing Cham, 2022, pp. 336–340

- [9] K. Alshmrany, M. Aldughaim, A. Bhayat, and L. Cordeiro, "Fusebmc v4: Improving code coverage with smart seeds via bmc, fuzzing and static analysis," *Form. Asp. Comput.*, vol. 36, no. 2, Jun. 2024, ISSN: 0934-5043. DOI: 10.1145/3665337. [Online]. Available: https://doi.org/10.1145/3665337

Terms and abbreviations

BMC	Bounded Model Checking
AI	Abstract Interpretation
CSP	Constraint Satisfaction Problem
ESBMC	Efficient SMT-Based Model Checker
FuSeBMC	Fuzzing with Symbolic Execution Based Model Checking
SMT	Satisfiability Modulo Theories
CSP	Constraint Satisfaction Problem
CP	Constraint Programming
TEST-COMP	Test Competition (context-specific)
SV-COMP	Software Verification Competition
IBEX	Interval-Based EXplorer (library)
PDR	Property-Directed Reachability
Z3	Z3 Theorem Prover (by Microsoft Research)
GOTO	GOTO Program (used in ESBMC context)
AST	Abstract Syntax Tree
CBMC	C Bounded Model Checker
SSA	Static Single Assignment
IR	Intermediate Representation
CFG	Control Flow Graph

Abstract

This thesis investigates the application of interval analysis and methods within the domain of software verification, with a particular focus on mitigating the state space explosion problem. State space explosion poses a significant challenge to static and dynamic software verification techniques, such as fuzzing, bounded model checking (BMC), and abstract interpretation. These methods, despite their robustness, struggle to scale when faced with complex programs that generate a vast number of execution paths and states. To address this, the thesis introduces the use of contractors—interval methods that refine the search space by eliminating non-solution regions—across several verification frameworks. By applying interval contractors in fuzzing, BMC, and abstract interpretation, the search space is systematically reduced without compromising the soundness or completeness of the verification process. Contractors are employed to navigate guard conditions, narrow down variable domains, and simplify control structures, leading to a more efficient exploration of execution paths. The thesis presents a detailed implementation of these methods and evaluates their performance through rigorous benchmarking. Results demonstrate that the integration of contractors significantly enhances verification efficiency, reducing both computational resource consumption and time while preserving accuracy in identifying potential software vulnerabilities. This research offers a novel contribution to improving the scalability of software verification methods, making them more practical for real-world applications.

Declaration of originality

I hereby confirm that no portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright statement

i The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made *only* in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in The University's policy on Presentation of Theses.

.

Chapter 1

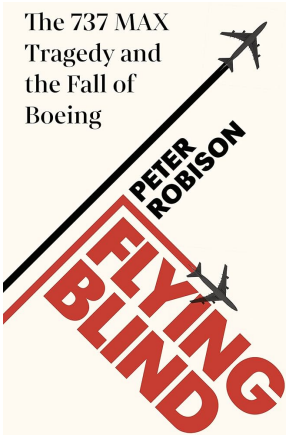
Introduction

The concept of testing encompasses a wide range of activities aimed at evaluating the functionality, reliability, and performance of various systems, from ancient civil engineering projects to modern digital innovations [10]–[12]. Throughout history, testing has been a crucial component of development and innovation, bridging theoretical design and practical application. In fields such as manufacturing, construction, and education, testing methodologies have evolved to ensure that products and structures meet specific standards and requirements. This evolution emphasises the importance of identifying and correcting issues before they lead to failure or underperformance. It calls for a more organised approach to quality assurance [13].

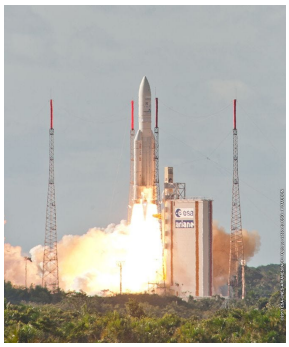
A prominent example of the implications of insufficient testing is the 1940 collapse of the Tacoma Narrows Bridge [14]. The bridge's collapse, resulting from an inadequate grasp of aerodynamic forces, prompted an examination of engineering methodologies and testing standards in civil construction. This catastrophe emphasised the imperative for rigorous testing frameworks, propelling progress in structural analysis and quality assurance to avert analogous failures in forthcoming infrastructure endeavours. Insights gained from these instances persist in shaping contemporary testing methodologies, guaranteeing that systems undergo rigorous examination prior to implementation.

As rigorous testing is necessary in engineering in order to avoid catastrophic failures, it is similarly imperative in software development. Testing is an essential component of the software lifecycle, impacting a wide range of applications, including online financial systems and toys for kids. It assures that software meets its specified requirements and operates effectively and securely. Neglecting its significance may result in substantial losses or endanger individuals' welfare. Testing activities include ensuring the software's correctness, safety, reliability, and security [15]. Correctness guarantees that the program functions as in-

(a) Therac-25



(b) Boeing 737 MAX



(c) Ariane 5 Flight 501

Fig. 1.1. Examples of software malfunction resulting in disastrous consequences.

tended in all scenarios, including the proper management of issues and unexpected situations. Safety is paramount in important systems, such as medical devices or automobile controls, where human lives may be endangered [16]. Reliability emphasises the software's capacity to function under defined settings throughout time without failure, whereas security verification seeks to safeguard against cyber threats by discovering and addressing potential vulnerabilities [17]. The importance of software testing is made clear by the serious consequences of software bugs, which can result in financial losses and pose significant safety and security risks, emphasising the necessity for rigorous verification and testing procedures throughout the software development lifecycle [18].

The impact of software failures, particularly in systems where human lives are at risk, cannot be overstated. A notable case is the Therac-25, a medical radiation therapy device illustrated in Figure 1.1a, which caused multiple fatalities due to software malfunctions in the mid-1980s, as reported by Leveson [19]. Similarly, the Boeing 737 MAX crashes, shown in Figure 1.1b, resulted in the tragic loss of 346 lives and highlighted the grave dangers of software failures in complex, safety-critical environments [20]. Another significant example is the Ariane 5 Flight 501 incident on June 4, 1996, illustrated in Figure 1.1c, where a software error caused the rocket to deviate from its intended trajectory and disintegrate shortly after launch [21]. These incidents underscore the vital importance of rigorous software verification and testing to protect human lives in advanced technological systems. We must learn from these tragedies and take decisive action to prevent their recurrence.

To avoid incidents, various methodologies have been progressively developed and refined in software development to enhance the effectiveness of quality assurance practices. These

methodologies are broadly classified into manual and automated techniques, each serving distinct purposes throughout the software development life cycle. Manual testing methods, such as code reviews and pair programming, remain prevalent in examining repositories and code bases on platforms such as GitHub [22]. In contrast, automated techniques have seen significant advancements, introducing tools that use various methods for more comprehensive testing [23].

1.1 Software Verification

The practice of *verification* is an integral part of the automated software testing umbrella. Verification evaluates whether the software meets its specified requirements and ensures the product is built correctly. Verification techniques are generally categorised into two main types: static and dynamic verification [24]. On the one hand, static verification involves analysing the code without executing it and detecting potential errors through tools such as linters [25]–[28] or static analysers. Dynamic verification, on the other hand, tests the software during runtime, identifying defects that manifest during execution, often through techniques such as unit testing or integration testing [29]–[32]. These verification methodologies work in tandem to ensure both code quality and functional correctness throughout the development process.

1.1.1 Static Analysis

Static verification, or *static analysis*, involves examining and evaluating a program's code without executing the program, providing a means to identify potential errors and issues in the codebase before runtime. Static analysis is crucial for several reasons. Firstly, it allows for the early detection of security vulnerabilities and coding errors, which can be much harder and costlier to fix once they reach production [33], [34]. Additionally, it ensures compliance with coding standards and helps maintain clean, maintainable codebases [34], [35]. By integrating static analysis into the development process, developers can "shift left" to catch issues early, reducing technical debt and ensuring smoother, faster software deployment [33], [36].

Moreover, static analysis contributes to the overall security of software systems by identifying potential risks such as buffer overflows, SQL injections, and other vulnerabilities [34].

It also helps enforce coding standards, making the code easier to maintain and less prone to errors in the long run [35]. The following is an exploration of the most used static analysis techniques in the industry and academia.

The journey of program analysis has evolved alongside the growing complexity of software development. Early on, *Syntax Checking* [37]–[40] emerged as a foundational tool, catching rudimentary errors such as typos and structural inconsistencies. While invaluable for immediate feedback, it quickly became evident that syntax alone could not address deeper logical or runtime issues that plagued more sophisticated code bases. Syntax checking was a necessary first step, but as software systems grew, developers demanded more [15].

This is where techniques such as *Data Flow and Control Flow Analysis* came into play. In the 1970s, these methods, developed by pioneers such as Allen [41] and Sharir & Pnueli [42], began to dig deeper into the program's inner workings. Data Flow Analysis traced how data moved through a program, identifying potential misuses or leaks, while Control Flow Analysis focused on the logical paths that the execution might follow. Though these methods were a leap forward, they were not without flaws. Their predictions about dynamic behaviour, especially in complex systems, could lead to inaccuracies, or what practitioners call "false positives".

With the rise of larger and more interconnected systems, the need to understand *dependencies* between different parts of the code became critical. *Dependency Analysis*, introduced in the 1990s by researchers like Jackson [43], mapped these interactions, offering invaluable insights into how changes in one module might ripple across the entire system. However, as systems grew to millions of lines of code, the challenge of managing and comprehending these dependencies became overwhelming.

In parallel, the industry was grappling with *Symbolic Execution*, a technique developed in the mid-1970s by J. C. King [44]. Symbolic execution offered the tantalising prospect of testing all possible execution paths by treating inputs symbolically rather than concretely. While powerful, this approach required enormous computational resources, limiting its use to specific cases where the potential payoff was worth the cost.

As the complexity of software continued to grow, *Semantic Analysis* emerged as a deeper, more nuanced approach. Aho [45], and others argued that understanding the meaning of code—its semantics—was key to detecting issues such as type safety violations or improper

usage of programming constructs. However, this approach required a sophisticated understanding of the language's semantics, making it challenging to apply without deep expertise [46].

Around the turn of the millennium, Bounded Model Checking (BMC) emerged as a prominent technique, targeting the detection of bugs within a predefined bound on the execution depth. This approach, pioneered by Biere [47] and further developed by Biere, Cimatti, Clarke, and Zhu [48], [49], offered a more scalable way to detect issues in finite sections of code. BMC examines the reachability of states within a predetermined number of steps, utilising formal methods and SAT or SMT solvers to verify the absence or presence of bugs within those constraints. While it is limited by its inability to see beyond its predetermined bounds, BMC's precision in identifying error conditions and providing counterexamples makes it an invaluable debugging tool, especially in hardware design, embedded systems, and safety-critical software verification.

Lastly, Abstract Interpretation, introduced by Patrick Cousot and Radhia Cousot in the late 1970s [50], became known for its ability to generalise and analyse all possible executions of a program. As a static analysis technique, it examines code without actual execution to deduce program behaviours by abstracting possible program states to identify potential errors and verify properties, utilising mathematical models to broadly approximate program behaviours. This technique is particularly adept at detecting runtime errors and compliance issues, offering insights into potential bugs such as null pointer dereferences and memory leaks preemptively. It is used in compiler optimisations and static analysis tools, aiding in pre-execution bug detection and software analysis. Its strength lies in scalability and soundness, but it is not immune to criticism. However, by over-approximating the behaviour of a program, it sometimes reports false positives, leaving developers to sift through unnecessary warnings [51].

Each technique offers unique advantages in static analysis, yet they also possess inherent limitations that necessitate careful selection based on specific analysis goals. Overall, two methods that stand out for their innovative approaches and transformative impact are Bounded Model Checking (BMC) and Abstract Interpretation. Bounded Model Checking (BMC) and Abstract Interpretation stand out due to their proven effectiveness and widespread adoption in the industry. BMC is valued for its precision in detecting complex bugs, especially in hardware and embedded systems, while Abstract Interpretation excels in analysing large-scale,

safety-critical software. Both techniques are recognised for their scalability and impact on real-world applications, making them key tools in static analysis [51], [52].

1.1.2 Dynamic Analysis

Dynamic analysis is essential to software testing [23]. It involves running the program in real time to uncover issues that static analysis alone might miss, such as runtime errors, memory leaks, and performance problems. Each dynamic testing method has its distinct advantages and challenges, and tracing their evolution provides valuable insight into their significance and adoption. Historically, these techniques emerged in response to the growing complexity of software systems, reflecting both academic inquiry and industrial need.

Among these techniques, Fuzz Testing, also known as fuzzing, has garnered significant attention due to its unparalleled ability to detect critical flaws. While methods such as Concolic Testing [53] explore specific execution paths with precision, and Performance Testing [54] benchmarks a system's response under varying workloads, fuzzing operates with a broader, almost chaotic scope. Professor Barton Miller's introduction of fuzzing in 1995 at the University of Wisconsin-Madison marks a pivotal moment in software testing [55]. Its breakthrough lies in its ability to generate vast amounts of random or semi-random input autonomously, pushing software systems to their breaking points.

Fuzzing excels in uncovering often elusive vulnerabilities, particularly in large and complex systems. This is not just due to the volume of inputs it generates but also because it introduces scenarios developers may never anticipate. Its success is evidenced by its widespread adoption in industries where security is paramount—such as operating systems, browsers, and cloud infrastructure-often finding vulnerabilities that other testing techniques overlook. The automation and scalability of fuzzing further enhance its appeal. Unlike Concolic Testing, which struggles with scalability in complex applications [56], fuzzing can handle large-scale systems by continuously generating new test cases with minimal human intervention.

However, fuzzing is not without its challenges. Its reliance on large computational resources to generate and process the high volume of test cases is a significant drawback [55]. Furthermore, while fuzzing effectively exposes vulnerabilities, it offers little systematic exploration of execution paths, leaving gaps in its coverage that more targeted methods, such as Concolic Testing, are better equipped to address. Despite this, the sheer efficiency of

fuzzing in security-critical applications makes it a preferred method for many organizations, even over more academically explored techniques such as abstract interpretation or bounded model checking.

What sets fuzzing apart is its random nature and its alignment with real-world attack vectors. In a world of increasingly sophisticated cyberattacks, the ability to simulate unpredictable input scenarios has become invaluable [57]. Fuzzing exposes the vulnerabilities that malicious actors might exploit, making it indispensable in the software industry's efforts to safeguard systems preemptively. This is why fuzzing has become a highly regarded and essential tool in security research and practical applications—its ability to find bugs that no one knew existed gives it a competitive edge in software testing.

In conclusion, dynamic testing methods such as fuzzing, Concolic Testing, and Performance Testing are indispensable for ensuring modern software systems' reliability, safety, and security. Fuzzing's automation and capacity for handling large datasets make it stand out, particularly in industries where security is non-negotiable. Ultimately, combining fuzzing with other dynamic and static testing methods offers the most robust strategy for uncovering defects and securing software, underscoring its pivotal role in the future of software engineering [54]–[56], [58].

1.2 Problem Description

Automated testing and software verification are essential tools for ensuring the reliability and correctness of modern software systems [59]. These techniques promise to streamline the testing process, catching bugs and issues early. However, despite their potential, they are still far from flawless. One of the biggest challenges they face is the Search Space Explosion problem (also known as State Space or Path Explosion) [60], which significantly limits the effectiveness of both static and dynamic analysis methods. This issue arises due to the exponential growth of possible states and execution paths, making it difficult to verify complex systems fully [61].

Bounded Model Checking (BMC) transforms the verification problem into a satisfiability problem (SAT) [47] or a satisfiability modulo theories (SMT) [62], [63] problem, which is then solved by a SAT or SMT solver. As the system's complexity grows, the number of

states increases exponentially, leading to a state space explosion, especially when there are unbounded loops. This explosion makes it increasingly difficult for SAT/SMT solvers to manage the verification process within reasonable time frames or memory constraints [64].

The state space explosion problem has not been completely solved, though significant progress has been made in mitigating its impact [65]. Techniques such as SAT/SMT solver optimisations, symbolic representations like Binary Decision Diagrams (BDDs) [66], interpolation methods [67], property-directed reachability (PDR) [68], and compositional or incremental verification methods have effectively managed this challenge [66]. However, several issues persist [69]. Scalability remains a concern, as these methods may not efficiently handle extremely large or infinite state spaces. Moreover, the trade-off between precision and efficiency is notable, with abstraction techniques simplifying the state space but requiring further refinement to ensure accuracy [49]. Many techniques also demand manual intervention or domain-specific knowledge, leaving the challenge of full automation unresolved. Memory limitations are also problematic, as even symbolic methods can struggle with large models containing numerous variables. Despite these advances, achieving a completely scalable and automated solution to state space explosion is still an open challenge [69], [70].

Although abstract interpretation inherently aims to mitigate state space explosion by working with abstract domains, the choice of abstraction can lead to a trade-off between precision and scalability. Too coarse an abstraction might miss errors (false negatives), while too fine an abstraction might not sufficiently reduce the state space, leading to performance issues similar to those in explicit state enumeration. [71], [72].

Fuzzing is a dynamic software verification technique that generates inputs to a program to find bugs and vulnerabilities in a given program [55]. The problem in fuzzing arises when executing a program with many inputs and complex guards. Because the potential input space can be vast, especially for complex programs with numerous input parameters. As the number of inputs and the program's complexity grows, the time and computational resources needed to thoroughly fuzz the program increase exponentially, making it challenging to achieve comprehensive coverage [73]–[75].

Several strategies have been developed to mitigate the state space explosion problem in these methodologies, including heuristic search techniques, state space reduction methods such as partial order reduction [76], abstraction techniques to simplify models [65], and par-

allel or distributed computing approaches to increase computational capacity [77]. Despite these efforts, the state space explosion remains a significant hurdle in scaling these verification and testing techniques to complex software systems.

Summary of the Problem: The exponential growth of possible states and execution paths in complex software systems leads to state space explosion, severely limiting the scalability and effectiveness of automated testing and verification methods.

1.3 Challenges

Automated software verification techniques, such as Bounded Model Checking (BMC), Abstract Interpretation, and Fuzzing, aim to ensure software correctness. However, they face several challenges, particularly as software systems grow more complex. This section outlines three key challenges that limit the effectiveness and scalability of these techniques: search space explosion, resource consumption, and the need to maintain soundness and completeness.

Challenge 1: *Search Space Explosion* The first major challenge is the problem of *search space explosion*. As software systems become more complex, the number of states or execution paths grows exponentially, overwhelming traditional verification methods such as BMC and Abstract Interpretation. This exponential growth in the search space makes it difficult for solvers to explore all possible states, often resulting in incomplete verification or excessive computational overhead. To mitigate this issue, reducing the search space while maintaining soundness and completeness is crucial for ensuring efficient and accurate verification. Research shows that symbolic techniques such as Binary Decision Diagrams (BDDs) and SAT/SMT solvers are helpful but still limited by scalability concerns when tackling vast or infinite state spaces [66], [72]. Advanced methods such as interpolants [67] and property-directed reachability (PDR) [68] have been developed to enhance model checking by generating concise over-approximations of reachable states and focusing on property violations. However, even these techniques face challenges when dealing with highly complex systems, as they can still encounter scalability issues due to the underlying state space explosion problem.

Challenge 2: *Resource Consumption* Another significant challenge is the excessive con-

sumption of computational resources, including memory and CPU time, in software verification processes. Verification tools such as BMC and fuzzing require substantial resources to explore large portions of the search space. For instance, fuzzing, though effective in detecting security vulnerabilities, often generates vast amounts of random input, requiring extensive computational power to achieve meaningful coverage. Reducing resource consumption without compromising verification quality remains an open challenge. Studies suggest that optimising solvers or employing parallel computing techniques can alleviate this issue, but further innovations are needed [74], [78].

Challenge 3: ***Maintaining Soundness and Completeness*** The third challenge is preserving the *soundness* and *completeness* of verification methods while reducing the search space. Soundness ensures that verified properties truly hold in the system, while completeness ensures that all potential errors are detected. Many techniques, including BMC and Abstract Interpretation, struggle to balance reducing the search space with maintaining soundness and completeness. Abstractions used to simplify verification may overlook important execution paths, resulting in false negatives, while overly precise methods can overwhelm the system with false positives or unsolved states. Ensuring that reductions in the search space do not compromise the verification's reliability is a critical area of ongoing research [59], [79].

By addressing these challenges, this research aims to enhance software verification techniques by integrating numerical methods and Interval Methods to reduce the search space and improve overall efficiency without sacrificing soundness and completeness.

1.4 Research Questions

Among the various challenges in software verification, this thesis specifically addresses the issue of search space explosion, which significantly hinders scalability and efficiency. While the related concerns of resource consumption and maintaining soundness and completeness are acknowledged, the primary focus is on mitigating search space explosion through the integration of numerical methods—particularly Interval Methods and Contractors—within existing verification frameworks.

The principle behind contractors is based on constraint propagation and narrowing down the search space. By applying constraints derived from the problem's specifications, contrac-

tors eliminate regions of the search space that cannot contain a solution, effectively reducing the computational burden. This involves evaluating the constraints over the interval representations of variables and adjusting these intervals to exclude values that do not satisfy the constraints.

Research Question 1. *To what extent can integrating a numerical method into verification techniques effectively reduce the search space?*

This thesis attempts to mitigate the search space problem by using Interval Methods, focusing on Contractors in particular. Contractors are an interval method that over-estimates the solution of a given Constraint Satisfaction Problem [80]. By representing variables as intervals, contractors enhance these methods by actively reducing the size of these intervals without losing any potential solutions.

Research Question 2. *To what extent can the integration of Interval Methods—particularly contractors—into verification tools reduce computational resource consumption, such as memory and processing time, during software verification?*

The reduction in the search space directly impacts the amount of computational resources required, such as processing power and memory usage. While reducing the state space can lead to faster algorithms and lower memory consumption, this reduction is not always guaranteed. In some cases, the process of calculating the contractor can introduce significant computational overhead, which may outweigh the potential benefits. This trade-off between the effort involved in contractor calculations and the possibility of reducing the state space is a critical consideration. In complex systems, this overhead might limit the overall effectiveness of the reduction techniques. Chapters 3, 4, and 5 will explore this trade-off and its impact through experimental evaluation.

Research Question 3. *Does the integration of Interval Methods, particularly contractors, into verification frameworks such as Fuzzing, Bounded Model Checking (BMC), and Abstract Interpretation preserve the soundness and completeness of these techniques?*

Soundness and completeness are important concepts in software verification. Soundness ensures that if the software is verified as correct, it truly is, preventing false positives. Completeness ensures that all potential errors are detected, leaving no bugs unnoticed. Together, they guarantee software verification is both accurate and exhaustive, ensuring software correctness and reliability without overlooking any possible errors [59], [66]. In each chapter,

we will discuss the effect of integrating contractors on the soundness and completeness of each method.

By addressing these research questions, this thesis aims to contribute to the advancement of software verification techniques, making them more efficient and reliable through the integration of numerical methods and Contractors.

1.5 Research Scope

This research focuses on advancing the state of software verification by addressing the prevalent issue of state space explosion in automated testing and verification techniques. It also focuses on designing, integrating, and applying interval analysis methods, particularly contractors, within various software verification approaches, namely fuzzing, bounded model checking (BMC), and abstract interpretation. The aim is to reduce the computational overhead associated with the verification of complex software systems while ensuring the soundness (cf. Definition 1) and completeness (cf. Definition 2) of the verification process.

This study encompasses the following key areas:

- **Interval Analysis and Contractors**: The research explores the use of interval analysis and contractors to reduce the search space in software verification systematically. Contractors are employed to refine variable domains, eliminate non-solution regions, and simplify constraint satisfaction problems, thereby enhancing the efficiency of verification processes.
- **Integrating Contractors in Software Verification Techniques**: The design and integration of contractors is applied across multiple software verification methods:
  - *Fuzzing*: Contractors are used to optimise the input space for fuzzes by bypassing infeasible paths and focusing on promising regions, thereby improving code coverage and detecting vulnerabilities.
  - *Bounded Model Checking (BMC)*: Contractors are applied to prune the state space in BMC by refining variable domains and constraints derived from BMC instances, resulting in more efficient verification of safety-critical properties.

  - *Abstract Interpretation*: Contractors are utilised to enhance the precision of abstract interpretation by narrowing the abstract domains and providing a more accurate program behaviour analysis.

- **Performance Evaluation and Benchmarking**: The research comprehensively evaluates the proposed methods through rigorous benchmarking. The effectiveness of contractors in reducing the required computational resources, such as memory and CPU time, is analysed across various case studies.

- **Soundness and Completeness**: The study investigates the impact of integrating contractors on the soundness and completeness of the verification process. Ensuring that the reductions in search space do not compromise the reliability and accuracy of the verification is a critical focus of this research.

1.5.1 Types of Vulnerabilities Targeted in This Research

The verification methods examined in this thesis—fuzzing, bounded model checking (BMC), and abstract interpretation—are particularly suited to uncover a variety of software vulnerabilities, especially those critical to safety, security, and reliability. Among the most commonly detected vulnerabilities are:

- **Buffer Overflows:** Violations in memory access boundaries leading to memory corruption.

- **Integer Overflows:** Arithmetic operations resulting in unexpected values due to exceeding variable limits.

- **Null Pointer Dereferencing:** Runtime errors caused by accessing memory through uninitialized or null pointers.

- **Memory Leaks and Use-After-Free Errors:** Faulty memory handling leading to unreleased resources or use of invalid memory.

- **Assertion Violations and Logic Bugs:** Violations of expected control flows or formal assertions.

- **Unreachable or Infeasible Paths:** Missed paths in traditional analysis that may contain critical logic or faults.

By integrating interval-based contractors into these frameworks, this research has shown that additional or previously undetected vulnerabilities can be revealed. This is achieved by improving state space exploration and refining variable domains more precisely than traditional techniques.

An analysis of benchmark test cases from SV-COMP and TEST-COMP demonstrates that contractors enabled the detection of vulnerabilities in cases where traditional static or symbolic analysis failed to produce conclusive results. In particular, cases involving complex loop constructs, tight constraint conditions, or infeasible branches benefited significantly from the interval narrowing achieved through contractors. These included benchmarks from categories such as `LoopLit`, `BitVectors`, and `Sequentialized`, where traditional tools either timed out or reported inconclusive results but were resolved correctly with contractor-enhanced techniques.

These findings are elaborated further in the evaluation sections of Chapters 3, 4, and 5, where benchmark-specific vulnerabilities and outcomes are discussed in detail.

Therefore, the scope of this research is defined by its objective to push the boundaries of knowledge by enhancing the scalability and practicality of software verification techniques by integrating interval methods, making them more suitable for real-world, large-scale software systems.

1.6 Contributions

This thesis explores the novel design and respective implementation of methods that integrate contractor techniques into various verification and testing frameworks to reduce the search space and enhance efficiency.

Contractors in Fuzzing: This work introduces a novel strategy that incorporates contractor techniques, grounded in interval analysis and related methodologies, to optimise software testing through fuzzing. By strategically bypassing guard conditions, this approach enables fuzzers to probe deeper execution paths, thereby improving code coverage. The integration

of abstract interpretation and static analysis further refines the search space, allowing the fuzzer to concentrate on paths with a higher likelihood of uncovering errors. This method significantly enhances the detection of potential vulnerabilities by prioritizing high-value exploration paths.

Contractors in Bounded Model Checking (BMC): This thesis also extends the application of contractors to BMC, employing contractors to model constraints and properties as a Constraint Satisfaction Problem (CSP) composed of variables, domains, and constraints derived from BMC instances. By analysing the program's GOTO-intermediate representation, which simplifies control constructs such as `switch` and `while` statements, contractors are applied to prune the CSP search space. The implementation of both outer and inner contractors refines variable domains by excluding non-satisfiable regions, thereby reducing computational complexity. Detailed algorithms and case studies demonstrate the effectiveness of contractors, particularly in handling iterative structures, leading to a marked improvement in verification efficiency. Empirical results corroborate that the introduction of contractors enhances performance, optimising resource usage while preserving the soundness and completeness of the BMC process.

Contractors in Abstract Interpretation: This thesis further contributes to the field by incorporating contractor operations into the abstract interpretation framework to improve the precision and scalability of program analysis. The use of contractors facilitates more refined constraint management and domain-specific approximations, resulting in enhanced accuracy and efficiency in analysing program behaviour. This method addresses several limitations of traditional abstract interpretation techniques, particularly in managing complex data structures and non-linear properties, and contributes to the development of more robust and scalable analysis tools.

This structured integration of contractor techniques across multiple verification and testing methods underscores the contributions of this thesis in reducing search space, improving computational efficiency, and maintaining rigorous verification standards.

1.7 Organisation of the Thesis

This thesis is structured to progressively build an understanding of advanced software verification techniques, the technologies employed, and their practical applications in software verification research. Each chapter focuses on distinct yet interrelated aspects of the research, providing a comprehensive overview and detailed analysis.

Chapter 2 presents a comprehensive overview of software verification techniques. It elucidates the various technologies employed in software verification research, assessing their effectiveness and limitations. The chapter introduces interval analysis and computations, explaining their utility in solving generic constraint satisfaction problems. Detailed discussions on the most effective interval methods are provided, demonstrating their application in enhancing software verification processes.

Each of Chapters 3, 4, and 5 explores a distinct verification technique—fuzzing, bounded model checking, and abstract interpretation, respectively—and examines how the integration of contractors affects their scalability and precision. While these techniques differ in methodology and application, they are unified in this thesis through the shared objective of reducing search space complexity using interval-based contractors. Chapter 3 demonstrates this integration in dynamic analysis via fuzzing, Chapter 4 applies it in a symbolic model checking setting (BMC), and Chapter 5 evaluates its impact within a static abstract interpretation framework. These chapters collectively answer the three central research questions introduced in Chapter 1 and provide cross-method insights into the effectiveness of contractors. Chapter 6 consolidates these findings and highlights their theoretical and practical implications.

Chapter 3 delves into the application and implementation of contractors within fuzzing. This chapter highlights the architecture of the selected tool, *FuSeBMC*, and explicates the integration of contractors and abstract interpretation into *FuSeBMC*. The discussion includes technical details on the methodology, showcasing how these integrations enhance the fuzzing process.

Chapter 4 explores the application of contractors in the context of Abstract Interpretation. This chapter focuses on implementing these methods within the emerging abstract interpretation framework in ESBMC (Efficient SMT-Based Model Checker). It provides an in-depth analysis of the results obtained from this application, discussing the improvements and chal-

lenges encountered.

Chapter 5 addresses applying interval methods via contractors in the Incremental Symbolic Bounded Model Checking (BMC) software. This chapter aims to enhance the verification process by introducing a novel method to model constraints and properties from BMC instances into a Constraint Satisfaction Problem (CSP). Contractors are utilised to refine the search space, thereby improving the efficiency of software verification. The chapter includes illustrative examples demonstrating the use of contractors to prune the search space in BMC by managing variable domains and constraints. Implementation steps are detailed, and the method's effectiveness is evaluated through experimental results, highlighting improvements in verification performance and resource consumption.

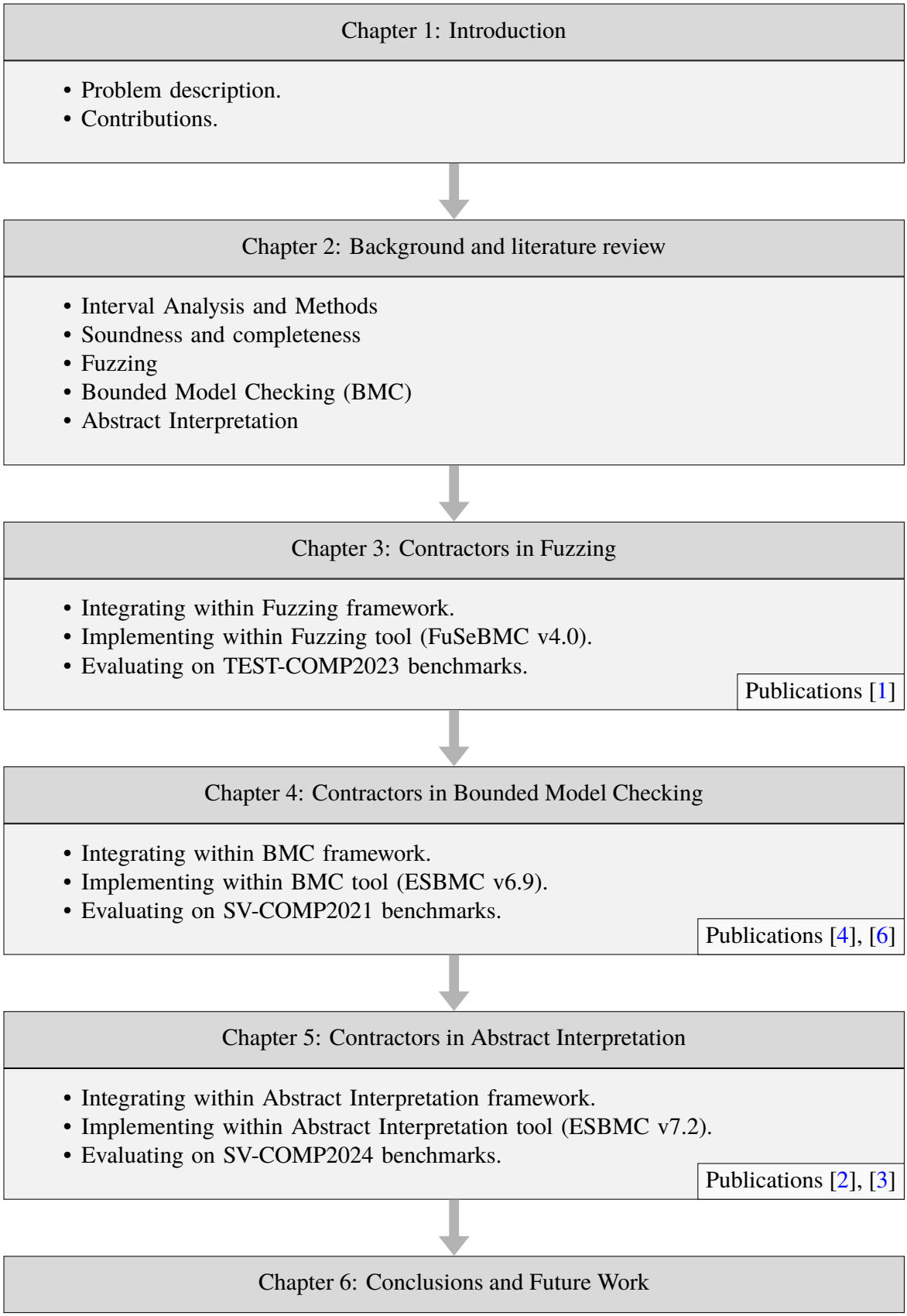


Fig. 1.2. Thesis structure.

Chapter 2

Background and Literature Review

2.1 Overview of Background and Related Concepts

This chapter describes the foundational concepts underpinning the various software analysis techniques explored in this thesis. The goal is to provide the reader with a clear understanding of the core principles and methods central to the discussions and implementations in the subsequent chapters.

We begin with a detailed exploration of interval analysis and its applications. Interval analysis is a powerful numerical method for dealing with uncertainty in mathematical computations, and its methods form the backbone of many verification techniques discussed later [50], [81]. This section also covers the essential set operations and arithmetic rules that apply to intervals, providing a rigorous mathematical foundation.

Next, we introduce the concept of constraint satisfaction problems (CSPs) within the context of interval analysis [82]. CSPs are pivotal in various verification methods, particularly in how they enable framing and solving interval-based problems. The chapter also discusses *contractors* interval methods that estimate solutions to CSPs by narrowing down the possible solution space, ensuring both correctness and efficiency.

Additionally, this chapter covers key verification concepts such as *soundness* and *completeness*, which are crucial in evaluating the reliability of software verification methods [81]. Finally, we explore fuzzing and bounded model checking (BMC), two dynamic analysis techniques integral to modern software testing and verification.

By the end of this chapter, the reader will have a comprehensive understanding of these preliminary concepts, enabling a deeper appreciation of the advanced methods and techniques

discussed in the later chapters.

2.2 Soundness and Completeness

In formal verification, there are two central concepts, soundness and completeness, to estimate the effectiveness of the verification process [81]. These are standards by verifiers on the level to which a program correctly implements given properties.

Definition 1 (***Soundness***)**.** For a verifier to be considered *sound*, it must reason about all program executions. Thus, if the verifier result concludes that there are no bugs, i.e., the verification outcome is TRUE, then the program is *safe* concerning the specified property. Formally, we can define *soundness* with the following equation:

$$\forall\, \texttt{p} \in \mathbb{L},\ \texttt{verifier(p)} = \textbf{TRUE} \implies \texttt{p} \text{ satisfies } \mathscr{P} \tag{2.1}$$

where $\mathbb{L}$ is the set of all programs, the program `verifier` is sound with respect to property $\mathscr{P}$ [81]

Soundness is concerned with the accuracy of the verification process in asserting that a program is free from errors [83]. Specifically, a sound verifier guarantees that if it concludes the program is safe (i.e., the verification outcome is TRUE), then the program indeed satisfies the intended property $\mathscr{P}$. This characteristic is formalised in Definition 1, emphasising that the verifier's assertion of correctness is trustworthy and that no errors have been overlooked in the analysis [83].

Definition 2 (***Completeness***)**.** For a verifier to be considered *complete*, it must have a concrete program execution that results in a property violation. Thus, if the verifier reports a bug, then the program is *unsafe* concerning the specified property. Similar to soundness 2.1, we can define completeness with the following equation:

$$\forall\, \texttt{p} \in \mathbb{L},\ \texttt{verifier(p)} = \textbf{TRUE} \impliedby \texttt{p} \text{ satisfies } \mathscr{P} \tag{2.2}$$

Completeness, as outlined in Definition 2, deals with the verifier's ability to detect violations of the specified property. A complete verifier ensures that a bug or violation within a

program will be identified, thereby confirming the program is unsafe if the verification outcome is FALSE. Completeness thus ensures that the verification process is thorough and that no potential errors are missed [84].

The relationship between soundness and completeness highlights the trade-offs that often arise in formal verification. A sound verifier minimises the risk of false positives—incorrectly indicating that a program is safe—while a complete verifier reduces the likelihood of false negatives—failing to detect a real issue [85]. Ideally, a verifier would be sound and complete, accurately identifying every genuine error without mistakenly flagging correct behaviour. However, achieving this balance is complex, and often, a compromise must be made depending on the specific goals of the verification process [86].

Understanding these concepts is key to developing and evaluating verification tools that ensure the reliability of software systems, particularly as their complexity and criticality continue to grow.

2.3 Fuzzing

Fuzzing is a technique used in software testing to uncover coding errors and security vulnerabilities by inputting large volumes of random data, known as "fuzz", into a system to provoke crashes or anomalous behaviour [87]. This technique automates unexpected but valid data input to reveal vulnerabilities such as buffer overflows, unhandled exceptions, and memory leaks. The primary objective of fuzzing is to identify weaknesses in a software system that could be exploited maliciously or lead to system failures.

Barton Miller pioneered fuzzing at the University of Wisconsin-Madison in 1988 [87]. Miller and his colleagues formulated the foundational principles of fuzzing and conducted a pioneering study demonstrating that simple random testing could unearth software application bugs.

Many companies, such as Microsoft, Google, and Mozilla, use fuzzing extensively to enhance the security of their software products. Before releasing, Microsoft uses fuzzing [88] to test operating systems and applications, such as Microsoft Office. Google utilises *OSS-Fuzz* [89], a tool designed to improve the security and robustness of open source software. This tool continuously applies fuzzing to open-source projects, discovering critical vulner-

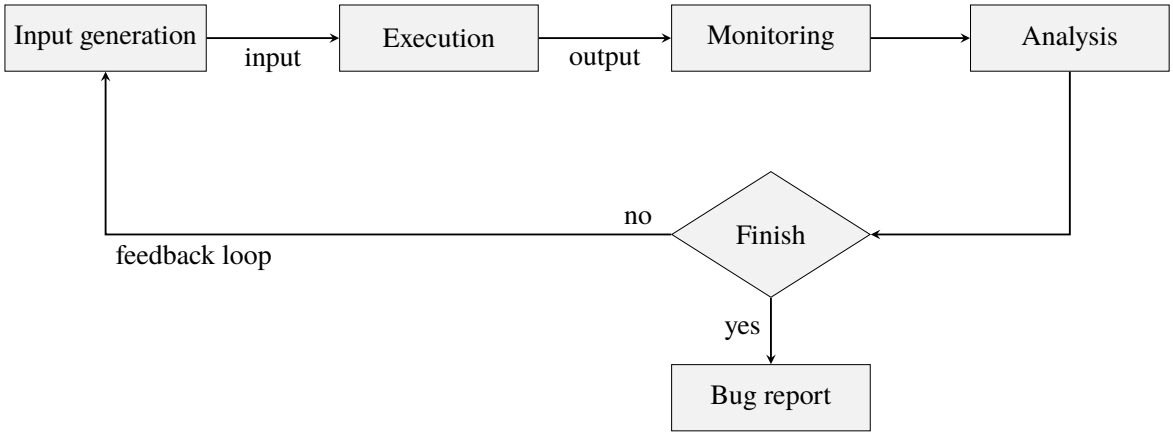


Fig. 2.1. Fuzzing Process [90].This diagram shows the iterative workflow of a fuzzing process, which includes six major steps: input generation, execution, monitoring, analysis, decision-making (finish or feedback), and bug reporting. The process uses a feedback loop to refine inputs for better bug detection efficacy and comprehensively assess vulnerabilities.

abilities in key libraries like *OpenSSL* and *libjpeg*. Mozilla uses fuzzing [88] to test components of the Firefox browser, including the rendering and JavaScript engines, which has significantly enhanced their security.

2.3.1 Fuzzing Process

First comes the input generation, in which the fuzzer generates inputs—either wholly arbitrary or crafted with care according to the expected input specifications of the program under test (PUT). The latter is an important step to ensure broad coverage of different possible input scenarios that the PUT may go through [90], [91].

Following the creation of such inputs, the next step is called **execution**, where such inputs are provided to the PUT. The execution may happen in different environments: locally on a tester's machine, remotely on servers, or even in controlled virtual environments so that no real-world systems are affected. The **execution** step runs the PUT to instigate problems that may arise during extreme or unexpected cases [90].

Next, it is essential to monitor the execution of the PUT and its response to each input, which should include checking for crashes, observing resource usage such as memory and CPU time, and validating the outputs against expected outcomes [92]. Such monitoring allows pinpointing where and when the PUT fails or behaves unexpectedly, giving the first insights into possible weaknesses or bugs.

If any bugs are detected, an **analysis** is performed to understand the root causes and effects of the failure. It involves using debugging tools to backtrack to the part of the code that instigated the failure and conduct an in-depth analysis of the nature of the bug. This step not only identifies the vulnerable part of the code but also allows assessing the severity of the identified issues, which provides information for prioritising bug fixes [93].

Finally, fuzzing includes a **feedback loop** in which the results of the testing period are fed into future tests. This makes the fuzzer, over time, adapt its testing strategies and focus more on input types or parts of the code that are more likely to discover new bugs. Advanced fuzzing techniques may even include machine learning algorithms to optimise this process better so that the tests become more effective with time [91].

2.3.2 Fuzzing Tools (Fuzzers)

Table 2.1. Comparison of fuzzing tools. The table below will go over the common fuzzing tools, their strengths and weaknesses, and best target applications. This will include AFL, LibFuzzer, Honggfuzz, and others, evaluating their applicability to domains including memory corruption detection, kernel fuzzing, and protocol testing.

Tool	Strengths	Weaknesses	Target Applications
AFL [94]	Effective for memory corruption bugs; coverage-guided; widely used	Requires source code instrumentation; limited for closed-source binaries	C/C++ programs
LibFuzzer [95]	Integration with LLVM; efficient memory error detection; unit testing style	Requires writing C++ harnesses; may not suit all developers	Library functions in C/C++
Honggfuzz [96]	High performance; supports user-space and kernel fuzzing; ASan integration	Steeper learning curve for beginners	User applications and Linux kernel
FuSeBMC [97]	Combines fuzzing with formal methods; finds deep bugs	Complex setup; higher resource requirements	Bounded model checking problems
OSS-Fuzz [98]	Automated large-scale fuzzing; continuous integration; finds numerous bugs	Mainly for open-source projects; not for proprietary software	Open-source libraries and applications
Radamsa [99]	Simple to use; effective for malformed inputs; easy integration	Lacks coverage guidance; may miss deeper bugs	Network protocols and file formats
Peach Fuzzer [100]	Comprehensive platform; extensible; supports many targets	Complexity; cost for commercial version	Protocols and file formats
Boofuzz [101]	Open-source; Python-based; easy to extend	Less efficient than compiled fuzzers	Network protocols, file formats, APIs
Atheris [102]	Python integration; handles complex data types; code coverage	Limited to Python; not for other languages	Python libraries and applications
Syzkaller [103]	Deep kernel integration; finds complex kernel bugs	Requires expertise; complex setup	Linux kernel fuzzing

Many tools implement fuzzing, each with unique approaches and target applications. Table 2.1 describes the strengths and weaknesses of each fuzzing tool. Notably, AFL (American

Fuzzy Lop) [94] is most recognised with wide efficiency in memory corruption-related bugs that could potentially yield vulnerabilities. AFL employs a genetic algorithm to create test cases while analysing the program's responses for every input supplied to optimise inputs and find new execution paths for the program's control flow. AFL technique has managed to detect many bugs in diverse software programs. LibFuzzer [95], on the other hand, a part of the LLVM compiler infrastructure, works primarily on the fuzzing of library functions individually. This strategy is most useful for projects that can identify functions apt for unit-testing-like scenarios. Honggfuzz [96] is another commonly used fuzzing tool that supports both coverage-guided and feedback-directed fuzzing approaches; it is also highly compatible with AddressSanitizer (ASan) in memory bug detection.

FuSeBMC [97] is a recent development in the area dedicated precisely to the task of fuzzing software that can be described as bounded model checking problems. This tool combines fuzzing techniques with symbolic execution and formal verification methods in order to achieve a deeper knowledge of the behaviour of the PUT. OSS-Fuzz [98], developed by Google, is a service that collaborates with open-source projects in order to run continuous testing of their software libraries and programs. Many security bugs and stability problems have been found with OSS-Fuzz. Radamsa [99] is a test case generator that mutates existing inputs to generate new test cases. This tool is important in that it is simple yet effective at producing deformed inputs to test the robustness of network protocols and file formats.

Moreover, Peach Fuzzer [100] is a comprehensive fuzzing framework that provides protocol and file format fuzzing. The extensibility of Peach Fuzzer, combined with its wide range of target systems applications, makes it suitable for industrial usage. Boofuzz [101] is an open-source fuzzing framework modelled from the Sulley [104] fuzzing framework. It supports fuzzing network protocols, file formats, and software APIs. Atheris [102] is a Python-based fuzzing tool tightly coupled with AFL that targets Python libraries and applications, aims to increase code coverage, and focuses on detecting run-time errors. Finally, Syzkaller [103] is a powerful and complex fuzzer designed exclusively for the Linux kernel by utilising a coverage-guided technique in finding security-related defects within the kernel source. These tools show how fuzzing techniques can be diverse depending on domains or software architectures because each fuzzing tool produces different test cases, has explored different execution paths, or facilitates integration with already existing development workflows. Continued development and improvement are significant in enhancing the reliability

and security of software in an increasingly complex technological landscape.

2.4 Bounded Model Checking (BMC)

Bounded Model Checking (BMC) is a verification technique used to verify the correctness of a system model within a specified number of steps [49]. It effectively identifies bugs in the early stages of system executions by transforming the verification problem into a satisfiability problem solvable by SAT or SMT solvers [105]. BMC is widely utilised in software through tools like CBMC and ESBMC [106], [107], which can analyse C and C++ programs for various safety properties and assertions by unwinding loops and analysing path constraints. This method is a preferred choice in academic and industrial applications for ensuring system reliability within bounded limits due to its capacity to provide a scalable solution to verify certain critical characteristics in the short term.

2.4.1 BMC Process

In BMC, the program is modelled as a state transition system (TS), which is derived from its control-flow graph [108], and then converted to a *Static Single Alignment* form (SSA). SSA is a representation where each variable is assigned exactly once, and every variable is defined before it is used, simplifying data flow analysis and optimisations [109]. Each control graph node will be converted to either an assignment or a conditional statement converted to a guard. Each edge represents a change in the program's control location [62]. When modelling the program, a Kripke structure [110] is used as TS $M = (S, T, S_0)$, which is an abstract machine that has a set of states S, initial states $S_0 \subseteq S$, and transition relation $T \subseteq S \times S$. The set of states $S = \{s_0, s_1, ...s_n\} : n \in \mathbb{N}$ contains all the states. Each state has the value of each variable in the program and a program counter pc. Each transition is denoted by $\gamma(s_i, s_{i+1}) \in T$, where it represents a logical formula that encodes all the changes in variables and pc from s_i to s_{i+1}. Next is to compute a *Verification Condition (VC)* denoted by Ψ, a quantifier-free formula in a decidable subset of first-order logic. Given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times to produce Ψ such that Ψ is satisfiable *iff* ϕ has a counterexample of length k or less.

$$\Psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} (\bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})) \wedge \neg\phi(s_i) \tag{2.3}$$

In the logical formula (2.3) [62], I denotes the set of initial states of M, $\gamma(s_j, s_{j+1})$ denotes the relation between two states in M. ϕ denotes the safety properties that should not be violated. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the execution of M of length i and if there exists some $i \leq k$ such that it satisfies Ψ_k at time-step i there exists a state in which ϕ is violated. Then, Ψ_k is given to an SMT solver to be checked for satisfiability. If it is satisfiable, then the SMT solver will provide an assignment that satisfies Ψ_k. This assignment constructs the counterexample using the values extracted from the program variables. For a property ϕ, a counterexample consists of a sequence of states $\{s_0, s_1, .., s_k\} | s_0 \in S_0$ and $s_i \in S | 0 \leq i < k$ and $\gamma(s_i, s_{i+1})$. If Ψ_k is unsatisfiable, no error state is reachable in k steps or less; therefore, no property was violated. Tools that implement BMC for software [62] produce two quantifier-free formulae, represented by C and P, which encode the constraints and properties, resp. Formula C serves as the first part of Ψ_k, which is $I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$. As for $\neg P$, it serves as the second part of Ψ_k which is $\bigvee_{i=0}^{k} \neg\phi(s_i)$. Then the SMT solver checks $C \models_{\mathcal{T}} P$ in the form of $C \wedge \neg P$.

BMC can also be used for soundness (cf. Definition 1) under certain conditions. A completion threshold (CT) is used to provide soundness [49]. This threshold checks whether k reaches all reachable states of M. This means a path of size k that can reach the program's last state starting from the initial state. CT_k can be defined as:

$$CT_k = I(s_0) \wedge \bigwedge_{i=0}^{k} \gamma(s_i, s_{i+1}) \wedge \neg(s_k = s_n) \tag{2.4}$$

Where n is the last reachable state of M. In BMC, if $\Psi_k \vee CT_k$ is UNSAT for a given k, then the program can be considered *safe*. Finding the optimal value for k is not always possible due to unbounded loops (i) or limited system resources (ii). Considering these limitations, two approaches can be used: k-induction (for i) and incremental (for ii).

2.4.2 BMC Strategies

Incremental Bounded Model Checking (Incremental-BMC) and k-induction are two fundamental strategies in BMC. Those techniques overcome the complexity of program property verification either by iterative examination of program states or by mathematical induction over bounded traces. Incremental-BMC incrementally increases the bound k to perform an exhaustive state space exploration until the program is proven safe or unsafe or resources are exhausted. On the other hand, k-induction uses a sequence of logical steps: base case, forward condition, and inductive reasoning to prove properties universally true for execution traces of programs. The following definitions formalize these methods and highlight their theoretical implementation developed in [106].

Definition 3 (*Incremental-BMC*). In Incremental-BMC approach, the value of k is incremented interactively until all states are reached (or resources are exhausted) [106]. We can define the incremental-BMC for a program P and bound k as $Inc(P, k)$:

$$Inc(P,k) = \begin{cases} \text{P is unsafe}, & \Psi_k \text{ is SAT} \\ \text{P is correct}, & \Psi_k \vee CT_k \text{ is UNSAT} \\ Inc(P, k+1), & \text{otherwise.} \end{cases} \tag{2.5}$$

[106]

Definition 4 (*k-Induction*). k-Induction approach requires three steps: base case, forward condition and inductive step. Base case and forward condition are represented through Ψ_k and CT_k, respectively. The inductive step (referred to as S) checks whether a property that holds for k will imply that it also holds for any next k. To achieve this, S is applied to a M', which adds lambda transitions between every state [106] as follows:

$$I_k = \exists n. \bigwedge_{i=n}^{n+k-1} (\phi(s_i) \wedge \gamma'(s_i, s_{i+1})) \wedge \neg\phi(s_{n+k}) \tag{2.6}$$

$$K(P,k) = \begin{cases} \text{P has a bug}, & \Psi_k \text{ is SAT} \\ \text{P is correct}, & \Psi_k \vee (CT_k \vee S_k) \text{ is UNSAT} \\ K(P, k+1), & \text{otherwise.} \end{cases} \tag{2.7}$$

2.4.3 Limitations

Bounded Model Checking is a technique that examines only a finite number of program runs [49]. Despite this, it generates verification conditions that provide detailed information about the program's execution path, the context in which functions are invoked, and the exact representation of expressions [62]. These verification conditions are logical formulas from a bounded program and desired correctness properties. If all the verification conditions of a bounded program are valid, then the program conforms to its specification up to the provided bound [111]. Users can specify correctness attributes using assert statements or code generated automatically from a specification language. Although BMC was developed two decades ago, recent developments in SMT have made it more practical [63]. However, due to the complexity of modern software systems, the impact of this technique is still limited [112].

It is common for BMC tools to encounter failures due to memory or time limitations [113]. These limitations are usually observed in programs that have loops with indeterminate or enormous bounds. Furthermore, even if a program does not violate any safety rules up to a certain bound, its safety is not guaranteed beyond that bound [113]. Researchers have developed new techniques to explore a program's search space to address these limitations while ensuring overall correctness.

2.4.4 Advancements and Tools

Bounded Model Checking (BMC) has had notable improvements that have increased its scalability, efficiency, and relevance in diverse fields, such as software verification, hardware design, and artificial intelligence systems. These advancements include enhancements in solver technology, symbolic methods, parallelisation techniques, hybrid verification methodologies, and the integration of machine learning strategies.

The development of SAT and SMT [114] solver technologies has significantly enhanced BMC's capability to handle complex systems efficiently. Solvers like MiniSAT [115] and Z3 [116], which employ conflict-driven clause learning (CDCL) and incremental solving techniques, have greatly improved BMC's performance [117], [118]. Furthermore, the introduction of Satisfiability Modulo Theories (SMT) solvers, such as Yices [119] and Z3 [116], has expanded BMC's applicability by enabling it to handle more complex theories involving

real numbers and arrays [118], [120].

Symbolic BMC methods employ different representations like Binary Decision Diagrams (BDDs) [121] and SMT [114] formulas to compactly encode state spaces. This symbolic representation reduces memory usage and allows for the exploration of larger or even infinite-state systems [122]. Tools such as the C Bounded Model Checker (CBMC) [107] extends BMC's application to C programs, enhancing the reliability and safety of critical software by efficiently detecting bugs and vulnerabilities. CBMC utilises symbolic execution and abstraction techniques to improve scalability and reduce computational complexity [66].

Parallel and distributed BMC techniques have been developed to further scale the verification process. By dividing verification tasks across multiple processors or computing environments, these methods significantly reduce verification time and enable the handling of large-scale industrial systems [123]. Tools like ABC [124] employ parallelisation strategies to leverage the computational power of multi-core processors, demonstrating substantial speed-ups in the verification of complex systems.

Hybrid verification approaches combine BMC with methods like abstraction refinement and interpolation, enhancing precision and effectiveness, especially for complex systems [125]. These techniques allow for a more scalable verification process by abstracting irrelevant details and focusing on critical components of the system under analysis.

In the field of software verification, tools like ESBMC (Efficient SMT-Based Bounded Model Checker) [62], [126] have made significant contributions. ESBMC translates the system under verification into a single static assignment (SSA) form and then into SMT formulas, utilising powerful SMT solvers to check for property violations. It can verify safety properties in both sequential and multi-threaded C programs, offering flexibility in selecting between fixed and floating-point arithmetic. ESBMC effectively detects various issues such as array bounds violations, pointer safety errors, deadlocks, data races, overflows, and memory leaks. It has been successfully applied to verify the safety and security of diverse systems, including digital control systems, digital filters, unmanned aerial vehicles, and telecommunication software [127]–[130].

In artificial intelligence and autonomous systems, BMC is used to verify the safety and correctness of machine learning models and decision-making algorithms, ensuring adherence to safety constraints [131]. The integration of BMC into Model-Based Design workflows

allows for the early detection of potential errors during the design phase, reducing costly mistakes in later development stages [132].

The incorporation of machine learning techniques into BMC tools has opened new avenues for optimising solver strategies based on system properties. Recent studies have demonstrated how learning from past verification runs can predict beneficial configurations and solver parameters, reducing the time and effort required for the verification process while maintaining high accuracy [133].

These advancements have significantly broadened BMC's scope, solidifying its role in verifying complex systems across various domains. By leveraging improved solver technologies, symbolic methods, parallelization, hybrid approaches, and machine learning integration, BMC continues to evolve, offering more efficient and scalable solutions for system verification challenges.

2.4.5 Hybrid Verification Tools

Several modern verification frameworks have explored hybrid approaches that combine static and dynamic analysis techniques to address the scalability and precision limitations of individual methods. For instance, **KLEE** [134] integrates symbolic execution with concrete execution for test case generation, enabling more effective path exploration and input derivation . Similarly, **Driller** combines fuzzing with concolic execution to discover complex bugs by exploring paths that are difficult for traditional fuzzers to reach [135]. **CPAchecker** and **Ultimate Automizer** [136], [137] are other prominent examples from SV-COMP that support configurable program analysis, allowing combinations of predicate abstraction, BMC, and value analysis.

These tools illustrate the benefits of unifying complementary techniques, much like this thesis, which augments fuzzing, BMC, and abstract interpretation with interval-based contractor methods. By narrowing search domains and refining input constraints, the contractor-based approach contributes to enhancing precision and efficiency across these traditionally independent verification paradigms.

2.5 Interval Analysis and Methods

In this section, we will explore interval analysis, methods, and powerful numerical tools that can be used to solve a wide range of estimation problems in real intervals. The interval methods we will discuss are guaranteed approaches that do not use sampling techniques, meaning they can obtain all possible solutions. The following chapters frequently use these methods to apply the contractors in different verification methods.

Section 2.5.1 of this chapter will cover basic interval computations and interval analysis. Section 2.5.4 will introduce constraint satisfaction problems in the context of intervals. Finally, in Section 2.5.5, we will explore interval methods in detail, where several algorithms are developed to solve various types of constraint satisfaction problems efficiently.

2.5.1 Interval Analysis

Definition 5 (***Real Interval***)**.** [80] A *real interval* is a connected, closed subset of $\mathbb{R}$ denoted by $[x]$. It has lower and upper limits that are scalars denoted by $\underline{x}$ and $\overline{x}$, respectively, where $\underline{x}, \overline{x} \in \mathbb{R}$. An interval is defined as $[x] = [\underline{x}, \overline{x}] \in \mathbb{IR}$, where $\mathbb{IR}$ is the set of all intervals of $\mathbb{R}$ [138].

Here, we assume that $\underline{x} \leq \overline{x}$, nonetheless, in the field of *Modal Interval Analysis* [138] it is possible to have $\underline{x} > \overline{x}$. However, this case is outside this thesis's scope, which does not mean that the opposite is invalid as it is used in the field of *Modal Interval Analysis* [138].

Definition 6 (***Width and Centre***)**.** [80] Each interval has a *width* and *centre*, which are respectively defined as:

$$w([x]) = \overline{x} - \underline{x} \tag{2.8}$$

$$c([x]) = \frac{\overline{x} + \underline{x}}{2} \tag{2.9}$$

where *width* and *centre* [82] are w and c, respectively. An interval is considered *degenerate* or *punctual* if its width is 0, i.e., $w = 0$.

Definition 7 (*Box*). [80] Intervals can be extended to higher dimensions. Equation (2.10) defines an interval in n dimensions is as vector of intervals for each dimension.

$$[\mathbf{x}] = [x_1] \times [x_2] \times ... \times [x_n] \tag{2.10}$$

where $[\mathbf{x}]$ is called a *box* and $[\mathbf{x}] \in \mathbb{IR}^n$. Given that the set of all n-dimensional boxes is denoted by $\mathbb{IR}^n$, A box is an axis-aligned hyper-rectangle in $\mathbb{R}^n$.

Interval width and centre are also extended to boxes as follows:

$$w([\mathbf{x}]) = \max_{i \in \{1,...,n\}} w([x_i]), \tag{2.11}$$

$$c([\mathbf{x}]) = [c([x_1]) \quad c([x_2]) \quad ... \quad c([x_n])]^T. \tag{2.12}$$

The *centre* of a box is a vector of each interval centre(cf.2.8 and 2.9); *width* remains a scalar and is the maximum width of all intervals.

2.5.2 Set Operations

Every interval is a set, but not vice-versa [80]. For a set to be a real interval, it has to be closed, connected, and defined in real numbers. Set operations include *intersection*, *union*, and *difference*.

Definition 8 (*Intersection*). [82] Intersection is applied on intervals as follows, Let $m_1 = \text{Max}(\underline{x}_1, \underline{x}_2)$ and $m_2 = \text{Min}(\overline{x}_1, \overline{x}_2)$ then:

$$[x_1] \cap [x_2] = \begin{cases} [\, m_1, m_2 \,], & \text{if } m_1 \leq m_2 \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.13}$$

Definition 9 (*Union and Interval Hull*). [82] For union, it can result in two disconnected intervals. *Interval Hull* encompasses the entire area between two disconnected intervals, denoted by $\sqcup$. *Interval Hull* is applied on intervals as follows, Let $m_1 = \min(\underline{x}_1, \underline{x}_2)$ and

$m_2 = \max(\overline{x}_1, \overline{x}_2)$ then:

$$[x_1] \sqcup [x_2] = [[x_1] \cup [x_2]] = \begin{cases} [\,m_1, m_2\,], & \text{if } [x_1] \neq \varnothing \wedge [x_2] \neq \varnothing \\ [x_1], & \text{if } [x_1] \neq \varnothing \wedge [x_2] = \varnothing \\ [x_2], & \text{if } [x_1] = \varnothing \wedge [x_2] \neq \varnothing \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.14}$$

Definition 10 (*Difference*). [82] Interval difference, denoted by the operator $[\setminus]$, Let $[y] = [x_1] \cap [x_2]$ then :

$$[x_1][\setminus][x_2] = [[x_1] \setminus [x_2]] = \begin{cases} [x_1], & \text{if } [y] = \varnothing \vee [y] = [x_2] \\ [\overline{y}, \overline{x}_1], & \text{if } [y] \neq [x_2] \wedge \overline{y} < \overline{x}_1 \\ [\underline{x}_1, \underline{y}], & \text{if } [y] \neq [x_2] \wedge \underline{y} > \underline{x}_1 \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.15}$$

Set operations can also be extended to higher dimensions. The Cartesian product involves the independent application of interval operations to each pair of corresponding intervals. For example, when presented with two boxes, $[\mathbf{x}]$ and $[\mathbf{y}]$, their intersection, union, and difference are precisely defined by Equations (2.16), (2.17), and (2.18) respectively.

$$[\mathbf{x}] \cap [\mathbf{y}] = ([x_1] \cap [y_1]) \times ([x_2] \cap [y_2]) \times \ldots \times ([x_n] \cap [y_n]) \tag{2.16}$$

$$[\mathbf{x}] \sqcup [\mathbf{y}] = ([x_1] \sqcup [y_1]) \times ([x_2] \sqcup [y_2]) \times \ldots \times ([x_n] \sqcup [y_n]) \tag{2.17}$$

$$[\mathbf{x}][\setminus][\mathbf{y}] = ([x_1] \setminus [y_1]) \times ([x_2] \setminus [y_2]) \times \ldots \times ([x_n] \setminus [y_n]) \tag{2.18}$$

2.5.3 Interval Arithmetic

Interval arithmetic encompasses a set of operations—such as addition, subtraction, multiplication, and division—that are widely applied in numerical methods and constraint solving [82]. That includes addition and multiplication at a basic level. Division and subtraction

are best described as interval functions [80]. Consider $[x_1]$ and $[x_2]$, and let $\diamond$ be a binary operator; then, interval arithmetic can be defined as:

$$[x_1] \diamond [x_2] = \{x_1 \diamond x_2 \,|\, x_1 \in [x_1], x_2 \in [x_2]\}, \tag{2.19}$$

If one interval is empty, the result of the binary operation is also an empty interval.

Definition 11 (***Addition***)**.** Addition is interpreted over intervals as follows:

$$[x_1] + [x_2] = \begin{cases} [\,\underline{x}_1 + \underline{x}_2\,,\, \overline{x}_1 + \overline{x}_2\,], & \text{if} [x_1] \neq \varnothing \wedge [x_2] \neq \varnothing \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.20}$$

Definition 12 (***Multiplication***)**.** Multiplication is applied on intervals as follows:

$$[x_1] * [x_2] = \begin{cases} [\,\text{Min}(\mathbb{S}_x), \text{Max}(\mathbb{S}_x)\,], & \text{if} [x_1] \neq \varnothing \wedge [x_2] \neq \varnothing \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.21}$$

where $\mathbb{S}_x = \{\underline{x}_1\underline{x}_2, \underline{x}_1\overline{x}_2, \overline{x}_1\underline{x}_2, \overline{x}_1\overline{x}_2\}$

Definition 13 (***Functions***)**.** Functions are applied on intervals as follows:

Let $f : \mathbb{R} \to \mathbb{R}$, then

$$[f]([x]) = \begin{cases} [\,f(\underline{x})\,,\, f(\overline{x})\,], & \text{if} f(x) \text{ is} \\ & \text{monotonically increasing.} \\ [\,f(\overline{x})\,,\, f(\underline{x})\,], & \text{if} f(x) \text{ is} \\ & \text{monotonically decreasing.} \end{cases} \tag{2.22}$$

For not monotonic functions, we split the domain into sub-domains where the function is monotonic and apply Equation 2.22, then take the union of the result.

Definition 14 (***Subtraction***)**.** Subtraction is applied on intervals as follows:

$$\text{Let } f(x) = -x, \text{ then } [x_1] - [x_2] = [x_1] + [f]([x_2]) \tag{2.23}$$

Definition 15 (***Division***)**.** Similar to subtraction, division can be defined as a function:

$$\text{Let } f(x) = \frac{1}{x}, \text{ then } \frac{[x_1]}{[x_2]} = \begin{cases} [x_1] * [f]([x_2]), & \text{if} \{0\} \notin [x_2] \\ \varnothing, & \text{otherwise.} \end{cases} \tag{2.24}$$

2.5.4 Constraint Satisfaction Problem

Definition 16 (**Constraint Satisfaction Problem**)**.** A constraint satisfaction problem (CSP), in the context of interval analysis, is defined as the triple $(\mathcal{X}, \mathcal{D}, \mathcal{F})$, where:

$\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n real-valued variables, represented in vector form as $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$; $\mathcal{D}$ is the set of domains, where $[\mathbf{x}] \in \mathbb{IR}^n$ is a box representing the interval domain of each variable (cf. Definition 5); $\mathcal{F}$ is the set of constraints expressed as $\mathbf{f}(\mathbf{x}) \leq \mathbf{0}$, where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a vector-valued function whose components are f_j, for all $j \in \{1, \ldots, m\}$ [80].

The solution set of the CSP is defined as:

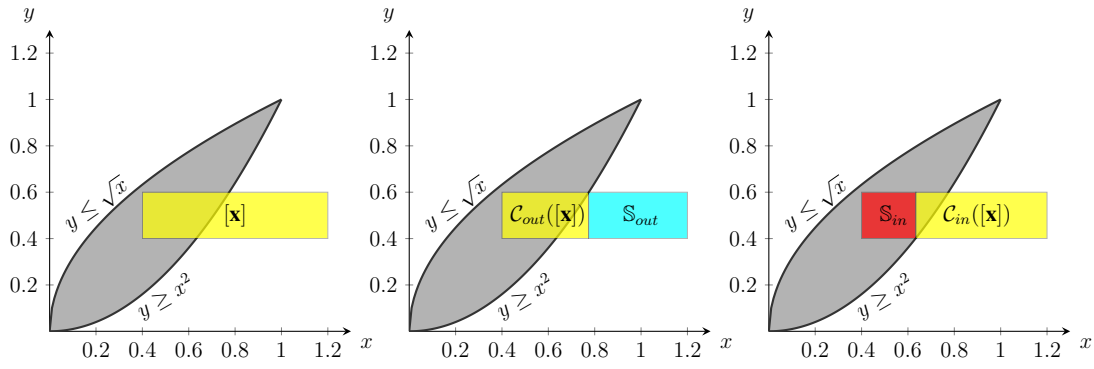
$$\mathbb{S}_{\mathbf{x}} = \{\mathbf{x} \in [\mathbf{x}] \mid \mathbf{f}(\mathbf{x}) \leq \mathbf{0}\}. \tag{2.25}$$

2.5.5 Interval Methods

Given a *CSP*, interval methods are heuristic search algorithms that compute upper-bound solutions [80].

Definition 17 (***Contractor***)**.** A *contractor* is an interval method that estimates the solution of a given CSP (cf. Definition 16) with the map $\mathcal{C} : \mathbb{R}^n \to \mathbb{R}^n$. Let $\mathbb{S}_{\mathrm{x}}$ be the solution set of the CSP, and $[\mathbf{x}]$ is an initial box then $\mathcal{C}([\mathbf{x}]) \subseteq [\mathbf{x}]$ and $\mathcal{C}([\mathbf{x}]) \cap \mathbb{S}_{\mathrm{x}} = [\mathbf{x}] \cap \mathbb{S}_{\mathrm{x}}$. The former satisfies the *contraction* condition, and the latter satisfies *correctness* condition [139].

As illustrated in Fig. 2.2, we can have inner contractor $\mathcal{C}_{in}$ and outer contractor $\mathcal{C}_{out}$, which complements each other in terms of constraints.



(a) Initial domain **(b)** outer contractor **(c)** inner contractor

Fig. 2.2. CSP with $\mathbf{x} \in \mathbb{R}^2$, initial box $[\mathbf{x}] = [0.4, 1.2] \times [0.4, 0.6]$, and two constraints: $y \geq x^2$, $y \leq \sqrt{x}$. Inner contractor versus an outer contractor. $\mathbb{S}_{out}$ and $\mathbb{S}_{in}$ are the remainder of $\mathcal{C}_{out}$ and $\mathcal{C}_{in}$ actions [80]

Lets start with Figure 2.2a where we see we have two constraints $y \leq \sqrt{x}$ and $y \geq x^2$ and the solution is the intersection between them (ie. the grey area). Our set of variables is $\mathbf{x} = \{x, y\}$ and our domains are the intervals $\mathbf{x} = [0.4, 1.2] \times [0.4, 0.6]$. In Figure 2.2b, we apply the outer contractor (i.e. remove the area outside the solution), and the resulting interval is the area inside the solution or on the boundaries. Similarly, in Figure 2.2c, when we apply the inner contractor and remove the area that is guaranteed to be inside the solution with the resulting area being outside the solution or on the boundaries.

$$\forall [\mathbf{x}] \in \mathbb{IR}^n,\ \mathcal{C}([\mathbf{x}]) = [[\mathbf{x}] \cap \mathbb{S}]. \tag{2.26}$$

After applying the contractor, the result is best described as three sets: $\mathbb{S}_{in}$ is the estimated solution of $\mathbb{S}_{\mathrm{x}}$ such that $\mathbb{S}_{in} \subset \mathbb{S}_{\mathrm{x}}$ color-coded in red. $\mathbb{S}_{out}$ is the *complement* of the estimated solution such that $\mathbb{S}_{out} \cap \mathbb{S}_{\mathrm{x}} = \varnothing$ color-coded in blue. $\mathbb{S}_{boundary}$ is the boundary set that represents the area that both includes the solution and its inverse such that $\mathbb{S}_{boundary} \cap \mathbb{S}_{\mathrm{x}} \neq \varnothing$ and $\mathbb{S}_{boundary} \not\subset \mathbb{S}_{\mathrm{x}}$ color-coded in yellow.

For different CSPs, there are many types of contractors[140], [141]. However, in this thesis, we will use *Forward-backward Contractor* [142] for its simplicity and efficiency.

Definition 18 (***Forward-backward Contractor***)**.** Forward-backward Contractor is a contractor (cf. Definition 17) that is applied to a CSP with one single constraint; it contracts in two steps: *forward evaluation* and *backward propagation* [80]; they are illustrated in Algorithm 1.

Generally, a CSP can have multiple constraints; to handle such a situation, we can use

Algorithm 1: Forward-backward Contractor $\mathcal{C}_{\uparrow\downarrow}$.

function $\mathcal{C}_{\uparrow\downarrow}([\mathbf{x}],\ f(\mathbf{x}),\ [I])$ **do**
  $[y] = [I] \cap [f]([\mathbf{x}])$
  for all $[x_i] \in [\mathbf{x}] : i \in \{1, ..., n\}$ **do**
    $[x_i] = [x_i] \cap [f_{x_i}^{-1}]([y], [\mathbf{x}])$
  end for
  return $[\mathbf{x}]$
end function

Outer Contractor, which applies forward-backward contractor on each constraint independently, as shown in Algorithm 2 [80].

Algorithm 2: Outer Contractor $\mathcal{C}_{out}$.

function $\mathcal{C}_{out}([\mathbf{x}],\ \mathbf{f}(\mathbf{x}))$ **do**
  for all $f_j \in \mathbf{f} : j \in \{1, ..., m\}$ **do**
    $[\mathbf{x}] \leftarrow \mathcal{C}_{\uparrow\downarrow}([\mathbf{x}],\ f_j(\mathbf{x}), (-\infty, 0])$
  end for
  return $[\mathbf{x}]$
end function

Forward-backwards contractor can be used to construct another type of contractor where the complementary CSP is considered; such contractor is called *Inner Contractor*, and its procedure is defined in Algorithm 3 [80].

Algorithm 3: Inner Contractor $\mathcal{C}_{in}$.

function $\mathcal{C}_{in}([\mathbf{x}],\ \mathbf{f}(\mathbf{x}))$ **do**
  $[\hat{\mathbf{x}}] \leftarrow \emptyset$
  for all $f_j \in \mathbf{f} : j \in \{1, ..., m\}$ **do**
    $[\hat{\mathbf{x}}] \leftarrow [\hat{\mathbf{x}}] \sqcup \mathcal{C}_{\uparrow\downarrow}([\mathbf{x}],\ f_j(\mathbf{x}), (0, \infty))$
  end for
  return $[\hat{\mathbf{x}}]$
end function

2.6 Abstract Interpretation

2.6.1 Introduction

Abstract interpretation is a theoretical framework used in computer science to analyse computer programs and extract information about their behaviour [50]. Patrick Cousot and Rad-

hia Cousot developed it in the 1970s [50], and it provides a way to simplify the program's semantics into a computable form, which is essential for practical program analysis. The central idea behind abstract interpretation is to map the complex semantics of a program into an abstract domain, which makes it simpler to work with. This abstract domain retains enough information to answer specific questions about the program's behaviour, such as identifying potential runtime errors, verifying compliance with type systems, or identifying optimisation opportunities [143].

Abstract interpretation defines an abstract domain that simplifies the values and states that a program can take [81]. For example, instead of considering the exact numerical values a variable can hold, an abstract domain might only distinguish between positive, negative, and zero values. The program's operations are then interpreted in this abstract domain, which allows the analysis to estimate the set of all possible states the program can reach during execution.

One of abstract interpretation's main strengths is its flexibility [81]. The same framework can be adapted to a wide range of analysis tasks by selecting different abstract domains. However, there is often a trade-off between the analysis's precision and its computational cost: a more detailed abstract domain can provide more accurate results at the expense of greater computational complexity [81], [144].

Abstract interpretation is used in many tools for static program analysis, including compiler optimisations, software verification, and security analysis [81]. Abstract interpretation formal definitions allow for the rigorous justification of the analyses' soundness, meaning that the analyses can guarantee the absence of specific errors in analysed programs [81].

In summary, abstract interpretation is a fundamental technique in program analysis that enables the systematic and scalable analysis of program behaviours. Abstracting away the complexities of exact execution semantics allows for the effective analysis of programs to ensure their correctness, security, and efficiency [51], [145].

2.6.2 Syntax

To comprehensively understand the fundamental principles of abstract interpretation, we need to establish a foundational language syntax that effectively showcases various techniques. In

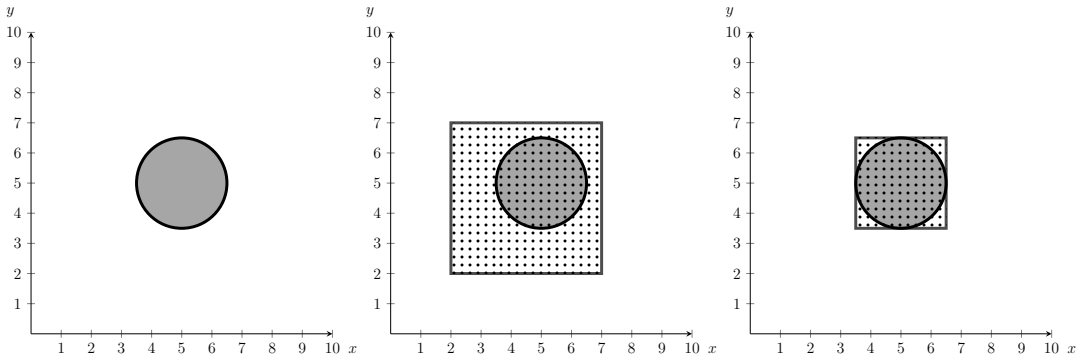
n	$\in$	$\mathbb{V}$	scalar values
$\mathtt{x}$	$\in$	$\mathbb{X}$	program variable
$\odot$	$::=$	$+ \mid - \mid * \mid \ldots$	binary operator
$\oslash$	$::=$	$< \mid \leq \mid == \mid \ldots$	comparison operator
E	$::=$		scalar expressions
	$\mid$	n	scalar constant
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$E \odot E$	binary operation
B	$::=$		Boolean expressions
	$\mid$	$E \oslash E$	Compare two expressions
L	$::=$		logical expression
	$\mid$	B	Boolean expression
	$\mid$	$L \wedge L$	and
	$\mid$	$L \vee L$	or
C	$::=$		commands
	$\mid$	$\mathtt{skip}$	command that ”does nothing”
	$\mid$	$C; C$	sequence of commands
	$\mid$	$\mathtt{x} := E$	assignment command
	$\mid$	$\mathtt{input(x)}$	command reading a value
	$\mid$	$\mathtt{if}(L)\{C\}\ \mathtt{else}\ \{C\}$	conditional command
	$\mid$	$\mathtt{while}(L)\{C\}$	loop command
P	$::=$	C	program

Fig. 2.3. A simple language syntax [81].

this regard, Figure 2.3 presents a rudimentary language designed with a limited set of features. This syntax is an extension of the originally proposed by Xavier Rival and Kwangkeun Yi in their seminal work on abstract interpretation [81].

The language syntax defines a set of scalar values representing constant values, $\mathbb{V}$. Additionally, a set of scalar-type variables, designated by $\mathbb{X}$ and defined by name, is also defined. The set of boolean values, $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$, is also included. Scalar expressions entail constant expressions, variables, and arithmetic operators applied to pairs, resulting in scalar values. Boolean expressions involve comparison operators that compare two expressions and result in $\mathbb{B}$ values. Logical expressions also result in $\mathbb{B}$ values but combine two boolean expressions into an ”AND” operation denoted by $\wedge$ and an ”OR” operation denoted by $\vee$.

The provided set of commands includes a skip statement that has no effect, a sequence of commands, assignment, input, conditional *if* statement, and simple loops. Although this language lacks many features commonly used in different languages, it can still express a large set of programs and demonstrate fundamental abstract interpretation concepts [81].



(a) A concrete set **(b)** Abstraction **(c)** Best abstraction

Fig. 2.4. This figure explains the abstraction principle in Abstract Interpretation: the first sub-figure (a) shows the concrete set containing all exact program states; the second sub-figure (b) shows an abstraction, which simplifies the states by merging them into some more general representation. Finally, the last sub-figure (c) shows the best abstraction, defined as the most precise approximation that contains all the concrete states and thus achieves a trade between precision and scalability, as described in [81].

2.6.3 Abstraction and Concretization

Definition 19 (***Abstraction***)**.** The concept of *abstraction* is defined as a collection $\mathscr{A}$ of logical properties that describe the program states [81]. These properties are referred to as *abstract properties* or *abstract elements*. An *abstract domain* is a set of abstract properties.

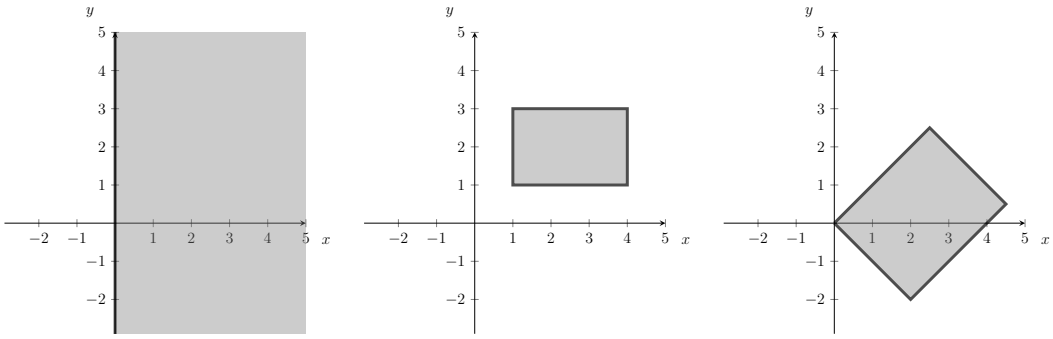
In this definition, the word abstract is used here instead of concrete. Moreover, all abstractions mentioned in this section are strictly an over-approximation.

Definition 20 (***Concretization***)**.** Concretization of an abstract element a of $\mathscr{A}$ is the set of program states that satisfy it, denoted by $\gamma(a)$ [81].

Definition 21 (***Best abstraction***)**.** We say that a is the *best abstraction* of the concrete set S if and only if $S \subseteq \gamma(a)$ and for any a' that is an abstraction of S (i.e., $S \subseteq \gamma(a')$), then a' is a coarser abstraction than a. If S has the best abstraction, then the best abstraction is unique. When it is defined, we let α denote the function that maps any concrete set of states into the best abstraction of that set of states.

2.6.3.1 Types of abstraction

Abstraction techniques are essential in simplifying the complexity of numerical values within programs, making it easier to analyse and verify their behaviour [81]. Among the various



(a) Sign abstraction where $x \geq 0$ **(b)** Interval Abstraction **(c)** Convex Polyhedra Abstraction

Fig. 2.5. Types of abstraction. The figure illustrates three types of abstraction in Abstract Interpretation. (a) Sign abstraction represents states based on variable signs, such as $x \geq 0$. (b) Interval abstraction approximates variable values within defined ranges, such as $x \in [a, b]$. (c) Convex polyhedra abstraction captures linear relationships between variables, providing higher precision at the cost of increased complexity. These abstractions offer a trade-off between precision and computational efficiency [81].

forms of abstraction, three notable methods are sign abstraction, convex polyhedra abstraction, and interval abstraction [81]. Each method has unique strengths and weaknesses, catering to precision and computational cost aspects. Sign abstraction is the simplest, categorising values by their signs with minimal computational expense but low precision. Convex polyhedra abstraction offers high precision by capturing exact linear relationships, albeit at a high computational cost. Interval abstraction balances these two, representing numerical values as intervals and providing moderate precision and computational efficiency. This balance makes interval abstraction a compelling choice for many applications, particularly those where ranges of variables are crucial, but the relationships between them are less so.

Sign Abstraction is one of the simplest forms of abstraction. This type of abstraction elements are over-approximated by sign [81]. It abstracts numerical values based on their sign. Each value is categorised as either positive (+), negative (-), or zero (0). Sign abstraction offers low precision as it only captures the sign of variables, losing any information about their magnitude or range. Its computational cost is very low, with simple operations as they only involve sign changes. This abstraction is useful for programs where the sign of a variable is important, such as detecting zero crossings. For example, Figure 2.5a shows a variable $x \geq 0$ the sign abstraction would represent it as $\{0, +\}$.

Interval abstraction represents numerical values as intervals $[a, b]$, where a and b are the lower and upper bounds of the possible values of a variable [81]. Interval abstraction provides medium precision when compared with convex polyhedra abstraction by capturing ranges of values but loses information about correlations between variables. Its computational cost

is low to moderate, as operations are relatively simple but can involve some overhead for managing interval bounds. This abstraction is suitable for programs where ranges of variables are important, but correlations are not. For example, for two variables x, y that range from 1 to 4 and 1 to 3, respectively, the interval abstraction would represent it as $[1, 4] \times [1, 3]$ as illustrated in Figure 2.5b.

Convex polyhedra abstraction represents numerical values using convex polyhedra, which can capture linear relationships between variables [81]. Convex polyhedra abstraction offers high precision by capturing exact linear constraints and correlations between variables. Its computational cost is high due to the complexity of operations on convex polyhedra, which involves linear programming techniques. This abstraction is ideal for programs where relationships between variables are crucial and precise linear constraints must be maintained. For example, For variables x and y that satisfy $x + y \leq 5$ and $x - y \leq 0$, $x + y \geq 0$ and $y - x < 0$, the convex polyhedra abstraction would capture this linear constraint precisely shown in Figure 2.5c.

In summary, interval abstraction is an efficient method for numerical abstraction in various verification contexts. While sign abstraction has low precision and convex polyhedra abstraction suffers from high computational costs, interval abstraction strikes a good balance between these competing aspects. It offers adequate precision by covering the range of variable values at a reasonable level of computational cost. The balance obtained by interval abstraction makes it especially well-suited for applications where the range of values is relevant but inter-variable dependencies are of minor interest.

2.6.4 Semantics

Semantics refers to the meaning of programs. Knowledge of semantics forms the basis for analyzing program behaviours at runtime and establishing correctness verification. Traditionally, semantics is classified into two complementary frameworks: concrete semantics and abstract semantics [81].

2.6.4.1 Concrete Semantics

Concrete semantics provides a precise and detailed description of the actual behaviour of a program [81]. It defines how program states change during execution. This is typically represented using sets of states and transitions between them. For example, concrete semantics can describe how the value of a variable changes over time as the program executes each instruction. This precise representation is crucial for understanding the exact behaviour of a program, but it often results in a vast amount of detail, making it impractical for large-scale analysis. The foundational work by Cousot & Cousot (1977) [50] laid the groundwork for this approach by establishing a unified model that captures these detailed behaviours.

Memory states, or simply a state, is a snapshot that captures the computer's configuration during program execution [81]. It includes the memory contents and the "program counter" value. Our language features (cf. Figure 2.3) a fixed set of variables, all of the same type, and does not include complex data structures. Therefore, a memory state is represented as a function m from the set of variables $\mathbb{X}$ to the set of values $\mathbb{V}$. For example, if $\mathbb{X} = x_1, x_2$, we write $\{x_1 \mapsto 8, x_2 \mapsto 4\}$ for the memory state that maps x_1 to 8 and x_2 to 4.

2.6.4.2 Abstract Semantics

Abstract semantics provides an abstract interpretation of the concrete semantics of a program. It involves mapping concrete states and operations into an abstract domain, where the operations are less detailed but more tractable for analysis. This abstraction simplifies the analysis by focusing on essential properties while ignoring irrelevant details [81], [144]. The abstract semantics is designed to over-approximate the program's behaviour, ensuring that all possible states and transitions in the concrete semantics are captured in the abstract domain [81], [144].

The importance of abstract semantics in abstract interpretation shows when it allows for analysing program behaviour without having to compute exact values or states. This is particularly useful in static analysis, where the goal is to prove properties about the program (such as the absence of errors) rather than to compute specific outcomes of the program's execution [81], [144].

The abstract semantics is defined by abstract transformers that approximate the effect of

each program operation on the abstract domain. These transformers are sound if they correctly over-approximate the effect of the corresponding concrete operations. The soundness of abstract semantics is key to ensuring that any conclusions drawn from the abstract interpretation hold for the actual program execution [50].

2.6.5 Interval Analysis

Interval analysis [81], [144] is a technique used within the framework of abstract interpretation, a theory used to reason about the behaviour of programs in a mathematically rigorous way. Abstract interpretation allows the analysis of programs by approximating their behaviours with a level of abstraction that makes the analysis feasible.

Definition 22 (***Interval analysis***)**.** refers explicitly to an abstract domain in abstract interpretation where the possible values of numerical variables in a program are represented as intervals [81]. Each variable is associated with an interval $[a, b]$, where a and b are the lower and upper bounds, respectively. This interval represents the variable's possible values during the program's execution.

Definition 23 (***Abstract Domain***)**.** In interval analysis, the abstract domain consists of intervals $[a, b]$ where a and b are elements of the set of real numbers, integers, or other numerical domains. The interval $[a, b]$ is an abstraction of the set of all possible values that a variable can hold.

Definition 24 (***Abstract Operations***)**.** Operations on variables in the program (like addition, subtraction, multiplication, etc.) are applied on intervals. For example, if a variable x is in the interval $[a_1, b_1]$ and another variable y is in the interval $[a_2, b_2]$, then the interval for $x + y$ would be $[a_1 + a_2, b_1 + b_2]$.

Definition 25 (***Widening and Narrowing***)**.** Since program loops can result in potentially infinite sequences of interval updates, interval analysis employs a technique called **widening** to ensure that the analysis terminates by over-approximating the intervals. **Narrowing** can then be applied to refine these over-approximations after widening.

Definition 26 (***Soundness and Precision***)**.** Interval analysis is sound, meaning that it correctly over-approximates the set of possible values that variables can take. However, it may not be very precise because the intervals can become quite large, especially after widening, leading to less precise information about the variable's value.

Interval analysis is used in various static analysis tools to detect potential runtime errors such as overflows, array out-of-bounds errors, and other numerical issues in programs [51], [146], [147]. By computing intervals for variables, it can check whether a variable might exceed certain limits, which could indicate a bug or vulnerability.

```
1 int x = 0;
2 while (x < 10) {
3     x = x + 2;
4 }
```

Listing 2.1. Interval analysis example

For example, consider the code in listing 2.1. When using interval analysis, initially x is in the interval $[0, 0]$. After the first iteration, x is in the interval $[2, 2]$. After several iterations, x could be in the interval $[0, 10]$ (over-approximation). In this case, interval analysis would allow the program analyser to determine that x will never exceed 10, avoiding potential overflows.

2.6.6 Abstract Interpretation Tools

Various tools have been developed for automating abstract interpretation, offering practical applications of the theoretical framework in actual software development and analysis. These tools are extensively utilised in industries for software verification, optimisation, and security analysis.

Astrée is one such tool [51]. This static analysis tool use abstract interpretation to identify runtime mistakes, including division by zero, out-of-bounds array accesses, and other important safety issues in embedded software. Astrée is proficient in analysing programs written in C, frequently utilised in safety-critical systems such as aerospace and automotive software.

Frama-C [146] is another significant tool, serving as a framework for the examination of C programs through several static analysis methodologies, including abstract interpretation. Frama-C enables developers to validate the attributes of their code, including accuracy, security, and adherence to particular coding standards.

Infer [147] is a tool created by Facebook for the analysis of Java, C, C++, and Objective-C code bases. It uses abstract interpretation to detect defects in mobile applications and server code, offering a scalable system capable of effectively analysing extensive code bases.

These tools demonstrate the practical application of abstract interpretation, showcasing its capacity to adapt to sophisticated software systems and offering automated support in guaranteeing software stability and security.

2.7 Summary

This chapter introduced the foundational concepts and methods that underpin the research presented in this thesis. It began with an overview of interval analysis, including definitions, arithmetic, and set operations relevant to interval computations. The role of Constraint Satisfaction Problems (CSPs) and interval contractors in narrowing variable domains was detailed, forming the mathematical basis for subsequent applications in verification.

The chapter also examined essential verification properties, such as soundness and completeness, which are critical for evaluating the reliability and exhaustiveness of verification methods. Fuzzing and Bounded Model Checking (BMC) were discussed as dynamic and semi-formal techniques respectively, highlighting their strengths and limitations in detecting software errors.

The final section focused on Abstract Interpretation, a powerful static analysis framework that enables reasoning over program properties without execution. Various abstraction techniques were introduced—such as sign abstraction, interval abstraction, and convex polyhedra—each offering trade-offs between precision and scalability. The use of Abstract Interpretation in detecting runtime errors and ensuring safety properties was also illustrated.

In summary, while Abstract Interpretation offers a scalable and mathematically grounded approach to program analysis, it is not without challenges. Balancing precision and performance, particularly in large or complex systems, remains an active area of research. Overall, this chapter established the theoretical and practical groundwork necessary to understand the integration of interval-based methods into software verification frameworks, as explored in the subsequent chapters.

Chapter 3

Contractors in Fuzzing

3.1 Introduction

In Test-Comp 2022 [148], cooperative verification tools showed their strength by being the best tools in each category. *FuSeBMC* [7], [8] is a test-generation tool that employs cooperative verification using fuzzing and BMC. *FuSeBMC* starts with the analysis to instrument the Program Under Test (PUT); then, based on the results from BMC/AFL, it generates the initial seeds for the fuzzer. Finally, *FuSeBMC* keeps track of the goals covered and updates the seeds, while producing test cases using BMC/Fuzzing/Selective fuzzer. In 2023, we introduce abstract interpretation to *FuSeBMC* to improve the test case generation. In particular, we use interval methods to help our instrumentation and fuzzing by providing intervals to help reach (instrumented) goals faster. The selective fuzzer is a crucial component of *FuSeBMC*, which generates test cases for uncovered goals based on information obtained from test cases produced by BMC/fuzzer [7]. This work is based on our previous study, where CSP/CP by contractor techniques are applied to prune the state-space search [149]. Our approach also uses Frama-C [146], [150] to obtain variable intervals, further pruning the state space exploration. Our original contributions are: (1) improve instrumentation to allow abstract interpretation to provide information about variable intervals; (2) apply interval methods to improve the fuzzing and produce higher impact test cases by pruning the search space exploration; (3) reduce the usage of resources (incl. memory and CPU time).

Fuzzing, also known as fuzz testing, is a widely used technique in software testing aimed at uncovering vulnerabilities and bugs by generating and injecting massive amounts of random data into the target program. The primary goal of fuzzing is to cause unexpected behaviour, such as crashes or security vulnerabilities, that might not be detected through conventional testing methods. Fuzzing faces significant challenges despite its effectiveness, particularly

when dealing with large and complex input spaces. The sheer volume of potential inputs can make it difficult to achieve comprehensive coverage, and the presence of complex conditional logic can lead to areas of the codebase being insufficiently tested.

To address these challenges, this chapter explores the use of contractors—powerful interval-based methods typically employed in constraint satisfaction problems (CSPs)—within the fuzzing process. Contractors help reduce the search space by applying constraints to intervals, thereby focusing the fuzzing effort on the most promising areas of the input space. This chapter delves into the theoretical underpinnings of contractors, their practical implementation in fuzzing, and the impact of this integration on the efficiency and effectiveness of fuzzing.

3.2 Methodology

To utilise contractors in the context of fuzzing, we need to combine the interval analysis component of abstract interpretation with fuzzing to help generate test cases. Interval analysis estimates the possible values that program variables can take during execution. Abstract interpretation allows for approximating program semantics, inferring properties that hold across all potential executions. Interval analysis focuses on representing variable values as intervals, defining the range within which a variable may reside. This approach leads to a more targeted exploration of the input space, potentially resulting in more efficient and effective test case generation.

To integrate interval analysis into fuzzing, the Program Under Test (PUT) is first instrumented to allow the use of abstract interpretation tools. This analysis identifies relevant variable intervals associated with specific testing goals, such as reaching particular branches or conditions within the code. These intervals are then used to narrow the input space that the fuzzer will explore, as opposed to traditional fuzzing techniques that rely on random input generation.

Contractor programming techniques are applied to refine the intervals obtained from the analysis further. Contractors are algorithms designed to reduce the domain of possible variable values by applying constraints, ensuring that the input space considered during fuzzing is more focused. The Forward-Backward contractor is one technique that iteratively refines

intervals by evaluating constraints in both forward and backward directions. This process excludes input values that do not satisfy the program's conditions.

A Constraint Satisfaction Problem (CSP)(cf. Definition 16) is constructed for each identified goal within the PUT, incorporating the relevant conditions, variables, and domains. Contractors are then applied to these CSPs, further refining the input intervals and producing a set of focused intervals for the fuzzer to explore. This approach directs the fuzzing process to consider inputs within these specified intervals, potentially influencing the execution paths related to the goals.

By incorporating interval analysis and contractor programming into fuzzing, the intervals derived from abstract interpretation guide the generation of test cases. This approach aims to manage the computational resources required for fuzzing while exploring the input space more directly.

3.3 Design and Implementation

This section presents both the design architecture and implementation details of the proposed contractor-based interval methods integrated within *FuSeBMC*. First, the high-level design of the enhanced system is outlined, providing a conceptual understanding of how contractors are incorporated into the fuzzing workflow. Following that, the technical implementation details are discussed, elaborating on the concrete steps taken to realise the design in practice.

3.3.1 Design Overview

FuSeBMC_IA improves the original *FuSeBMC* using Interval Analysis and Methods [139]. Fig. 3.1 illustrates the *FuSeBMC_IA*'s architecture. For an architectural overview of *FuSeBMC* and its extension in this work, refer to Appendix B. Our approach starts from the analysis phase of *FuSeBMC* [7], [8]. It parses statement conditions required to reach a goal, to construct a Constraint Satisfaction Problem/Constraint Programming (CSP/CP) [80] with three components: constraints (program conditions), variables (used in a condition), and domains (provided by the static analyzer Frama-C via eva plugin [151]). We instrument the PUT with

Frama-C intrinsic functions to obtain the domains, which generate intervals of a given set of variables at a specific program location. Then, we apply the contractor to each goal's CSP and output the results to a file used by the selective fuzzer.

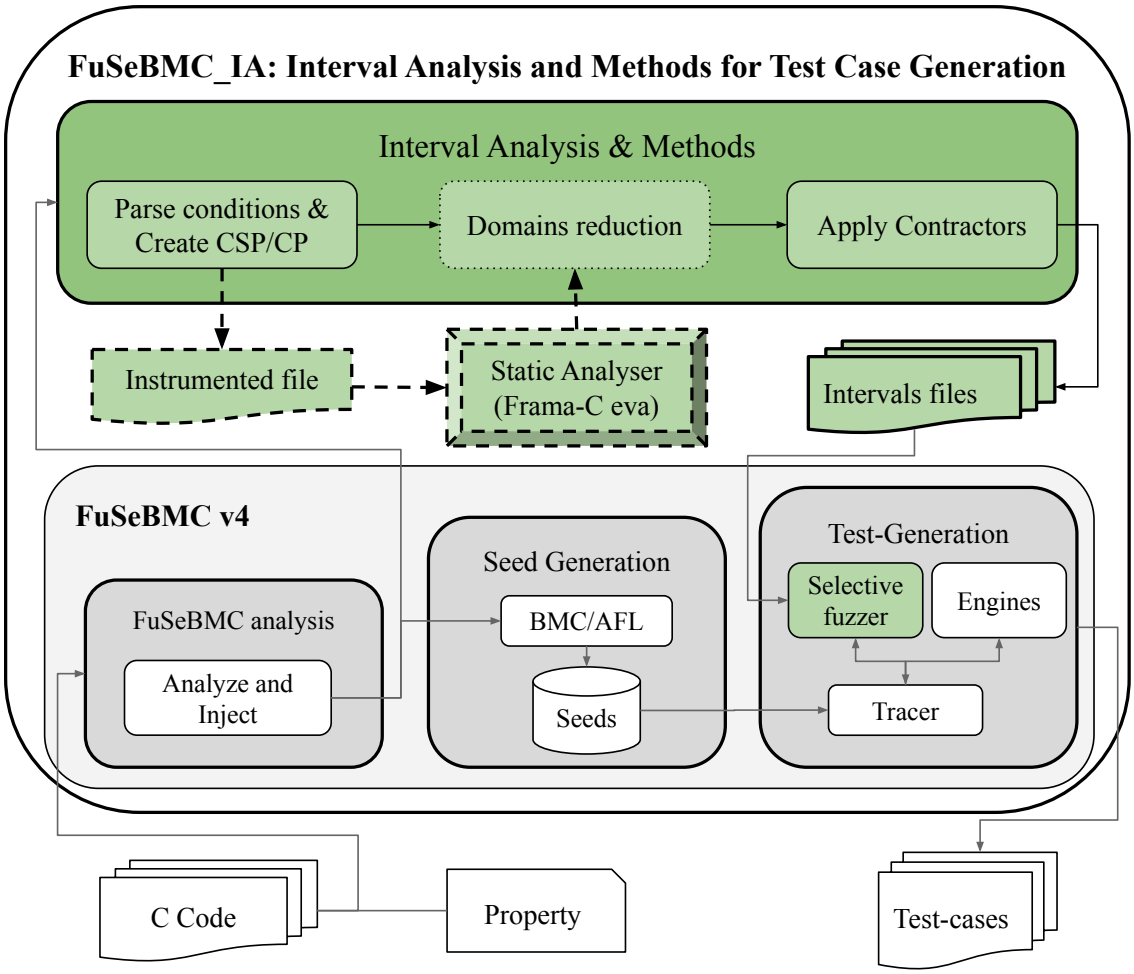


Fig. 3.1. *FuSeBMC_IA*'s architecture. The changes introduced in *FuSeBMC_IA* for Test-Comp 2023 are highlighted in green. The new Interval Analysis & Methods component generates intervals to be used by the selective fuzzer.

Contractor Programming is a set of interval methods that estimate the solution of a given CSP [80]. The used contractor technique is the Forward-Backward contractor, which is applied to a CSP/CP with a single constraint [139], which is implemented in the IBEX library [152]. IBEX is a C++ library for constraint processing over real numbers that implement contractors. More details regarding contractors can be found in our current work-in-progress [149].

3.3.2 Implementation Details

3.3.2.1 Parsing Conditions and CSP/CP creation for each goal.

While traversing the PUT clang AST [153], we consider each statement's conditions that lead to an injected goal: the conditions are parsed and converted from Clang expression [153] to IBEX expression [152]. The converted expressions are used as the constraints in CSP/CP when creating a contractor. After parsing the goals, we have a CSP/CP for each goal. If a goal does not have a CSP/CP, the intervals for the variables are left unchanged. We also create a constraint for each condition in case of multiple conditions and take the intersection/union. At the end of this phase, we have a list of each goal and its contractor. Also, a list of variables for each contractor will be used to instrument the Frama-C file in the next phase.

```
1   int main() {
2   fuSeBMC_init:;
3     int x = __VERIFIER_nondet_int();
4     int y = 0;
5     if (x <= y) {
6     GOAL_1:;
7       x++;
8     }
9     if (x >= y) {
10      if (x <= 0) {
11      GOAL_2:;
12        x = y;
13      }
14    }
15    if (x > 1 && x < -1) {
16    GOAL_3:;
17      y++;
18    }
19    return 0;
20  }
```

Instrumented file

```
1   int main() {
2     int x = __VERIFIER_nondet_int();
3     int y = 0;
4     if (x <= y) {
5       Frama_C_show_each_GOAL_1_2_(x, y);
6       x++;
7     }
8     if (x >= y)
9       if (x <= 0) {
10        Frama_C_show_each_GOAL_2_2_(x, y);
11        x = y;
12      }
13    if (x > 1 && x < -1) {
14      Frama_C_show_each_GOAL_3_1_(x);
15      y++;
16    }
17    return 0;
18  }
```

Instrumented file for Frama-C

```
1   Goal 1:
2   x
3   -2147483648.000000
4   0.000000
5   y
6   0.000000
7   0.000000
8   Goal 2:
9   x
10  0.000000
11  0.000000
12  y
13  0.000000
14  0.000000
15  Goal 3:
16  Unreachable
```

Intervals file

Fig. 3.2. The figure illustrates an example of files produced. We are starting from the instrumented file that shows the goals injected. Then, we instrument the file with the Frama-C intrinsic function. Finally, we produce a file with each goal and the intervals to satisfy the conditions for each goal.

3.3.2.2 Domains reduction.

In this step, we attempt to reduce the domains (primarily starting from $(-\infty, \infty)$) to a smaller range. This is done via Frama-C eva plugin (evolved value analysis) [151]. First, during the instrumentation, we make an instrumented file aimed to be used by Frama-C using its intrinsic functions `Frama_c_show_each()` (cf. Fig. 3.2). This function allows us to add custom text to identify goals and how many variables are in each call. Second, we run Frama-C to obtain the new variable intervals. Finally, we update the domains for the corresponding CSP/CP.

3.3.2.3 Applying contractors.

Contractors will help prune the domains of the variables by removing a subset of the domain that is guaranteed not to satisfy the constraints. With all the components for a CSP/CP available, we now apply the contractor for each goal and produce the output file in Figure 3.2. The result will be split per goal into two categories. The first category lists each variable and the possible intervals (lower bound followed by upper bound) to enter the condition given. The second category contains unreachable goals, i.e. when the contractor result is an empty vector.

3.3.2.4 Selective Fuzzer.

The Selective Fuzzer parses the file produced by the analyser, extracts all the intervals, applies these intervals to each goal, and starts fuzzing within the given interval. Thus, it prunes the search space from random intervals to informed intervals. The selective fuzzer will also prioritise the goals with smaller intervals and set a low priority to goals with unreachable results.

3.4 Evaluation

3.4.1 Objectives

This chapter defines the experimental goals of integrating interval-based contractors into fuzzing-based verification, specifically within the FuSeBMC framework. These goals are derived from the research questions presented in Section 1.4, with a particular focus on evaluating the effectiveness and efficiency of the proposed approach.

The evaluation in this chapter is designed to address the following two research questions:

RQ1 : To what extent can integrating a numerical method into verification techniques effectively reduce the search space?

RQ2 : To what extent can the integration of Interval Methods—particularly contractors—into verification tools reduce computational resource consumption, such as memory and processing time?

RQ3, which concerns the preservation of soundness and completeness, is not addressed in this chapter. This decision is based on the inherent limitations of the Test-Comp benchmarking framework, which does not penalise incorrect verification results nor explicitly report false positives or false negatives. As a result, it is not possible to formally or empirically assess the soundness and completeness of the verification process based on Test-Comp outcomes alone. This issue will be revisited qualitatively in later chapters, where broader implications and limitations are discussed.

Accordingly, the experimental goals for this chapter are:

EG1 **Search Space Reduction Efficiency:** To evaluate the extent to which interval-based contractors can reduce the search space in fuzzing-based verification, and whether this reduction is achieved efficiently.

EG2 **Resource Consumption Impact:** To assess the impact of contractor integration on computational resource usage—specifically CPU time and memory consumption—when applied within the FuSeBMC framework.

3.4.2 Description of Benchmarks

To evaluate the performance of *FuSeBMC_IA*, we analysed the final results from Test-Comp 2023 [148] and compared them to those achieved by *FuSeBMC* v4 during the same competition. Test-Comp is an internationally recognized software testing competition where tools compete in automated test-case generation. The competition categorizes all test-case generation tasks into two primary groups: Cover-Branches and Cover-Error.

The Cover-Branches category focuses on maximizing branch coverage within a given C program by generating a comprehensive set of test cases. In contrast, the Cover-Error category

requires participants to produce a test case capable of triggering a predefined error location (i.e., an explicitly marked error function) within the program.

Performance in the Cover-Branches category is evaluated using the TestCov tool [154], which assigns a coverage score between 0 and 1 for each task. For instance, achieving 65% branch coverage on a given task results in a score of 0.65. Subcategory scores are then computed by summing the scores for all tasks within that subcategory and rounding the total. Tools are awarded a binary score in the Cover-Error category: 1 for successfully reaching the error function and 0 otherwise.

Each category is further subdivided into multiple subcategories, which are organized based on key program features or the origin of the program. The majority of programs in Test-Comp originate from SV-COMP [155], the largest and most diverse open-source repository for software verification tasks. This repository includes both hand-crafted and real-world C programs, encompassing a variety of features such as loops, arrays, bit-vectors, floating-point operations, dynamic memory allocation, recursive functions, event-condition action systems, concurrent programming, and BusyBox2 software. For detailed information about the benchmark structure and scoring methodology used in this evaluation, refer to Appendix A.

3.4.3 Setup

The evaluations for Test-Comp 2023 were performed on servers equipped with an 8-core (4 physical cores) Intel Xeon E3-1230 v5 CPU running at 3.4 GHz, 33 GB of RAM, and operating on x86-64 Ubuntu 20.04 with Linux kernel 5.4. Each test suite generation task was constrained to 8 CPU cores, 15 GB of RAM, and a maximum of 15 minutes of CPU time. FuSeBMC allocated its computational resources across its engines based on predefined time distributions, which were adjusted in 2023.

For benchmarks in both categories, 20 seconds were assigned to seed generation. The fuzzer was allocated 200 seconds for Cover-Error benchmarks and 250 seconds for Cover-Branches benchmarks. The bounded model checker received 650 seconds for Cover-Error tasks and 600 seconds for Cover-Branches tasks. Additionally, the time allocated to the selective fuzzer was reduced to 30 seconds compared to its allocation in the previous year.

In the BMC evaluation, we executed ESBMC using the same configuration options em-

ployed in SV-COMP 2024, with the only variation being the strategy selected (i.e., k-induction or incremental-bmc). The following options were used: `--incremental-bmc` to enable incremental BMC, `--k-induction` to activate k-induction, `--unlimited-k-steps` to remove the upper iteration limit for the incremental BMC algorithm, and `--interval-analysis-ibex-contractor` to enable interval analysis (ESBMC's abstract interpretation engine) and apply the contractor-based method.

The results are presented in terms of testing time and scores. All execution times reported are CPU times, representing only the periods during which the allocated CPUs were actively engaged. Memory consumption was measured as the amount of RAM occupied by the testing process, excluding swapped or non-resident memory. Measurements for CPU time and memory usage were performed using the *benchexec* tool [156], with detailed command configurations available on the supplementary page[1]. Swapping and turbo boost were disabled during the experiments to ensure consistency, and all tools were restricted to a single CPU core.

3.4.4 Results

Regarding EG1, Table 3.1 provides a comprehensive overview of the tools' performance in Test-Comp 2023, highlighting their scores and CPU times. Notably, *FuSeBMC_IA* demonstrated significantly lower CPU time compared to *FuSeBMC*, achieving an overall reduction of 35%. However, this improvement came at the cost of a 3% decrease in the score.

Examining individual categories in Table 3.4, *FuSeBMC_IA* maintained the same score and CPU time in subcategories such as Bitvector, Controlflow, Heap, ProductLines, XCSP, and DeviceDriversLinux64. Conversely, its performance declined in Loops, where it achieved a lower score with the same CPU time, and in Recursive, where the score remained unchanged but CPU time increased. In other subcategories, such as Arrays, ECA, Floats, and Sequentialized, *FuSeBMC_IA* managed to reduce CPU time but occasionally lost 1–2 points. Remarkably, in the Hardware category, *FuSeBMC_IA* achieved the same score while consuming less than half the CPU time compared to *FuSeBMC*.

Similarly, the results for Cover-branches, detailed in Table 3.5, exhibit a consistent trend. In categories like Arrays, Bitvectors, ControlFlow, Floats, Heaps, Loops, Recursive, and Se-

[1] https://Test-Comp.sosy-lab.org/2023/results/results-verified/

Table 3.1. Test-Comp 2023 overall results. With *FuSeBMC* in first place, VeriFuzz is second, and *FuSeBMC_IA* in third. Note on meta-categories: The score is not the sum of scores of the sub-categories (normalization). The run time is the sum of the run times of the sub-categories, rounded to two significant digits.

Participants	Cover-Error 1173 tasks		Cover-Branches 2933 tasks		Overall 4106 tasks	
	Score	CPU time	Score	CPU time	Score	CPU time
CoVeriTest [157], [158]	581	120000 s	1509	1700000 s	2073	1800000 s
ESBMC-kind [159], [160]	289	3100 s				
FuSeBMC [7], [8]	**936**	260000 s	**1678**	2600000 s	**2813**	2800000 s
FuSeBMC_IA [6], [161]	**908**	**130000 s**	**1538**	**1700000 s**	**2666**	**1800000 s**
HybridTiger [162], [163]	463	240000 s	1170	1600000 s	1629	1900000 s
KLEE [164], [165]	721	10000 s	999	990000 s	1961	1000000 s
Legion [166], [167]			838	2300000 s		
Legion/SymCC [167]	349	2700 s	1027	2500000 s	1329	2500000 s
PRTest [168], [169]	222	240000 s	770	2400000 s	927	2600000 s
Symbiotic [170], [171]	644	20000 s	1430	1600000 s	2128	1600000 s
TracerX [172], [173]			1400	780000 s		
VeriFuzz [174]	**909**	16000 s	**1546**	2600000 s	**2673**	2600000 s
WASP-C [175]	570	9300 s	1103	1100000 s	1770	1100000 s

quentialized, *FuSeBMC_IA* experienced slight score reductions while significantly reducing CPU time. Meanwhile, categories such as ECA, ProductLines, XCSP, SQLite, MainHeap, and DeviceDriversLinux64 maintained their scores with reduced CPU time. Notably, when compared to Verifuzz, the second-place competitor, *FuSeBMC_IA* emerged as the most efficient in CPU time among the podium finishers.

Regarding EG2, Table 3.2 presents the differences in CPU time and score between the methods. The *FuSeBMC_IA* approach reduced the score by 2.99% in cover-error and 8.34% in cover-branches, resulting in an overall score reduction of 5.23%. However, *FuSeBMC_IA* significantly decreased CPU time, achieving a 50% reduction in cover-error and 34.62% in cover-branches, culminating in an overall CPU time reduction of 35.71%.

To evaluate efficiency, we compared the number of points processed per hour for each method, as shown in Table 3.2. The *FuSeBMC* method achieved 3.62 points per hour, while *FuSeBMC_IA* reached 5.332 points per hour—a 47.43% improvement. This substantial in-

EG1 **(Efficiency)** Overall, the results show that *FuSeBMC_IA* consistently reduced CPU time compared to *FuSeBMC*, achieving up to a 35% improvement while experiencing only a minor decrease (about 3%) in the overall score. In many categories, *FuSeBMC_IA* maintained comparable scores but significantly lowered CPU time, in some cases by half—for example, in the Hardware category. Even where small score reductions occurred (1–2 points), they were generally accompanied by substantial gains in CPU efficiency. Furthermore, *FuSeBMC_IA* outperformed other leading tools, such as VeriFuzz, in terms of CPU usage.

While reduced CPU time indicates improved efficiency, it does not alone constitute definitive evidence that the search space was pruned. However, through close manual inspection of selected benchmarks, we observed clear instances where the application of contractors led to constrained variable domains and the elimination of infeasible paths. These observations support the interpretation that contractors contribute to effective search space reduction.

Therefore, based on both performance metrics and benchmark analysis, we conclude that contractors help to prune the search space in a way that reduces computational effort.

crease in processing efficiency demonstrates that the trade-off in the score is justified, making *FuSeBMC_IA* a more effective choice overall.

Regarding energy consumption, *FuSeBMC_IA* showed excellent resource optimisation. Table 3.3 shows the podium winners and compares their energy consumption along with scores and CPU time. We notice that *FuSeBMC_IA* requires significantly less CPU time and energy than both *FuSeBMC* and VeriFuzz. Therefore, *FuSeBMC_IA* is an environmentally friendly and economically viable alternative. Moreover, it attains a CPU time reduction of 37% and energy consumption reduction of 30% compared to VeriFuzz, and a reduction of 58% and 35%, respectively, compared to *FuSeBMC*. This efficiency is paramount in large-scale or continuous testing scenarios where computational costs and environmental impact are among the key concerns.

FuSeBMC_IA provides a sustainable solution for organizations looking to achieve high-

Table 3.2. This table compares *FuSeBMC* with *FuSeBMC_IA* in terms of score and CPU time in seconds. It also shows the increase percentages where *FuSeBMC_IA* was less than *FuSeBMC* in Cover-Error by 2.99% in terms of score and 50% in terms of CPU time. The table also shows the increase in points per hour in each tool and the percentage increase by 47%.

Category		FuSeBMC	FuSeBMC_IA	% Increase
Cover-Error	$Score$	936	908	-2.99%
	$Time(s)$	260000	130000	-50%
Cover-Branches	$Score$	1678	1538	-8.34%
	$Time(s)$	2600000	1700000	-34.62%
Overall	$Score$	2813	2666	-5.23%
	$Time(s)$	2800000	1800000	-35.71%
Points per hour		3.62	5.332	47.43%

Table 3.3. Overview of the top-three test generators for each category (measurement values for CPU time and energy rounded to two significant digits) [176].

Rank	Tester	Score	CPU Time (h)	CPU Energy (kWh)
		Cover-Error		
1	*FuSeBMC*	**936**	72	0.96
2	VeriFuzz	909	**4.5**	**0.049**
3	*FuSeBMC_IA*	908	37	0.48
		Cover-Branches		
1	*FuSeBMC*	**1678**	720	9.2
2	VeriFuzz	1546	730	9.1
3	*FuSeBMC_IA*	1538	**470**	**6.0**
		Overall		
1	*FuSeBMC*	**2813**	790	10
2	VeriFuzz	2673	730	9.2
3	*FuSeBMC_IA*	2666	**500**	**6.5**

quality test generation and sustainability in their practices. Strong performance with minimal use of resources makes it stand out when an organisation's priorities are efficiency and speed. Many researchers and pioneers have expressed the importance of speed in software testing in the industry [177]–[179]. By adopting *FuSeBMC_IA*, an organization can provide reliable outcomes of testing activities while addressing sustainable and cost-conscious development practices.

EG2 (Trade-off) *FuSeBMC_IA* has a great balance between performance and resource consumption. Where it involves a small sacrifice in terms of score, it hugely reduces CPU time and power consumption, leading to nearly 30% better energy efficiency compared to its counterparts. Together with this improvement of 47.43% in processing power, the placement of *FuSeBMC_IA* is sound for widespread testing setups—both on grounds of economics and being eco-friendly.

3.5 Strengths and Weaknesses

Using abstract interpretation in *FuSeBMC_IA* improved the test-case generation regarding resources. Our selective fuzzer uses the new contractors generated by the Interval Analysis and Methods component: (1) the information provided helps the selective fuzzer to start from a given range of values rather than a random range (as was our strategy in the previous version); (2) the selective fuzzer uses the information about unreachable goals to set their priority low for reachability; (3) when compared to *FuSeBMC* v4, this improvement helped saving CPU time by 37% and memory by 13%, which leads to saving 40% of energy; (4) although our approach produces fewer test cases for a given category, the impact of these test cases is higher in terms of reaching instrumented goals; (5) there is potential for future work to use the information provided by Frama-C, especially regarding overflow warnings. Finally, the intervals provided may not affect the *FuSeBMC_IA*'s outcome in the worst case. i.e., the selective fuzzer performs no better than not having interval information for seed generation. The time it takes to generate the intervals is only a tiny fraction of the time it takes to produce the test cases; its impact when the information is not useful is negligible.

Our approach suffers from a significant technical limitation: *FuSeBMC_IA* cannot create complementary contractors; we can only create intervals that satisfy the constraints of a branch (i.e., outer contractors). In practice, we can only create intervals to `if`-statements and ignore its `else`-statements (the inner contractor). We also skip any `if`-statement inside `else`-statements, as this may lead to unsound intervals. This is a technical limitation rather than a theoretical one: we use run-time type information (RTTI) to identify ibex expressions. However, we link our tool with Clang, which requires compilation with no RTTI information. We are investigating approaches to address this limitation, e.g., to encapsulate all ibex

expressions and manually store expression information, but currently, no proper fix has been implemented. Additionally, a bug has been found that caused *FuSeBMC_IA* to crash on some benchmarks that made *FuSeBMC_IA* scores much less than *FuSeBMC* in the coverage category.

3.6 Conclusion

This chapter presents a method based on contractors to improve fuzzing-based test generation by integrating interval analysis and contractor programming techniques. Embedding interval-based abstract interpretation in the fuzzing process allows for a directed search toward more promising input domains, significantly reducing the computational resources needed to generate test cases. The introduction of contractors traditionally used in constraint satisfaction problems—allowed for more efficient pruning of the input space, improving resource usage and shortening execution time.

Empirical assessments, like those performed as part of Test-Comp 2023 [148], [176], indicated this could substantially raise CPU utilization and reduce memory consumption without compromising the thoroughness of test coverage. The approach showed substantial efficiency gains and considerable improvement in energy consumption despite minor trade-offs, like slightly lower coverage scores for some categories. Moreover, using tools like Frama-C [146], [150], which provide variable intervals via the eva plugin [151], helped ensure that obtained intervals were reliable and feasible, leading to more effective test cases. By constraining the search space and focusing on input values that are more likely to expose previously unexplored execution paths, the selective fuzzer increased the effectiveness of fuzzing efforts.

While the implementation described in Chapter 3 is specifically built on C programs using Frama-C for instrumentation, the underlying methodology of applying interval contractors to refine input domains prior to fuzzing is generalisable to other languages and frameworks. The core requirements for applying the approach are: (1) the ability to extract guard conditions or symbolic constraints from source code, and (2) a mechanism to represent input domains in a form amenable to interval analysis. Many modern analysis frameworks in languages such as C++, Rust, and Java support intermediate representations (e.g., LLVM IR, Java bytecode) that expose program control-flow and constraints. By adapting the parsing and CSP construction phases to these representations, the interval contractor integration could be extended beyond

C/Frama-C environments. However, the efficiency and precision of the method may depend on the richness and precision of the intermediate representation used in the target system.

In the current implementation of *FuSeBMC_IA*, no external sanitisation tools such as AddressSanitizer (ASan) or UndefinedBehaviourSanitizer (UBSan) are integrated into the fuzzing or verification pipeline. The detection of vulnerabilities relies primarily on assertion violations, memory model checking, and symbolic analysis conducted by the underlying bounded model checking and static analysis components. While sanitisation frameworks could potentially augment the detection of runtime errors such as buffer overflows or undefined behaviours, integrating them was considered out of scope for the primary goal of this research, which was focused on reducing search space and improving test input precision through interval-based domain contraction. Nevertheless, incorporating sanitisation tools could be a promising extension for future work, particularly to enhance runtime bug detection capabilities.

Technical challenges remain, for example, in synthesising complementary contractors and handling complex branching structures. Future work will investigate improvements in the treatment of `else`-statements and nested conditions to increase the applicability of interval-based pruning. Moreover, exploiting more advanced static analysis information—for instance, overflow warnings—offers good prospects for further optimization of test generation strategies. In other words, combining fuzzing, bounded model checking, abstract interpretation, and contractor programming leads to a possibly powerful framework for more effective and efficient test case generation. This research is, therefore, expected to continue to yield such hybrid methodologies with a significant impact on automated test generation and the development of more safe and reliable software systems.

Table 3.4. Note on meta-categories: The score is not the sum of scores of the sub-categories (normalization). The run time is the sum of run times of the sub-categories, rounded to two significant digits.

Participants $\frac{score}{CPUtime(s)}$	**# tasks**	**CoVeriTest**	**ESBMC-kind**	**FuSeBMC**	**FuSeBMC_IA**	**HybridTiger**	**KLEE**	**Legion**	**Legion/SymCC**	**PRTest**	**Symbiotic**	**TracerX**	**VeriFuzz**	**WASP-C**
RS-Arrays	90	71 12000	5 44	90 15000	88 13000	67 40000	85 110	67 60000	19 17	36 5000	72 630	0 0	88 510	78 410
RS-BitVectors	9	7 250	0 0	9 880	9 880	5 1400	8 5.5	1 300	3 890	5 4500	7 88	0 0	9 340	7 500
RS-ControlFlow	5	1 910	0 0	5 450	5 460	0 0	4 1.9	0 0	3 480	0 0	5 140	0 0	4 330	4 97
RS-ECA	18	3 390	0 0	12 2100	11 2000	1 900	14 490	0 0	1 1.3	0 0	13 900	0 0	14 840	2 180
RS-Floats	32	24 2800	13 96	32 4500	31 3100	22 4600	7 2.8	0 0	2 8.1	2 1800	0 0	0 0	31 420	22 2800
RS-Heap	47	42 13000	6 1	45 1800	45 1800	37 9100	45 170	3 18	42 690	11 9900	45 190	0 0	45 260	36 760
RS-Loops	130	64 36000	70 910	128 28000	127 28000	47 30000	82 3400	3 2700	34 70	89 80000	71 2700	0 0	123 2400	96 770
RS-ProductLines	169	160 3900	169 380	169 1000	169 1000	45 41000	169 82	34 210	159 360	92 83000	159 3100	0 0	169 1300	5 37
RS-Recursive	20	7 3100	0 0	19 870	19 1300	5 680	16 8.8	0 0	16 150	1 900	17 230	0 0	18 330	12 41
RS-Sequentialized	98	55 4300	40 60	94 11000	92 9900	86 75000	78 3600	0 0	0 0	0 0	71 4000	0 0	95 1000	43 810
RS-XCSP	54	45 2200	49 1600	47 1200	47 1200	47 22000	32 2500	0 0	0 0	0 0	16 6700	0 0	49 1100	49 1800
RS-Hardware	494	85 11000	0 0	288 190000	288 69000	35 20000	0 0	0 0	1 1.2	57 51000	34 930	0 0	319 7400	39 1100
SS-BusyBox-MemSafety	5	0 0	0 0	1 280	0 0	0 0	1 1.8	0 0	0 0	0 0	0 0	0 0	0 0	0 0
SS-DeviceDriversLinux64	2	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
Cover-Error	1173	581 120000	289 3100	936 260000	908 130000	463 240000	721 10000		349 2700	222 240000	644 20000		909 16000	570 9300

Table 3.5. Note on meta-categories: The score is not the sum of scores of the sub-categories (normalization). The run time is the sum of run times of the sub-categories, rounded to two significant digits.

Participants $\frac{score}{CPUtime(s)}$	**# tasks**	**CoVeriTest**	**ESBMC-kind**	**FuSeBMC**	**FuSeBMC_IA**	**HybridTiger**	**KLEE**	**Legion**	**Legion/SymCC**	**PRTest**	**Symbiotic**	**TracerX**	**VeriFuzz**	**WASP-C**
RS-Arrays	292	208 240000s	0 0	253 260000s	249 170000s	202 240000s	73 34000s	171 210000s	99 250000s	127 110000s	204 230000s	200 81000s	238 250000s	191 140000s
RS-BitVectors	61	48 20000s	0 0	48 55000s	42 28000s	16 7400s	31 22000s	33 51000s	44 40000s	33 54000s	48 36000s	48 33000s	45 55000s	34 29000s
RS-ControlFlow	11	0 7000s	0 0	2 9200s	1 3800s	0 5800s	1 800s	0 8100s	1 9900s	0 2900s	1 1400s	1 14s	1 9000s	1 1200s
RS-ECA	29	5 24000s	0 0	9 26000s	9 24000s	2 23000s	7 26000s	3 21000s	3 19000s	2 26000s	9 21000s	7 14000s	10 25000s	4 24000s
RS-Floats	197	92 63000s	0 0	100 180000s	96 71000s	76 62000s	16 11000s	54 160000s	48 170000s	40 180000s	47 6300s	45 490s	98 180000s	53 26000s
RS-Heap	110	76 75000s	0 0	80 98000s	79 54000s	63 47000s	74 65000s	61 80000s	77 95000s	34 57000s	74 69000s	77 12000s	78 99000s	63 39000s
RS-Loops	661	526 420000s	0 0	543 600000s	534 300000s	431 360000s	356 190000s	319 500000s	242 570000s	325 590000s	502 260000s	504 160000s	538 590000s	381 170000s
RS-ProductLines	263	76 61000s	0 0	77 200000s	77 170000s	55 220000s	74 9800s	70 190000s	73 160000s	48 240000s	69 5500s	77 10000s	77 240000s	1 1800s
RS-Recursive	51	40 46000s	0 0	44 46000s	43 25000s	37 27000s	21 21000s	26 20000s	39 46000s	10 14000s	44 41000s	39 15000s	39 43000s	37 27000s
RS-Sequentialized	91	73 74000s	0 0	80 82000s	71 31000s	53 71000s	28 15000s	1 82000s	1 82000s	10 82000s	46 32000s	48 25000s	80 82000s	57 35000s
RS-XCSP	114	114 5600s	0 0	105 100000s	105 40000s	114 5800s	101 17000s	2 100000s	73 100000s	101 100000s	107 77000s	90 16000s	106 100000s	76 16000s
RS-Combinations	671	233 550000s	0 0	340 600000s	305 490000s	167 500000s	197 470000s	182 530000s	219 580000s	79 600000s	285 500000s	277 360000s	252 600000s	152 520000s
SS-BusyBox-MemSafety	62	10 55000s	0 0	21 52000s	1 430s	5 42000s	18 50000s	0 56000s	0 56000s	13 54000s	12 34000s	14 27000s	0 56000s	0 610s
SS-DeviceDriversLinux64	287	59 36000s	0 0	59 260000s	59 250000s	6 24000s	24 59000s	54 260000s	53 260000s	16 240000s	42 230000s	56 20000s	56 260000s	0 1600s
SS-SQLite-MemSafety	1	0 900s	0 0	0 900s	0 0.35s	0 900s	0 28s	0 900s	0 810s	0 900s	0 3.0s	0 46s	0 900s	0 170s
Termination-MainHeap	32	31 7600s	0 0	30 29000s	30 17000s	27 7200s	16 53s	23 26000s	27 22000s	11 11000s	29 27000s	31 10000s	29 27000s	29 23000s
Cover-Branches	2933	1509 1700000s	0 0	1678 2600000s	1538 1700000s	1170 1600000s	999 990000s	838 2300000s	1027 2500000s	770 2400000s	1430 1600000s	1400 780000s	1546 2600000s	1103 1100000s

Chapter 4

Contractors in BMC

4.1 Introduction

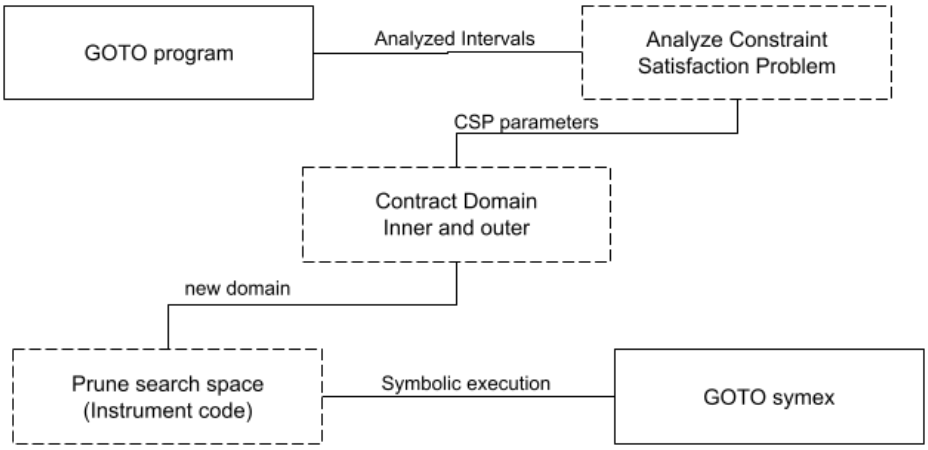


Fig. 4.1. This figure illustrates the integration of interval contractors into the bounded model checking (BMC) workflow. The process begins with parsing the C program into an intermediate representation (IR), where verification properties are extracted. Constraints relevant to these properties are then identified, forming a constraint satisfaction problem (CSP). Interval contractors are applied to narrow variable domains by removing infeasible values, effectively pruning the search space. Following contraction, the IR is updated by inserting assumptions that restrict variables to the contracted intervals. Symbolic execution proceeds over this refined program, producing verification conditions that are subsequently solved using an SMT solver. A satisfiable outcome indicates a property violation, while an unsatisfiable result confirms the property's correctness within the given bounds. By integrating contractors early in the workflow, the system achieves a more efficient and scalable verification process without compromising soundness or completeness.

Here, we introduce contractors described in Section 2.5 to BMC of software. In particular, we model the constraints and properties generated from BMC instances to a CSP [180]. A CSP has three inputs: variables (dimensions), domain, and constraints. The term "constraints" has a different meaning when dealing with BMC or CSP. So, we decided to differentiate between the two by stating whether the constraints are BMC constraints or CSP constraints. We obtain the domains by analysing the declaration of independent variables, *assume* directive and variable assignment (BMC constraints). The directive *assert*, repre-

senting the property, will give the contractor CSP constraints. Assert statements will also dictate which variables are included in our defined domains. We apply the contractors in the intermediate representation (IR) of programs. Our IR is a GOTO-program that simplifies the program representation (e.g., replaces *switch* and *while* by *if* and *goto* statements) [62], [181][1]. Fig. 4.1 shows how contractors can be used in BMC, where the *GOTO-symex* component performs the program's symbolic execution.

4.2 Approach

As an illustrative example, we consider the code fragment illustrated in Fig. 4.2a. We model this code into a CSP (cf. definition16) by having our variables x as x_1 and y as x_2. Assume the maximum value that an unsigned integer can hold, defined as Max_{uint}. Therefore, we define our intervals for each variable from the declaration of the variables. $[x_1] = [0, \text{Max}_{uint}], [x_2] = [0, \text{Max}_{uint}]$. With the *assume* directive, variable x_1 interval will be $[x_1] = [0, 20]$. With the given assertion x $\geqslant$ y, we determine the inequality used being $x_1 \geq x_2$ as our constraint. Now we have our CSP, we model our input as parameters for a contractor such that: Variables x_1, x_2; Domains, $[x_1] = [0, 20], [x_2] = [0, \text{Max}_{uint}]$; Constraint, $x_2 - x_1 \leq 0$.

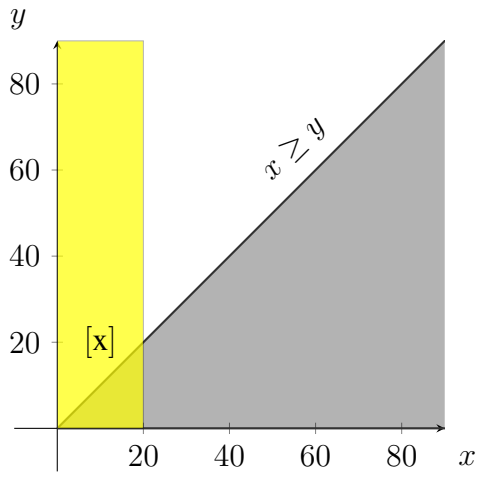
We use the *forward-backward* contractor [142] mentioned in Definition 18. After we plug our values in the equations in Fig. 4.2d, we obtain $[y] = [-20, 0]$, $[x_1] = [0, 20]$ and $[x_2] = [0, 20]$. We notice that $[x_2]$ was contracted from $[0, \text{Max}_{uint}]$ to $[0, 20]$ because that is where our solution lies. Regarding BMC, having the domain contracted by the outside contractor ensures a violation of property since that area is outside the solution $\mathbb{S}_{out}$. We can visualize the results in Fig. 4.2, where the left graph shows our initial domain, the right graph shows the contracted domain in blue, and the area remaining in yellow.

However, if we use the inner contractor, which is the complement of our constraint $f(x) > 0$, we may prune the area inside our solution $\mathbb{S}_{in}$ to possibly prune our search space and only have the $\mathbb{S}_{boundary}$ area to be checked by BMC. Consider the example in Fig. 4.3a, we form our CSP by having our variables: let x be x_1 and y be x_2, and domain be $[x_1] = [20, 40]$ and $[x_2] = [0, 70]$ and constraint as $x_1 \geq x_2$.

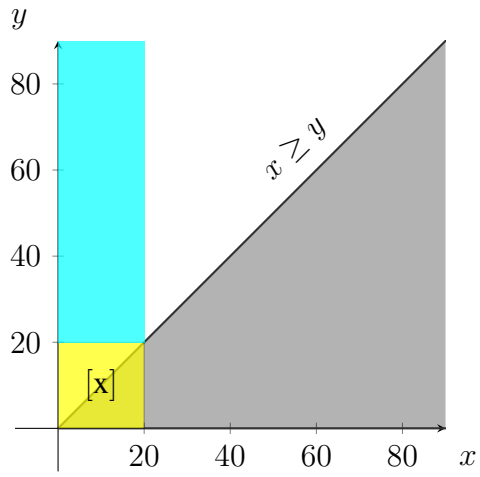
[1]Documentation of GOTO-program can be found at https://ssvlab.github.io/esbmc/documentation.html

```
1 unsigned int x=nondet_uint();
2 unsigned int y=nondet_uint();
3 __ESBMC_assume(x <= 20);
4 assert(x >= y);
```

(a) Outer contractor example.



(b) Before Outer Contractors.



(c) After Outer Contractors.

$$
\begin{array}{lll}
f(x) \leq 0 & I = (-\infty, 0] & \\
f(x) : y = x_2 - x_1 & [y] = I \cap ([x_2] - [x_1]) & \text{Forward-step} \\
f(x)^{-1}_{x_1} : x_1 = x_2 - y & [x_1] = [x_1] \cap ([x_2] - [y]) & \text{Backward-step} \\
f(x)^{-1}_{x_2} : x_2 = y + x_1 & [x_2] = [x_2] \cap ([y] + [x_1]) & \text{Backward-step}
\end{array}
$$

(d) Outer contractor steps for the example in Fig 4.2a.

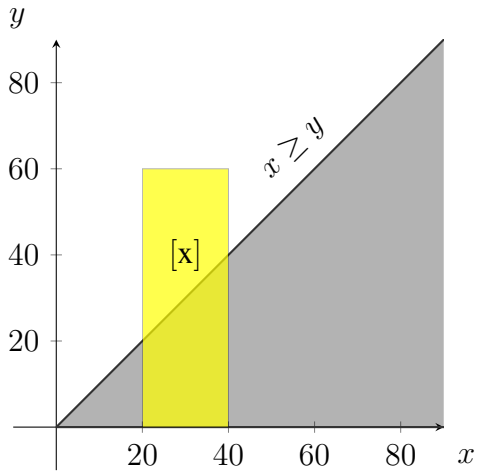
Fig. 4.2. Example illustrating outer contractor.

For the inner contractor, we list our formulas as illustrated in Fig. 4.3d. So we plug our values for $[x_1]$ and $[x_2]$; we obtain $[y] = [0, 10]$, $[x_1] = [20, 40]$ and $[x_2] = [20, 60]$. Here, $[x_2]$ was contracted from $[0, 60]$ to $[20, 60]$. To visualize the solution, in Fig. 4.3c, the red area is contracted from our domain; it represents the part of our domain that belongs to $\mathbb{S}_{in}$. For BMC, this area is guaranteed to hold the property; therefore, we only prune our search space to the orange area $\mathbb{S}_{boundary}$.

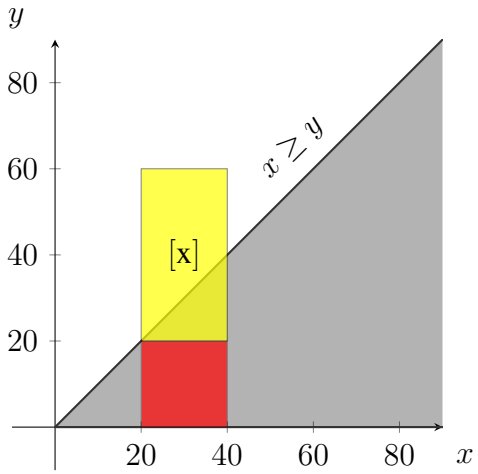
Algorithms 4 and 5 contain the main steps necessary for our approach. The first four lines describe each variable we will use throughout the process. We will analyse the variables $\mathbf{x}$ and their intervals (or domain) $[\mathbf{x}]$ and the properties (or constraints in the context of CSP) $\mathbf{f(x)} \leq \mathbf{0}$. Then, after calling $Apply\_Contractor$, we initialise the contractors based on variables and constraints. The outer contractors $\mathcal{C}_{out}$ are initialised based on the property in the form of $f(x) \leq 0$ or $f(x) = 0$, which depends on the property. In contrast, the inner

```
1 unsigned int x=nondet_uint();
2 unsigned int y=nondet_uint();
3 __ESBMC_assume(x >= 20 && x <= 30);
4 __ESBMC_assume(y <= 30);
5 assert(x >= y);
```

(a) Inner contractor example.



(b) Before inner Contractors.



(c) After inner Contractors.

$f(x) > 0$	$I = [0, \infty)$	
$f(x) : y = x_2 - x_1$	$[y] = I \cap ([x_2] - [x_1])$	Forward-step
$f(x)^{-1}_{x_1} : x_1 = x_2 - y$	$[x_1] = [x_1] \cap ([x_2] - [y])$	Backward-step
$f(x)^{-1}_{x_2} : x_2 = y + x_1$	$[x_2] = [x_2] \cap ([y] + [x_1])$	Backward-step

(d) Inner contractor steps for the example in Fig 4.3a.

Fig. 4.3. Example illustrating inner contractor.

contractor $\mathcal{C}_{in}$ is initialized in the form $f(x) > 0$ or $f(x) \neq 0$.

Algorithm 4: Proposed method algorithm.

Input: Program P

1: $\mathbf{x}$: set of Variables
2: $[\mathbf{x}]$: domain
3: $\mathbf{f(x)} \leq \mathbf{0}$: set of Constraints
4: $\mathbf{f(x)} \leq \mathbf{0} \leftarrow Analyze\_Properties(P)$
5: $[\mathbf{x}] \leftarrow Analyze\_Intervals(P)$
6: $[\mathbf{x}] \leftarrow Apply\_Contractor(\mathbf{x}, [\mathbf{x}], \mathbf{f(x)} \leq \mathbf{0})$
7: $P' \leftarrow instrument(P, [\mathbf{x}])$
8: **return** P'

With CSP components being set up, we contract our domain starting with $\mathcal{C}_{out}$, where

Algorithm 5: *Apply_Contractor*.

Input: Varibles $\mathbf{x}$, Domains $[\mathbf{x}]$, Constraints $\mathbf{f}(\mathbf{x}) \leq 0$

1: $[\mathbf{x}]' \leftarrow \mathcal{C}_{out}([\mathbf{x}],\ \mathbf{f}(\mathbf{x}))$
2: **if** $[\mathbf{x}]' = \emptyset$ **then**
3: $\quad P' \leftarrow P$
4: **else**
5: $\quad \mathbb{S}_{out} \leftarrow [\mathbf{x}] \setminus [\mathbf{x}]'$
6: $\quad [\mathbf{x}]'' \leftarrow \mathcal{C}_{in}([\mathbf{x}]',\ \mathbf{f}(\mathbf{x}))$
7: $\quad \mathbb{S}_{in} \leftarrow [\mathbf{x}]' \setminus [\mathbf{x}]''$
8: $\quad \mathbb{S}_{boundary} \leftarrow [\mathbf{x}]''$
9: **end if**
10: **return** $\mathbb{S}_{boundary} \cup \mathbb{S}_{out}$

the domain will be reduced to exclude the region outside the solution, which violates the property. If the resulting set of the domain $[\mathbf{x}]'$ is empty, it means that the domain will violate the property. There is no need to prune the search space (lines 2 and 3 of Algorithm 5). Otherwise, the difference between the result from the outer contractor $[\mathbf{x}]'$ and the original domain $[\mathbf{x}]$ will yield $\mathbb{S}_{out}$, as demonstrated in line 10 of Algorithm 4. After that, we apply the inner contractor $\mathcal{C}_{in}$ to remove the intervals inside the solution where the property holds. Similar to the outer contractors, the difference between the two sets $[\mathbf{x}]''$ and $[\mathbf{x}]'$ will yield the set inside the solution $\mathbb{S}_{in}$. Note that box difference will yield not necessarily one box but rather a finite set of boxes [80]. The resulting domain from the contractor $[\mathbf{x}]''$ will be our boundary area, where $[\mathbf{x}]'' \subseteq \mathbb{S}_{boundary}$. With $\mathbb{S}_{out}$, $\mathbb{S}_{in}$ and $\mathbb{S}_{boundary}$, now we can instrument the program for the BMC engine to search for a counterexample in $\mathbb{S}_{out}$ and $\mathbb{S}_{boundary}$ while removing $\mathbb{S}_{in}$ from the search-space.

The instrumentation will be done using the `assume(expr)` directive, where we constrain the search space indicated by the expression $expr$, which the contractors produced. Note that instrumenting `assume(expr)` directive is not a trivial task since we need to ensure where the pruned intervals are in the program context, especially regarding loops. If $\mathbb{S}_{in}$ is at the end of the loop, we add the `assume`$(\mathbb{S}_{out} \cup \mathbb{S}_{boundary})$ to remove these unnecessary loop steps in the loop. However, if $\mathbb{S}_{in}$ is at the beginning or middle of the loop, we discussed it further in Section 4.5.

Algorithm 4 describes the high-level domain application of the contractor process. Initially, the CSP components are defined in lines 1,2,3 and the set of program constraints is collected, each constraint being modelled as a relation over program variables. The contractor associated with each constraint is then applied iteratively. These contractors operate by

removing portions of the variable domains that cannot possibly satisfy the constraint, progressively refining the domains until a fixpoint is reached or a limit is reached. In implementation, contractors are instantiated using libraries such as IBEX, where standard contractors (e.g., forward-backward contractors for inequalities) are composed to form a contractor vector. Algorithm 4 thus formalises the overall framework for domain pruning prior to symbolic execution.

Algorithm 5 details the application of a single contractor to a specific variable domain. Given a current domain and a constraint, the contractor checks whether a refinement is possible based on interval arithmetic and constraint propagation rules. If so, it updates the domain accordingly; otherwise, it preserves the original domain. The key to this operation is maintaining soundness: no feasible solutions are removed. The implementation realises this by carefully distinguishing between outer and inner contractors depending on the type of constraint involved, ensuring that feasible intervals are not inadvertently excluded.

The impact of these algorithms is substantial. By integrating domain reduction prior to symbolic execution, the verification process avoids exploring infeasible program paths, significantly reducing SMT solver query sizes and improving overall verification performance.

We will use the illustrative example in Fig. 4.4 to demonstrate the application of the steps of Algorithm 4. Note that we have two variables in this program `x` and `y`, which will be x_1 and x_2 in the CSP space, respectively, where $\mathbf{x} = \{x_1, x_2\}$. For the domains, after the analysis, they will be $x_1 = [1, \text{Max}_{int}]$ and $x_2 = [0, 1000]$ with $[\mathbf{x}] = [1, \text{Max}_{int}] \times [0, 1000]$, while the constraint will be $x_1 \geq x_2$ (because the assert is `x >= y`) where $\mathbf{f}(\mathbf{x}) \leq \mathbf{0} \Rightarrow f(x) = x_2 - x_1 \leq 0$. Therefore, our constraints will be $x_2 - x_1 \leq 0$ for the outer contractor and $x_2 - x_1 > 0$ for the inner contractor. In this case, however, the outer contractor returned the same domain, which means it cannot be contracted from the outside because there are no values in the domain $[\mathbf{x}]$ guaranteed to be outside the solution, thus $\mathbb{S}_{out} = \emptyset$. Fig. 4.5 illustrates the outcome of the inner contractor on the program (line 6 of Algorithm 4), where the domain was reduced from $[\mathbf{x}]' = [1, \text{Max}_{int}] \times [0, 1000]$ to $[\mathbf{x}]'' = [1, 1000] \times [1, 1000]$ where $\mathbb{S}_{in} = \{[1000, \text{Max}_{int}] \times [0, 1000], [0, 1000] \times [0, 1]\}$ and $\mathbb{S}_{boundary} \leftarrow [\mathbf{x}]''$. An example of the instrumentation employed is illustrated in Fig. 4.6. The `assume(expr)` directive here is placed in the loop, where the values change for variables. It is placed in this location to guide the verifier not to check steps further in the loop because these steps represent $\mathbb{S}_{in}$, where the property is guaranteed to hold.

```c
#include <assert.h>
int main() {
    int x = 1, y = 0;
    while (y < 1000
            && __VERIFIER_nondet_int()) {
        x = x + y;
        y = y + 1;
    }
    assert(x >= y);
    return 0;
}
```

Fig. 4.4. C program extracted from *sv-benchmarks/c/loop-lit/afnp2014.c*

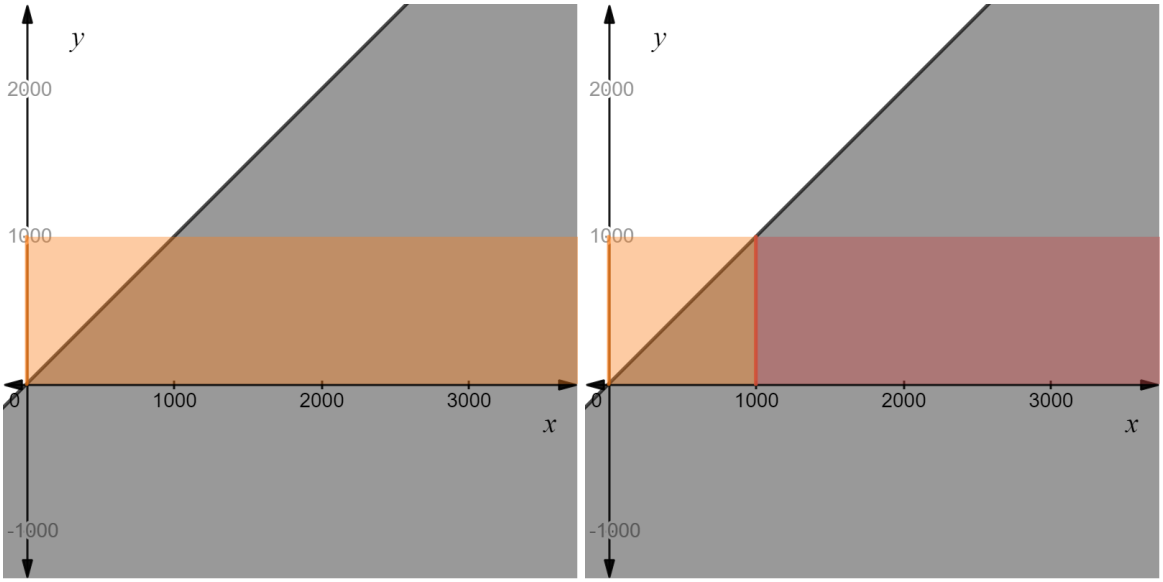


Fig. 4.5. `afnp2014.c` search space before and after contractors.

4.3 Implementation

The implementation of the proposed approach follows the Algorithm 4 and goes into the following steps, as described below. Though it has limitations, we will discuss them further in Section 4.3.5. A background description of ESBMC and its extension with contractors is available in Appendix B.

4.3.1 Property Analysis

This step will analyse the property used as a constraint in the CSP (cf. definition16). We start with parsing each expression in the `assert` directive, which is carried by converting from the syntax of ESBMC GOTO-program to the IBEX syntax. Figures 4.7b and 4.7a illustrate

```
1 #include <assert.h>
2 int main() {
3     int x = 1, y = 0;
4     while (y < 1000
5             && __VERIFIER_nondet_int()) {
6         x = x + y;
7         y = y + 1;
8         assume(x <= 1000);
9     }
10    assert(x >= y);
11    return 0;
12 }
```

Fig. 4.6. `afnp2014.c` after applying contractors.

both syntaxes, where we notice that IBEX syntax is more strict in its structure than ESBMC's syntax. For example, $(x > 1) + 1$ is allowed in ESBMC's syntax but not IBEX. Furthermore, IBEX syntax does not have an operator for $!=$, which makes it hard to convert some expressions. While parsing an expression, we build a list of variables inside the assertions, which we will use later to analyse intervals. We have two lists updated from this procedure: a list of variables and a list of constraints.

4.3.2 Interval Analysis

Interval analysis builds intervals for each variable in the assert statements. These intervals will serve as domains for the CSP. The current implementation now utilizes `--interval-analysis` option in ESBMC and parses each assume directive inserted by it. At each assume directive, we check whether the variable used is in our list, and if it is, we update its interval.

4.3.3 Interval Methods (Apply Contractor)

For contractors, we chose IBEX library.[2] IBEX is a C++ library for constraint processing over real numbers. It started in 2007 as an open-source academic project that provides algorithms for handling non-linear constraints. RealPaver [182] is also a C++ interval solver that deals with CSPs involving non-linear constraints. It is more focused on robust global optimisation using interval arithmetic. We choose IBEX over RealPaver because of its ability to easily integrate with other solvers and tools, providing more versatility in mixed-integer programming

[2] http://www.ibex-lib.org/

n	$\in$	$\mathbb{V}$	scalar values
$\mathtt{x}$	$\in$	$\mathbb{X}$	program variable
$\odot$	$::=$	$+ \mid - \mid * \mid ...$	binary operator
$\circledcirc$	$::=$	$< \mid \leq \mid == \mid ...$	comparison operator
E	$::=$		scalar expressions
	$\mid$	n	scalar constant
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$E \odot E$	binary operation
B	$::=$		Boolean expressions
	$\mid$	$E \circledcirc E$	Compare two expressions
L	$::=$		logical expression
	$\mid$	B	
	$\mid$	$B \wedge B$	and
	$\mid$	$B \vee B$	or

(a) IBEX Syntax.

E	$::=$		scalar expressions
	$\mid$	n	scalar constant
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$E \odot E$	binary operation
	$\mid$	$E \circledcirc E$	Compare two expressions
	$\mid$	$E \wedge E$	and
	$\mid$	$E \vee E$	or
	$\mid$	$\neg E$	not
	$\mid$	$Typecast(E)$	typecast
...			

(b) ESBMC goto-program Syntax.

Fig. 4.7. Syntax

and broader application support.

The contractor type used is the *forward-backward* contractor. With the constraint and the variables parsed from the first step and domains from the second step, we create the contractor with the constraint complement to reduce the intervals inside the solution. As described previously, the contractor will need a CSP; at this point, we should have all the components to apply it. We have the list of constraints, the variables, and their domains. The resulting domains from the contractor will represent where the property may be violated, thus limiting the search space for BMC.

4.3.4 Modify Original Program

With the resulting intervals, we compare them to the original and insert the `assume` directive based on the difference. This is to tell BMC to ignore the instructions (loop iterations), which will result in holding the property.

4.3.5 limitations

In property analysis (cf. Section 4.3.1), we support a limited range of one-to-one conversions, restricting the types of transformations applicable to expressions. While these conversions work for simple cases, more complex expressions need different handling. For example, IBEX does not support the `!=` operator, which can be represented as a union of intervals. Similarly, expressions such as `(x>1)+1`, valid in GOTO-program syntax, are not allowed in IBEX due to stricter rules that prevent adding logical expressions to scalars. As shown in Fig. 4.7a, IBEX enforces a clear distinction between logical and numerical expressions, making these conversions more complicated. Additionally, we handle only one constraint at a time, and expressions with unsupported operators such as `!=`, `==`, `&&`, and `||` are not converted as they fall outside IBEX's capabilities.

A key challenge with ESBMC's interval analysis is that the resulting intervals can be too broad, leading to overly conservative results that may obscure useful insights for contractors. This reduces the analysis effectiveness. Furthermore, the lack of floating-point interval analysis may present challenges in future applications since IBEX exclusively handles `double` types. Systems requiring precise numerical analysis would struggle with this limitation. Thus, the current gaps in operator support, handling of complex conversions, and interval precision highlight improvements, such as better interval accuracy, support for more complex expressions, and floating-point interval analysis to enhance system precision and usability.

4.4 Experimental Evaluation

4.4.1 Objectives

The objective of this experimental evaluation is to assess the impact of integrating contractors—an interval method—into the Bounded Model Checking (BMC) process. This assessment is structured to provide empirical evidence addressing the core research questions formulated in Chapter 1 (Section 1.4). Specifically:

EG1 To evaluate the extent to which contractors reduce the state space in BMC. This objective aligns directly with Research Question 1, which investigates whether numerical methods, particularly contractors, can effectively reduce the search space in verification techniques.

EG2 To measure computational improvements in terms of CPU time and memory consumption. This objective supports Research Question 2, which examines the impact of contractor integration on resource consumption, including processing time and memory efficiency.

EG3 To determine whether the incorporation of contractors preserves soundness and completeness of the verification. This objective addresses Research Question 3, which seeks to verify that the integration of contractors does not compromise the soundness or completeness of BMC as a verification framework.

4.4.2 Description of the Setup

We chose ESBMC [106] as our BMC engine due to its incremental algorithms and its integration with benchexec. We executed ESBMC with the same set of options of SV-COMP 2022, differing only on the strategy chosen (i.e., incremental-bmc). following set of options: `--incremental-bmc`, which enables the incremental BMC; `--unlimited-k-steps`, which removes the upper limit of iteration steps in the incremental BMC algorithm; `--goto-contractor` to enable our method. Experimental results that do not use the contractor were taken from SV-COMP 2022[3]. We compare our results with state-of-the-art tools from SV-

[3] https://sv-comp.sosy-lab.org/2022/results/results-verified/

COMP 2022. These tools include CBMC [107] for its success over the years. Veriabs [183], which is the SV-COMP 2022 champion in this category and the runner-up CPA Checker 2.1 [184] and Verifuzz [185]. SV-COMP score system consists in: (+2) for correct true, (+1) for correct false, (-32) for incorrect true, and (-16) for incorrect false.

All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 160GB of RAM. We set time and memory limits of 900 seconds and 15GB for each benchmark, respectively. Finally, we present the results by showing the verification time and scores. All presented execution times are CPU times, i.e., only the elapsed periods spent in the allocated CPUs. Furthermore, memory consumption is the amount of memory that belongs to the verification process and is currently present in RAM (i.e., not swapped or otherwise not-resident). CPU time and memory consumption were measured with the *benchexec* tool [156] (commands available on the supplementary page). We did not enable swapping or turbo during our experiments, and all executed tools were restricted to a single core.

4.4.3 Description of the Benchmarks

We evaluate our approach using 7044 tasks in `unreach-call` benchmarks from SV-COMP 2022 [186], which can be described as *verifying whether exists an execution path that can lead a benchmark to an assertion failure*. This category contains benchmarks extracted from various domains (e.g., linux drivers, recursion, unbounded loops, algorithms, real-world, etc.). Due to massive arithmetic computation within loops, this category can result in many timeouts (i.e., a tool can not verify it in a given time constraint); they fit our approach. All tools, benchmarks, and evaluation results are available on a supplementary web page[4]. For detailed information about the benchmark structure and scoring methodology used in this evaluation, refer to Appendix A

4.4.4 Results

ESBMC with contractors had been evaluated on *unreach-call* property in 13 subcategories in SV-COMP. Table 4.1 shows our comparison results with other state-of-the-art BMC tools. Our approach showed its effectiveness in multiple subcategories regarding score and resource

[4] https://doi.org/10.5281/zenodo.6949341

consumption. In detail, we improved ESBMC incremental-BMC in `Arrays`, `ControlFlow`, `ECA`, `Heap`, and `Loops`.

> The experimental results show that our method helped BMC efficiently reduce resource consumption, including a reduction of 75% of memory usage while verifying 1% more benchmarks than our baseline, thus successfully answering **EG2**.

Our approach demonstrated its capabilities in these benchmarks by achieving scores equal to or higher than those of ESBMC Incremental-BMC. We have the lead in subcategories `XCSP` and `AWS-C-Common`, and we are a close second in `Floats`. Also, our approach ranked third in our comparison of `Loops`, `Arrays`, and `Heap` categories. This shows how valuable contractors are in real-world programs because we scored higher than other tools and decreased the verification time and memory. However, contractors could not improve the score in other categories, for it was not applicable in the current implementation. The cases where contractors could not be applied are due to our naive approach to detecting monotonicity and due to it only supporting a subset of operators (cf. Subsection 4.3.5).

> While the observed consistency in correct and incorrect verification results—maintained within a margin of 0.01%—serves as empirical evidence suggesting that the integration of contractors does not negatively impact the verification outcomes, this observation alone does not constitute a formal proof of soundness or completeness. As defined in Definitions 15 and 16, soundness and completeness are theoretical properties that require formal guarantees beyond empirical behaviour.
> Thus, although our results are aligned with the expected outcomes of a sound and complete verification process, they should be interpreted as supporting evidence rather than definitive proof. This empirical consistency strengthens our confidence in the implementation's reliability and supports the achievement of Experimental Goal 3 (**EG3**), but we acknowledge that formal verification or theoretical analysis would be necessary to rigorously prove soundness and completeness in the general case.

One of the main issues in BMC is dealing with lengthy loops. Our approach focuses on dealing with loops and pruning the unnecessary steps to conclude the verification. `Loops` category has 14 subcategories categorized the same as SV-COMP 2022 as shown in Table 4.2. In particular, our approach performed better in four of these subcategories than plain ES-

BMC. For example, we checked the subcategory of `loops`; we got remarkable results compared to plain ESBMC. The same applies to `loop-new`, `loop-zilu`, `loops-crafted-1` and `nla-digbench-scaling`. We also got better scores and time consumption results than the champion tool Veriabs. In particular, in the subcategory `loop-simple`, `verifythis` and `nla-digbench-scaling`.

The greatest performance improvements with contractors were observed in the `LoopLit` and `BitVectors` subcategories. In `LoopLit`, contractors effectively narrowed loop bounds, significantly reducing the number of iterations explored during symbolic execution. In `BitVectors`, contractors constrained variable ranges involved in bitwise operations, simplifying the generated verification conditions. These gains are attributed to the presence of tight, statically analyzable constraints that lend themselves well to interval-based domain reduction. In contrast, benchmarks involving dynamic memory and concurrency exhibited more modest improvements due to the complexity and unpredictability of their constraints.

In the `loops` category, we find our approach shows improvement of ESBMC performance by at least 2% and achieves **EG1**.

However, we scored lower in `loop-acceleration` because some benchmarks timed out, which cost us 3 scores. Our approach also has some shortcomings: we do not deal with multiple lengthy loops in this version. That is why we got the same score in some subcategories. We achieved the most critical objectives by saving CPU time, decreasing memory consumption, and improving the scores. Additionally, we scored reasonably well in most categories, with 2 being the lead. However, this version of our approach does not apply to some benchmarks, so we see some score differences. Also, we noticed our approach did not perform well in one particular benchmark and reported a false negative. The benchmark is in `bitvectors`, representing the overflow problem. Another issue with the implementation is that we do not apply the contractors when it comes to an unsupported operator. This is also a reason for the score not improving in some subcategories. Moreover, the lack of accuracy in the interval analysis option in ESBMC hindered the performance of the contractors. Because the intervals were not accurate and the result of the contractor did not help reduce the search space. In the future, we will work to overcome this limitation.

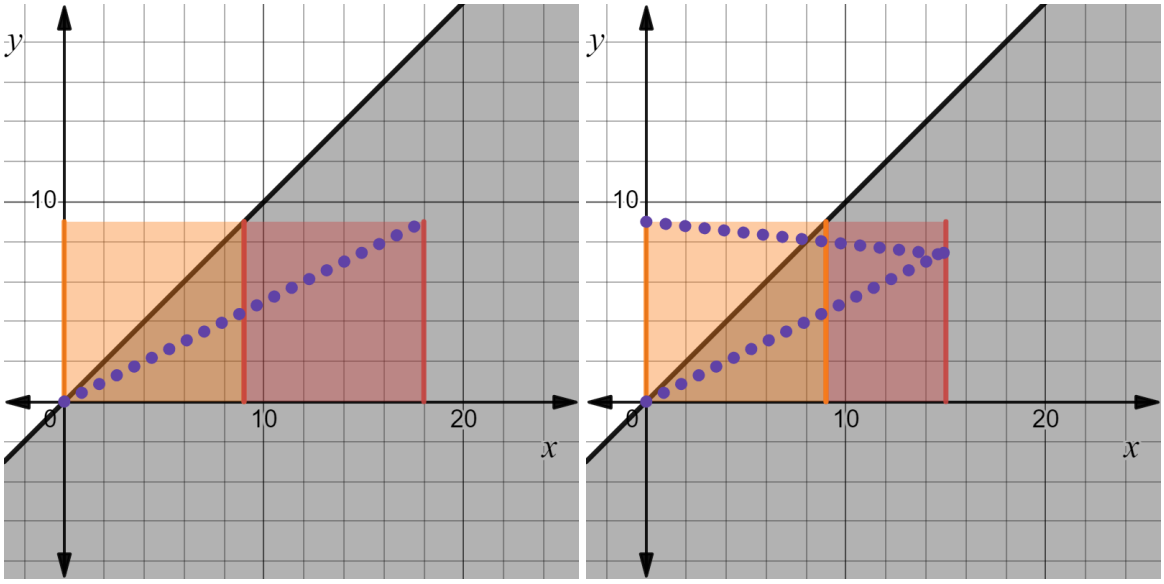


Fig. 4.8. Monotone vs non-monotone.

4.5 Threats to the validity

As mentioned in Section 4.2, we only apply the contractor if the contracted area is at the end of the loop. For example, given a variable $x = X_0$ that is incremented inside a non-deterministic loop `x++` up to a max value of X_{max}, if the contractor result (e.g., $X = [X_0, X_{max}]$) was reduced from the upper bound (i.e., $X < X_n < X_{max}$, where $X < X_n$ is the contracted region) we can safely skip the instructions (iterations) that exceed the upper bound. However, suppose the contracted area was at the beginning of the loop (i.e., the reduced interval from the lower bound). In that case, we cannot guarantee the correctness of the verification results, as the method cannot reason about the minimum quantity of loop iterations. In the previous example, reducing from the lower bound (i.e., $X > X_n \wedge X < X_{max}$) cannot be contracted as it would skip instructions.

Similarly, the program's monotonicity for variables also affects the verification results. For example, a variable y that may be non-deterministically incremented or decremented cannot be contracted. Illustrated in Figure 4.8 are two programs; one is monotone (here, y would always increment), and the other is not (here, y can increment and decrement at any state). Dots represent states, and blank lines and grey areas represent a property. Assume we start from the origin point; the first program can safely use an *assume* directive to skip the states guaranteed to hold the property. However, that is not the case with the second program, and we will end up in the same situation as mentioned before, where the area reduced is at the

beginning of the loop. Nevertheless, this time, it is in the middle.

4.6 Conclusion

In the domain of Interval Analysis and Methods, the creation of contractors utilizing Interval Methods has rendered their applications more practical by mitigating the complexity inherent in such methods [139]. We assert that a similar advantageous impact can be achieved by incorporating contractors in interval analysis applied to the static analysis of programs. In this research, we have extended the application of contractors to prune the search space within Bounded Model Checking (BMC) techniques. Experimental results illustrate the consequential impact on open-source C benchmarks, wherein certain categories witnessed an increase in scores accompanied by a reduction in resource consumption.

Specifically, employing interval methods via contractors expedited the incremental BMC verification process. A comparative analysis between verification with and without contractors' applications reveals notable differences, with 32 additional benchmarks successfully verified, particularly evident when pruning the search space of programs featuring loops. Furthermore, our approach demonstrated competitive performance against state-of-the-art tools in BMC, achieving a lead in two subcategories.

Importantly, the proposed methodology preserves the soundness and completeness of the BMC technique by consistently producing the same verification outcome. However, it necessitates further development to encompass additional cases and enhance overall scores. Prioritising the implementation of support for more operators and refining interval analysis stands out as critical for achieving the most impact improvements in verification results in scores and resource consumption.

Table 4.1. Results for each subcategory. Each cell in the table shows two values. The top value represents the score, while the bottom represents time in seconds.

Subcategory / $\frac{Score}{Time(s)}$	**ESBMC** incremental-BMC with contractors	**ESBMC** incremental-BMC v6.9	**ESBMC** k-induction v6.8	**CBMC** v5.43.0	**Veriabs** v1.4.2	**CPA Checker** v2.1	**Verifuzz** v1.2.11
Arrays	159 272607	151 273123	234 224050	-66 133708	**729** 17449	77 352273	109 274219
BitVectors	24 13876	55 14145	55 12924	41 8294	**76** 6077	71 9680	15 28691
ControlFlow	58 47384	55 46727	64 23883	67 30696	**128** 24763	113 13987	40 46734
ECA	270 947500	268 949801	1098 539341	170 989186	**1312** 449183	930 687361	429 799332
Floats	800 48147	802 46768	779 36501	782 46743	**827** 44156	749 62876	-64 368963
Heap	256 55180	227 53887	226 45732	**292** 48786	204 40801	282 50371	57 161308
Loops	669 296102	652 294764	609 273893	637 236902	**1044** 170916	696 292803	209 492657
ProductLines	-595 214485	-595 214554	783 62777	256 128144	903 149945	**903** 80606	216 48716
Recursive	-64 31790	-64 31771	109 22736	100 18695	**125** 22073	92 23215	43 54177
XCSP	**159** 14306	159 14334	158 13273	158 12087	152 19869	153 20442	55 59186
AWS-C-Common	**174** 32234	174 33479	157 19637	6 5279	- -	43 81193	0 1999
BusyBox	0 2141	0 1812	2 10	0 7755	- -	**2** 2555	0 206
DeviceDriversLinux64	66 2133570	66 2102103	2020 802304	-3257 1697560	- -	**3114** 948362	0 328643
total	1976 4109322	1950 4077268	6294 2077061	-814 3363835	5500 945232	**7225** 2625724	1109 2664831

Table 4.2. Results for loops category. Each cell in the table shows two values. The top value represents the score, while the bottom represents time in seconds.

Subcategory / $\frac{Score}{Time(s)}$	**ESBMC** incremental-BMC with contractors	**ESBMC** incremental-BMC v6.9	**ESBMC** k-induction v6.8	**CBMC** v5.43.0	**Veriabs** v1.4.2	**CPA Checker** v2.1	**Verifuzz** v1.2.11
loops	29 18353	25 18544	1 13261	52 14172	**85** 5817	78 10579	28 32777
loop-acceleration	26 18568	29 15803	28 13734	28 5800	**49** 2967	32 16468	17 18698
loop-crafted	3 3846	3 3608	8 1723	1 1450	**11** 459	5 3306	0 5403
loop-invgen	11 18610	11 17661	15 15683	3 21295	**41** 8529	37 12144	1 25219
loop-lit	15 12628	15 12645	13 11945	23 10011	**37** 4317	21 11150	1 18914
loop-new	8 6309	4 8354	12 6285	12 3996	**16** 2835	10 6977	0 9906
loop-industry-pattern	16 9061	16 9061	28 2923	16 8137	32 2363	**36** 1826	0 16208
loops-crafted-1	-24 42493	-28 40099	-17 19040	3 27417	**96** 6344	9 42570	8 39663
loop-invariants	3 7124	3 7122	3 6950	1 6141	**17** 144	7 4914	1 7208
loop-simple	**13** 961	13 920	11 903	9 605	11 1001	7 2419	1 6307
loop-zilu	20 10932	14 13657	16 12650	14 13135	**42** 1194	38 4887	0 19812
verifythis	**2** 3631	2 3634	2 3618	2 2069	0 2392	-30 1939	2 2883
nla-digbench	7 23947	7 23953	8 23617	5 21905	**23** 16700	5 25427	4 24412
nla-digbench-scaling	540 119639	538 119702	481 141561	468 100768	**584** 115853	441 148198	146 265249
total	669 296102	652 294764	609 273893	637 236902	**1044** 170916	696 292803	209 492657

Chapter 5

Contractors in abstract interpretation

5.1 Introduction

In formal methods and static analysis, abstract interpretation is widely regarded as a powerful method for approximating program behavior [81], [144]. However, one of the central difficulties with abstract interpretation is balancing the requirement for precision with ensuring the analysis remains computationally tractable [144]. This chapter introduces the concept of contractors as one solution to alleviate this problem, particularly in the context of interval analysis. Contractors are interval-based methodologies for contracting the intervals representing possible values for program variables to improve the precision of abstract interpretations. This chapter discusses the use of contractors on conditional statements—more precisely, on *if*-statements and loops—illustrating how contractors can be systematically integrated into interval analysis to obtain more precise results by pruning intervals using contractors and reducing the search space which is the goal of this thesis.

5.2 Methodology

In this section, we explore contractors' application in the context of interval analysis within the abstract interpretation framework. Contractors are tools for improving the precision of abstract interpretation in numerical computations by contracting intervals. Our primary focus is the application of contractors in conditional expression evaluations, typically if statements and loops. Such types of expressions may involve variables, often represented by intervals. We do this by constructing the three basic components of a constraint satisfaction problem (CSP) (cf. Definition 16): the condition as constraint, the variables involved in the condition,

and the domains of these variables obtained by interval analysis (cf. Section 2.6.5). In doing so, we utilise contractors to narrow down these intervals.

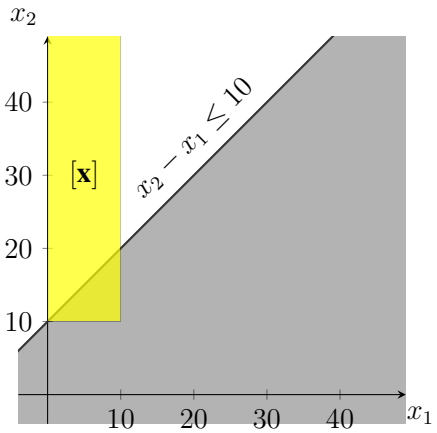
When applicable, contractors utilize rigorous interval arithmetic to narrow the intervals representing the possible values of variables iteratively. The contraction process reduces the width of these intervals, converging to a more precise and accurate state space description. This iterative aspect of contraction allows the progressive elimination of infeasible values to improve the intervals until they nearly represent the actual range of possibilities.

A particularly valuable benefit that comes along with the use of contractors in abstract interpretation is its ability to decrease the precision loss occurring while widening (cf. Definition 25) is taking place. Widening is one approach to ensuring the termination of the analysis by state space over-approximation; quite often, this leads to a loss in precision [187]. The contractors overcome this issue by mitigating some lost precision by ensuring soundness and preserving the precision of the analysis itself. This improvement is beneficial for applications requiring accuracy, such as those in safety-critical systems or numerical calculations, where even small inaccuracies may cause large deviations.

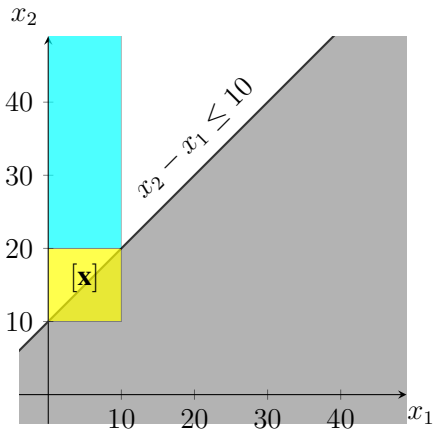
For example, consider the code fragment in Figure 5.1a. It starts with assigning the variables with random values (i.e., $[-\infty, \infty]$) and then limiting the values with assume directive to $[0, 10]$ and $[10, \infty]$ for `x_1` and `x_2` respectively. Next, we encounter the `if` statement. Now, we construct our CSP in this context where the variables are `x_1` and `x_2` are our variables x_1, x_2, domains are $[0, 10]$ and $[10, \infty]$ for x_1 and x_2 respectively, and our constraint which is the `if` statement condition $x_2 - x_1 \leq 10$. With all CSP components ready, we construct our contractor as seen in Figure 5.1d. We plug in our intervals for the two variables, and we get $x_1 = [0, 10]$ with no change and $x_2 = [10, 20]$ with a notable reduction. Thus, we gained precision in intervals inside the `if` statement. Listing 5.1 shows the difference of goto program in ESBMC execution where the red highlighted lines are from interval analysis without the contractor and the green lines were added when using the contractor.

```c
#include <assert.h>
int main() {
    int x_1 = __VERIFIER_nondet_int();
    int x_2 = __VERIFIER_nondet_int();

    assume(x_1 <= 10 && x_1 >= 0 && x_2 >= 10);

    if(x_2 - x_1 <= 10)
        assert(x_1 - x_2 <= 0);

    return 0;
}
```

(a) contractor example in abstract interpretation.



(b) Before Contractors.



(c) After Contractors.

$f(x) \leq 0$	$I = (-\infty, 0]$	
$f(x) : y = x_2 - x_1 - 10$	$[y] = I \cap ([x_2] - [x_1] - 10)$	Forward-step
$f(x)^{-1}_{x_1} : x_1 = x_2 - y - 10$	$[x_1] = [x_1] \cap ([x_2] - [y] - 10)$	Backward-step
$f(x)^{-1}_{x_2} : x_2 = y + x_1 + 10$	$[x_2] = [x_2] \cap ([y] + [x_1] + 10)$	Backward-step

(d) Contractor steps for the example in Fig 5.1a.

Fig. 5.1. Example illustrating contractors in abstract interpretation.

5.3 Implementation: Ibex Contractor

For contractors, we chose the IBEX library.[1] IBEX is a C++ library dedicated to constraint processing over real numbers. It started in 2007 as an open-source academic project that provides algorithms for handling non-linear constraints. RealPaver [182] is also a C++ interval solver that solves constraint satisfaction problems (CSPs) with non-linear constraints. Its main focus is on rigorous global optimization using interval arithmetic. We choose IBEX

[1] http://www.ibex-lib.org/

```
1 main (c:@F@main):
2 DECL signed int x_1;
3 DECL signed int return_value$___VERIFIER_nondet_int$1;
4 ASSIGN return_value$___VERIFIER_nondet_int$1=NONDET(signed int);
5 ASSIGN x_1=return_value$___VERIFIER_nondet_int$1;
6 DECL signed int x_2;
7 DECL signed int return_value$___VERIFIER_nondet_int$2;
8 ASSIGN return_value$___VERIFIER_nondet_int$2=NONDET(signed int);
9 ASSIGN x_2=return_value$___VERIFIER_nondet_int$2;
10 ASSUME x_1 <= 10 && 0 <= x_1 && x_2 <= 2147483647 && 10 <= x_2
11 IF !(x_2 - x_1 <= 10) THEN GOTO 2
12 -ASSUME x_2 <= 2147483647 && 10 <= x_2 && x_1 <= 10 && 0 <= x_1
13 +ASSUME x_2 <= 20         && 10 <= x_2 && x_1 <= 10 && 0 <= x_1
14 ASSERT x_1 - x_2 <= 0
15 2: RETURN: 0
16 END_FUNCTION // main
```

Listing 5.1. The GOTO program for the example in Figure 5.1a. The red lines are the lines removed and replaced with green lines as a result of using contractors.

over RealPaver because it can easily be integrated with multiple solvers and tools, thus providing more flexibility in mixed-integer programming and broader application support.

The following steps are taken to apply the contractor in the given program.

1. **Enable Ibex Contractor Flag**: Configuration of ESBMC to utilise the Ibex library for interval analysis. The flag `--interval-analysis-ibex-contractor` is used along with `--interval-analysis`.

2. **Conditional GOTO and ASSERTS in ESBMC Analysis**: ESBMC identifies a conditional GOTO statement during the execution path analysis. These conditional statements are in the form `IF cond THEN GOTO label`. They represent either an `if`, `if-else`, or a loop. Asserts can be in two forms: `assert()` from `assert.h` or user-defined `__VERFIER_assert()`.

3. **Parsing Condition to Ibex**: The condition from the GOTO statement is parsed into Ibex for analysis. Figure 4.7 illustrate the difference in syntax. A GOTO expression may not be converted to IBEX expression because it does not fit the syntax. There are two exceptions; one is when the parser encounters a `A != B` expression, it will convert it into `A > B || A < B`. The other exception is when the parser encounters a negation or unary operator `!`, where it will take the complement of the expression encapsulated in the negation.

4. **Reading Current Intervals for Variables**: Retrieval of current intervals (ranges of values) for variables involved in the condition. This is done via a list created by interval analysis to represent the program's current state. Primarily, it has three lists: one for integers, floats, and one for wrapped intervals. In this work, we are only interested in the integers list.

5. **Applying the Ibex Contractor**: Ibex contractor is applied to refine the intervals based on the condition. The contractor used is the forward-backward contractor 18, and it is the outer contractor variant. The inner contractor has not been used explicitly here because interval analysis will analyse a conditional statement's TRUE and FALSE paths. In the TRUE path, IBEX gets the condition as it is. Meanwhile, in the FALSE path, IBEX gets the negation or the complement of that condition, which makes ESBMC apply the outer and inner contractors. An optimisation could be done here by storing the expressions and creating complements rather than repeatedly parsing them.

6. **Converting Results to ESBMC Expressions**: The results from Ibex are converted back into expressions that ESBMC can understand. This step is not as trivial as it may seem due to the conversion from `double` to `int` and its variations (`long, unsigned int, char`). One case to be aware of is converting from the max `int` number represented in `double` back to `int`. So, conditions are set in place to prevent getting the wrong values. Another measure is rounding. When rounding numbers back to integers, we take the floor for the upper bound and the ceiling for the lower bound.

7. **Updating ESBMC Interval Analysis**: The updated intervals from Ibex are passed back to ESBMC for continued analysis.

5.4 Evaluation

5.4.1 Objectives

This experiment will compare results from performing two strategies *incremental-BMC* and *k-induction* (cf. Definitions 3, 4). Our experiment will include three methods in each strategy: plain strategy, strategy with interval analysis, and strategy with interval analysis utilizing contractors. The comparison will be made regarding CPU time, memory consumption, and

SV-COMP score. SV-COMP score system consists of (+2) for correct true, (+1) for correct false, (-32) for incorrect true, and (-16) for incorrect false. This evaluation will answer the following main experimental goals:

EG1 **(Efficiency)** Do the contractors prune the search space to consume fewer resources (CPU time, Memory) when verifying programs with ESBMC?

EG2 **(Soundness and completeness)** Does the use of contractors affect the correctness or completeness of the verification results for both incremental BMC and *k*-induction?

EG3 **(Trade-off)** What are the trade-offs in time and memory usage when using contractors, and is there a point where using contractors becomes inefficient?

5.4.2 Setup

We executed ESBMC with the same set of options of SV-COMP 2024, differing only on the strategy chosen (i.e., *k*-induction or incremental-bmc), following this set of options:

`--incremental-bmc`, which enables the incremental BMC.

`--k-induction`, which enables the k-induction.

`--unlimited-k-steps` removes the upper limit of iteration steps in the incremental BMC algorithm.

`--interval-analysis` to enable interval analysis (ESBMC's abstract interpretation engine).

`--interval-analysis-ibex-contractor` to enable our method utilizing contractors.

We limit our test to compare with baseline ESBMC to focus the test on the effect of contractors on ESBMC's abstract interpretation engine. This means we will not compare it with other tools as their performance is irrelevant in this comparison because we want to focus on the performance of the contractors in the same context.

All experiments were conducted on an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 160GB of RAM. We set time and memory limits of 900 seconds and 15GB for each

benchmark, respectively. Finally, we present the results by showing the verification time and scores. All presented execution times are CPU times, i.e., only the elapsed periods spent in the allocated CPUs. Furthermore, memory consumption is the amount of memory that belongs to the verification process and is currently present in RAM (i.e., not swapped or otherwise not-resident). CPU time and memory consumption were measured with the *benchexec* tool [156] (commands available on the supplementary page). We did not enable swapping or turbo during our experiments, and all executed tools were restricted to a single core.

5.4.3 Benchmarks

We evaluate our approach using 11282 tasks in `unreach-call` benchmarks from SV-COMP 2024 [155], which can be described as *verifying whether exists an execution path that can lead a benchmark to an assertion failure*. This category contains benchmarks extracted from different domains (e.g., linux drivers, recursion, unbounded loops, algorithms, real-world, etc.). Due to massive arithmetic computation within loops, this category can result in many timeouts (i.e., a tool can not verify it in a given time constraint); they fit our approach. All tools, benchmarks, and evaluation results are available on a supplementary web page[2]. For detailed information about the benchmark structure and scoring methodology used in this evaluation, refer to Appendix A

5.4.4 Results

Given the experimental goals described above (EG1-EG3), the results offer insights into how the contractor method performs relative to incremental-BMC and k-induction strategies. The discussion below focuses on the implications of using contractors for each experimental goal. Tables 5.6 and 5.7 show this evaluation's results overview across categories.

Overall, in incremental-BMC, contractors took slightly more CPU time by 0.18% and memory increased by 0.02% than interval analysis but still less than plain incremental. Similarly, k-induction showed insignificant changes in CPU time (0.43% increase) and memory (2.20% decrease) if we compare interval analysis with and without contractors.

[2] https://drive.google.com/drive/folders/1gQw8Gf5uAsXKkbVTjndbkB6n1bs4Cil_?usp=sharing

```
1 ...
2 int init(void)
3 {
4   int tmp ;
5
6   {
7   if ((int )r1 == 0) {
8     if ((int )id1 >= 0) {
9       if ((int )st1 == 0) {
10        if ((int )send1 == (int )id1) {
11          if ((int )mode1 == 0) {
12          ...
13                if ((int )id6 != (int )id8) {
14                    if ((int )id7 != (int )id8) {
15                            tmp = 1;
16                    } else {
17                        tmp = 0;
18                }
19                } else {
20                    tmp = 0;
21            }
22        }
23 ...
```

Listing 5.2. `pals_lcr.8.ufo.BOUNDED-16.pals+Problem12_label03.c` code snippet. It shows many `if` nested to finally reach the assignment of `tmp` to 1.

However, looking closely at the individual benchmark results, we notice a major decrease in time in some benchmarks. For example, one benchmark[3] time with interval analysis was 875 seconds. With contractors, the benchmark was verified in 211 seconds. This decrease is because contractors made some variable intervals (within interval analysis) smaller by half. In Listing 5.2, we show part of the code where intervals differ from interval analysis without and with contractors. When this code was transformed into GOTO language (ESBMC internal representation [4]), we show the difference in Listing 5.3 where both GOTO files are identical with the lines added because of the contractor are highlighted in green. Particularly, the variables starting with the prefix `id` are of type `char`, meaning they have an initial interval from -128 to 128 (i.e., $[-128, 127]$). Some intervals were reduced to $[0, 127]$ with the contractor. This reduction in intervals made the solver determine the satisfiability faster, which led to faster verification time.

```
1 init (c:@F@init):
2 DECL signed int tmp;
3 ASSIGN tmp=NONDET(signed int);
```

[3] `pals_lcr.8.ufo.BOUNDED-16.pals+Problem12_label03.c`

[4] Documentation of GOTO-program can be found at https://ssvlab.github.io/esbmc/documentation.html

```
4 IF !((signed int)r1 == 0) THEN GOTO 121
5 IF !((signed int)id1 >= 0) THEN GOTO 119
6 IF !((signed int)st1 == 0) THEN GOTO 117
7 +ASSUME id1 <= 127 && 0 <= id1
8 IF !((signed int)send1 == (signed int)id1) THEN GOTO 115
9 IF !((signed int)mode1 == 0) THEN GOTO 113
10 IF !((signed int)id2 >= 0) THEN GOTO 111
11 IF !((signed int)st2 == 0) THEN GOTO 109
12 +ASSUME id2 <= 127 && 0 <= id2
13 IF !((signed int)send2 == (signed int)id2) THEN GOTO 107
14 IF !((signed int)mode2 == 0) THEN GOTO 105
15 ...
16 +ASSUME id8 <= 127 && 0 <= id8 && id5 <= 127 && 0 <= id5
17 IF !((signed int)id5 != (signed int)id8) THEN GOTO 7
18 IF !((signed int)id6 != (signed int)id7) THEN GOTO 5
19 +ASSUME id8 <= 127 && 0 <= id8 && id6 <= 127 && 0 <= id6
20 IF !((signed int)id6 != (signed int)id8) THEN GOTO 3
21 IF !((signed int)id7 != (signed int)id8) THEN GOTO 1
22 ASSIGN tmp=1;
23 GOTO 2
24 1: ASSIGN tmp=0;
25 2: GOTO 4
```

Listing 5.3. GOTO program produced by ESBMC from `pals_lcr.8.ufo.BOUNDED-16.pals+Problem12_label03.c` with interval analysis and another time with interval analysis contractors. The files are almost identical with the lines added by using contractors in green

Over the Sequentialized category, we see a decrease in memory usage. However, in Combination, we see an increase. In both cases, the difference is less than 5% of total memory. However, when we look at some benchmarks, we see a substantial decrease in memory, especially in the Sequentialized category; therefore, we make two additional Tables 5.8 5.9, which show only benchmarks that resulted in either TRUE or FALSE and removed all timeouts and out of memory. In this table, all methods have the same score. However, we cannot say the same for CPU time and memory. We notice that the Sequentialized category took less time (11.62%) and less memory (24.65%) when using contractors with incremental-BMC and took less time (5.28%) and less memory (17.38%) when using contractors with *k*-induction.

To determine why interval analysis with and without the contractor differ in time and memory, we will closely examine one of the benchmarks that showed a decrease in CPU time and memory. `pals_lcr-var-start-time.3.2.ufo.UNBOUNDED.pals.c.v+sep-reducer.c` showed CPU time of $519\,\text{s}$, $532\,\text{s}$, and $265\,\text{s}$ in plain *k*-induction, *k*-induction with interval analysis, and *k*-induction with interval analysis with contractors, respectively. It also showed 1GB, 1GB, and 0.5GB in memory. When we examine the code, we notice a lot of conditions leading to values that would reach the error state. Listing 5.4 shows a snippet of the code we are examining. When we compare GOTO files for interval analysis without contractors

Table 5.1. Here, we show the percentage increase in CPU time (CPU) and memory consumption (Mem) when comparing interval analysis with and without contractors when they both reach the same status. This table is an extension of Tables. 5.8 and 5.9. In category *Sequentialized*, contractors consumed less CPU by 11.62%. While in category *ProductLines* contractors consumed more CPU by 13.49%

Category	Incremental-BMC		K-induction	
	CPU	Mem	CPU	Mem
Arrays	1.16%	1.69%	0.78%	1.69%
BitVectors	-1.78%	3.60%	-2.72%	3.74%
ControlFlow	6.68%	2.21%	4.74%	2.40%
ECA	2.73%	2.24%	2.56%	1.25%
Floats	-3.19%	0.82%	0.60%	1.10%
Heap	0.54%	3.53%	1.64%	3.12%
Loops	-0.08%	2.21%	0.90%	1.64%
ProductLines	**13.49%**	2.97%	**15.95%**	2.94%
Recursive	1.05%	1.81%	0.82%	1.82%
Sequentialized	**-11.62%**	**-24.65%**	**-5.28%**	**-17.38%**
XCSP	0.78%	2.51%	-0.70%	2.92%
Combinations	0.00%	2.30%	2.37%	1.38%
Hardware	-0.21%	0.19%	0.16%	0.16%
Hardness	2.56%	3.36%	-0.20%	1.46%
Total	0.05%	0.70%	0.86%	0.78%

to GOTO files for interval analysis with contractors, we notice that some if statements were removed.

Illustrated in Listing 5.5, we show a sample of the if statements removed. The reason for removing them is that the contractor was able to keep track (through interval analysis) of variables (namely `st1`, `st2`, and `st3`) involved in each if statement in Listing 5.4. Thus, the contractor guaranteed that these `if` statement conditions would always be true, which made the other branch of this `if` statement entirely omitted because it was unreachable. Likewise, if the condition is always false, the `if` branch is omitted, and a GOTO would take the program's execution directly to the `else` branch. Therefore, The memory required to verify such benchmarks is reduced from the baseline.

On the other hand, we see an increase in CPU time in *Productlines* category. We closely examined the benchmarks and noticed that most verification tasks were solved in an average of 1.9 and 2.2 seconds in interval analysis without and with contractors, respectively. This leads us to the conclusion that the overhead introduced when using the contractors is amplified here due to the low values and verification time.

```
int init__tmp;
        if (((int)r1) == 0)
        {
            if (((((int)alive1) + ((int)alive2)) + ((int)alive3)) >= 1)
            {
                if (((int)id1) >= 0)
                {
                    if (((int)st1) == 0)
                    {
                        if (((int)send1) == ((int)id1))
                        {
                            if (((int)mode1) == 0)
                            {
                                if (((int)id2) >= 0)
                                {
                                    if (((int)st2) == 0)
                                    {
                                    ...
```

Listing 5.4. `pals_lcr-var-start-time.3.2.ufo.UNBOUNDED.pals.c.v+sep-reducer.c` file from *Sequentialized* category. Where contractors consumed significantly less CPU and memory. The code shows nested `if` statements that manipulate variable values to reach an error state eventually. All the conditions in the `if` statements are considered constraints for the contractors.

```
1
2 DECL signed int check__tmp;
3 ASSIGN check__tmp=NONDET(signed int);
4 -IF !((signed int)st1 + (signed int)st2 + (signed int)st3 <= 1) THEN
      GOTO 38
5 -IF !((signed int)r1 < 3) THEN GOTO 37
6 ASSIGN check__tmp=1;
7 ASSIGN __return_1471=1;
8
9
10 IF (signed int)mode1 == 0 THEN GOTO 36
11 -IF !((signed int)r1 == 255) THEN GOTO 5
12 +GOTO 5
13 RETURN: 0
14 -5: ASSIGN r1=(unsigned char)((signed int)r1 + 1);
15 +5: ASSIGN r1=1;
16 ASSIGN node1__m1=p3_old;
17
18
19 ASSIGN check__tmp=NONDET(signed int);
20 -IF !((signed int)st1 + 2 <= 1) THEN GOTO 47
21 +GOTO 45
22 -RETURN: 0
23 -47: ASSIGN check__tmp=0;
24 +45: ASSIGN check__tmp=0;
25 ASSIGN __return_4237=0;
```

Listing 5.5. GOTO file comparison for file `pals_lcr-var-start-time.3.2.ufo.UNBOUNDED.pals.c.v+sep-reducer.c` produced by ESBMC interval analysis with and without contractors. We are comparing the files where the use of contractor removes the red highlighted lines, and the use of contractor adds the green highlighted lines.

EG1 **(Efficiency)** When we look at the total CPU time and memory increases by the contractor, in both strategies, they are less than 1%. The addition of contractors introduces minimal overhead regarding CPU time and memory usage. However, the contractors also helped save time and memory in certain categories. Thus, EG1 is achieved.

In the case of EG2, we can use the score metric to check whether there is any effect on soundness (cf. for Definition 1) and completeness (cf. for Definition 2). The score is the number of benchmarks where verification succeeded, either by proving correctness (TRUE) or showing counterexamples (FALSE). The scores show some interesting trends for all categories. We will discuss the changes in Incremental-BMC and k-induction separately.

Table 5.2. Scores of Incremental-BMC, incremental-BMC with interval analysis, incremental-BMC with interval analysis utilizing contractors.

Category	# of files	Incremental-BMC	Interval analysis	Contractors
Arrays	431	112	112	112
BitVectors	49	57	57	57
ControlFlow	66	-463	-462	-462
ECA	1263	297	297	**299**
Floats	1076	867	864	867
Heap	240	289	289	289
Loops	790	759	759	759
ProductLines	597	453	453	453
Recursive	162	132	132	132
Sequentialized	584	231	231	**236**
XCSP	119	159	159	159
Combinations	671	431	429	428
Hardware	1224	349	349	349
Hardness	4010	568	568	**570**
Total	11282	4241	4237	**4248**

In Table 5.2, contractors scored higher than other configurations. The categories where scores increased: The ECA category, with its $1,263$ benchmarks, showed a slight increase in performance. The score increased from 297, obtained using interval analysis, to 299 when contractors were used. Another category, Sequentialized, also improved after the introduction of contractors, with scores increasing from 231 to 236. Similarly, Hardness category increased from 568 to 570.

Table 5.3. This table shows the number of benchmarks that resulted in all statuses. Comparing Incremental-BMC, incremental-BMC with interval analysis, incremental-BMC with interval analysis utilizing contractors. We see that contractors have increased the number of correctly verified benchmarks while maintaining the same number in incorrectly verified benchmarks.

status	Incremental-BMC	Interval analysis	Contractor
correct	3386	3383	**3391**
correct true	1447	1446	**1449**
correct false	1939	1937	**1942**
incorrect	27	27	27
incorrect true	17	17	17
incorrect false	10	10	10
unknown	7869	**7872**	7864

On the other hand, some categories remained stable. The scores in the *BitVectors*, *Heap*, and *XCSP* categories did not change even after adding contractors, meaning no negative impact was caused on these benchmarks. In the *Combination* category, the score has decreased to 428 when using the contractors which is the only negative result in this table.

These results show that adding contractors to interval analysis for incremental BMC either preserves or slightly increases the verification score. Importantly, these improvements are obtained without sacrificing any soundness or completeness of the analysis. Moreover, Table 5.3 shows that contractors were able to solve more benchmarks and did not introduce any new false positives or false negatives.

Table 5.4. Scores of k-induction, k-induction with interval analysis, k-induction with interval analysis utilizing contractors.

Category	# of files	K-induction	Interval analysis	Contractors
Arrays	431	114	114	114
BitVectors	49	57	57	57
ControlFlow	66	-439	-439	-439
ECA	1263	1064	**1071**	1030
Floats	1076	626	**628**	625
Heap	240	295	295	295
Loops	790	765	777	**779**
ProductLines	597	605	783	783
Recursive	162	132	132	132
Sequentialized	584	227	229	**230**
XCSP	119	159	159	159
Combinations	671	**444**	441	440
Hardware	1224	**593**	591	591
Hardness	4010	7092	7090	7092
Total	11282	11734	**11928**	11888

Incremental-BMC benefited from using contractors for score metrics. However, the same cannot be said for k-induction. Table 5.5 shows the difference in scores for using contractors with k-induction. Most categories remained the same (*Arrays*, *Bitvectors*, *ControlFlow*, *Heap*, *ProductLines*, *Recursive*, *XCSP*, *Hardness*), two categories benefited slightly (Loops, Sequentialized), and three categories have reduced scores (ECA, Floats, Combination).

These results indicate that while contractors can enhance scores in some cases, they may also lead to slight reductions in others. This variability suggests that contractors' impact on scores depends on the specific characteristics of the benchmarks, as discussed in Listing. 5.5. Moreover, the ECA category has seen the most discrepancy in benchmark status. Some benchmarks reached a status when using interval analysis without contractors; when using contractors, they timeout and vice versa. Because ECA contains one of the largest `if` statements in all benchmarks. Listing 5.6 shows a sample of the many `if` statements in each file, and each file contains around 5000 lines of code. Examining the verification time of some benchmarks, we notice that the overhead introduced is high in this category, especially with k-induction. While interval analysis benefited from contractors in some benchmarks, in other benchmarks, it suffered greatly, leading to this decrease in score.

```
1  if((((a24==13) && ( a17 <=  -108  && (((a5==4) && (input == 2)) &&   ((189 <
       a2) && (281 >= a2))))) && (a7==12))){
2          a17 = ((((((a17 - 0) * 9) / 10) / 5) % 109) - -93);
3          a7 = 14;
4          a24 = 15;
5          a5 = 3;
6          return -1;
7      } else if((((a5==3) && (((-108 < a17) && (111 >= a17))  && ((a24==13) &&
       (((a7==12) || (a7==13)) && (input == 4))))) &&   ((189 < a2) && (281 >= a2
       ))))){
8          a7 = 12;
9          return 26;
10     } else if(...
11
```

Listing 5.6. sample from ECA category benchmark files.

Table 5.5 shows the number of benchmark files with each status. Unlike incremental BMC, contractors suffered from k-induction. However, the number of incorrect results remained the same, meaning we maintained the same soundness and completeness in k-induction.

Table 5.5. This table shows the number of benchmarks that resulted in all statuses. Comparing k-induction, k-induction with interval analysis, k-induction with interval analysis utilizing contractors. We notice that while interval analysis without contractors solved more files overall, contractors were able to tie with plain k-induction when it comes to the number of correct false. Overall, all methods produced the same number of incorrect results.

status	K-induction	Interval analysis	Contractor
correct	7282	**7377**	7359
correct true	5364	**5463**	5441
correct false	1918	1914	**1918**
incorrect	37	37	37
incorrect true	17	17	17
incorrect false	20	20	20
unknown	3963	3868	3886

EG2 (Soundness and completeness) Although contractors did not increase the score in *k*-induction, in both strategies, the number of incorrect results remained the same for the same benchmark files. Thus, achieving EG2.

Integrating contractors with interval analysis in incremental BMC significantly enhances verification. While this introduces some overhead, the benefits of the improvement in the verification score pay off for the added costs. Contractors improve the precision of interval analysis and, thus, increase the chances of either proving properties or finding bugs.

In *k*-induction, contractors' application is especially useful when they enhance scores. However, the impact of these will vary from case to case. When contractors result in a reduced score, deeper analysis is required to understand the root cause and develop strategies to mitigate the negative effects. Employing heuristics to determine when contractors are most effective in a given codebase can further optimize their application and maximize the overall verification efficiency.

EG3 (Trade-off) In incremental-BMC, the overhead introduced when using contractors is minimal. While *k*-induction has seen a slight decline in score due to the overhead in the ECA category. This makes us conclude that EG3 is partially achieved and can be enhanced in future work.

The results show that contractors produced greater performance improvements in Incremental-BMC than in k-induction. This difference is primarily due to the overhead introduced by contractors during k-induction. In Incremental-BMC, contractors are applied at each bounded

step to refine variable domains, resulting in immediate reductions in SMT query complexity and solver runtime. However, in k-induction, contractors must be re-applied at every induction step, across both the base case and the inductive step, which significantly increases preprocessing time. The cumulative overhead of repeated domain contraction offsets much of the potential performance gain. Consequently, while contractors still help reduce the search space in k-induction, their net benefit is less pronounced compared to Incremental-BMC.

Generally, adding contractors to interval analysis is rewarding and worth considering in both incremental BMC and k-induction. The minimal impact on efficiency and the potential for improved scores make contractors a valuable addition to interval analysis techniques. However, researchers should assess their verification tasks' specific needs and constraints to make informed decisions about their use.

5.5 Conclusion

This chapter shows the application of contractors within the framework of abstract interpretation. Its aim was to tackle a number of challenges within the software verification area, particularly state space explosion, through the proper embedding of interval methods and contractors to enhance both precision and performance in an abstract interpretation analysis.

Implementation emphasized the Ibex contractor, a tool capable of iteratively reducing variable domains in abstract domains and preserving *soundness* and *completeness* of the verification. The contractors' applications demonstrate considerable solution space reduction and the establishment of a basis for further accurate handling of the constraints. The integration optimised the detection of runtime errors, such as null pointer dereferences and memory leaks, by increasing the analysis of program behaviours while maintaining the accuracy of results.

Moreover, we presented a comprehensive evaluation that detailed the objectives, experimental setup, metrics, and results. The study's outcome revealed that, in certain cases, it significantly reduced the consumption of computational resources, notably CPU time and memory consumption, when compared with conventional abstract interpretation approaches. The use of contractors enabled a more focused exploration of the state space, thus partially overcoming the drawbacks of over-approximation inherent in abstract interpretation. Indeed, as shown by the experimental results, it is an important improvement in dealing with com-

plex program structures and nonlinear properties, where traditional approaches usually suffer from scalability issues.

Looking ahead, the lessons learned from this chapter open up new avenues for investigation. Subsequent research in this area could focus on integrating contractors with other abstract domains or developing hybrid methods that combine abstract interpretation with dynamic techniques, such as fuzzing. Additionally, performance optimizations, by parallelization or distributed computing, would also be very effective in increasing the applicability of these contractors to large-scale real-world software systems.

In summary, Chapter 5 describes the effect of contractors in the field of abstract interpretation, focusing on how they can be used to scale up and increase the effectiveness of software verification methods. This effort marks a step toward more solid and practical verification frameworks that can cope with the growing complexity of modern-day software systems.

Table 5.6. Comparison of Incremental-BMC, Interval Analysis, and Contractors Performance. The first set is plain Incremental-BMC, the second is incremental-BMC with interval analysis, and the third is incremental-BMC with interval analysis with contractors

Category	# files	Incremental-BMC			Interval Analysis			Contractors		
		CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score
Arrays	431	288,092	969,753	112	286,661	1,057,806	112	287,754	1,060,058	112
BitVectors	49	13,381	16,274	57	13,427	16,188	57	13,413	16,313	57
ControlFlow	66	16,319	48,718	-463	16,005	47,109	-462	16,069	47,165	-462
ECA	1263	902,156	4,967,636	297	909,188	3,945,520	297	911,044	3,979,001	299
Floats	1076	383,538	4,887,401	867	378,771	5,343,631	864	379,271	5,340,145	867
Heap	240	54,544	118,037	289	54,673	118,181	289	54,674	117,348	289
Loops	790	299,182	512,963	759	300,151	511,120	759	300,288	509,641	759
ProductLines	597	215,082	366,580	453	215,166	390,959	453	215,261	391,673	453
Recursive	162	63,310	172,592	132	63,743	171,805	132	63,677	170,575	132
Sequentialized	584	168,302	4,455,324	231	168,649	4,456,330	231	177,712	4,395,890	236
XCSP	119	13,813	13,760	159	13,811	13,783	159	13,827	13,946	159
Combinations	671	297,522	600,128	431	303,227	629,835	429	303,596	646,228	428
Hardware	1224	812,656	1,745,222	349	772,702	2,873,987	349	772,625	2,874,847	349
Hardness	4010	3,342,413	2,866,775	568	3,342,407	2,875,038	568	3,341,542	2,893,309	570
Total	11282	6,870,309	21,741,163	4241	6,838,582	22,451,292	4237	6,850,752	22,456,137	4248

Table 5.7. Comparison of plain K-induction, Interval Analysis, and Contractors Performance. The first set is plain K-induction, the second is K-induction with interval analysis, and the third is K-induction with interval analysis with contractors

Category	# files	K-Induction			Interval Analysis			Contractors		
		CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score
Arrays	431	282,120	1,086,093	114	281,896	1,141,289	114	283,388	1,144,968	114
BitVectors	49	13,392	16,164	57	13,390	16,061	57	13,369	16,317	57
ControlFlow	66	5,536	37,832	-439	5,466	37,826	-439	5,487	37,958	-439
ECA	1263	606,040	4,723,721	1064	616,488	4,487,395	1071	617,567	4,074,409	1030
Floats	1076	340,525	4,743,063	626	337,533	5,210,996	628	338,349	5,200,962	625
Heap	240	52,096	106,886	295	52,022	114,354	295	51,974	112,774	295
Loops	790	300,350	519,488	765	294,776	513,684	777	294,289	513,938	779
ProductLines	597	146,810	252,973	605	66,653	122,039	783	66,796	123,067	783
Recursive	162	63,273	172,947	132	63,460	172,784	132	63,477	170,948	132
Sequentialized	584	172,317	4,533,890	227	171,890	4,533,091	229	183,079	4,489,660	230
XCSP	119	13,830	13,791	159	13,839	13,715	159	13,825	13,921	159
Combinations	671	298,901	614,374	444	300,565	612,350	441	301,445	616,614	440
Hardware	1224	703,722	1,589,182	593	664,817	2,776,495	591	664,873	2,779,016	591
Hardness	4010	542,813	711,184	7092	543,294	712,692	7090	542,915	719,042	7092
Total	11282	3,541,726	19,121,590	11734	3,426,089	20,464,771	11928	3,440,834	20,013,593	11888

Table 5.8. Comparison of Incremental-BMC, Interval Analysis, and Contractors Performance. The first set is plain Incremental-BMC, the second is incremental-BMC with interval analysis, and the third is incremental-BMC with interval analysis with contractors. This table shows only the benchmarks we solved by all methods and produced either TRUE or FALSE. This comparison is made to compare resource consumption fairly.

Category	# files	Incremental-BMC			Interval Analysis			Contractors		
		CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score
Arrays	431	996	9,927	112	993	10,237	112	1,005	10,410	112
BitVectors	49	766	2,288	57	812	2,279	57	798	2,361	57
ControlFlow	66	459	4,101	-463	345	4,194	-463	368	4,286	-463
ECA	1263	33,946	272,697	292	40,569	293,025	292	41,677	299,598	292
Floats	1076	18,321	88,747	862	17,033	89,571	862	16,491	90,303	862
Heap	240	708	8,456	289	623	8,399	289	626	8,696	289
Loops	790	15,266	37,492	759	16,176	38,162	759	16,164	39,007	759
ProductLines	597	622	20,972	453	704	21,163	453	799	21,791	453
Recursive	162	2,253	9,616	132	2,280	9,613	132	2,304	9,787	132
Sequentialized	584	5,269	31,634	231	5,314	31,638	231	4,696	23,839	231
XCSP	119	2,100	7,272	159	2,098	7,273	159	2,114	7,455	159
Combinations	671	40,407	160,770	428	45,885	172,784	428	45,885	176,762	428
Hardware	1224	24,196	306,015	349	24,630	311,095	349	24,580	311,675	349
Hardness	4010	1,387	18,792	568	1,386	18,885	568	1,421	19,519	568
Total	11282	146,697	978,778	4228	158,849	1,018,317	4228	158,927	1,025,490	4228

Table 5.9. Comparison of plain K-induction, Interval Analysis, and Contractors Performance. The first set is plain K-induction, the second is K-induction with interval analysis, and the third is K-induction with interval analysis with contractors. This table shows only the benchmarks we solved by all methods and produced either TRUE or FALSE. This comparison is made to compare resource consumption fairly.

Category	# files	K-Induction			Interval Analysis			Contractors		
		CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score	CPU Time (s)	Memory (MB)	Score
Arrays	95	1,051	10,110	114	1,348	10,500	114	1,358	10,678	114
BitVectors	35	777	2,291	57	775	2,277	57	754	2,362	57
ControlFlow	60	406	4,685	-439	397	4,712	-439	416	4,825	-439
ECA	637	75,850	423,016	986	72,733	413,610	986	74,598	418,790	986
Floats	554	15,710	78,886	623	15,732	79,757	623	15,827	80,635	623
Heap	183	733	8,641	295	728	8,652	295	740	8,922	295
Loops	462	18,845	48,868	763	18,710	49,376	763	18,879	50,184	763
ProductLines	435	833	26,090	605	848	25,905	605	983	26,667	605
Recursive	93	2,251	9,616	132	2,229	9,620	132	2,247	9,795	132
Sequentialized	187	4,807	28,428	227	4,872	28,395	227	4,615	23,461	227
XCSP	106	2,116	7,249	159	2,126	7,240	159	2,111	7,451	159
Combinations	383	41,496	183,419	433	43,214	184,387	433	44,237	186,938	433
Hardware	470	24,913	328,269	591	25,822	408,295	591	25,863	408,938	591
Hardness	3562	139,194	396,833	7088	139,649	398,621	7088	139,373	404,456	7088
Total	7262	328,982	1,556,400	11634	329,183	1,631,345	11634	332,000	1,644,102	11634

Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

This thesis has explored the integration of *contractor-based interval methods* into three key software verification techniques: *Fuzzing*, *Bounded Model Checking (BMC)*, and *Abstract Interpretation*. The primary aim was to mitigate the *state space explosion problem*, which presents a major barrier to the scalability of these methods.

Research Questions

This research was structured around the following core research questions:

1. **RQ1:** To what extent can integrating a numerical method into verification techniques effectively reduce the search space?
2. **RQ2:** To what extent can the integration of interval methods—particularly contractors—into verification tools reduce computational resource consumption, such as memory and processing time?
3. **RQ3:** Does the integration of interval methods, particularly contractors, into verification frameworks preserve the soundness and completeness of these techniques?

Research Objectives

The research was conducted with the following objectives:

- **O1:** Integrate contractor-based interval methods into fuzzing, BMC, and abstract interpretation frameworks.
- **O2:** Evaluate the impact of these integrations on reducing the search space.
- **O3:** Assess the improvements in computational performance, including CPU time and memory consumption.
- **O4:** Determine whether the proposed methods preserve the soundness and completeness of verification results.

Key Findings and Answers to Research Questions

RQ1 (Search Space Reduction): Across the case studies, contractors were effective in reducing the search space by refining input domains and eliminating infeasible paths. In BMC (Chapter 4), they enabled more targeted symbolic execution through domain pruning. In fuzzing (Chapter 3), contractors guided the generation of meaningful inputs by skipping unsatisfiable conditions. These outcomes validate that the application of contractors aligns with Objectives **O1** and **O2**.

RQ2 (Computational Resource Consumption): Benchmarks from Chapters 3–5 reveal that contractors generally reduced computational costs. For example, in the `Sequentialized` category (Chapter 5), CPU time was reduced by 11.62%. However, some overheads were observed in specific categories such as `ProductLines`. These results indicate that while contractors improve performance in most cases, their benefit is context-dependent, thus fulfilling Objective **O3**.

RQ3 (Soundness and Completeness): Verification outcomes remained consistent with the baseline tools, with correct and incorrect results maintained within a 0.01% margin. This empirical stability provides strong evidence that soundness and completeness are preserved in practice. However, this does not constitute a formal proof, and further theoretical validation is recommended. The empirical evidence supports the achievement of Objective **O4**.

In summary, the integration of contractors has been shown to reduce search space, improve computational efficiency, and maintain verification reliability across multiple software

verification paradigms. These findings substantiate the thesis's contribution to enhancing the scalability and practicality of automated software verification methods.

6.2 Discussion of Research Implications

The results have important implications for the domain of software verification. The methods developed during this thesis have made tools like fuzzers and model checkers more scalable for analysing complex, real-world software systems by tackling the problems caused by the state space explosion problem. These advances are important in safety-critical systems, such as aviation, healthcare, and the automotive industry. Furthermore, the integration of contractors fills in existing gaps in verification methods, thus increasing accuracy while reducing computational needs. This dual benefit has the potential to change industry standards, making automated verification more accessible to a broader range of developers and researchers.

The study makes a methodological contribution by showing the possibility of integrating interval methods into different verification frameworks. Contractors' flexibility extends beyond a single domain, allowing them to be used in a broad range of software testing applications, including those related to security-critical applications and general software quality assurance.

6.3 Future Work

The findings of this study pave the way for multiple directions for future inquiry:

- *Advanced Optimisation Techniques:* Develop adaptive contractors that change dynamically according to the state space characteristics. Machine learning models for predicting optimal parameters are to be explored; such an approach may enhance the efficiency and adaptability of contractors in all sorts of scenarios.
- *Improved Integration of Tools:* It is necessary to improve the integration of contractors with complementary software verification tools and frameworks, especially hybrid approaches that combine static and dynamic analysis. This improvement would enable

more widespread use in different classes of software systems, including those deployed in industrial and real-time settings.

- *Real-time Applications:* Explore the use of contractors in real-time systems where computational efficiency and accuracy are critical. This exploration includes applications in autonomous vehicles, where immediate decision-making depends on trustworthy software verification.
- *Collaborative Approaches:* Discuss how contractors might be combined with other advanced verification techniques, like AI-based fuzzing or formal verification, to construct hybrid approaches that leverage the strengths of each.

The present thesis has demonstrated that the integration of interval methods and contractors in software verification frameworks improves their scalability and effectiveness. This thesis has made improvements to current methods to mitigate the state space explosion problem further, making automated verification more possible and effective for complex systems. The contributions presented in this work bring theoretical improvements and provide concrete tools and methodologies that researchers and practitioners can apply. While there are challenges and opportunities to improve, the methodologies and results presented in this work represent an important step toward ensuring the reliability and safety of software systems in a range of critical domains. The potential of the techniques presented here is huge and opens the way for further innovation within the field, laying the foundations for future research in the push of boundaries in automated software verification even further.

6.4 Concluding Remarks

The history of testing, from the early engineering approach to modern software development, demonstrates its critical role in validating system performance, reliability, and safety. testing is a bridge from theoretical design to practical implementation, saving the world from failures whose consequences could be disastrous.

Testing is vital in software development to confirm that systems meet specified requirements while being safe, reliable, and secure. Inadequate testing may lead to significant financial consequences, safety hazards, and security vulnerabilities. More comprehensive testing

strategies, both manual and automated, have become indispensable in validating correctness, reliability, and security in software, especially in safety-critical environments.

This thesis substantially contributes to addressing the challenges of software verification by introducing interval analysis and contractors to mitigate the state space explosion problem. The proposed techniques systematically extend the search space while preserving soundness and completeness by integrating contractors in fuzzing, bounded model checking (BMC), and abstract interpretation frameworks. This new approach increases the scalability and efficiency of software verification, as shown by extensive benchmarking while reducing computational costs and maintaining accuracy in identifying potential vulnerabilities.

This research widens the scope of existing verification methodologies and responds to one of the overarching needs: ensuring that software systems are resilient, secure, and dependable. These principles can drive further improvements in software quality assurance, reducing risks associated with the security of technological frameworks supporting our modern existence.

References

[1] M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro, "Fusebmc_ia: Interval analysis and methods for test case generation," in *Fundamental Approaches to Software Engineering*, L. Lambers and S. Uchitel, Eds., Cham: Springer Nature Switzerland, 2023, pp. 324–329, ISBN: 978-3-031-30826-0 (cited on pp. 11, 35).

[2] R. S. Menezes, M. Aldughaim, B. Farias, *et al.*, "Esbmc v7.4: Harnessing the power of intervals," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2024, pp. 376–380, ISBN: 978-3-031-57256-2 (cited on pp. 11, 12, 35).

[3] R. S. Menezes, E. Manino, F. Shmarov, M. Aldughaim, R. de Freitas, and L. C. Cordeiro, *Interval analysis in industrial-scale bmc software verifiers: A case study*, 2024. arXiv: 2406.15281 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2406.15281 (cited on pp. 11, 35).

[4] J. Deveza, M. Aldughaim, R. S. Menezes, L. Cordeiro, and R. de Freitas, *Reducing the bmc formal verification state-space by a cp preprocessing with variable-domain interval contraction*, 2024 (cited on pp. 11, 35).

[5] M. Aldughaim, K. Alshmrany, and L. Cordeiro, "Working title: Interval analysis and methods in testcase generation," 2024 (cited on p. 11).

[6] M. Aldughaim, K. Alshmrany, R. Menezes, L. Cordeiro, and A. Stancu, "Incremental symbolic bounded model checking of software using interval methods via contractors," *arXiv preprint arXiv:2012.11245*, 2020 (cited on pp. 12, 35, 73).

[7] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs," in *International Conference On Tests And Proofs*, Springer, 2021, pp. 85–105 (cited on pp. 12, 64, 66, 73).

[8] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro, "Fusebmc v4: Smart seed generation for hybrid fuzzing:(competition contribution)," in *International Conference On Fundamental Approaches To Software Engineering*, Springer International Publishing Cham, 2022, pp. 336–340 (cited on pp. 12, 64, 66, 73).

[9] K. Alshmrany, M. Aldughaim, A. Bhayat, and L. Cordeiro, "Fusebmc v4: Improving code coverage with smart seeds via bmc, fuzzing and static analysis," *Form. Asp. Comput.*, vol. 36, no. 2, Jun. 2024, ISSN: 0934-5043. DOI: 10.1145/3665337. [Online]. Available: https://doi.org/10.1145/3665337 (cited on p. 12).

[10] J. M. Juran, *Juran's quality handbook*. 1999 (cited on p. 18).

[11] M. F. Bado, D. Tonelli, F. Poli, D. Zonta, and J. R. Casas, "Digital twin for civil engineering systems: An exploratory review for distributed sensing updating," *Sensors*, vol. 22, no. 9, p. 3168, 2022 (cited on p. 18).

[12] A. Miczo, *Digital logic testing and simulation*. John Wiley & Sons, 2003 (cited on p. 18).

[13] D. Hoyle, *ISO 9000 Quality Systems Handbook-updated for the ISO 9001: 2015 standard: Increasing the Quality of an Organization's Outputs*. Routledge, 2017 (cited on p. 18).

[14] H. Petroski, *Engineers of Dreams: Great Bridge Builders and the Spanning of America*. New York: Alfred A. Knopf, 1992 (cited on p. 18).

[15] I. Sommerville, *Software Engineering*, 10th. Pearson, 2016, ISBN: 978-0133943030 (cited on pp. 18, 21).

[16] P. Koopman and M. Wagner, *Better Embedded System Software*. Phil Koopman, 2019 (cited on p. 19).

[17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004 (cited on p. 19).

[18] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 510–525, 2009 (cited on p. 19).

[19] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, 1993 (cited on p. 19).

[20] D. Travis, "Boeing 737 max: An artificial stability problem," *IEEE Spectrum*, 2019, Available online: https://spectrum.ieee.org/boeing-737-max8-crashes (cited on p. 19).

[21] I. Board, "Ariane 5 flight 105 inquiry board report," tech. rep., European Space Agency Press, Tech. Rep., 1996 (cited on p. 19).

[22] S. Al-Saqqa, S. Sawalha, and H. AbdelNabi, "Agile software development: Methodologies and trends.," *International Journal of Interactive Mobile Technologies*, vol. 14, no. 11, 2020 (cited on p. 20).

[23] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2017 (cited on pp. 20, 23).

[24] S. Desikan and G. Ramesh, *Software testing: principles and practice*. Pearson Education India, 2006 (cited on p. 20).

[25] PyCQA, *Pylint*, Version 2.7.4, Python Code Quality Authority. [Online]. Available: https://pylint.pycqa.org/ (cited on p. 20).

[26] PyCQA, *Flake8*, Version 3.9.2, Python Code Quality Authority, 2021. [Online]. Available: https://flake8.pycqa.org/ (cited on p. 20).

[27] N. C. Zakas and the ESLint Team, *Eslint*, Version 7.32.0. [Online]. Available: https://eslint.org/ (cited on p. 20).

[28] LLVM Project, *Clang-tidy*, Version 12.0.0, LLVM Foundation. [Online]. Available: https://clang.llvm.org/extra/clang-tidy/ (cited on p. 20).

[29] SonarSource, *Sonarqube: Continuous inspection of code quality*, Accessed: 2024-09-23, SonarSource S.A. [Online]. Available: https://www.sonarqube.org/ (cited on p. 20).

[30] LLVM Developer Group, *Clang static analyzer*, Accessed: 2024-09-23, LLVM Project. [Online]. Available: https://clang-analyzer.llvm.org/ (cited on p. 20).

[31] M. Focus, *Fortify static code analyzer*, Accessed: 2024-09-23, Micro Focus International. [Online]. Available: https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview (cited on p. 20).

[32] Synopsys, *Coverity static analysis*, Accessed: 2024-09-23, Synopsys, Inc. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html (cited on p. 20).

[33] Perforce, *What is static analysis? static code analysis overview*, Accessed: 2023-09-04, 2023. [Online]. Available: https://www.perforce.com (cited on p. 20).

[34] GrammaTech, "The role of static analysis in a secure software development life cycle (sdlc)," *GrammaTech.com*, 2023, Accessed: 2023-09-04. [Online]. Available: https://www.grammatech.com/ (cited on p. 20).

[35] Codacy, "Static code analysis: Everything you need to know," *Codacy Blog*, 2023, Accessed: 2023-09-04. [Online]. Available: https://blog.codacy.com/ (cited on pp. 20, 21).

[36] S. Times, "The importance of prevention: How shifting left, static analysis and unit testing create better code quality," *SD Times*, 2023, Accessed: 2023-09-04. [Online]. Available: https://www.sdtimes.com/ (cited on p. 20).

[37] G. Sherwood, *Php codesniffer: Coding standard compliance for php*, Accessed: 2024-09-23, Squiz Labs. [Online]. Available: https://github.com/squizlabs/PHP_CodeSniffer (cited on p. 21).

[38] C. Team, *Checkstyle: Static code analysis for java*, Accessed: 2024-09-23, Checkstyle Project. [Online]. Available: https://checkstyle.sourceforge.io/ (cited on p. 21).

[39] T. Parr, *Antlr (another tool for language recognition)*, Accessed: 2024-09-23, ANTLR Project. [Online]. Available: https://www.antlr.org/ (cited on p. 21).

[40] Koalaman, *Shellcheck: A shell script static analysis tool*, Accessed: 2024-09-23, Open Source. [Online]. Available: https://www.shellcheck.net/ (cited on p. 21).

[41] F. E. Allen, "Control flow analysis," *Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970 (cited on p. 21).

[42] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., 1981 (cited on p. 21).

[43] D. Jackson, "Dependency analysis for software architecture," in *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM, 1994 (cited on p. 21).

[44] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976 (cited on p. 21).

[45] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007 (cited on p. 21).

[46] M. L. Scott, *Programming Language Pragmatics*. Morgan Kaufmann, 2009 (cited on p. 22).

[47] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science (LNCS), vol. 1579, Springer, Berlin, Heidelberg, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14 (cited on pp. 22, 24).

[48] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, pp. 7–34, 2001 (cited on p. 22).

[49] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," *Adv. Comput*, vol. 58, no. C, pp. 117–148, 2003 (cited on pp. 22, 25, 42, 43, 45).

[50] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252 (cited on pp. 22, 36, 54, 55, 60, 61).

[51] B. Blanchet, P. Cousot, R. Cousot, *et al.*, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003, pp. 196–207. DOI: 10.1145/781131.781153 (cited on pp. 22, 23, 55, 62).

[52] E. M. Clarke, D. Kroening, and F. 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 (cited on p. 23).

[53] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2005, pp. 213–223 (cited on p. 23).

[54] R. Jain, *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. Wiley-Interscience, 1991, Covers methodologies and techniques for performance testing, emphasizing the importance of measuring system behavior under various conditions. (cited on pp. 23, 24).

[55] B. P. Miller *et al.*, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," University of Wisconsin-Madison Department of Computer Sciences, CS-TR 1995-1268, 1995, Introduces fuzzing as an automated technique to generate random inputs for finding security vulnerabilities. (cited on pp. 23–25).

[56] K. Sen *et al.*, "Cute: A concolic unit testing engine for c," *SIGSOFT Software Engineering Notes*, vol. 30, no. 5, 2005, Describes concolic testing, a technique that merges concrete and symbolic execution for thorough execution path exploration. (cited on pp. 23, 24).

[57] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007, ISBN: 9780321446113 (cited on p. 24).

[58] T. Ball, "The concept of dynamic analysis," *ESSYM*, 1999, Explores the methodology and application of observing software execution to identify runtime errors. (cited on p. 24).

[59] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008 (cited on pp. 24, 27, 28).

[60] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 1996 (cited on p. 24).

[61] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997 (cited on p. 24).

[62] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012, ISSN: 00985589. DOI: 10.1109/TSE.2011.59. [Online]. Available: http://www.esbmc.org (cited on pp. 24, 42, 43, 45, 46, 82).

[63] C. Barrett, R. Sebastiani, S. A. Seshia, *et al.*, "Handbook of satisfiability," *Satisfiability modulo theories*, vol. 185, pp. 825–885, 2009 (cited on pp. 24, 45).

[64] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30, ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. [Online]. Available: https://doi.org/10.1007/978-3-642-35746-6_1 (cited on p. 25).

[65] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, 1994 (cited on p. 25).

[66] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999 (cited on pp. 25, 26, 28, 46).

[67] K. L. McMillan, "Interpolation and SAT-based model checking," in *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, Springer, 2003, pp. 1–13 (cited on pp. 25, 26).

[68] A. R. Bradley, "Sat-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2011, pp. 70–87 (cited on pp. 25, 26).

[69] R. Bloem, K. Greimel, T. A. Henzinger, and B. Könighofer, "Synthesizing robust systems," *Acta Informatica*, vol. 51, no. 3-4, pp. 193–220, 2014 (cited on p. 25).

[70] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive (1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012 (cited on p. 25).

[71] P. Cousot and R. Cousot, "Abstract interpretation: Past, present and future," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2014, pp. 1–10 (cited on p. 25).

[72] P. Cousot, "Formal verification by abstract interpretation," in *NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings 4*, Springer, 2012, pp. 3–7 (cited on pp. 25, 26).

[73] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–38, 2023 (cited on p. 25).

[74] J. Ruthruff, R. C. Armstrong, B. G. Davis, J. R. Mayo, R. J. Punnoose, *et al.*, "Leveraging formal methods and fuzzing to verify security and reliability properties of large-scale high-consequence systems," Sandia National Lab.(SNL-CA), Livermore, CA (United States), Tech. Rep., 2012 (cited on pp. 25, 27).

[75] J. Bozic, S. Kremer, A. Legay, and S. Sedwards, "Model-based fuzz testing for robustness analysis," in *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2017, pp. 123–132 (cited on p. 25).

[76] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996, ISBN: 9783540614513 (cited on p. 25).

[77] U. Stern and D. L. Dill, "Parallelizing the Murphi verifier," in *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, Springer, 1997, pp. 256–278 (cited on p. 26).

[78] K. Sen, D. Marinov, and G. Agha, "Concolic testing," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New York, NY, USA: ACM, 2005, pp. 252–262 (cited on p. 27).

[79] E. M. Clarke, "Model checking: Back and forth between theory and practice," *IFIP Advances in Information and Communication Technology*, vol. 52, no. 1, pp. 5–26, 2012 (cited on p. 27).

[80] M. M. A. Mustafa, "Guaranteed SLAM—An interval approach," Ph.D. dissertation, Manchester University, 2017 (cited on pp. 28, 48, 49, 51–54, 66, 67, 85).

[81] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020, ISBN: 9780262356657. [Online]. Available: https://books.google.co.uk/books?id=-aLLDwAAQBAJ (cited on pp. 36, 37, 55–61, 99).

[82] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, "Interval Analysis," in *Applied Interval Analysis*, Springer, London, 2001, ch. 2, pp. 11–43, ISBN: 978-1-4471-0249-6. DOI: 10.1007/978-1-4471-0249-6{_}2 (cited on pp. 36, 48–50).

[83] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2018 (cited on p. 37).

[84] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004 (cited on p. 38).

[85] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *ACM Symposium on Principles of Programming Languages*, 2002, pp. 238–252 (cited on p. 38).

[86] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969 (cited on p. 38).

[87] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990 (cited on p. 38).

[88] C. Miller and Z. N. J. Peterson, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Pearson Education, 2006 (cited on pp. 38, 39).

[89] Google, *Oss-fuzz - continuous fuzzing for open source software*, 2020. [Online]. Available: https://github.com/google/oss-fuzz (cited on p. 38).

[90] M. Eceiza, J. L. Flores, and M. Iturbe, "Fuzzing the internet of things: A review on the techniques and challenges for efficient vulnerability discovery in embedded systems," *IEEE Internet of Things Journal*, vol. 8, no. 13, pp. 10 390–10 411, 2021 (cited on p. 39).

[91] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007 (cited on pp. 39, 40).

[92] J. Silveira and N. Ferreira, "Tpvis: A visual analytics system for exploring test case prioritization methods," *Available at SSRN 4820768*, 2024. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4820768 (cited on p. 39).

[93] MITRE, *Common vulnerabilities and exposures (cve)*, https://cve.mitre.org/, A list of publicly known cybersecurity vulnerabilities. (cited on p. 40).

[94] *American fuzzy lop*, http://lcamtuf.coredump.cx/afl/, Accessed: 2024-05-15 (cited on pp. 40, 41).

[95] *Libfuzzer - a library for coverage-guided fuzz testing*, https://llvm.org/docs/LibFuzzer.html, Accessed: 2024-05-15 (cited on pp. 40, 41).

[96] *Honggfuzz*, https://github.com/google/honggfuzz, Accessed: 2024-05-15 (cited on pp. 40, 41).

[97] *Fusebmc*, https://fusebmc.github.io/, Accessed: 2024-05-15 (cited on pp. 40, 41).

[98] *Oss-fuzz - google's continuous fuzzing service for open source software*, https://google.github.io/oss-fuzz/, Accessed: 2024-05-15 (cited on pp. 40, 41).

[99] *Radamsa*, https://gitlab.com/akihe/radamsa, Accessed: 2024-05-15 (cited on pp. 40, 41).

[100] *Peach fuzzer*, https://www.peach.tech/, Accessed: 2024-05-15 (cited on pp. 40, 41).

[101] *Boofuzz*, https://github.com/jtpereyda/boofuzz, Accessed: 2024-05-15 (cited on pp. 40, 41).

[102] *Atheris: A coverage-guided python fuzzing engine*, https://github.com/google/atheris, Accessed: 2024-05-15 (cited on pp. 40, 41).

[103] *Syzkaller - unsupervised, coverage-guided kernel fuzzer*, https://github.com/google/syzkaller, Accessed: 2024-05-15 (cited on pp. 40, 41).

[104] P. Amini, "Fuzzing framework," *Black Hat USA*, vol. 14, pp. 211–217, 2007 (cited on p. 41).

[105] A. Author and B. Author, "Title of chapter," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. Amsterdam, Netherlands: IOS Press, 2009, ch. 18, pp. xxx–xxx (cited on p. 42).

[106] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: An industrial-strength C model checker," in *ASE*, ACM, 2018, pp. 888–891 (cited on pp. 42, 44, 91).

[107] D. Kroening and M. 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 (cited on pp. 42, 46, 92).

[108] S. S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers Inc., 1997, ISBN: 1558603204 (cited on p. 42).

[109] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991 (cited on p. 42).

[110] S. A. Kripke, "Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi," *Mathematical Logic Quarterly*, vol. 9, no. 5-6, pp. 67–96, 1963, ISSN: 15213870. DOI: 10.1002/malq.19630090502 (cited on p. 42).

[111] A. R. Bradley and Z. Manna, *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007 (cited on p. 45).

[112] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: An industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 888–891 (cited on p. 45).

[113] O. M. Alhawi, H. Rocha, M. R. Gadelha, L. C. Cordeiro, and E. Batista, "Verification and refutation of c programs based on k-induction and invariant inference," *International journal on software tools for technology transfer*, vol. 23, pp. 115–135, 2021 (cited on p. 45).

[114] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t)," *Journal of the ACM (JACM)*, vol. 53, no. 6, pp. 937–977, 2006 (cited on pp. 45, 46).

[115] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, no. 53, pp. 1–2, 2005 (cited on p. 45).

[116] L. De Moura and N. 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 (cited on p. 45).

[117] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and applications of satisfiability testing*, Springer, 2003, pp. 502–518 (cited on p. 45).

[118] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, 2008, pp. 337–340 (cited on pp. 45, 46).

[119] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*, Springer, 2014, pp. 737–744 (cited on p. 45).

[120] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for dpll(t)," in *Computer Aided Verification*, Springer Berlin Heidelberg, 2006, pp. 81–94 (cited on p. 46).

[121] Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986, ISSN: 1557-9956. DOI: 10.1109/TC.1986.1676819 (cited on p. 46).

[122] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1999, pp. 193–207 (cited on p. 46).

[123] M. Christakis, P. Müller, and S. Schneider, "Distributed bounded model checking," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, 2012, pp. 89–99 (cited on p. 46).

[124] A. Mishchenko and R. Brayton, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification*, Springer Berlin Heidelberg, 2010, pp. 24–40 (cited on p. 46).

[125] E. M. Clarke and O. Grumberg, "Abstraction and refinement for model checking," *Logic Journal of the IGPL*, vol. 10, no. 4, pp. 421–432, 2004 (cited on p. 46).

[126] M. R. Gadelha, R. S. Menezes, and L. C. Cordeiro, "Esbmc 6.1: Automated test case generation using bounded model checking," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 857–861, 2021 (cited on p. 46).

[127] I. Bessa, H. Ismail, R. M. Palhares, L. C. Cordeiro, and J. E. C. Filho, "Formal non-fragile stability verification of digital control systems with uncertainty," *IEEE Trans. Computers*, vol. 66, no. 3, pp. 545–552, 2017. DOI: 10.1109/TC.2016.2601328. [Online]. Available: https://doi.org/10.1109/TC.2016.2601328 (cited on p. 46).

[128] R. B. Abreu, M. Y. R. Gadelha, L. C. Cordeiro, E. B. de Lima Filho, and W. S. da Silva Jr., "Bounded model checking for fixed-point digital filters," *J. Braz. Comput. Soc.*, vol. 22, no. 1, 1:1–1:20, 2016. DOI: 10.1186/s13173-016-0041-8. [Online]. Available: https://doi.org/10.1186/s13173-016-0041-8 (cited on p. 46).

[129] L. C. Chaves, I. Bessa, H. Ismail, A. B. dos Santos Frutuoso, L. C. Cordeiro, and E. B. de Lima Filho, "Dsverifier-aided verification applied to attitude control software in unmanned aerial vehicles," *IEEE Trans. Reliab.*, vol. 67, no. 4, pp. 1420–1441, 2018. DOI: 10.1109/TR.2018.2873260. [Online]. Available: https://doi.org/10.1109/TR.2018.2873260 (cited on p. 46).

[130] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022. DOI: 10.1002/stvr.1793. [Online]. Available: https://doi.org/10.1002/stvr.1793 (cited on p. 46).

[131] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *arXiv preprint arXiv:1611.03688*, 2016 (cited on p. 46).

[132] L. Kovács and A. Voronkov, "Automated reasoning for software verification: Bmc and beyond," in *International Conference on Automated Deduction*, Springer, 2013, pp. 337–354 (cited on p. 47).

[133] T. N. Nguyen, A. Legay, P. Godefroid, and D. Monniaux, "Using machine learning to improve automatic verification," in *Artificial Intelligence and Statistics*, PMLR, 2017, pp. 261–269 (cited on p. 47).

[134] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008 (cited on p. 47).

[135] N. e. a. Stephens, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016 (cited on p. 47).

[136] D. Beyer and M. Keremoglu, "Cpachecker: A tool for configurable software verification," in *CAV*, 2011 (cited on p. 47).

[137] M. e. a. Heizmann, "Ultimate automizer with unsatisfiable cores for error location," in *SV-COMP*, 2022 (cited on p. 47).

[138] M. A. Sainz, J. Armengol, R. Calm, P. Herrero, L. Jorba, and J. Vehi, *Modal Interval Analysis*. Springer, 2014, vol. 2091 (cited on p. 48).

[139] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter, "Applied Interval Analysis," in *Springer London*, 2001 (cited on pp. 52, 66, 67, 96).

[140] E. Hansen and G. W. Walster, *Global optimization using interval analysis: revised and expanded*. CRC Press, 2003, vol. 264 (cited on p. 53).

[141] A. Neumaier and A. Neumaier, *Interval methods for systems of equations*. Cambridge university press, 1990, vol. 37 (cited on p. 53).

[142] L. Granvilliers, "Revising hull and box consistency," *Logic Programming*, pp. 230–244, 1999. DOI: 10.7551/mitpress/4304.003.0024 (cited on pp. 53, 82).

[143] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999, ISBN: 978-3-540-65410-0 (cited on p. 55).

[144] P. Cousot, *Principles of abstract interpretation*. MIT Press, 2021 (cited on pp. 55, 60, 61, 99).

[145] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1979, pp. 269–282. DOI: 10.1145/567752.567778 (cited on p. 55).

[146] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*, Springer, 2012, pp. 233–247 (cited on pp. 62, 64, 77).

[147] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Moving fast with software verification," in *NASA Formal Methods Symposium*, Springer, 2015, pp. 3–11 (cited on p. 62).

[148] D. Beyer, "Advances in automatic software testing: Test-comp 2022.," in *FASE*, 2022, pp. 321–335 (cited on pp. 64, 70, 77).

[149] M. Aldughaim, K. Alshmrany, R. Menezes, L. Cordeiro, and A. Stancu, "Incremental symbolic bounded model checking of software using interval methods via contractors," *arXiv preprint arXiv:2012.11245*, 2020 (cited on pp. 64, 67).

[150] P. Baudin, F. Bobot, D. Bühler, *et al.*, "The dogged pursuit of bug-free c programs: The frama-c software analysis platform," *Communications of the ACM*, vol. 64, no. 8, pp. 56–68, 2021 (cited on pp. 64, 77).

[151] D. Bühler, "Eva, an evolved value analysis for frama-c: Structuring an abstract interpreter through value and state abstractions," Ph.D. dissertation, Rennes 1, 2017 (cited on pp. 66, 68, 77).

[152] G. Chabert, *Https://github.com/ibex-team/ibex-lib*. [Online]. Available: https://github.com/ibex-team/ibex-lib/ (cited on pp. 67, 68).

[153] *Clang documentation*, http://clang.llvm.org/docs/index.html, [Online; accessed August-2019], 2015 (cited on p. 68).

[154] D. Beyer and T. Lemberger, "Testcov: Robust test-suite execution and coverage measurement," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 1074–1077 (cited on p. 71).

[155] D. Beyer, "State of the art in software verification and witness validation: SV-COMP 2024," in *Proc. TACAS (3)*, ser. LNCS 14572, Springer, 2024, pp. 299–329. DOI: 10.1007/978-3-031-57256-2_15 (cited on pp. 71, 105).

[156] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2016, pp. 887–904 (cited on pp. 72, 92, 105).

[157] D. Beyer and M.-C. Jakobs, "Coveritest: Cooperative verifier-based testing.," in *FASE*, 2019, pp. 389–408 (cited on p. 73).

[158] M.-C. Jakobs and C. Richter, "Coveritest with adaptive time scheduling (competition contribution)," *Fundamental Approaches to Software Engineering*, vol. 12649, p. 358, 2021 (cited on p. 73).

[159] M. R. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "Esbmc v6. 0: Verifying c programs using k-induction and invariant inference: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III 25*, Springer, 2019, pp. 209–213 (cited on p. 73).

[160] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *International journal on software tools for technology transfer*, vol. 19, no. 1, pp. 97–114, 2017 (cited on p. 73).

[161] M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro, "Fusebmc_ia: Interval analysis and methods for test case generation: (competition contribution)," in *International Conference on Fundamental Approaches to Software Engineering*, Springer, 2023, pp. 324–329 (cited on p. 73).

[162] J. Bürdek, M. Lochau, S. Bauregger, *et al.*, "Facilitating reuse in multi-goal test-suite generation for software product lines," in *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18*, Springer, 2015, pp. 84–99 (cited on p. 73).

[163] S. Ruland, M. Lochau, and M.-C. Jakobs, "Hybridtiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution)," *Fundamental Approaches to Software Engineering*, vol. 12076, p. 520, 2020 (cited on p. 73).

[164] Cadar, Cristian, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224 (cited on p. 73).

[165] C. Cadar and M. Nowack, "Klee symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 867–870, 2021 (cited on p. 73).

[166] D. Liu, G. Ernst, T. Murray, and B. I. Rubinstein, "Legion: Best-first concolic testing (competition contribution).," in *FASE*, 2020, pp. 545–549 (cited on p. 73).

[167] D. Liu, G. Ernst, T. Murray, and B. I. Rubinstein, "Legion: Best-first concolic testing," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 54–65 (cited on p. 73).

[168] T. Lemberger, "Plain random test generation with prtest," *International Journal on Software Tools for Technology Transfer*, vol. 23, pp. 871–873, 2021 (cited on p. 73).

[169] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking: A comparative evaluation of the state of the art," in *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13*, Springer, 2017, pp. 99–114 (cited on p. 73).

[170] M. Chalupa, J. Novák, and J. Strejcek, "Symbiotic 8: Parallel and targeted test generation," *Fundamental Approaches to Software Engineering*, pp. 368–372, 2021 (cited on p. 73).

[171] M. Chalupa, J. Strejček, and M. Vitovská, "Joint forces for memory safety checking," in *Model Checking Software: 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings 25*, Springer, 2018, pp. 115–132 (cited on p. 73).

[172] J. Jaffar, R. Maghareh, S. Godboley, and X.-L. Ha, "Tracerx: Dynamic symbolic execution with interpolation (competition contribution)," *Fundamental Approaches to Software Engineering*, vol. 12076, p. 530, 2020 (cited on p. 73).

[173] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "Tracer: A symbolic execution tool for verification," in *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, Springer, 2012, pp. 758–766 (cited on p. 73).

[174] R. Metta, R. K. Medicherla, and H. Karmarkar, "Verifuzz: Good seeds for fuzzing (competition contribution).," in *FASE*, 2022, pp. 341–346 (cited on p. 73).

[175] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic execution for webassembly," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022 (cited on p. 73).

[176] D. Beyer, "Software testing: 5th comparative evaluation: Test-comp 2023.," in *FASE*, 2023, pp. 309–323 (cited on pp. 75, 77).

[177] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002. DOI: 10.1109/32.988497 (cited on p. 75).

[178] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012. DOI: 10.1002/stvr.430 (cited on p. 75).

[179] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' perceptions of software testing: An exploratory study," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2015, pp. 1–10. DOI: 10.1109/ESEM.2015.7321199 (cited on p. 75).

[180] J. Little and E. Tsang, *Foundations of Constraint Satisfaction.* (Computation in cognitive science 5). Academic Press, 1995, p. 666, ISBN: 9780127016108. DOI: 10.2307/2584541. [Online]. Available: https://books.google.co.uk/books?id=TnxQAAAAMAAJ (cited on p. 81).

[181] L. C. Cordeiro, "Smt-based bounded model checking of multi-threaded software in embedded systems," Ph.D. dissertation, University of Southampton, UK, 2011. [Online]. Available: http://eprints.soton.ac.uk/186011/ (cited on p. 82).

[182] L. Granvilliers and F. Benhamou, "Algorithm 852: Realpaver: An interval solver using constraint satisfaction techniques," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 1, pp. 138–156, 2006 (cited on pp. 88, 101).

[183] P. Darke, S. Agrawal, and R. Venkatesh, "Veriabs: A tool for scalable verification by abstraction (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2021, pp. 458–462 (cited on p. 92).

[184] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 184–190 (cited on p. 92).

[185] A. B. Chowdhury, R. K. Medicherla, and R. Venkatesh, "Verifuzz: Program aware fuzzing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019, pp. 244–249 (cited on p. 92).

[186] D. Beyer, "Progress on software verification: Sv-comp 2022," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., Cham: Springer International Publishing, 2022, pp. 375–402 (cited on p. 92).

[187] A. Cortesi and M. Zanioli, "Widening and narrowing operators for abstract interpretation," *Computer Languages, Systems & Structures*, vol. 37, no. 1, pp. 24–42, 2011 (cited on p. 100).

Appendices

Appendix A

Benchmark Suites

This appendix provides a detailed description of the benchmark suites employed in this thesis—**SV-COMP** (Software Verification Competition) and **Test-Comp** (Testing Competition)—as well as the associated scoring mechanisms. These benchmarks are the de facto standards for evaluating software verification and test-generation tools. Their inclusion in this work ensures methodological rigour, fairness, and reproducibility of results.

A.1 SV-COMP: Software Verification Competition

SV-COMP[1] is an annual competition that assesses the capabilities of automated software verifiers. It focuses on C and C++ programs, providing a wide-ranging collection of verification tasks that reflect real-world and synthetic use cases. The primary goal of SV-COMP is to compare verification tools based on their *precision, soundness, scalability, and efficiency* in proving or refuting properties of software systems.

Each verification task comprises a program annotated with specifications, typically expressed as assertions. Tools are required to determine whether these specifications hold (`TRUE`) or are violated (`FALSE`). If the analysis is inconclusive—due to a timeout, crash, or incomplete reasoning—the result is recorded as `UNKNOWN`.

Tasks are organised into categories, each addressing a distinct class of verification challenges. These are further subdivided into subcategories that reflect varying levels of complexity, domain-specific characteristics, or particular features of the source programs. The major categories and their subcategories for SV-COMP 2024 include:

[1] https://sv-comp.sosy-lab.org/

- `ReachSafety`: Focuses on verifying reachability properties, particularly the absence of assertion violations. Subcategories include:
  - `ReachSafety-Arrays` – programs involving array manipulations,
  - `ReachSafety-BitVectors` – bit-level operations and overflows,
  - `ReachSafety-ControlFlow` – intricate control flow patterns,
  - `ReachSafety-ECA` – event-condition-action systems,
  - `ReachSafety-Floats` – floating-point computations,
  - `ReachSafety-Heap` – dynamic memory and pointer analysis.
- `MemSafety`: Evaluates memory safety properties such as absence of invalid dereferences, memory leaks, or double-free errors. Subcategories include:
  - `MemSafety-InvalidDereference`,
  - `MemSafety-InvalidFree`,
  - `MemSafety-MemCleanup`.
- `ConcurrencySafety`: Focuses on multi-threaded programs, aiming to detect data races, deadlocks, and assertion violations in concurrent contexts.
- `NoDataRace`: Verifies the absence of data races in concurrent programs.
- `Termination`: Concerns itself with proving program termination, which is crucial for sound reasoning about loops and recursive functions.
- `NoOverflow`: Verifies the absence of signed and unsigned integer overflows.
- `SoftwareSystems`: Consists of larger and more complex programs taken from real-world software projects. This category assesses the scalability and robustness of tools under realistic conditions.
- `ProductLines`: Focuses on software with configurable features (e.g., via preprocessor macros).

- `ControlFlowIntegers`: Targets programs with intensive control-flow and integer variable use.

The scoring system in SV-COMP rewards correctness and penalises unsoundness. A correct `TRUE` or `FALSE` yields positive points, while an incorrect classification results in significant penalties. For instance, correctly reporting a violation might grant +1 point, while a wrong result (e.g., reporting `TRUE` when a violation exists) incurs a penalty as large as −16 points. `UNKNOWN` results typically receive a neutral score (0), but frequent `UNKNOWN`s can negatively affect a tool's overall standing by reducing its coverage.

The SV-COMP scoring model promotes not only high precision but also discourages speculative or unsound outputs. This directly aligns with this thesis's emphasis on preserving soundness and completeness, particularly when introducing new analysis mechanisms such as interval contractors.

A.2 Test-Comp: Testing Competition

Test-Comp[2] focuses on automated test-case generation rather than formal verification. Its goal is to evaluate how effectively tools can produce test inputs that expose bugs, achieve high coverage, or satisfy specific criteria in C programs. The competition benchmarks are diverse, ranging from simple toy examples to complex system-level programs, and are chosen to challenge the breadth and depth of test-generation strategies.

In contrast to SV-COMP, the scoring in Test-Comp is based on *quantitative metrics* such as statement and branch coverage, bug detection, and efficiency. Tools are ranked based on the amount of code they can exercise through generated tests. Higher code coverage and successful bug findings lead to increased scores. Execution time and energy consumption are also considered, with more efficient tools receiving favourable assessments.

Test-Comp 2023 organises its benchmarks into two primary categories, each containing multiple subcategories:

- **Cover-Error**: This category assesses the ability of tools to generate test cases that reach error conditions. Subcategories include:

[2] https://test-comp.sosy-lab.org/

  - `ReachSafety-Arrays`
  - `ReachSafety-BitVectors`
  - `ReachSafety-ControlFlow`
  - `ReachSafety-ECA`
  - `ReachSafety-Floats`
  - `ReachSafety-Heap`
  - `ReachSafety-Loops`
  - `ReachSafety-ProductLines`
  - `ReachSafety-Recursive`
  - `ReachSafety-Sequentialized`
  - `ReachSafety-XCSP`
  - `ReachSafety-Hardware`
  - `SoftwareSystems-BusyBox-MemSafety`
  - `SoftwareSystems-DeviceDriversLinux64-ReachSafety`

- **Cover-Branches**: This category evaluates the extent of branch coverage achieved by the generated test cases. Subcategories mirror those of Cover-Error, focusing on the same program sets but with coverage specifications.

The scoring system in Test-Comp is based on quantitative metrics:

- **Bug Finding**: A tool earns +1 point for each test suite that successfully triggers a specification violation within the time limit. No points are awarded otherwise.
- **Code Coverage**: The score corresponds to the fraction of branches covered, as reported by the `TestCov` tool. For example, achieving 80% branch coverage yields 0.8 points.

- **Efficiency**: Tools are ranked based on the total CPU time consumed for successful test generations. In cases of equal scores, tools with lower cumulative CPU time are ranked higher.

Meta-categories aggregate performance across multiple subcategories, with scores normalised to ensure fairness regardless of category size. This structure allows for a comprehensive assessment of a tool's capabilities across diverse testing scenarios.

In this thesis, Test-Comp is used to benchmark the fuzzing-based tool *FuSeBMC* and its contractor-augmented version, *FuSeBMC_IA*. The results demonstrate how contractors contribute to improving coverage and bug-finding capability by refining input generation and pruning infeasible execution paths. These insights are key to answering the research questions on scalability and effectiveness.

A.3 Relevance to This Thesis

By employing SV-COMP and Test-Comp, this thesis benefits from standardised, peer-reviewed benchmarks that ensure comparability and reproducibility. These benchmarks provide not only a robust framework to assess the correctness and performance of verification tools but also a neutral environment to evaluate the integration of interval-based contractors.

The scoring methodologies reinforce the importance of sound and efficient analysis—central themes of this research. By examining the results within these benchmarks, the thesis is able to provide concrete, empirical answers to the stated research questions concerning search-space reduction, computational overhead, and the preservation of verification guarantees.

Appendix B

Tool Foundations

This appendix presents the foundational tools utilised in this thesis: **FuSeBMC**[1] and **ESBMC**[2]. These verification frameworks provide the infrastructure upon which the proposed contractor-based interval analysis techniques were implemented and evaluated. Their capabilities and extensibility were key to validating the methods developed throughout the thesis.

B.1 FuSeBMC

FuSeBMC, or *Fuzzing with Symbolic Execution-Based Model Checking*, is a hybrid software testing tool designed to enhance the efficiency of test case generation for C programs. It achieves this by combining dynamic fuzzing, symbolic execution, and lightweight static analysis. Originally developed for the Testing Competition (Test-Comp), FuSeBMC is optimised to maximise code coverage and detect assertion violations and memory errors. It uses smart seed generation techniques and abstract interpretation to increase the likelihood of exposing bugs, especially in complex or constraint-heavy program paths.

In this thesis, FuSeBMC was extended to incorporate interval analysis using contractors, resulting in a modified tool referred to as **FuSeBMC_IA**. The integration of contractors allows the tool to discard infeasible paths early in the fuzzing process, thereby guiding input generation toward valid and high-value execution traces. This enhancement not only reduces computational overhead but also improves the quality and precision of generated test cases. The empirical impact of this integration is discussed in detail in Chapter 3.

[1] https://github.com/fusebmc/fusebmc
[2] https://github.com/esbmc/esbmc

B.2 ESBMC

ESBMC, the *Efficient SMT-Based Model Checker*, is a state-of-the-art verification framework for C and C++ programs. It employs bounded model checking techniques supported by SMT solvers to verify a range of software properties including memory safety, arithmetic correctness, and control-flow reachability. ESBMC operates by translating input code into an intermediate representation (GOTO-programs), performing symbolic execution, and encoding verification conditions as SMT queries.

In the context of this thesis, ESBMC was used as the host platform for two complementary investigations. First, as described in Chapter 4, contractors were integrated into the bounded model checking pipeline to refine variable domains prior to SMT solving. This optimisation was designed to reduce the complexity of SMT encodings by narrowing the state space without losing precision. Second, Chapter 5 details the application of contractors within ESBMC's abstract interpretation engine. This integration improved the expressiveness of interval-based abstract domains, enabling more accurate approximations of program behaviour during fixpoint computations.

B.3 Contextual Relevance to the Thesis

Both FuSeBMC and ESBMC were selected not only for their established performance in verification competitions, but also for their open architecture and flexibility. These attributes enabled the seamless integration of interval analysis and contractor methods developed in this research. Rather than constructing new tools from the ground up, the thesis demonstrates how existing, widely-used frameworks can be effectively enhanced with numerical techniques to improve software verification outcomes.

The originality of the work lies in the novel design and integration of contractor-based methods within these tools. The extended versions—**FuSeBMC_IA** and the contractor-enhanced **ESBMC**—represent new contributions to the field. They are capable of addressing core verification challenges such as state space explosion and resource overhead while preserving soundness and completeness. These extensions significantly improve the scalability and precision of verification across both dynamic and static analysis domains.

Appendix C

Teaching and Volunteering Experience

C.1 Graduate Teaching Assistant

During my PhD studies, I worked as a Graduate Teaching Assistant (GTA) at the University of Manchester for two years. This role gave me invaluable experience in teaching and assisting students in various courses. My responsibilities included:

- **Robotics Course**: I assisted students in practical and theoretical aspects of robotics, helping them understand key concepts and guiding them through hands-on projects. Additionally, I participated in demonstrations for UCAS visitors, where I showcased the robotics projects to prospective students and answered their queries about the course and university life.
- **Software Security Course**: I also supported the Software Security course, providing guidance during lab sessions and helping students resolve technical issues related to security vulnerabilities, secure coding practices, and verification techniques.

This teaching experience allowed me to develop my communication and mentoring skills while reinforcing my knowledge of robotics and software security.

C.2 Volunteering Experience

In addition to my teaching responsibilities, I actively participated in volunteering activities. I was involved with the **Saudi Students Club in Manchester**, where I contributed to organizing an event to celebrate **Saudi Founding Day**. My role in the event included:

Fig. C.1. Certificate of Appreciation Awarded outstanding voluntary contributions during the Saudi Founding Day event organised by the Saudi Students Club in Manchester, United Kingdom, on 27 February 2024.

- Assisting in the planning and organising of the event, ensuring that all logistical details were taken care of.
- Engaging with attendees, promoting cultural awareness, and highlighting the historical significance of Saudi Founding Day.

Volunteering for this event was a fulfilling experience, allowing me to contribute to my community and promote Saudi culture in Manchester.