MANCHESTER 1824
The University of Manchester

A formal semantics for GOTO in the CProver ecosystem

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

2025

Rafael Sá Menezes
School of Engineering
Department of Computer Science

Contents

Word count: 36257

List of figures

List of tables

List of publications

1. WU, Tong et al. ESBMC v7. 7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction: (Competition Contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer Nature Switzerland, 2025. p. 223-228.

2. Li, X. Gadelha, M. R. Braße, F. **Menezes, R. S.** Korovin, K. Cordeiro, L. C. ESBMC v7.6: Enhanced Model Checking of C++ Programs with Clang AST.

3. Wei, C., Wu, T., **Menezes, R. S.**, Shmarov, F., Aljaafari, F., Godboley, S., Alshmrany K. de Freitas, R., Cordeiro, L. C. (2025, May). ESBMC v7.7: Automating Branch Coverage Analysis Using CFG-Based Instrumentation and SMT Solving. In Fundamental Approaches to Software Engineering. Cham: Springer Nature Switzerland

4. **Menezes, R. S.**, Tihanyi, N., Jain, R., Levin, A., de Freitas, R. & Cordeiro, L. C. (2025, April). VO-GCSE: Verification Optimization through Global Common Subexpression Elimination. In International Conference on Foundations of Software Engineering. Cham: Springer Nature Norway.

5. Farias, B., **Menezes, R.**, de Lima Filho, E. B., Sun, Y., & Cordeiro, L. C. (2024, September). ESBMC-Python: A Bounded Model Checker for Python Programs. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 1836-1840).

6. **Menezes, R. S.**, Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., ... & Cordeiro, L. C. (2024, April). ESBMC v7. 4: Harnessing the Power of Intervals: (Competition Contribution). In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 376-380). Cham: Springer Nature Switzerland.

7. Aljaafari, F., Shmarov, F., Manino, E., **Menezes, R.**, & Cordeiro, L. C. (2023, April). EBF 4.2: Black-Box Cooperative Verification for Concurrent Programs: (Competition Contribution). In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 541-546). Cham: Springer Nature Switzerland.

8. Song, K., Gadelha, M. R., Braußе, F., **Menezes, R. S.**, & Cordeiro, L. C. (2023, December). ESBMC v7.3: Model Checking C++ programs using Clang AST. In Brazilian Symposium on Formal Methods (pp. 141-152). Cham: Springer Nature Switzerland.

9. E. Manino, **R. S. Menezes**, F. Shmarov, and L. C. Cordeiro. NeuroCodeBench: a Plain C Neural Network Benchmark for Software Verification. In Workshop on Automated Formal Reasoning for Trustworthy AI Systems, 2023

10. Silvestrim, R. G., Trigo, F. V., Rocha, W., Vieira, M. R., Junior, J. V., Mendes, O. D. C., ... & Cordeiro, L. C. (2023, November). Towards Integrity and Reliability in Embedded Systems: The Synergy of ESBMC and Arduino Integration. In 2023 XIII Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 1-6). IEEE.

11. Aljaafari, F. K., **Menezes, R.**, Manino, E., Shmarov, F., Mustafa, M. A., & Cordeiro, L. C. (2022). Combining BMC and fuzzing techniques for finding software vulnerabilities in concurrent programs. IEEE Access, 10, 121365-121384.

12. Braußе, F., Shmarov, F., **Menezes, R.**, Gadelha, M. R., Korovin, K., Reger, G., & Cordeiro, L. C. (2022, July). ESBMC-CHERI: towards verification of C programs for CHERI platforms with ESBMC. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 773-776).

13. **Menezes, R.**, Moura, D., Cavalcante, H., de Freitas, R., & Cordeiro, L. C. (2022, July). ESBMC-Jimple: verifying Kotlin programs via Jimple intermediate representation. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 777-780).

Terms and abbreviations

AI Abstract Interpretation

API Application Programming Interface

AST Abstract Syntax Tree

AWS Amazon Web Services

BDD Binary Decision Diagram

BMC Bounded Model Checking

CBMC C Bounded Model Checker

CEGAR Counter Example Guided Abstraction Refinement

CFG Control Flow Graph

CWE Common Weakness Enumeration

EBMC Efficient Bounded Model Checker

ESBMC Efficient SMT-Based Bounded Model Checker

GCSE Global Common Subexpression Elimination

HOL Higher Order Logic

IR Intermediate Representation

JBMC Java Bounded Model Checker

JPF Java Pathfinder

Irep Internal Representation

PC Program counter

RoS Rule of Signs

SAT Boolean Satisfiability

SMT Satisfiability Modulo Theories

SSA Static Single Assignmen

SV-COMP Software Verification Competition

UB Undefined Behavior

VCC Verification conditions

VSA Value-Set Analysis

Abstract

This thesis develops a formal and extensible verification framework for the CProver family of tools. Tools such as CBMC, 2LS, and ESBMC share a common intermediate representation, the GOTO IR, as well as core verification steps (symbolic execution, loop unwinding, decision procedure). However, their independent implementations have lacked formal semantics and reusable infrastructure. This fragmentation impedes rigorous reasoning about tool behavior, complicates the adoption of new verification methods, and restricts interoperability.

We introduce the GOTO language, a formal abstraction that puts CProver's existing GOTO IR on solid semantic foundations. Verification proceeds by symbolically executing GOTO programs to generate loop-free symbolic traces, which are encoded into logical formulae for automated decision procedures. We formalize the operational semantics of GOTO and trace evaluation, and provide a machine-checked Isabelle/HOL development to ensure soundness.

To enable modular verification, we design a transformation infrastructure analogous to compiler passes. This includes slicing, common-subexpression elimination, memory modeling via intrinsic symbols, and control-flow simplification. Multiple backends – bounded model checking, abstract interpretation, and others – can be applied over the same GOTO programs.

We implement the framework in ESBMC and assess its effectiveness on extensive benchmark suites (including SV-COMP categories). Our evaluation demonstrates significant performance and precision improvements arising from GOTO-based optimizations. We further enhance tool interoperability by translating GOTO binaries from alternative verifiers (such as Kani for Rust) into our framework, supporting hybrid workflows that leverage frontend strengths of CBMC, Kani, or other tools with ESBMC's backend.

By unifying theory and practice, this work establishes a principled foundation for the design, implementation, and verification of the CProver ecosystem.

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.

Acknowledgements

First and foremost, I thank my parents for their unwavering support, guidance during challenges, and steadfast belief in me. Their encouragement has been my greatest strength.

My deepest gratitude goes to Professor Lucas Cordeiro, who has supervised my research since the end of my undergraduate studies. From guiding my master's research in Brazil to nurturing my development as a researcher, his technical expertise and personal mentorship have been invaluable. I am especially grateful for our collaborative discussions—which extended beyond formal supervision—and for his friendship throughout this journey. I also thank Felipe Monteiro, whose early collaboration led to one of this thesis's key contributions.

I extend heartfelt thanks to my colleagues at the University of Manchester. Moving to a new country can be daunting, but Dr Fatimah Aljaafari, Dr Edoardo Manino, and Dr Fedor Shmarov welcomed me warmly. Their insightful feedback and challenging questions sharpened my ideas and honed my research skills. I am particularly indebted to Dr Edoardo Manino, whose contributions evolved into a co-supervisory role, enriching this work with his expertise. I also acknowledge Dr Giles Reger for his role as co-supervisor.

My sincere appreciation goes to Alexander Levin and Gadi Haber at Intel for their invaluable industrial insights. Their collaboration was crucial in maturing ESBMC and validating its integration into real-world workflows. Moreover, I am thankful to Intel and the University of Manchester for sponsoring my PhD.

Finally, I acknowledge Dr Daniel Kroening, whose pioneering work on CProver laid the foundation for ESBMC and this thesis. His vision continues to inspire the field and drives advances in software verification.

To all who supported, challenged, and inspired me, thank you. This thesis reflects your generosity as much as my own efforts.

Chapter 1

Introduction

The complexity of software systems has made it more difficult to ensure their correctness, reliability, and security across various platforms [1]. As software pervades critical infrastructure, the consequences of defects can escalate from minor inconveniences to severe economic and safety repercussions [2]. Inadequate verification practices may lead to the deployment of flawed or vulnerable systems [3], [4], with far-reaching consequences. A fatal example is the *Horizon IT scandal* [5], where software bugs in an accounting software system led the British Post Office to wrongly persecute hundreds of sub-postmasters; Prime Minister Rishi Sunak even called it one of the biggest miscarriages of justice in British history [6]. In a more recent case, a series of software failures at Southwest Airlines led to financial losses exceeding one billion dollars [7].

To increase confidence in the software, testing approaches are used [8], often leading to 50% cost of the development process [9]–[11]. Although software testing is a widely used technique for detecting defects, it provides only limited confidence in the absence of errors [1], [8]. In particular, testing explores only a subset of possible program executions. Consequently, many defects, especially those involving rare or complex interactions, can remain undetected. As noted in a recent White House report, testing alone is insufficient for assuring software quality, and formal methods are required to provide stronger guarantees [4].

In contrast to concrete test cases, assertions abstractly describe expected properties of program states [12], [13]. Formal methods aim to verify that such assertions hold for all possible executions, thereby increasing confidence in software correctness [14]. Software verifiers are tools that automate this reasoning [15]; they analyze program logic using formal techniques to determine whether assertions are valid under all execution paths. For programming languages that allows low-level memory manipulation, verifiers can detect critical vulnerabilities, including buffer overflows, use-after-free errors, integer overflows, and memory leaks [16]. Competitions such as SV-COMP [15] foster the development of these tools by providing benchmarks that target specific classes of weaknesses.

Within this domain, the CProver framework [17] provides a family of software verification tools for C and C++ programs. C Bounded Model Checker (CBMC) [18], the flagship tool in this ecosystem, has had significant academic and industrial impact [19], [20], culminating in a Test of Time Award at ETAPS 2025. CBMC has also served as the foundation for other tools, including ESBMC [21], 2LS [22], and CSeq [23]. CProver tools share a standard verification pipeline: (1) translation from a high-level language to the GOTO IR, (2) symbolic execution to generate an execution trace bounded by a given depth, and (3) encoding the trace into a logical formula checked by a decision procedure. However, despite these shared principles, each tool implements its components differently, which offers limited reuse between them.

This thesis proposes a framework for the CProver ecosystem, grounded in an abstract language named *GOTO*, which is based on the intermediate representation used by all CProver tools (the GOTO IR). By formalizing the semantics and verification procedures around this shared core, the framework aims to improve modularity, correctness, and collaboration across tools. The remainder of this chapter presents the problem statement and objectives, outlines the proposed approach, summarizes our contributions, and provides an overview of the thesis structure.

1.1 Problem description

Regardless of their use in both academia and industry [20], [24], CProver tools lack a formal framework to which tools can be built upon. This deficiency manifests in several ways: the absence of formal definitions for core components, the lack of correctness proofs for key algorithms, and limited sharing of infrastructure among tools. As a result, the development and maintenance of these tools remain challenging, with steep learning curves and limited modularity, hindering collaboration.

A key reason for this situation is historical. CBMC originated as a prototype to check behavioral consistency between Verilog and ANSI-C implementations [25]. As a prototype, it was initially designed for experimentation rather than extensibility or formal rigor.[1] Over time, tools evolved to support increasingly complex C constructs and incorporated sophisticated analysis techniques such as Value-Set Analysis (VSA) [26], Abstract Interpretation

[1]The Tech Report [25] focuses on the translation from C to equations. It does not detail the techniques employed by CBMC, e.g., symbolic execution, intermediate representation, and parser. In Section 2.3 we will describe how the prototype originated from a SyMP (a theorem prover) plugin.

(AI) [27], and Satisfability Modulo Theories (SMT) [28]. However, these additions were often ad hoc, making it harder to compare results across different tools or build upon previous work. In several cases, the implementation diverged significantly from the original published algorithms. For instance, ESBMC's k-Induction – an inductive proof strategy – algorithm was initially based on program transformation [29], but later versions partially replaced this with symbolic execution, which now lacks the formalism.

These informal practices pose significant challenges to research groups. Without formal descriptions of the translation to intermediate representations, symbolic execution, or SMT encoding, it becomes difficult to reason about the soundness of the verification pipeline because critical components (e.g., control-flow construction, variable renaming, and encoding strategies) are often implicit, or implementation-dependent. When proposing new features or optimizations, researchers and developers are routinely faced with the question: *"Is this transformation correct?"*. In many cases, no rigorous answer can be provided.

For example, consider the simple program in Figure 1.1a. The verification process involves unrolling the loop (Figure 1.1b) and encoding the result into a logical formula (Figure 1.1c). At each step, key correctness questions arise: *Does the unwinding preserve the program's semantics? Is the SMT formula semantically equivalent to the original C program?* These questions remain largely unaddressed due to the lack of a formalized and shared infrastructure.

```
1 int x = 1;
2 int y = *;
3 assume(y>=0&&y<3);
4 while(*) {
5   x = x*y;
6   assert(x != 8) }
```

(a) C program with error

```
1 x1 = 1;
2 y1 = *;
3 assume(y1 >= 0 &&
    y1 < 3);
4 x2 = x1 * y1;
5 assert(x2 != 8);
```

(b) K=1 unwinding

```
1 (= x1 1)
2 (>= y1 0)
3 (<= y1 3)
4 (= x2 (* x1 y1))
5 (= x2 8)
```

(c) Decision procedure

Fig. 1.1. Example of ESBMC flow for C program with $k = 1$ (adapted from [30]). Note that assigning the value "*" means a non-deterministic value.

The lack of a formal framework also complicates the extension and reuse of tools. Although tools such as CBMC and ESBMC share foundational ideas, their implementations have diverged considerably. For example, CBMC integrates support for program contracts, while ESBMC focuses on inductive strategies. However, due to incompatible code bases and the absence of a shared intermediate language, combining these features is impractical. From a development standpoint, the lack of a modular and well-documented infrastructure leads to prolonged development times. In our experience, new contributors require at least six

months before they can engage meaningfully with the codebase.

By contrast, other verification and compiler infrastructures provide successful examples of shared intermediate representations. The Boogie framework [31], which describes itself as a modular and reusable verifier, enables multiple verification strategies to coexist and evolve through a well-defined intermediate language. Similarly, the LLVM project [32] integrates front-ends, optimizers, and analyzers around its LLVM IR, enabling powerful tooling reuse and modularity. Regarding CProver, this could allow different approaches for each step of the flow: *language front-ends*, *passes*, *symbolic execution algorithms*, *decision procedures*, *counterexamples*, etc. These examples illustrate the value of designing frameworks around a principled and shared representation. Unfortunately, CProver tools evolved in such a manner that it would be impractical to introduce such a formalized representation without significant development effort.

To our knowledge, no formal framework currently exists for CProver tools, one that defines their internal intermediate language, transformation rules, or verification strategies. This thesis addresses this gap by investigating the following question: *Is it feasible to develop a language that serves as the basis for a verification framework applicable to all CProver tools?*

1.2 Objectives

The main objective of this thesis is to design, formalize, and evaluate a unifying framework for CProver tools, based on a shared intermediate representation. This framework aims to serve as a formal foundation for implementing and validating software verification algorithms across tools such as CBMC [19] and ESBMC [21].

To achieve this, the thesis pursues the following specific objectives:

1. **Formalize an abstract intermediate language**, capturing the essential structure and semantics of CProver's GOTO IR (cf. Chapter 3).
2. **Define a methodology for encoding and verifying algorithms** over the GOTO language, including symbolic execution, and decision procedure (cf. Chapter 3).
3. **Introduce a framework for GOTO-based transformations**, enabling the definition, optimization, and analysis of verification strategies within a modular and extensible architecture (cf. Chapter 4).

4. **Implement selected GOTO-based algorithms** and evaluate them through integration with existing CProver tools, including real-world applications, tool interoperability, and reusability of verification components across tools (cf. Chapter 5).

1.3 Outline of the Solution

The proposed solution builds upon the similarities between CBMC and ESBMC to define a unifying framework based on an abstract GOTO language. Our solution proposes a formal framework that abstracts and generalizes these core components, enabling both theoretical reasoning and practical reuse. This framework is organized into two major pillars:

- **GOTO Generation:** The first stage of the framework defines a common language, GOTO (named after the existing GOTO IR it formalizes), which acts as the intermediate representation for all front-ends (e.g., C, C++, Rust, Java, Solidity, Python). Unlike existing implementations, this GOTO language is formally specified, allowing reasoning about correctness-preserving transformations. The generation process supports transformations such as loop unrolling, overflow instrumentation, and control-flow simplification. These transformations can be precisely defined and verified within the framework. Moreover, different verification strategies—e.g., k-Induction, contract checking, or abstract interpretation—can be expressed in terms of GOTO programs, enabling modularity and reusability.
- **GOTO Trace Generation:** The second stage concerns the generation of symbolic traces over the GOTO program. The framework provides a formal description of how bounded symbolic execution constructs execution traces, ensuring that the resulting formulas preserve the semantics of the original GOTO program. Once the trace is generated, it is translated into logical formulas suitable for decision procedures (SAT, SMT, or abstract domains). The framework supports alternative backends and optimization passes at this stage. This modular design enables the use of different tools, allowing for experimentation, swapping, or combining solvers and trace generation techniques while maintaining a shared semantic foundation.

Figure 1.2 illustrates the architecture of the proposed framework. The pipeline begins with a supported input language (e.g., C, C++, or Rust), which is compiled to a GOTO program (blocks 1–3). This program is then symbolically executed to produce a bounded trace (block

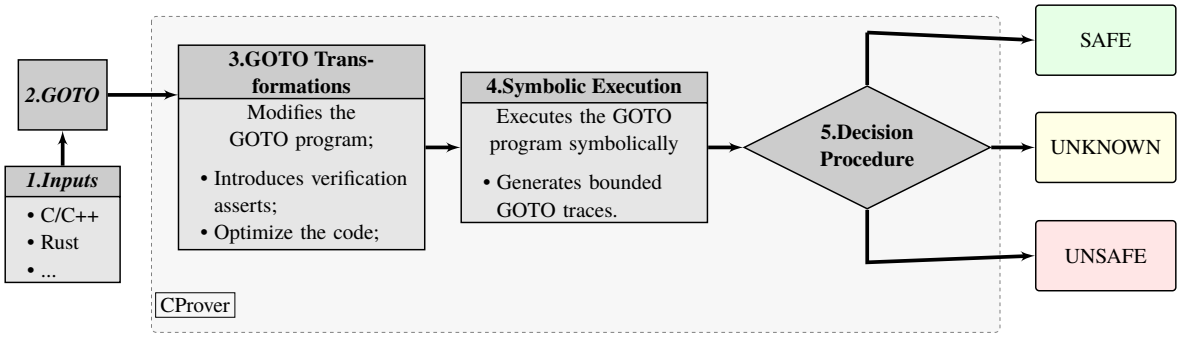


Fig. 1.2. The proposed architecture.

4), which is finally encoded into a satisfiability problem (block 5). The modular design of each block ensures that they can be reused, reasoned about, and extended independently.

By isolating and formalizing the GOTO generation and trace construction processes, the proposed framework addresses the objectives identified in Section 1.2:

- It enables correctness proofs for key components (e.g., transformation passes, symbolic execution steps).
- It encourages reuse and interoperability between tools (e.g., shared GOTO front-ends or verification strategies).
- It facilitates the combination of verification techniques, improving the practical power of CProver tools.

Ultimately, this framework transforms the ad hoc evolution of the CProver tools into a principled architecture with formal semantics, reusable components, and a shared language for verification research.

To assess the validity and applicability of the proposed framework, this thesis adopts a two-pronged validation strategy. First, we implement core algorithms—such as symbolic execution and program transformation—within the framework and evaluate them experimentally in Chapter 5, using both synthetic and real-world benchmarks. Second, we provide a formalization of selected components of the framework in Isabelle/HOL (Appendix A), focusing on the semantics of GOTO traces and the correctness of bounded symbolic execution. This dual validation approach ensures that the proposed framework is not only practically effective but also grounded in rigorous formal reasoning.

1.4 Contributions

This thesis addresses the challenge of fragmentation in the CProver ecosystem by developing a formal, extensible framework based on a shared intermediate representation. The core contributions of this work are aligned with the objectives defined in Section 1.2 and respond directly to the limitations outlined in the problem description (Section 1.1).

The first contribution is the design and formalization of an abstract intermediate language, GOTO (named after the GOTO IR it formalizes), that encapsulates the control flow, memory model, assertion structure, and data types (represented uniformly through naturals) used in CProver tools. Concurrency primitives are discussed as language extensions (Section 3.5.2), though standardizing a single concurrency model remains future work, as CBMC and ESBMC adopt different approaches (explicit thread instructions vs. function-call-based threading). This language serves as the foundation for the proposed framework, enabling a principled description of program behavior, transformations, and verification strategies. By capturing the essential semantics in a formally defined language, the framework enables precise reasoning about verification procedures and forms the basis for encoding symbolic execution and decision procedures. The formal semantics of this language are specified in Isabelle/HOL: Isabelle automatically verifies the totality and termination of the trace evaluation functions, and the load-after-store property of the memory model is proved as a theorem.

The second contribution is the development of a formal methodology for symbolic execution and trace construction, centered around bounded verification. This methodology defines how program executions are encoded into logical formulas, and includes a machine-checked formalization of symbolic execution semantics and properties.

The third contribution focuses on enabling verification-oriented program transformations through a modular transformation infrastructure. Inspired by compiler design (e.g., LLVM passes), the framework supports the definition and combination of GOTO-level transformations such as slicing, overflow instrumentation, and path pruning. This infrastructure not only enables reuse across tools but also facilitates experimentation with alternative verification strategies, including contract checking, abstract interpretation, and test generation. These transformations are implemented and integrated into ESBMC, enabling the practical deployment and benchmarking of these modules.

The fourth contribution is an empirical evaluation of the framework through real-world

benchmarks and tool interoperability studies. Key algorithms and transformations were implemented in ESBMC and evaluated using SV-COMP-style benchmarks and industrial verification cases. These experiments demonstrate the scalability, effectiveness, and reusability of the proposed components. For example, we show that CBMC and ESBMC can share front-end processing and symbolic execution infrastructure by emitting and consuming GOTO representations, enabling workflows that leverage the strengths of both tools. This resulted in publications [21], [33].

Finally, this thesis contributes to the broader goal of promoting modularity and collaboration within the CProver ecosystem. By establishing a shared formal foundation, the proposed framework reduces duplication of effort and lowers the barrier for extending tools or integrating new analysis techniques. The ability to interchange components—such as symbolic executors or SMT encoders—without modifying core logic promotes both maintainability and extensibility. This work also enables new research directions by providing a formal basis upon which future optimizations, strategies, and correctness proofs can be built.

Together, these contributions represent a step toward transforming the CProver family of tools from a collection of loosely connected implementations into a coherent, formally grounded verification ecosystem.

1.5 Thesis Structure

This chapter provides an overview of the background, motivation, and issue addressed by this thesis, along with its goals, proposed solution, and research contributions. Figure 1.3 shows the thesis structure. The subsequent chapters of this thesis are structured as follows:

Chapter 2, *Theoretical Foundations for CProver Verification*, provides an introduction to the foundational concepts essential for this research. It commences with a brief history of CProver tools. It then proceeds to other topics relevant to this thesis, including Software Verification and Program Semantics.

Chapter 3, *GOTO Language*, constitutes the primary basis for this study. Initially, the chapter frames an architecture with the GOTO abstract language. Afterward, we define the GOTO language and the Trace Language. Concluding the chapter, we examine the differences between the framework and real-world tools.

Chapter 4, *GOTO Transformations*, introduces a series of transformation algorithms within

the framework, designed to facilitate verification of a GOTO program. This approach aims to enhance the proof strategies for their applications.

Chapter 5, *Impact of GOTO transformations in ESBMC*, adopts a more practical approach, illustrating the efficacy of the transformation within the CProver tool employed by our research group, ESBMC. We evaluate each transformation utilizing a methodology similar to that of SV-COMP. Additionally, we examine the impact of the transformations on the internal components of the tool.

Chapter 6, *Conclusions* concludes this work. It commences by providing a summary of the presented works and associated findings. Furthermore, it concludes by suggesting future research directions.

Appendix A, *GOTO formalization in Isabelle*, includes a partial implementation of the framework as proposed within the Isabelle theorem prover [34].

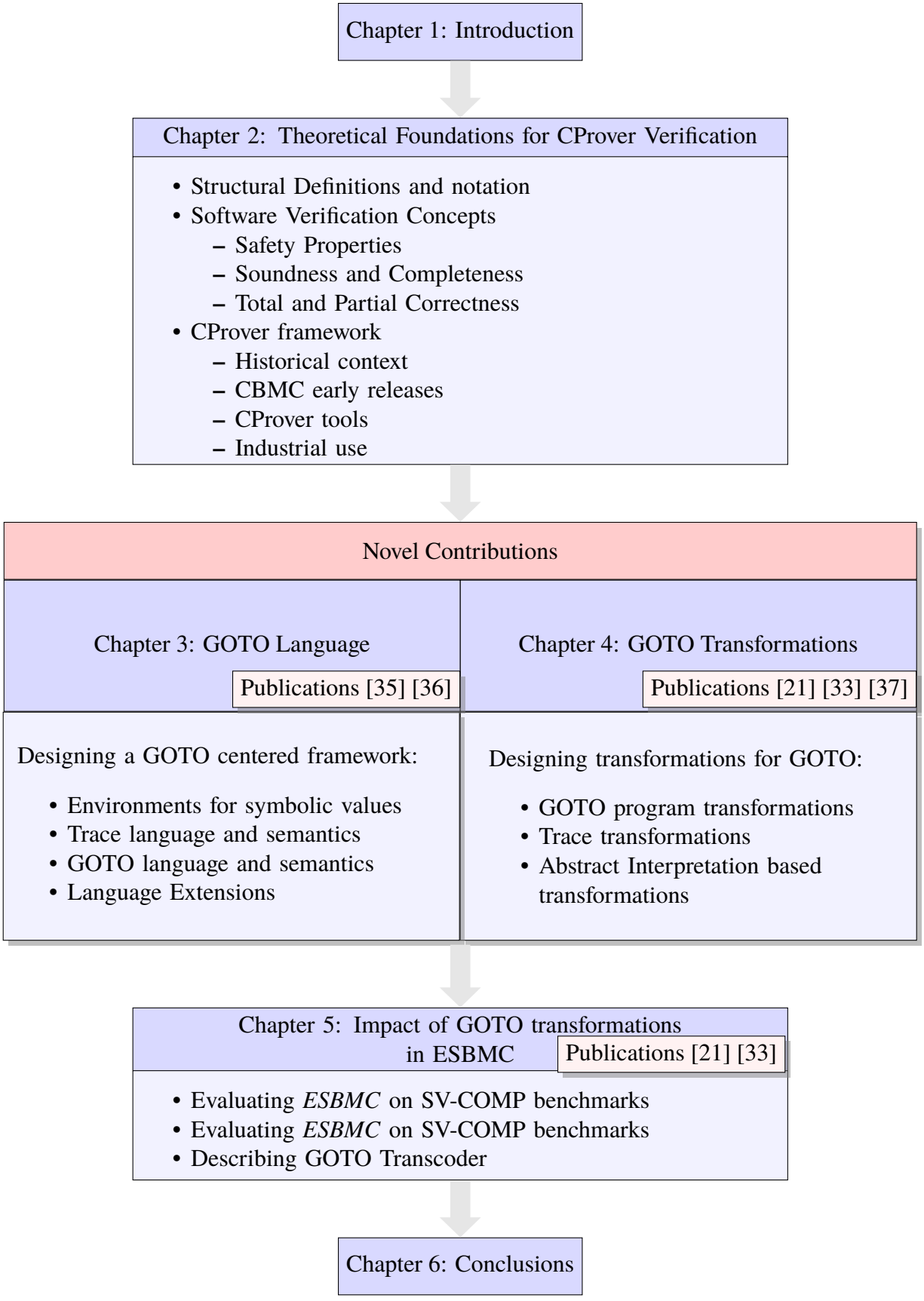


Fig. 1.3. Thesis structure.

Chapter 2

Theoretical Foundations for CProver Verification

To support the formal framework developed in this thesis, this chapter introduces several foundational concepts that are revisited in later chapters. Other topics—such as the memory model and abstract interpretation—are deferred to the points where they are used (Chapters 3 and 4, respectively).

We begin with structural definitions and functional notation—including inductive data types, curried functions, and higher-order abstractions—which underpin the syntax of the GOTO and trace languages introduced in Chapter 3 and are used extensively in the mechanized semantics in Isabelle/HOL [34].

We then review core verification notions—safety properties, soundness, and partial vs. total correctness—that provide the background needed to reason about program behavior and verification outcomes. These notions inform the formal semantics in Chapter 3, the transformation design in Chapter 4, and the experimental evaluation in Chapter 5.

Finally, we provide a source-based historical overview of the CProver ecosystem, from early CBMC to the current landscape, to situate the contributions of this thesis within existing toolchains (Section 2.3).

2.1 Structural Definitions and Notation

Throughout this work, we make extensive use of functional and structural definitions inspired by the declarative programming paradigm [38]. This section provides a brief introduction to these concepts to prepare the reader for the notational and semantic style adopted in the thesis.

2.1.1 Inductive Types and Pattern Matching

In the declarative style, types can be defined recursively. A *well-formed* inductive type is one that always terminates in a base constructor.[1] In Example 2.1, consider the definition of natural numbers using *Peano numbers* [39]:

Example 2.1. Definition of natural numbers through *Peano numbers*.

$$\begin{aligned} \mathbb{N} ::= & \mathit{Zero} \\ & \mid \mathit{Suc}(\mathbb{N}) \end{aligned}$$

Throughout this thesis, we use rounded boxes to highlight constructors, operators, and key terms when they appear inline.

This representation models natural numbers as chains of *Suc* applications starting from a base value *Zero*. For instance, the number two is expressed as *Suc(Suc(Zero))*. Every natural number constructed in this manner eventually reaches the base case, making the type inductively well-formed. Moreover, we can construct predicates such as $\forall n \in \mathbb{N},\ n < \mathit{Suc}(n)$.

Such inductive structures support recursive function definitions. In particular, *pattern matching* enables functions to branch on the structure of their inputs. Consider the function `add`, which sums two Peano numbers:

Example 2.2. Addition between two naturals.

$$\text{add}(x, y) = \begin{cases} \texttt{y}, & \text{if } x = \texttt{Zero} \\ \texttt{Suc}(\text{add}(n, y)), & \text{if } x = \texttt{Suc}(n) \end{cases}$$

This function is structurally recursive: each call is made on a structurally smaller argument. The base case yields a result directly, and the recursive step applies `add` to a smaller subterm. While this case-based format is readable, it becomes verbose in text. Functional languages typically express the same logic more succinctly using pattern-matching syntax:

Example 2.3. Addition between two naturals (pattern matching).

[1] We adopt the definition used in Isabelle/HOL [34].

$$\text{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\text{add}(\texttt{Zero}, y) \triangleq y$$

$$\text{add}(\texttt{Suc}(n), y) \triangleq \texttt{Suc}(\text{add}(n, y))$$

Although Peano's original axioms define natural numbers starting from *one*, in the remainder of this thesis, we adopt $\boxed{\textit{Zero}}$ as the base constructor. This choice aligns with the conventions used in programming language semantics and proof assistants such as Isabelle [34]. The principles of well-founded recursion and structural reasoning discussed here apply equally in either formulation [34].

These inductive foundations underpin many definitions throughout this work. In particular, they support the formal semantics and recursive procedures defined in Chapter 3, as well as the termination arguments and structural induction proofs formalized in Appendix A.

2.1.2 Currying and Functional Style

In the functional programming paradigm, programs are often constructed as compositions of pure functions [40]. A function is *pure* if it always produces the same output given the same input and has no observable side effects [40]. This style has significant implications for the structure of data and computation.

For example, consider the operation of modifying an array:

- In an *imperative* style, one would directly mutate the array, possibly using a pointer.
- In a *functional* style, one would construct a new array based on the original, with the desired modification applied.

Purity enables key features such as function composition (e.g., $\boxed{f \circ g}$) and partial application. The process of transforming a multi-argument function into a sequence of unary functions is known as *currying*, a foundational idea in languages such as Haskell [41].

For instance, instead of modeling addition as a function from a pair of naturals to a natural, $\boxed{\texttt{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}}$, we can define it in curried form:

Example 2.4. Addition between two naturals (curried, with pattern matching).

$$\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

$$\text{add } \texttt{Zero}\ y \triangleq y$$

$$\text{add } \texttt{Suc}(n)\ y \triangleq \texttt{Suc}(\text{add } n\ y)$$

This means that `add` takes a natural number and returns a new function that expects another natural number and produces their sum. This small change introduces a powerful compositional mechanism. For example, we can define a partially applied function $\boxed{add2}$ as $\boxed{add\ 2}$, and then compose it with another function such as $\boxed{add1}$ (addition by $\boxed{Suc(Zero)}$), yielding an expression like $\boxed{add2 \circ add1}$. Example 2.5 illustrates partial application and composition.

Example 2.5. Partial application and function composition.

$$\text{add2} : \mathbb{N} \to \mathbb{N}$$

$$\text{add2}\ y \triangleq add\ (Suc(Suc(Zero)))\ y$$

$$= Suc(Suc(y))$$

$$(\text{add2} \circ \text{add1})\ y \triangleq Suc(Suc(Suc(y)))$$

Lastly, we will rely on λ-calculus notation as a concise way to define computable functions [42]. We will not explore its equivalence to Turing machines [43], but will use it as a convenient shorthand for inline function definitions. We also assume that standard constructs such as *if-then-else* are already defined. A formula has the form $\boxed{\lambda\{vars\}.\{definition\}}$, where $\boxed{\{\text{vars}\}}$ is a sequence of bound variables appearing in $\boxed{\{\text{definition}\}}$.

In Example 2.6, we show the representation of Peano numbers in λ-calculus form, where $\boxed{\text{a} \mapsto Suc}$ and $\boxed{\text{b} \mapsto Zero}$. Here it becomes clearer that the encoding corresponds to repeated application of a function.

Example 2.6. Naturals as λ-calculus terms (replicated from [42], p. 3).

$$1 \to \lambda ab.\ a(b)$$

$$2 \to \lambda ab.\ a(a(b))$$

$$3 \to \lambda ab.\ a(a(a(b)))$$

We can apply arguments to these formulas just as we do when invoking functions. Example 2.7 shows how to define the add function in this framework. The intuition is that it takes as arguments: (1) the first operand, (2) the second operand, (3) the *successor* function, and (4) the *unit* function. This definition also illustrates the use of *higher-order functions*, i.e., functions that take other functions as arguments or return them as results [44].

Example 2.7. Application of λ-calculus for addition.

$$
\begin{aligned}
add &\triangleq \lambda mnab.\ m\, a\, (n\, a\, b) \\
add\ 1\ 1 &\triangleq (\lambda mnab.\ m\, a\, (n\, a\, b))\ (\lambda ab.\ a(b))\ (\lambda ab.\ a(b)) \\
&= \lambda ab.\ (\lambda ab.\ a(b))(a)\ ((\lambda ab.\ a(b))\ a\ b) \\
&= \lambda ab.\ (\lambda b.\ a(b))(a(b)) \\
&= \lambda ab.\ a(a(b)) \\
&= 2
\end{aligned}
$$

Currying and λ-calculus abstraction are fundamental in functional programming [44]. They allow for generic definitions, transformations, and analysis procedures over syntax trees or program traces. For example, a higher-order function may take a predicate and return a verifier that applies it to each step of a symbolic trace. In proof assistants such as Isabelle/HOL, higher-order functions enable the definition of reusable proof combinators, semantic transformers, and modular evaluation strategies [34]. This expressiveness is essential for the mechanized formalization presented in Appendix A, where many core components—such as trace evaluation and symbolic execution—are parameterized by user-supplied functions or predicates [34].

These functional definitions are not merely illustrative: they provide the building blocks for specifying programming languages and reasoning about program traces, which we use in the semantics of the GOTO language in Chapter 3.

2.2 Software Verification Concepts

The previous chapter introduced CProver tools in practice as software verifiers. Here, we expand on *what* a software verifier is, what it seeks to achieve, what defines the correctness and limitations of its verdicts. We begin with the notion of safety properties, then discuss common categories of software defects, before reviewing key concepts such as soundness,

completeness, and (partial vs. total) correctness.

2.2.1 Safety Properties and CWE

To verify a program, one must first specify its *intended* behavior. In the context of this thesis, correctness is expressed primarily as a collection of *safety properties*, which state that "nothing bad happens" during execution [45][2]. These can be formalized as assertions over the program state [46].

In practice, programming languages differ in the hazards they expose. Low-level languages such as C permit direct memory manipulation, which leads to many of the vulnerabilities listed in the Common Weakness Enumeration (CWE). Indeed, all of the CWE Top 25 most dangerous vulnerabilities apply to C programs [47]. Compliance with stricter coding standards, such as the SEI CERT C guidelines [48], requires ruling out additional classes of errors.

From its earliest versions, CBMC included dedicated checks for such vulnerabilities [49]. Since version 1.2, the tool has been able to detect:

- **Out-of-bounds write (CWE 787) and read (CWE 125)**;
- **Use after free (CWE 416)**;
- **Null pointer dereference (CWE 476)**;
- **Integer overflow or wraparound (CWE 190)**.

Within CProver, these properties are encoded as assertions and are checked exhaustively through the verification pipeline [49].

2.2.2 Soundness and Completeness

In this work, we characterise soundness and completeness of a verifier V applied to a program P with property φ [14], [50], [51]. A verifier may return *True* (property holds), *False* (property violated, with a counterexample), or *Unknown* (no verdict, e.g., due to timeout or resource exhaustion). Two goals are distinguished: *verification* ($P \models \varphi$) and *falsification* ($P \not\models \varphi$), each with its own soundness and completeness criterion:

[2] Its counterpart—*liveness*—requires that something good eventually happens [45].

- **Complete w.r.t. verification:** If $P \models \varphi$, then V does not return *False*.
- **Sound w.r.t. verification:** If V returns *True*, then $P \models \varphi$.
- **Sound w.r.t. falsification:** If V returns *False*, then $P \not\models \varphi$ (the counterexample is genuine).
- **Complete w.r.t. falsification:** If $P \not\models \varphi$, then V does not return *True*.

A verifier is *sound* if every property it proves is guaranteed to hold in the actual program [51]. Thus, if the tool reports that P is "safe," then no real execution of P violates the property under consideration. Soundness is preserved only if all stages of the verification pipeline – translation to an intermediate representation, symbolic execution, abstraction, and solver encoding – faithfully preserve program semantics [51]. An unsound transformation at any stage may lead to a faulty program being incorrectly reported as safe [51].

Returning *Unknown* is compatible with soundness in both senses: when V abstains from a verdict, no incorrect claim is made. It does, however, reduce completeness.

Soundness does not imply completeness. A sound tool may still produce false alarms (spurious counterexamples) or inconclusive results, but it will never wrongly classify an unsafe program as safe [51]. CProver tools are sound w.r.t. both verification and falsification, while accepting incompleteness as a trade-off for scalability [46]. For instance, BMC is sound w.r.t. both goals but complete w.r.t. neither: bugs beyond the exploration bound may be missed, and safe programs may not receive a verdict within the bound.

2.2.3 Partial and Total Correctness

Partial and total correctness are properties of a program with respect to a specification, formulated in Hoare logic [52]:

- **Partial correctness:** If the program terminates when started in a state satisfying the precondition, the postcondition holds. Termination itself is not guaranteed [52].
- **Total correctness:** The program terminates and the postcondition holds [52].

This thesis is primarily concerned with *safety properties*: conditions that must hold at every point during execution (e.g., no assertion violation, no null dereference). CProver tools support both notions [46]. To verify total correctness within a bounded exploration, tools insert

unwinding assertions after loops and recursive calls, requiring all iterations to be fully explored [19]. Without such assertions, only partial correctness is established for the explored executions.

Other techniques, such as k-induction or predicate abstraction, aim to reason beyond bounded exploration [29] [22]. These approaches remain sound, but they are not complete:

- Properties may hold vacuously. For example, an assertion placed after a non-terminating loop (e.g., `assert(0)`) is unreachable and therefore trivially satisfied [53]. Proof attempts may fail to establish safety, even if the program is actually safe.

These notions – safety, soundness, and partial vs total correctness – form the theoretical baseline for the verification outcomes considered in this thesis. They also clarify why bounded model checkers such as CBMC and ESBMC focus on sound but incomplete analyses, and motivate the extensions introduced in Chapters 3 and 4 for reasoning about assertions and symbolic traces in a bounded yet precise framework.

2.3 The CProver Ecosystem

The work presented in this thesis builds directly on the technologies and methodologies developed within the *CProver* ecosystem [17]. This section introduces the historical and technical foundations of the ecosystem, with particular emphasis on the evolution of the verification engines *CBMC* [19] and *ESBMC* [21]. We begin with a brief overview of model checking as applied to software and then situate these tools within that context.

2.3.1 Model Checking and Symbolic Verification

Model checking is a verification technique that systematically explores the state space of a system to ensure that the specified properties hold [50]. In classical hardware-oriented settings, the system is modeled as a finite-state transition system, and properties are expressed in temporal logics [54]. The verification engine enumerates all reachable states to check for violations.

A canonical example involves verifying safety in concurrent programs such as semaphore protocols [50], where the number of reachable states grows exponentially with the num-

ber of processes and variables. For simple programs, explicit-state model checking suffices. However, as software complexity increases, the state space quickly becomes intractable.

To address this challenge, symbolic model checking was introduced [55], replacing explicit enumeration with symbolic representations of state sets, typically encoded as Binary Decision Diagrams (BDDs). While this improved scalability, BDD-based methods still struggled with the complexity of modern software. This limitation motivated the development of bounded model checking (BMC) [56], which explores the system only up to a finite depth.

BMC reduces verification to a satisfiability (SAT) query [3]: if an error exists within a bounded number of steps, a counterexample is returned; otherwise, the system is safe within that bound. Biere and collaborators further showed that, under certain conditions (e.g., small diameter or clique width of the transition system), safety guarantees could be established even with bounded exploration [56]. Around the same time, researchers began applying model checking directly to software. Rather than encoding a hardware-like transition system, software programs—often at the source level—were translated into verification problems. Several tools emerged, including Java PathFinder (JPF) [59] for Java. At the time, JPF started as a front-end translating Java into Promela for the SPIN model checker [60].

For ANSI-C, Daniel Kroening decided not to translate the language into an existing model checker. His intuition was that theorem provers could be used to reason about the sub-problems that arise in software analysis [61]. He observed that languages such as ANSI-C and Java rely heavily on low-level (bitwise) operations, which theorem provers at the time struggled with due to their reliance on higher-level abstractions [61]. Conversely, model checkers were naturally suited to such operations [62]. To bridge this gap, Kroening developed a plugin for the Symbolic Model Prover (SyMP) [63]. The plugin symbolically explored programs, generating execution paths that a model checker then checked. In an e-mail exchange [64], Kroening recalled:

> *"The plugin had an 'unwind' rule, which did one step of unwinding. People ultimately only used that, which then became CBMC."*

The *CBMC* (C Bounded Model Checker) project emerged from this plugin, initially as part of research on equivalence checking between hardware models and C software representations [18]. Its core design translated C programs into an internal representation, executed them symbolically up to a fixed bound, and generated a corresponding SAT formula. The

[3] Later BMC was also applied via SMT [28], accelerators [57] and even newer approaches MoXI[58].

resulting Boolean formula encoded safety properties, and SAT solving determined whether counterexamples existed. To support termination arguments, CBMC also inserted assertions after loops and recursive calls, enabling verification of total correctness within the bounded setting. Its procedure consisted of two checks: (1) ensuring that all assertions are valid, and (2) ensuring that all loops terminate. If both held, the program was deemed safe.

Reconstructing the broader history of the CProver tools is non-trivial: early releases were poorly documented, and several resources are no longer accessible. In the next subsection, we trace the early stages of CBMC, how it gave rise to the broader CProver family of tools, and its eventual industrial adoption.

2.3.2 Early CBMC

We begin with the initial CBMC releases, focusing on the features that will be revisited in Chapter 3. As mentioned above, it is difficult to obtain all early releases. To recover them, we collected binaries and source code from the Internet Archive:

1. We consulted the historical CBMC downloads page.[4]
2. The directory server (an Apache/2.4.10 instance) originally listed all files publicly, from which we obtained filenames corresponding to CBMC releases.
3. The Apache server remains online, so we removed the Internet Archive prefix from the URLs to match the current CProver website.

The earliest source code we recovered was release 2.4.1.[5] For earlier versions, we relied on reverse engineering using Ghidra [65]. Since the binaries were optimized and statically built, we could not extract high-level structures such as class names. Instead, we used the source code from v2.4.1 as a reference. Our analysis focused primarily on string data, since *CBMC*'s internal representation makes heavy use of them. All binaries and sources have been archived on Zenodo for reproducibility [66]. Figure 2.1 illustrates the release timeline of CBMC versions from 1.0 up to 2.9, which we selected as the point where all features assumed by our abstract CProver framework were in place.

[4] `https://web.archive.org/web/20160509132334/http://www.cprover.org/cbmc/download`

[5] We also contacted Kroening by e-mail [64], but at the time of writing earlier source code was not available.

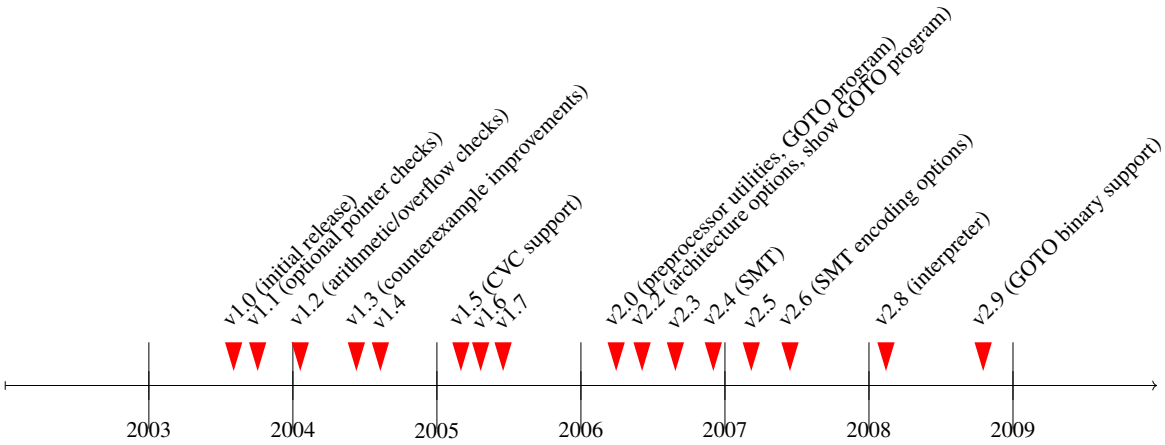


Fig. 2.1. Timeline of early CBMC releases and their main features. Dates are obtained from binary timestamps. Some intermediate releases (e.g., v2.7) were not recovered.

The oldest available binary corresponds to version 1.0, created by the Unix user `kroening` on August 3rd, 2003. This version lacked explicit references to symbolic execution or the GOTO language as an intermediate representation. Instead, it appeared to operate directly on ANSI-C statements and parsing. Notably, it did not support preprocessing, requiring users to ensure their code was ANSI-C compliant [25]. This changed in version 2.0, which introduced preprocessing capabilities. Furthermore, newer versions added a bundled `libc` implementation, distributed separately.

Nevertheless, version 1.0 already exhibited several concepts central to later releases:

- *Symbolic execution:* Classes implementing symbolic execution were already present, capable of generating SSA traces.
- *Ireps (internal representations):* Ireps were used as containers for program structures, types, and expressions. We adopt the term *internal representation* as it appears to originate from the SyMP plugin [61]. Examples of this representation are shown in Section 5.8.
- *Arbitrary-precision integers:* The `BigInt` class was already available and used to store and manipulate values from source programs efficiently.
- *Non-determinism:* CBMC encoded bounded models over *all possible* values that each SSA variable could assume.

Our formalization (cf. Chapter 3) minimizes the number of *ireps* while constructing symbolic executions. Later versions introduced the explicit notion of the GOTO program (v2.0) and the `goto` binary (v2.9). In the same period, experimental SMT solver support appeared

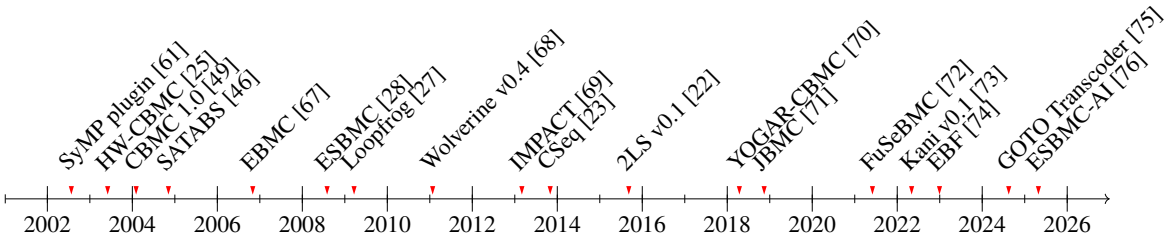


Fig. 2.2. Timeline of CProver-derived tools. Dates are based on publication, binary timestamps, *git* commits, or conference presentations. For visual clarity: CBMC 1.0 is placed in February 2004 instead of July 2003; Loopfrog is placed in March 2009 instead of August 2008; IMPACT is placed in March 2013 instead of October 2013; JBMC is placed in November 2018 rather than its first release in July.

(v2.4), setting the stage for later forks. Starting with version 5.6, development moved to GitHub, significantly improving accessibility. The GOTO IR introduced in v2.0 is the one we will reference throughout this thesis, and in Chapter 5 we show how the GOTO binary can serve as a compatibility layer between CProver tools.

2.3.3 CProver Tools

The methodology pioneered in CBMC served as the foundation for a wide range of related tools. Some extended CBMC directly, others reused its infrastructure (front-ends, internal representations, or symbolic execution engines), while others introduced specialized techniques for concurrency, abstraction, or new programming languages. Figure 2.2 summarizes the timeline of major forks and derivatives.

Below, we briefly describe each tool and its contribution to the ecosystem:

- **HW-CBMC:** The original tool for checking equivalence, accepting inputs both in C and in Verilog [25].
- **SATABS:** A SAT-based predicate-abstraction model checker for ANSI-C that implements a complete CEGAR loop [46]. Abstraction, simulation, and refinement are all performed using SAT queries. Predicate abstraction is accelerated by clustering, spurious counterexamples are filtered via per-transition SAT checks, and refinement uses weakest preconditions.
- **EBMC:** The *Efficient Bounded Model Checker* [67] extended SAT-based BMC with over-approximations. By inserting cut-points into feedback loops, EBMC reduced completeness thresholds and abstracted pipelines, enabling proofs of safety properties

beyond bounded depths. Refinement was driven by UNSAT cores, yielding a verification procedure for many hardware designs.

- **ESBMC:** The *Efficient SMT-Based Bounded Model Checker* forked from CBMC to exploit SMT solvers directly via solver APIs [28]. ESBMC integrated Clang/LLVM as its front end, expanded the GOTO language with SMT-aware constructs, and added k-induction for unbounded proofs [29]. It also introduced explicit concurrency modeling and richer internal representations, aligned with SMT theories, facilitating the translation layer.

- **Loopfrog:** an algorithm to compute an over-approximation of the set of reachable states of a program by replacing loops [27]. It was integrated within CBMC and implemented as an *Abstract Interpretation*.

- **Wolverine:** A verifier based on *lazy abstraction with interpolants* [68]. Wolverine incrementally unwinds control-flow graphs into reachability trees, using Craig interpolants to generalize safe states. Fixed points of interpolants yield inductive invariants. The tool bundled an interpolating solver for equality logic with uninterpreted functions, with limited bit-vector support, and exposed APIs for external SMT solvers.

- **IMPACT:** a CProver-based model checker that implements lazy abstraction with interpolants (the "Impact" algorithm) [69]. It leverages the CProver parsing and GOTO infrastructure but replaces CBMC's monolithic BMC with a CEGAR loop driven by SMT-solver–derived interpolants.

- **CSeq:** A *sequentializer* for concurrent C programs [23]. CSeq translated `pthread`-based concurrency into nondeterministic sequential code, reusing CProver parsing and symbolic execution. Assertions, synchronization primitives, and thread scheduling were encoded in GOTO form, enabling CBMC and ESBMC to verify concurrent programs without modification.

- **2LS:** A program analysis and verification tool built explicitly on the CProver infrastructure [22]. 2LS combines bounded model checking, k-induction, and template-based invariant synthesis. Programs are converted into SSA with loop heads cut, and invariants are synthesized incrementally, discharging conditions to SAT with clause reuse across refinements.

- **YOGAR-CBMC:** A CBMC fork optimized for multi-threaded programs [70]. YOGAR tackled the cubic blow-up of thread-interleaving encodings by abstracting away

scheduling constraints. It refined only when counterexamples proved spurious, using *Event Order Graphs* (EOGs) to capture read–write relations and generate compact refinement constraints, significantly improving scalability.

- **JBMC:** A CProver-based verifier for Java bytecode [71]. JBMC translates class files into the GOTO IR and symbolically executes them. It introduced the *Java Operational Model* (JOM) to capture library semantics and a dedicated string solver to handle operations such as concatenation or substring. This made JBMC the first bit-precise BMC verifier for Java.

- **FuSeBMC:** FuSeBMC [72] combines ESBMC counterexamples with fuzzing for test generation methodologies.

- **Kani:** A bounded model checker for Rust developed by AWS in collaboration with Oxford and Manchester [73]. Rust code is compiled into MIR, then lowered to GOTO, reusing the CBMC pipeline. Kani verifies memory safety, panics, pointer validity, and user-specified contracts. Its Rust-specific front end made BMC applicable to modern systems programming.

- **EBF:** The *Ensembles of Bounded Model Checking with Fuzzing* framework [74] combines state-of-the-art BMC tools (e.g., CBMC, ESBMC) with a new concurrency-aware gray-box fuzzer called OpenGBF.

- **GOTO Transcoder:** Originating from this thesis, the Transcoder bridges differences between CBMC and ESBMC's GOTO encodings. It normalizes instruction formats, typing rules, and solver constructs, enabling CBMC-generated GOTO binaries to be verified directly by ESBMC. This work highlights the thesis' central claim: a unified formal GOTO semantics can serve as a compatibility layer across the ecosystem. Chapter 5 presents its design and evaluation.

- **ESBMC-AI:** A recent extension of ESBMC integrating Large Language Models (LLMs) with counterexample-guided repair [76]. ESBMC generates counterexamples, which are embedded into prompts to guide LLM-generated patches. Candidate repairs are re-verified, iterating until a safe fix is found. This approach exploits ESBMC's low false-positive rate to tame the nondeterminism of LLMs, achieving higher repair accuracy than LLM-only baselines.

Tracing the development of CBMC, ESBMC, and their derivatives shows both the strengths of bounded model checking and the fragmentation that arises from diverging implementa-

tions. This motivates the central contribution of the thesis: by giving GOTO programs a formal semantics, we provide a unifying framework that can reconcile these differences, enable interoperability across tools, and support new extensions such as the GOTO Transcoder (Chapter 5).

2.3.4 Industrial Use

In recent years, CProver tools have achieved significant industrial uptake in domains demanding safety, reliability, and compliance:

- **AWS and CBMC [20]:** Amazon Web Services integrated CBMC into the verification of the `s2n` TLS library, ensuring memory safety, constant-time execution, and contract compliance in CI/CD pipelines.
- **Intel and ESBMC [33]:** Intel applied ESBMC to power-management firmware, verifying pointer safety and arithmetic properties critical to hardware reliability.
- **Rust verification with Kani [73]:** Developed with AWS, Kani extended CProver's methodology to Rust, focusing on memory safety and concurrency. Its infrastructure also enabled this thesis' GOTO Transcoder (Section 5.8) and ESBMC [77].
- **Diffblue [78]:** A University of Oxford spin-out that commercialized CProver principles for enterprise Java testing, automatically generating unit tests via symbolic execution.
- **Veribee [79]:** A University of Manchester spin-out leveraging ESBMC and FuseBMC to deliver cybersecurity-focused verification in CI/CD pipelines, tailored to industrial partners.
- **Byte Repair [80]:** Another University of Manchester spin-out leveraging ESBMC-AI (as the verification engine) to verify and repair code from different domains from users.
- **Ethereum and ESBMC [36]:** the Python front-end of ESBMC was used to ensure safety properties on the Ethereum spec. ESBMC was able to identify an overflow bug.

2.4 Summary

This chapter introduced the theoretical foundations that underpin the verification framework developed in this thesis. We began with structural definitions—inductive datatypes, pattern matching, and curried functions—that establish the notational style and functional reasoning used throughout the thesis and in the Isabelle/HOL mechanization (Appendix A). These foundations enable the definition of recursive procedures, the use of structural induction for termination arguments, and principled modeling of symbolic execution semantics.

Next, we examined core concepts in software verification: safety properties, partial and total correctness, and the inherent limitations of automated reasoning. These notions clarify the scope of CProver tools, which primarily target bounded safety checking rather than liveness properties, and provide the theoretical basis for the semantics of assertions, symbolic traces, and program transformations developed in Chapters 3 and 4.

Finally, we traced the historical evolution of the CProver ecosystem. The development of CBMC and ESBMC illustrates both the strength of bounded model checking and the challenges introduced by diverging implementations. Despite relying on different solver back ends and verification strategies, these tools share the GOTO IR as their central intermediate representation. This observation motivates the central contribution of this thesis: by formalizing the semantics of GOTO and its transformations, we can unify, extend, and interconnect CProver tools. The Industrial Use subsection of Section 2.3 further highlights the growing relevance of these tools in practice, reinforcing the need for a principled and extensible foundation.

The next chapter – Chapter 3 – builds on this foundation by introducing the abstract GOTO language and its symbolic trace semantics, which serve as the formal core of the proposed unification framework.

Chapter 3

GOTO Language

This chapter defines the syntax and semantics of the **Abstract GOTO Language** used throughout this thesis [28]. This language serves as the formal foundation for addressing the challenges outlined in Chapter 1 and underpins several verification algorithms used in CProver tools, such as bounded model checking [49], k-induction [81], and sequentialization [23]. Our focus is on two core components:

- $\mathbb{G}$: an *abstract programming language* that models C-like control flow and data structures;
- $\mathbb{T}$: a *trace language* representing bounded, single-path executions derived from $\mathbb{G}$ programs.

Figure 3.1 illustrates the overall verification flow. A program expressed in $\mathbb{G}$ is symbolically executed up to a bound k, producing a trace $\mathbb{T}$. This trace is then translated into a decision procedure (e.g., SAT [46] or SMT [82]) that determines whether the program behavior satisfies its assertions.

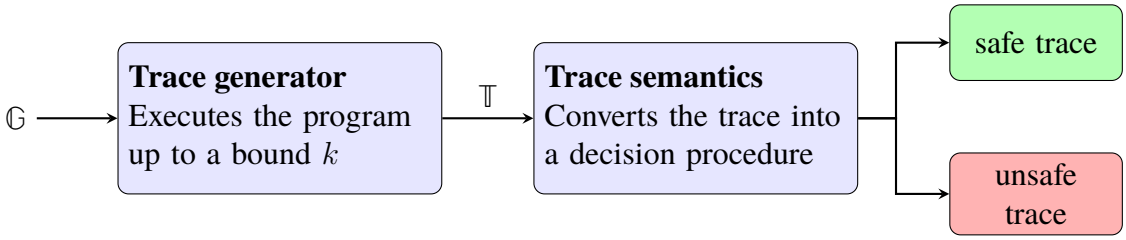


Fig. 3.1. Verification flow of a GOTO program. A bounded program trace ($\mathbb{T}$) is generated from a program ($\mathbb{G}$) and evaluated as either *safe* or *unsafe*. Verification algorithms may then increase the bound or terminate based on trace safety.

The trace language $\mathbb{T}$ captures a symbolic path of a GOTO program up to a bound. Compared to $\mathbb{G}$, it enforces stronger restrictions, including *immutability* (variables cannot be reassigned) and guaranteed termination (the trace execution always terminates). These properties enable translation into logical formulas for decision procedures. Section 3.2 elaborates on the semantics and structure of the trace language.

By contrast, the abstract programming language $\mathbb{G}$ permits constructs such as loops and mutable variables, making it closer in spirit to imperative languages such as C. This expres-

siveness eases the translation from high-level input programs (see Section 3.3). Figure 3.2 presents a mock example demonstrating how a simple C program is translated into both the GOTO language ($\mathbb{G}$) and its corresponding trace ($\mathbb{T}$). The first subfigure shows the original C program, the second shows the $\mathbb{G}$ equivalent, and the third shows the corresponding trace in $\mathbb{T}$, with all variables being assigned based on the program's control flow. Unlike C, the abstract GOTO language does not contain undefined behavior. Potential undefined behavior in the source program is either (i) explicitly encoded as an assertion in the GOTO program (e.g., bounds checks, overflow checks), or (ii) resolved by nondeterministic choice or fixed semantics. This design isolates verification concerns from language-level vagueness, ensuring that GOTO has a total, well-defined semantics.

```
1 int cond;
2 int x = 1;
3 if(cond)
4   x = 0;
5 assert(x);
```

(a) C program with error

```
1 ASSIGN(x,1)
2 IfThenGoto(
3   Not(cond),5)
4 ASSIGN(x,0)
5 ASSERT(x)
```

(b) Mock $\mathbb{G}$ equivalent

```
1 ASSIGN(x0, 1)
2 ASSIGN(x1, 0)
3 ASSIGN(x2,
4   ITE(cond,x1,x0
    )
5 ASSERT(x2)
```

(c) Mock $\mathbb{T}$ equivalent

Fig. 3.2. Mock example from a C program into the GOTO program and trace.

The design of both $\mathbb{G}$ and $\mathbb{T}$ is inspired by the structure of ESBMC and CBMC, the two most active tools in the CProver ecosystem. At the time of writing, CBMC [83] has 946 GitHub stars, and ESBMC [84] has 377. While ESBMC began as an SMT-based Context-Bounded Model Checker for Multi-threaded Programs fork of CBMC [28], both tools have since evolved independently. Their languages differ notably in areas such as concurrency, exception handling, and mathematical constructs. Section 3.7 provides a comparison of the main features.

In this chapter, we first define a common core language that captures the key abstractions used across tools (Section 3.2 and Section 3.3). We then introduce *language extensions* (Section 3.5) that accommodate additional constructs found in practical verification scenarios. Although prior work [28], [35], [36] has described parts of the language, these efforts have been partial—focusing on grammar definitions without formal semantics or trace modeling.

We encourage readers to review background material in Chapter 2, particularly Section 2.1, which introduces key notation used in the structural definitions. Our notation adopts standard functional programming conventions, such as: λ-functions, and currying [44].

All definitions introduced in this chapter are formalized in the Isabelle/HOL proof assistant [34], and are presented in a *literate programming* style [85] to improve accessibility. These formalizations are available in Appendix A and are written with the assumption that the reader may be unfamiliar with functional languages or theorem provers.

Finally, we emphasize that the GOTO language is abstract and is not intended for direct programming use. Instead, it serves as a semantic foundation for verification tools, providing a structured intermediate representation that enables formal reasoning and supports the verification of software correctness (see Section 2.2).

3.1 Basics

This section introduces the basic constructs used throughout the GOTO language. GOTO programs are composed of two kinds of terms: concrete values (natural numbers) and symbolic values (placeholders representing unknowns). The set of symbols $\mathbb{S}$ is defined as the set s_0, s_1, . . ., where each s_i represents a different symbolic placeholder for the values used in the program. Each symbol is uniquely identified by its natural number index, i.e., $\mathbb{S} = \{s_n \mid n \in \mathbb{N}\}$. The constructor $Symbol : \mathbb{N} \to \mathbb{S}$ (used in Section A.2.1) maps an index to its corresponding symbol. We start with Definition 3.1, which contains the primitives used.

Definition 3.1. Primitives used in GOTO programs.

$$\begin{aligned} \mathit{nat} \in \mathbb{N}, &\quad \textit{natural numbers} \\ \mathit{bool} \in \mathbb{B} = \{\top, \bot\}, &\quad \textit{boolean values (used in both syntax and evaluation)} \\ s_0, s_1, \ldots \in \mathbb{S}, &\quad \textit{symbols} \end{aligned}$$

The GOTO language is intentionally untyped: all values are represented uniformly through naturals and arrays. This simplifies the formal semantics by avoiding a type system at the abstract level. Typing is handled at the CProver tool level via the Irep representation (Section 3.7), and type information (e.g., bit-widths) is encoded as parameters in expressions such as $Constant\ n\ w$.

Next, we start by defining structures that use these primitives. Arrays are modeled as infinite sequences indexed by natural numbers, as shown by Definition 3.2. An array can hold any other set α (similar to a parametric type [86] [87] from modern programming languages), represented as A_α. Any primitive can be used for the array, for example: $A_\mathbb{N}$ (array of natu-

rals), $A_{\mathbb{B}}$ (array of booleans), or $A_{\mathbb{S}}$ (array of symbols).

Definition 3.2. Arrays with parametric type α. The array is a total function:

$$A_\alpha : \mathbb{N} \to \alpha, \textit{ array of type } \alpha$$

This total function formulation ensures that all array accesses are defined (e.g., no out-of-bounds behavior). A simple way to define arrays is to use λ-expressions [88], which provides a quick way to define functions (see Section 2.1). For example, an array of naturals (that is, $A_{\mathbb{N}}$) consisting of $\langle 1, 0, 0, \ldots \rangle$ can be defined as: $\lambda i.$ **if** $i = 0$ **then** 1 **else** 0.

In addition to arrays, we define *lists* as finite sequences constructed inductively [89]. The inductive structure of lists guarantees that any recursive or iterative algorithm over them will terminate. For our purposes, lists serve as a means to iterate over a finite trace. Definition 3.3 shows how a list is defined.

Definition 3.3. List constructors with the parametric type α.

$$\begin{aligned} L_\alpha ::= \;& Nil \\ \mid \;& Cons\ \alpha\ L_\alpha \end{aligned}$$

Since this list is defined with the parametric type α, it can be replaced with any other construct. Example 3.1 shows a list over $L_{\mathbb{N}}$.

Example 3.1. List for the sequence $\langle 2, 3, 1 \rangle$.

$$Cons\ 2\ (Cons\ 3\ (Cons\ 1\ Nil))$$

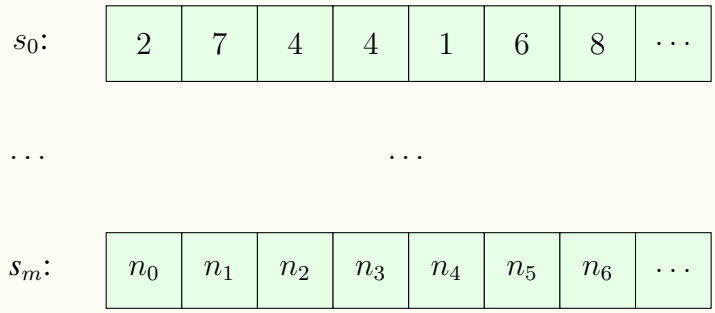
Before defining trace semantics, we must describe how symbols are assigned concrete values during execution. For that, we will use environments [14] [88], which in this context is a mapping between symbols and concrete values. Definition 3.4 describes how an environment is defined as a function. Obtaining environments for a trace or program goes beyond the scope of this work. Previous works from CBMC [18], [90] and ESBMC [28], [91] go in-depth about encoding traces in satisfiability formulas to produce this environment. For this work, we will rely on the existence of an environment to execute the program trace.

Definition 3.4. Environment.

$$ev : \mathbb{S} \to A_{\mathbb{N}},\ \textit{environment}$$

In practice, an environment is a function that maps every symbol in the program into an array of naturals. Example 3.2 contains an example of an environment; each symbol has an array of associated values in the figure.

Example 3.2. Environment *env*.



In this example, we can obtain the value 7 by calling $\textit{env}\ s_0\ 1$.

We choose to represent environments as maps to arrays because this provides a single uniform representation for all values. Memory, pointers, and aggregate types are naturally modeled as arrays; treating primitives as constant arrays (via $ToArray$) avoids special-casing in the semantics. For primitives, the value is replicated across all indices, and by convention the first index is used to extract the scalar result. With the basics in place, we can proceed to formalize the trace language, which constitutes a subset of the abstract language.

3.2 GOTO Trace ($\mathbb{T}$)

The *GOTO Trace* results from executing a *GOTO Program*, by interpreting or running a symbolic execution (as will be shown in Section 3.3.4). We begin by defining the trace language, as its expressions form a syntactic and semantic subset of those available in GOTO Programs. A trace consists of assignments, restrictions, and properties over symbols defined in a way that the execution procedure evaluates (see Section 3.2.2) to true if a safety violation is found. The assertions are negated to generate a failing environment. Example 3.3 illustrates a mock trace as an example of a *GOTO Trace*.

Example 3.3. Mock trace with a property violation.

```
1 ASSUME(s0 > 10)          // Continues only if s0 > 10
2 ASSIGN(s1, s0 + 1)       // s1 = s0 + 1
3 ASSERT(s1 < s0)          // Checks whether s1 < s0
```

The sequence can be interpreted as: $\boxed{(s_0 > 10) \wedge ((s_0 + 1) \geq s_0)}$. Note that, during trace interpretation, the assertion is semantically negated to identify failing environments. This formula is satisfied by $\boxed{s_0 \to 11}$, which triggers the assertion failure.

The concept of memory in the GOTO trace is that each symbol is mapped to an array of values. Any operations that cause a side effect (such as an assignment) causes the environment to be updated for that symbol. Additionally, the trace has some properties that make it simpler to reason:

Immutable Once a symbol is assigned, its value is constant and cannot change.

No loads/stores In the program trace, there are no operations over memory, only over symbols. Since everything is immutable, all operations are directly defined in terms of symbols and their values.

Loop-free Traces are finite sequences and do not involve iteration constructs, ensuring guaranteed termination.

3.2.1 Expressions

Expressions are the central unit used for constructing a GOTO program. They are used to express logical and arithmetic operations. They can be recursively grouped into many subexpressions. Let $\mathbb{T}_E$ be the expressions which are defined in Definition 3.5.

Definition 3.5. Trace Expressions Constructors.

$$
\begin{aligned}
\mathbb{T}_E ::= \quad & ConstantBool\ (\mathbb{B}), \textit{truth value} \\
| \quad & And\ \mathbb{T}_E\ \mathbb{T}_E, \textit{conjunction} \\
| \quad & Not\ \mathbb{T}_E, \textit{complement} \\
| \quad & GreaterThan\ \mathbb{T}_E\ \mathbb{T}_E, \textit{comparison} \\
| \quad & Constant\ \mathbb{N}\ \mathbb{N}, \textit{constant natural} \\
| \quad & Add\ \mathbb{T}_E\ \mathbb{T}_E\ \mathbb{N}, \textit{addition between naturals} \\
| \quad & ConstantArray\ A_{\mathbb{N}}, \textit{array of naturals} \\
| \quad & Select\ \mathbb{T}_E\ \mathbb{T}_E, \textit{select an element of an array} \\
| \quad & With\ \mathbb{T}_E\ \mathbb{T}_E\ \mathbb{T}_E, \textit{array update} \\
| \quad & ITE\ \mathbb{T}_E\ \mathbb{T}_E\ \mathbb{T}_E, \textit{if-then-else} \\
| \quad & SymbolValue\ \mathbb{T}_E, \textit{value of the symbol}
\end{aligned}
$$

The semantics of trace expressions are given by the function *EvalExpr* (see its signature in Definition 3.6), which maps an expression and environment to an array of natural numbers. Additionally, any *Language Extension* (as will be described in Section 3.5) that introduces a new expression must also implement this function. While arrays are the standard output, some computations only require a single element (such as logical operators). In these cases, the convention is to *use the first value of the array as the result*. The contents of this section are formalized in Isabelle in Section A.2.1.

Definition 3.6. *EvalExpr* syntax.

$$EvalExpr : \mathbb{T}_E \to ev \to A_{\mathbb{N}}$$

Numerical Expressions

Numerical expressions are the ones used to apply operations to numbers. As mentioned earlier, when dealing with primitives as the unit, the convention is to replicate them across all indices of the array. For that, we define the function $ToArray_{\mathbb{N}}$, which converts a natural number to an array consisting of that same value. The function is defined as: $\boxed{\lambda i.\ n}$, where n is the natural number used. Definition 3.7 shows the semantics for Constants.

Definition 3.7. Semantics of Constant Expressions.

$$EvalExpr\ (Constant\ n_0\ n_1)\ env \triangleq ToArray_{\mathbb{N}}\ (n_0\ \textit{mod}\ n_1)$$

and Definition 3.8 for the addition operation:

Definition 3.8. Semantics of Add Expressions.

$$\begin{aligned} EvalExpr\ (Add\ e_0\ e_1\ n_0)\ env \triangleq\ & ToArray_{\mathbb{N}}(((EvalExpr\ e_0\ env)(0) \\ & + (EvalExpr\ e_1\ env)(0))\ \textit{mod}\ n_0) \end{aligned}$$

Note that all arithmetic operations end with modulo operations. In Example 3.4 we show how these expressions can be used.

Example 3.4. Examples of Numerical Expressions.

Considering an arbitrary environment E. The following expressions are computed as follows:

$$\begin{aligned} EvalExpr\ & (Constant\ 2\ 32)\ E \triangleq ToArray_{\mathbb{N}}\ (2\ \textit{mod}\ 32) \\ & = ToArray_{\mathbb{N}}\ (2) = \lambda i.2 = \langle 2, 2, \ldots \rangle \end{aligned}$$

$$\begin{aligned} EvalExpr\ & (Add\ (Constant\ 2\ 32)\ (Constant\ 2\ 32))\ E \triangleq ToArray_{\mathbb{N}}(\\ & ((EvalExpr\ (Constant\ 2\ 32)\ E)(0) \\ & + (EvalExpr\ (Constant\ 2\ 32)\ E)(0))\ \textit{mod}\ n_0) \\ & = ToArray_{\mathbb{N}}(((\lambda i.2)(0) + (\lambda i.2)(0))\ \textit{mod}\ 32) \\ & = ToArray_{\mathbb{N}}((2+2)\ \textit{mod}\ 32) \\ & = \lambda i.4 \end{aligned}$$

Logical Expressions

These are expressions that work on Boolean values. Similarly to numerical, we also need to define the function $ToArray_{\mathbb{B}}$ to encode booleans into arrays: $\lambda i.\textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 0$. Definition 3.9 contains the semantics for all the logical expressions.

Definition 3.9. Semantics for the logical expressions.

$$
\begin{aligned}
&EvalExpr \quad (ConstantBool\ b)\ env \triangleq ToArray_{\mathbb{B}}\ b \\
&EvalExpr \quad (Not\ e_0)\ env \triangleq ToArray_{\mathbb{B}}((EvalExpr\ e_0\ env)(0) = 0) \\
&EvalExpr \quad (And\ e_0\ e_1)\ env \triangleq ToArray_{\mathbb{B}}(\\
&\qquad ((EvalExpr\ e_0\ env)(0) \neq 0) \wedge ((EvalExpr\ e_1\ env)(0) \neq 0)) \\
&EvalExpr \quad (GreaterThan\ e_0\ e_1)\ env \triangleq \\
&\qquad ToArray_{\mathbb{B}}((EvalExpr\ e_0\ env)(0) > (EvalExpr\ e_1\ env)(0))
\end{aligned}
$$

In these operations, we also use another convention: 0 means *false* while non-zero means *true*. Example 3.5 demonstrates the logical expressions.

Example 3.5. Examples of Logical Expressions.

Considering an arbitrary environment $\boxed{\mathrm{E}}$. The following expressions are computed as follows:

$$
\begin{aligned}
&EvalExpr \quad (ConstantBool\ \bot)\ E \triangleq ToArray_{\mathbb{B}}\ (\bot) \\
&\qquad \triangleq \lambda i.0 \triangleq \langle 0, 0, \ldots \rangle
\end{aligned}
$$

$$
\begin{aligned}
&EvalExpr \quad (Not\ (ConstantBool\ \bot))\ E \triangleq ToArray_{\mathbb{B}} \\
&\qquad (EvalExpr\ (ConstantBool\ \bot)\ E\ (0) = 0) \\
&\qquad \triangleq ToArray_{\mathbb{B}}\ ((\lambda i.0\ 0) = 0) \triangleq \lambda i.1
\end{aligned}
$$

$$
\begin{aligned}
&EvalExpr \quad (And\ (ConstantBool\ \bot)\ (ConstantBool\ \bot))\ E \triangleq \\
&\qquad (EvalExpr\ (ConstantBool\ \bot)\ E\ (0) = 0) \\
&\qquad \wedge (EvalExpr\ (ConstantBool\ \bot)\ E\ (0) = 0) \triangleq \lambda i.0
\end{aligned}
$$

$$
\begin{aligned}
&EvalExpr \quad (GreaterThan\ (Constant\ 2\ 32)\ (Constant\ 2\ 32))\ E \triangleq ToArray_{\mathbb{B}}(\\
&\qquad (EvalExpr\ (Constant\ 2\ 32)\ E)(0) \\
&\qquad > (EvalExpr\ (Constant\ 2\ 32)\ E)(0)) \\
&\qquad \triangleq ToArray_{\mathbb{B}}(2 > 2) \triangleq \lambda i.0
\end{aligned}
$$

Symbol Expressions

Symbol expressions are containers for symbols, mainly used to obtain values from the environment. We define $Symbol : \mathbb{N} \to \mathbb{S}$ as the constructor that maps a natural number n to

the symbol s_n. This computation consists of two main steps: a) evaluate the index expression to obtain n, and b) look up s_n in the environment. Its semantics are defined in Definition 3.10.

Definition 3.10. Semantics of Symbol Expressions.

$$EvalExpr\ (SymbolValue\ e)\ env \triangleq env\ (Symbol\ (EvalExpr\ e\ env\ 0))$$

In Example 3.6 we demonstrate how the environment and symbol expressions interact.

Example 3.6. Examples of Symbol Expressions.

Considering an environment E used in Example 3.2. The following expressions are computed as follows:

$$\begin{aligned} EvalExpr\quad & (SymbolValue\ (Constant\ 0\ 32))\ E \triangleq E \\ & (Symbol\ (EvalExpr\ (Constant\ 0\ 32)\ 0)) \\ & \triangleq E\ (Symbol\ 0) \triangleq \langle 2, 7, 4, 4, \ldots \rangle \end{aligned}$$

Array Expressions

Operations over arrays are the base unit for the trace. Definition 3.11 defines the semantics for the operations:

Definition 3.11. Semantics for the Array Expressions.

$$\begin{aligned} EvalExpr\quad & (ConstantArray\ a)\ env \triangleq a \\ EvalExpr\quad & (With\ a\ o\ u)\ env \triangleq \lambda i.\ \textbf{if}\ i = (EvalExpr\ o\ env)\ (0) \\ & \textbf{then}\ (EvalExpr\ u\ env)\ (0) \\ & \textbf{else}\ (EvalExpr\ a\ env)\ (i) \\ EvalExpr\quad & (Select\ a\ o)\ env \triangleq \\ & ToArray_{\mathbb{N}}((EvalExpr\ a\ env)(EvalExpr\ o\ env)(0)) \end{aligned}$$

Example 3.7. Examples of Array Expressions.

Considering an environment E used in Example 3.2. The following expressions are computed as follows:

$$EvalExpr \quad (ConstantArray\ (\lambda i.1))\ E \triangleq \lambda i.1$$

$$EvalExpr \quad (With\ (ConstantArray\ (\lambda i.1))\ (Constant\ 2\ 32)\ (Constant\ 3\ 32))\ E \triangleq \lambda i.\ \textbf{if}\ i = 2\ \textbf{then}\ 3\ \textbf{else}\ 1$$

$$EvalExpr \quad (Select\ (ConstantArray\ (\lambda i.1))\ (Constant\ 2\ 32))\ E \triangleq \lambda i.1$$

ITE expressions

The ITE expressions consist of *if-then-else* that enables conditional values for expressions. Defined in Definition 3.12:

Definition 3.12. Semantics for the ITE expressions.

$$EvalExpr \quad (ITE\ e_0\ e_1\ e_2)\ env \triangleq \textbf{if}\ (EvalExpr\ e_0\ env)(0) \neq 0\ \textbf{then}\ EvalExpr\ e_1\ env\ \textbf{else}\ EvalExpr\ e_2\ env)$$

Example 3.8. Examples of Array Expressions.

Considering an environment E used in Example 3.2. The following expressions are computed as follows:

$$EvalExpr \quad (ITE\ (Constant\ 0\ 32)(Constant\ 1\ 32)(Constant\ 2\ 32))\ E \triangleq \lambda i.2$$

3.2.2 Statements

A GOTO trace consists of a sequence of statements; these statements are:

Assign This sets the value of a symbol as the result of an expression. A symbol can only be written once.

Assume Preconditions for the trace. If the expression evaluates to *false*, the execution does not proceed further and the trace is considered *safe* for that environment (i.e., no violation is reachable from that point).

Assert A safety condition. If the expression evaluates to *false*, it indicates a property violation and the trace is *unsafe* for that environment; otherwise, execution proceeds.

Definition 3.13 shows the syntax of the statements.

Definition 3.13. Syntax of trace statements.

$$\begin{aligned} \mathbb{T} ::= \quad & Assign\ \mathbb{S}\ \mathbb{T}_E \\ | \quad & Assume\ \mathbb{T}_E \\ | \quad & Assert\ \mathbb{T}_E \end{aligned}$$

For semantics, in Definition 3.14 we define the function *EvalTrace*, which evaluates a *list* of trace statements into a truth value that represents whether the trace is *unsafe* against an *environment*. We use $\bot$ (false) to denote a *safe* outcome and $\top$ (true) to denote an *unsafe* outcome. The contents of this section are formalized in Isabelle in Section A.2.2.

Definition 3.14. Syntax and Semantics of *EvalTrace*.

$$EvalTrace:\ L_{\mathbb{T}} \to ev \to \mathbb{B}$$

First, an empty list is a safe trace.

$$EvalTrace\ Nil\ env \triangleq \bot$$

An assignment updates the environment and proceeds to the following statement:

$$\begin{aligned} EvalTrace\ (Cons\ (Assign\ s\ e)\ l)\ env \triangleq\ & EvalTrace\ l\ (\lambda i. \\ & \textbf{if}\ i = s\ \textbf{then}\ EvalExpr\ e\ env\ \textbf{else}\ env\ i) \end{aligned}$$

Assumptions evaluate the condition, and if it does not hold, it implies that the execution cannot proceed. Therefore, the trace is considered *safe* for that environment.

The execution continues otherwise:

$$EvalTrace\ (Cons\ (Assume\ e)\ l)\ env \triangleq \textbf{if}\ EvalExpr(e, env)(0) \neq 0 \\ \textbf{then}\ EvalTrace\ l\ env\ \textbf{else}\ \bot$$

Similarly, for an *Assert* statement, *EvalTrace* evaluates the condition; if it does not hold, the trace is considered *unsafe*. Otherwise, the execution continues:

$$EvalTrace\ (Cons\ (Assert\ e)\ l)\ env \triangleq \textbf{if}\ EvalExpr(e, env)(0) \neq 0 \\ \textbf{then}\ EvalTrace\ l\ env\ \textbf{else}\ \top$$

If $EvalTrace$ returns $\top$ for a given environment, we say that the trace is *unsafe* under that environment. However, we cannot conclude safety from a single environment; to say that a trace is *safe*, we need to prove that no environment causes $EvalTrace$ to return $\top$. To illustrate how the interpretation of a trace depends on the environment, consider the example in Example 3.9. This trace performs an assignment followed by an assertion. The evaluation of the assertion determines whether the trace is considered safe or unsafe under a particular environment. As mentioned above, the computation of environments and their proofs goes beyond the scope of this work. With the trace syntax and semantics defined, we can start defining the GOTO programming language.

Example 3.9. A GOTO trace whose safety depends on the environment.

```
1 ASSIGN(s1, s0 + 1)          // s1 = s0 + 1
2 ASSERT(s1 < 10)             // Checks whether s1 < 10
```

We examine two possible environments:

- *Environment 1 (safe):* $\boxed{env_1 = \lambda s.\ \lambda x.\ 5}$

  Here, all symbols resolve to 5:

  - $s_0 = 5 \rightarrow s_1 = 6$
  - Assertion: $6 < 10$ is **true**

  The trace completes successfully with no violations, so `EvalTrace` returns $\bot$, meaning the trace is *safe* under this environment.

- *Environment 2 (unsafe):* $\boxed{env_2 = \lambda s.\ \textbf{if}\ s = s_0\ \textbf{then}\ \lambda x.\ 10\ \textbf{else}\ \lambda x.\ 5}$

  In this case, the evaluation proceeds as follows:

- $s_0 = 10 \rightarrow s_1 = 11$
- Assertion: $11 < 10$ is **false**

Thus, `EvalTrace` returns $\top$, indicating an *unsafe* trace under this environment.

This example illustrates that a GOTO trace is deemed *unsafe* if there exists at least one environment under which an assertion fails. Conversely, to prove that a trace is *safe*, one must ensure that no environment exists where an assertion evaluates to false.

3.3 GOTO program

The GOTO program is the representation used to generate program traces. In the CProver framework, this language is intended to be used as an intermediate representation of the high-level programming languages in the analysis. Unlike GOTO traces, GOTO programs may include mutable state and loops.

3.3.1 Instructions

The GOTO program instructions consist of:

Assign This sets the value of a symbol as the result of an expression. With respect to execution, this statement constructs a new symbol to store the expression.

Assume The assumption consists of preconditions for the program. The program interpretation can only continue if the assumption holds; otherwise, it ends the program as safe.

Assert Asserts are properties that ensure correctness. The program interpretation can only continue; otherwise, it ends the program as unsafe.

IfThenGoto A conditional jump. If the condition holds, then the program counter moves to the Goto value.

These statements, defined as G_I, have the following grammar:

Definition 3.15. Primitives used in GOTO programs.

$$
\begin{aligned}
G_I ::= \quad & Assign\ \mathbb{S}\ \mathbb{T}_E \\
\mid \quad & Assume\ \mathbb{T}_E \\
\mid \quad & Assert\ \mathbb{T}_E \\
\mid \quad & IfThenGoto\ \mathbb{T}_E\ \mathbb{N}
\end{aligned}
$$

Note that they use the same expressions as in the trace. All control flow is captured through `IfThenGoto` alone: unstructured control flow constructs from the source language (e.g., `goto`, `break`, `continue`, `switch`) are lowered into conditional and unconditional jumps by the CProver front-ends during translation. An unconditional jump is expressed as `IfThenGoto(`$\top$`, n)`. With the statements, the CProver tools can verify high-level programming languages. Example 3.10 contains a mock example of a C snippet next to the generated GOTO. Note that from now on we will *omit and simplify the constructs of the GOTO language, such as the arithmetic and the logical operators*.

Example 3.10. C code and its GOTO counterpart. The code contains variables with different widths and loops.

```
1 uchar a = 0;       // ASSIGN (a, Constant(0,8))
2 uint sum = 0;      // ASSIGN (sum, Constant(0,32))
3 do {               //
4   sum += a;        // ASSIGN (sum, Add(sum,a,32))
5   a++;             // ASSIGN (a, Add(Constant(1,8),a,8))
6 } while (a);       // IFTHENGOTO (a, 4)
7 assert(sum > a); // ASSERT (GreaterThan(sum,a))
```

To produce the program traces needed for execution, we use a symbolic execution approach to explore the program paths. Symbolic execution [92] allows us to explore the program space while keeping the values as symbols. Since programs may contain unbounded loops, the number of execution paths can be infinite. Therefore, we limit the exploration to an upper bound.

The main idea of symbolic execution is the use of symbolic values instead of concrete values to store variable values. As a result, the output of a program can be represented as a function [93] (similar to the environment).

In the next sections, we will focus on the features needed for the engine. First, we will focus on identifying loops; the bounded aspect of the engine needs to track where the loops are in the execution. Next, we will focus on how to deal with a variable that has many values depending on the path conditions. Finally, we define the bounded symbolic execution algorithm.

3.3.2 Extracting loops

A GOTO program is just a sequence of instructions. We can then consider a program as a *list of statements*. A *loop* is an *IfThenGoto* statement that jumps to a previous instruction.[1] A loop is then defined as the pair $\langle l_s, l_e \rangle$ where l_s is the index of the beginning of the loop and l_e is the backward jump. Example 3.11 contains an example of a loop and its pair.

Example 3.11. A program loop. At line 3 the conditional jump goes back to line 1. The loop in this program is defined as $\langle 1, 3 \rangle$.

```
1 ASSIGN(s0, Constant(1,32))      // unsigned s0 = 1
2 ASSERT(s0)                      // assert (s0 != 0)
3 IFTHENGOTO(s0, 1)               // if (s0 != 0) goto 1
4 ...                             // program continues...
```

The algorithm 1 contains an algorithm to extract all loops of the program given a list of statements. The algorithm consists of iterating over all instructions and capturing backward jumps.

Algorithm 1 Extract loops

```
1: procedure EXTRACT-LOOPS(I)
2:     loops ← { }                                  ▷ Initialize result
3:     pc ← 0                                       ▷ Initialize Program Counter
4:     while pc ≤ length(I) do
5:         i ← I[pc]                                ▷ Get instruction
6:         if IsIfThenGoto(i) ∧ i.target ≤ pc then
7:             loops ← ∪ { ⟨ i.target, pc ⟩ }       ▷ Store loop
8:         pc ← pc + 1                              ▷ Increment program counter
       return loops
```

[1] CProver tools commonly refers to this as a backwards jump.

3.3.3 Merging paths

The main difficulty lies in determining the correct symbolic value to use for a variable, especially when multiple control paths define different values. Compilers also have to deal with this problem to work with static optimizations (that optimizes the code generated), which they solve by applying transformations and introducing explicit $\phi - nodes$ into the code. Explicit $\phi - nodes$ add an important property to the program that *all definitions dominate their uses* in the CFG [94]. However, this is not the approach used by CProver tools (as the time of this writing). In the Future Works (see Section 6.2) we will introduce the concept of $\phi - nodes$.

In order to deal with the issue of updating symbols, CProver tools rely on introducing ϕ expressions during access of the symbol. For example, a symbol s_0 that has the value "2" but when some condition "cond" is true is "1" would be represented as:

Example 3.12. Assignment over a ϕ expression.

$$ASSIGN(s_0, \phi(cond_1 \rightarrow 1, \neg cond_1 \rightarrow 2))$$

These expressions are then translated during the decision procedure. For this work, we decide against adding this ϕ expression directly and we rely on a chain of ITE operations. This is akin to how ESBMC converts its ϕ expessions [28]; another natural conversion is into a chain of implication disjunctions. Example 3.13 shows an example of the merging with the ITE operation.

Example 3.13. Trace for a merging.

The program starts with a conditional jump on the a_0 variable; each conditional path following the jump sets a different value for a_0. At the end, the value of a_0 is asserted. On the right, we show the resulting trace of this program, including the ITE.

```
1 IFTHENGOTO(a0,4)   // -- a0 != 0
2 ASSIGN(a0, 2).     // ASSIGN(a1, 2)
3 IFTHENGOTO(1,5)    // -- True
4 ASSIGN(a0, 0)      // ASSIGN(a2, 0)
5 ASSERT(a0)         // ASSERT(ITE(a0,a1,a2))
```

Example 3.14. Trace generation for a loop with bound $k = 2$, where x_0 is a nondeterministic initial value.

```
1 ASSIGN(x, x+1)
2 IFTHENGOTO(x<3, 1)    // Do-While Loop
3 ASSERT(x > 0)
```

With bound $k = 2$, the loop body is unrolled twice, introducing a fresh symbol at each iteration. As in Example 3.13, the assertion receives an ITE expression that merges all possible exit points:

```
1 ASSIGN(x1, x0+1) // always happens
2 ASSIGN(x2, x1+1) // 1st unwind
3 ASSIGN(x3, x2+1) // 2nd unwind
4 ASSERT(ITE(x1<3, ITE(x2<3, x3, x2), x1) > 0)
```

The outer ITE selects x_1 if the loop was never entered; the inner ITE selects x_2 if one iteration was taken and x_3 if two were taken.

Examples 3.13 and 3.14 both contain an extra transformation implicitly, the renumbering. This operation is responsible for updating the expressions with their latest values. For that, we define the pair $\langle g \in T_E, s \in \mathbb{S} \rangle$. We can then keep track of all symbols and their updates through a renumbering function, which maps each symbol to its list of guarded versions (guards are formally defined in Section 3.3.4): Here, L_X denotes a list of elements of type X, and $\mathbb{S}$ is the set of symbols (Definition 3.1).

Definition 3.16. Renumbering syntax

$$\textit{renumbering} : S \to L_{\langle g \in T_E, s \in \mathbb{S} \rangle}$$

The *renumbering* can then be used to know what are current symbols and guards. For instance, in the same example (Example 3.13), when arriving at Line 5 instruction, the realization of the function is:

Example 3.15. ITE expression for ϕ expression.

$$\lambda s.\ if\ s = a_0\ then\ \langle \langle a_0 = 0, a_2 \rangle, \langle a_0 \neq 0, a_1 \rangle \rangle\ else \langle \langle \top, s \rangle \rangle$$

Each pair $\langle g, s \rangle$ in the list means "under path condition g, the symbol takes the value of s."

The list is converted to a nested ITE expression: from $\langle\langle a_0 = 0, a_2\rangle, \langle a_0 \neq 0, a_1\rangle\rangle$, we obtain $ITE(a_0 = 0,\ a_2,\ a_1)$, selecting the appropriate version based on which guard holds.

The *renumbering* is dynamically changed based on the current trace. Every assignment update it based on the current path condition. In the next section we will describe the algorithm that updates this function.

3.3.4 Bounded Symbolic Execution

Using loop detection and merging operations, we can finally start building the algorithm for symbolic execution. Defining this algorithm in a declarative approach results in the addition of many auxiliary procedures. For this reason, for this part we will define this algorithm in an imperative approach while keeping the full declarative approach with the Isabelle approach (Appendix A).

The procedure *symex-k(k, instrs)* (Algorithm 2), computes a bounded symbolic execution up to a bound k using the list of instructions *instrs*. The algorithm keeps stacks of each execution path, tracking how many times a symbol was renamed or a loop was taken. In the end, it will result in a GOTO trace that can be used to analyze the program properties (up to that bound). To keep the different stacks and executions, we also added the procedure *symex-k-it* (Algorithm 3) which keeps track of all the symbolic paths.

Before describing the algorithm, we first would like to informally describe some helper functions:

Renumber: The *Renumber* procedure takes a $\mathbb{T}_E$ expression and a *renumbering* function and produces a new $\mathbb{T}_E$ in which symbols are replaced by their ITE-merged forms. Concretely, it traverses the expression e and for each symbol s encountered, replaces it with the ITE chain built from the renumbering map: if $r(s) = \langle\langle g_1, s_1\rangle, \ldots, \langle g_n, s_n\rangle\rangle$, the result is $ITE(g_1, s_1, ITE(g_2, s_2, \ldots, s_n)\ldots)$. We will also assume that the *Total-Symbols()* procedure will give us the current number of symbols in the program. Also, for implementation, we will consider the *renumbering* as a hash map.

Is X: This procedure returns a boolean that evaluates to true if an instruction is of type "X". This will be used mainly to detect what type of instruction we are dealing with.

Accessing elements of instructions: To access an element of an instruction, we will use the "." operator followed by an alias for that instruction field. The alias was chosen for

the sake of clarity.

Concatenation ($++$): The operator $++$ denotes list concatenation, appending the elements of the right operand to the left.

Algorithm 2 Symbolic Execution Initialization

```
1: procedure SYMEX-K(inst, k)
2:     pc ← 0
3:     limit ← size(inst)
4:     guard ← ⊤
5:     renumbering ← λx. ⟨⟨⊤, x⟩⟩
6:     return symex-k-it(inst, k, pc, limit, guard, renumbering)
```

The main challenge of the algorithm is that the program counter (PC) may be in multiple points of the execution; therefore, each condition in the program path that leads to that counter needs to be tracked. These conditions will be referred to as *guards* (or path guards) and consist of a $\mathbb{T}_E$ that represents a truth value for the execution of the path.

3.4 Verifying GOTO Programs

To verify a GOTO program, we are interested in knowing whether there exists a program execution that results in a violation of an `ASSERT` statement. For that, we define the following function:

Definition 3.17. EvalProgram definition

$$\mathit{EvalProgram} : G \to \mathbb{N} \to \mathbb{B}$$

where *EvalProgram* g k takes a GOTO program g and a bound k, and returns $\top$ if there exists an environment ev such that the symbolic execution up to depth k results in an assertion violation:

$$\exists ev.\ \mathit{EvalTrace}\ (\texttt{symex-k}\ g\ k)\ ev = \top$$

Using this function, we can define when a GOTO program is considered *safe* or *unsafe*:

- A program g is **unsafe at bound** k if *EvalProgram* g $k = \top$.
- A program g is **safe at bound** k if *EvalProgram* g $k = \bot$.
- A program g is **safe** (in general) if for all $k \in \mathbb{N}$, *EvalProgram* g $k = \bot$.

Algorithm 3 Symbolic Execution Interactive

```
1:  procedure SYMEX-K-IT(inst, k, pc, lastInst, g, r)
2:      if pc ≥ lastInst ∨ k = 0 then                                  ▷ Base case
3:          return Nil
4:      i ← inst[pc]                                                   ▷ Get instruction
5:      expr ← renumber(i.expr, r)                                     ▷ Renumber expression
6:      if IsIfThenGoto(i) then
7:          if i.target > pc then                                      ▷ If-Else
8:              stmt ← symex-k-it(inst, k, pc+1, i.target, Implies(g, Not(expr)), r)
9:              g ← Implies(g, (expr))
10:             pc ← i.target
11:         else                                                       ▷ Loop
12:             it ← 0
13:             stmt ← Nil
14:             while it < k do                                        ▷ incremental k interactions
15:                 expr ← renumber(i.expr, r)
16:                 stmt ← stmt ++ symex-k-it(inst, it, pc+1, i.target, expr, r)
17:                 it ← it + 1
18:             expr ← renumber(i.expr, r)
19:             g ← Implies(g,Not(expr))                               ▷ Assume loop termination
20:     else if IsAssume(i) then                                       ▷ Assumption
21:         g ← Implies(g,expr)
22:         stmt ← Assume(expr)
23:     else if IsAssert(i) then                                       ▷ Assertion
24:         g ← Implies(g,expr)
25:         stmt ← Assert(expr)
26:     else if IsAssignment(i) then                                   ▷ Assignment
27:         g ← g
28:         r ← prepend(r[i.var], Implies(g, TotalSymbols() + 1))
29:         stmt ← Assign(renumber(i.var,r), expr)
30:     it-call ← symex-k-it(inst, k, pc+1, limit, g, r)
31:     return stmt ++ it-call
```

Bounded Verification and Termination

Symbolic execution with a bound k checks safety properties up to a finite exploration depth: it can find property violations (counterexamples) reachable within k steps, but cannot conclude that a program is safe for unbounded executions.

BMC with *unwinding assertions* [95] strengthens this by asserting that all loop iterations are fully explored within the bound; if the assertion holds, the absence of violations is not an artifact of prematurely stopped unrolling. Techniques such as k-induction [29] go further by attempting to prove safety for all iterations via loop invariants. Both bounded and inductive approaches are concerned with safety properties throughout execution.

3.5 Language extensions

While the core GOTO language captures basic imperative flow, modern programming languages include dynamic memory, concurrency, and numerical extensions that require more expressive intermediate representations. We will be brief here, as these features can be referred to their original works.

3.5.1 Memory

Many programming languages (such as C) allow the developers to manipulate memory directly through the use of pointers. Furthermore, the language allows the user to dynamically allocate more memory for the program. To support these constructs, we rely on extending our set of expressions to also contain: *store* and a *load* operations over arrays. In CProver, Kroening proposed to use symbols and offsets as indexes for memory operations [61]. In here, we will do in a similar fashion, with more strict constraints. Considering an Offset $\in \mathbb{N}$ and a value $\in \mathbb{N}$.

Definition 3.18. Memory definitions

Let Memory be any total function defined as:

$$\mathsf{Memory} : \mathbb{S} \to \mathsf{Offset} \to \mathbb{N}$$

Let load be a function defined as:

$$\begin{aligned} &\mathit{load} : \mathsf{Memory} \to \mathbb{S} \to \mathsf{Offset} \to \mathbb{N} \\ &\mathit{load}(m, s, o) = m(s, o) \end{aligned}$$

Let store be a function defined as:

$$\begin{aligned} &\mathit{store} : \mathsf{Memory} \to \mathbb{S} \to \mathsf{Offset} \to \mathbb{N} \to \mathsf{Memory} \\ &\mathit{store}(m, s, o, v) = \lambda s'\ o'.\ \textbf{if}\ s = s' \wedge o = o'\ \textbf{then}\ v\ \textbf{else}\ m(s', o') \end{aligned}$$

Theorem 3.1. For any Memory m, the *load* function performed after the *store* function should yield the same value if the same symbol and offset are used as in:

$$\mathit{load}(\mathit{store}(m, s, o, v), s, o) = v$$

Proof. We aim to prove that loading from memory after storing a value returns that same value.

$$
\begin{aligned}
&\textit{load}(\textit{store}(m, s, o, v), s, o) \\
&= \textit{store}(m, s, o, v)(s, o) && \text{(by Definition 3.18)} \\
&= \begin{cases} v & \text{if } s = s \wedge o = o \\ m(s, o) & \text{otherwise} \end{cases} && \text{(by Definition 3.18)} \\
&= v && \text{(since } s = s \wedge o = o \text{ always holds)}
\end{aligned}
$$

□

Theorem 3.2. For any Memory m, the *store* function can not affect *load* operations over different offsets or symbols $(s_0 \neq s_1) \vee (o_0 \neq o_1) \implies \textit{load}(\textit{store}(m, s_0, o_0, v), s_1, o_1) = \textit{load}(m, s_1, o_1)$

Proof. We aim to prove that loading from memory after storing a value in another symbol will not result in a change of the original value. We will omit $(s_0 \neq s_1) \vee (o_0 \neq o_1)$ and refer to it when needed.

$$
\begin{aligned}
&\textit{load}(\textit{store}(m, s_0, o_0, v), s_1, o_1) && \text{Left-side} \\
&= \textit{store}(m, s_0, o_0, v)(s_1, o_1) && \text{(by Definition 3.18 (\textit{load}))} \\
&= \begin{cases} v & \text{if } s_0 = s_1 \wedge o_0 = o_1 \\ m(s_1, o_1) & \text{otherwise} \end{cases} && \text{(by Definition 3.18 (\textit{store}))} \\
&= m(s_1, o_1) && \text{(since } s_0 \neq s_1 \vee o_0 \neq o_1 \text{ by premise)} \\
&\textit{load}(m, s_1, o_1) && \text{Right-side} \\
&= m(s_1, o_1) && \text{(by Definition 3.18 (\textit{load}))}
\end{aligned}
$$

□

In practice, CProver tools make use of an intrinsic symbol to abstract away the memory of the program [28]. For example, Example 3.16 contains an approximation of this transformation.

Example 3.16. C with dynamic memory code and its GOTO counterpart.

In the GOTO comments, "*" denotes a nondeterministic value assigned by the front-end during allocation (not the C dereference operator). The symbols *mem* and *mem_obj* are intrinsic memory symbols introduced by the front-end to model the heap.

```
1 int *a = malloc(4);//ASSIGN (mem_obj, *)
2                    //ASSIGN (a, *)
3 *a = 42;           //ASSIGN (mem, WITH(mem, a, 42))
4 assert(*a == 42);  //ASSERT (SELECT(mem, a) = 42)
```

More intrinsic symbols can be added to track information about the allocated memory, such as length and access. These extra symbols can be added to automatically ensure safety properties (according to the Common Weakness Enumeration framework [96]) such as: out-of-bounds access, after-free use or double-free. Concretely, the front-end maintains auxiliary symbols that record the allocation size and validity status of each memory object. Before each memory access, the front-end instruments an assertion checking that the offset lies within the allocated bounds and that the object has not been freed, following the same UB instrumentation approach described in Section 3.7.

However, when considering path merging, the problem becomes more complicated. As a pointer might be pointing to multiple places depending on path conditions. Therefore, the CProver tools rely on a value-set analysis [26] to compute the possible values of pointers. Historically, this was performed as a pre-processing of the GOTO language, but later it was moved to the symbolic execution.

3.5.2 Concurrency

There is a divergence in the CProver tools on how to deal with multithreaded programs in both the language and symbolic execution levels. Dealing with concurrency involves a way to describing threads from the higher-level language up to how to explore and encode all possible interleavings.

To start, CSeq [23] has an approach that consists in sequentializing the original C source-program into a non-deterministic program that contains all the interleavings. This approach essentially removes the need of the GOTO framework to support the concurrency directly.

This approach however limits the reasoning that we can do over concurrent programs.

CBMC proposed new statements specifically to deal with threads manipulation. Additionally, more statements were added to ensure atomicity of certain operations. The following statements are used:

Begin Thread: Creates a new thread and sets its entry point.

End Thread: Signals the end of the instructions that belong to the thread.

Begin Atomic: Starts an atomic block, no other interleavings can affect the next block of code until it terminates.

End Atomic: End the atomic block

For the symbolic execution of such statements the initial approach is to fully encode all possible interleaving into one unique trace. Another approach is to encode the formulas limiting how many interleavings are allowed [97]. Both approaches require the original proposed symbolic execution algorithm to be updated.

Furthermore, compared to CSeq encoding the formulas during symbolic execution allows optimizations to prune interleavings that results in the same states (for example, partial order reduction [98]).

3.5.3 Other extensions

Several language features are commonly expected from programming languages. Most of them were implemented by previous works [19], [83], [84], [99], [100].

Here we will give a brief summary on them:

Non-integer numbers: Dealing with other numerical types such as fixed or floating-point numbers requires new instructions to encode such expressions. In Gadelha's thesis [99], the thesis proposes an encoding for such numerical types using bitvectors.

Functions: The current version of GOTO does not have support for functions (as in procedures). This is a pretty standard CProver construct which allows the definition of functions, call and return instructions. For this work, we did not add such construct directly to avoid unnecessary complexity of the formalization.

Exception handling: For exception handling, there are two main approaches: adding new instructions or by mapping the exceptions into *if-then-else* chains (as in JBMC [71] and ESBMC-Jimple [35]).

Nondeterminism: In practice, software verifiers rely on explicit nondeterministic calls [15] to model unknown or arbitrary input values, allowing the verifier to reason about all possible concrete executions simultaneously. In the current framework, this can be supported by introducing two symbols during symbolic execution: a counter (e.g. s_i) and a generator (e.g. s_{nd}). The counter is incremented each time a nondeterministic value v is requested: $v \mapsto s_{nd}[s_i + +]$

3.6 Example: Verification of programs

This section illustrates the end-to-end flow: Source code, GOTO program, Trace generation, and Verdict. It makes concrete: (i) how source-level Undefined Behavior (UB) is encoded as assertions or choices in GOTO; (ii) how the trace is SSA; and (iii) what "sound up to bound k" means in practice for the generated traces.

For the sake of clarity, we will show the byproducts of each step in a more clean way (e.g., using $\boxed{+}$ instead of $\boxed{Add}$). This full example is shown in the exact form at Section A.2.5, including other cases (such as loops and memory management).

3.6.1 Source program

We will use one of the motivational examples for SMT-based verification work [28], the program is shown in Example 3.19.

Example 3.17. C program with error (adapted from [28]).

```
1 int main() {
2   int a[2], i, x;
3   if (x == 0)
4     a[i] = 0;
5   else
6     a[i+2] = 1;
7   assert(a[i+1]>=1);
```

```
8 }
```

In the program, the uninitialized variables (line 2) are Undefined Behavior and treated as nondeteministic. The assertion (line 7) can fail if $\boxed{a \mapsto \langle 0, 0\rangle, x \mapsto 0, i \mapsto 0}$. Other UBs, such as out of bounds when $\boxed{i \mapsto 20}$ can be encoded throught explicit assertions in the conversion.

3.6.2 GOTO program

Here is a translation from the program of Example 3.19. Note that this was done by hand and each CProver tool has its own translation algorithm in place.

Example 3.18. GOTO program with error (adapted from [28]).

```
1 IFTHENGOTO (Not x) (4)
2 ASSIGN a (With a i 0)
3 IFTHENGOTO (True) (5)
4 ASSIGN a (With a (i+2) 1)
5 ASSERT (Select a (i+1) >= 1)
```

In practice, the CProver front-ends produce the instrumented version directly; the listing above is shown without UB assertions only to separate the translation concerns for clarity. UB instrumentation works by inserting an assertion immediately before each potentially unsafe operation, checking the precondition that must hold for the operation to be well-defined. For example, each array access is preceded by a bounds check asserting that the index lies within the declared size of the array:

```
1 IFTHENGOTO (Not x) (4)
2 ASSERT (1 >= i)
3 ASSIGN a (With a i 0)
4 IFTHENGOTO (True) (5)
5 ASSERT (1 >= i)
6 ASSIGN a (With a (i+2) 1)
7 ASSERT (1 >= i)
8 ASSERT (Select a (i+1) >= 1)
```

3.6.3 Bounded symbolic execution and the trace

This program has no loops (see Section A.2.5 for examples with loops), we will just pick the arbitrary $\boxed{k \mapsto 1}$.

Example 3.19. GOTO trace with error (adapted from [28]).

```
1 ASSIGN a1 (With a0 i0 0)
2 ASSIGN a2 (With a0 (i0+2) 1)
3 ASSIGN a3 (ITE (Not x) a1 a2)
4 ASSERT (Select a3 (i0+1) >= 1)
```

3.7 Abstracting the CProver tools

As discussed at the start of this chapter, our framework aims to generalize the core behavior of two major tools in the CProver ecosystem: CBMC and ESBMC. While both tools share many verification principles, they differ in implementation strategies, language support, and analysis techniques.

- **Encoding Backend:** The most significant distinction lies in the underlying decision procedures. ESBMC is built around SMT-based encodings, while CBMC focuses on SAT-based encodings. This distinction allows each tool to leverage different optimizations such as incremental solving, parallelism, and specialized theories.
- **Concurrency Support:** CBMC introduces explicit GOTO-level concurrency instructions (Section 3.5.2), such as `BEGIN_THREAD` and `END_THREAD`. ESBMC, in contrast, interprets thread behavior from explicit calls to `pthreads` and introduces concurrency semantics during symbolic execution. This limits ESBMC's compatibility with alternative threading APIs (e.g., `WinThreads`) but enables techniques such as context-bounded analysis. Meanwhile, CBMC encodes all interleavings in a single symbolic execution trace.

These differences manifest at multiple abstraction layers. For example, CBMC encodes concurrency via explicit GOTO statements, while ESBMC introduces it dynamically during symbolic execution. To unify both under our framework, we treat these differences as language extensions (see Section 3.5). This allows us to model each tool's additions either as

extensions to the instruction set (G_E, i.e., new syntactic constructs) or to the semantics (G_S, i.e., changes in how existing instructions are interpreted).

Another point of divergence lies in the internal representation of expressions and statements, known as Irep (intermediate representation). Example 5.1 shows an example of an Irep encoding the constant `42`. While both tools use this format, their semantics differ in two key ways:

1. Irep expressions may recursively contain statements (e.g., `side effects`).
2. Expressions in both tools are strongly typed.

The first difference does not impact our abstraction significantly, as both tools simulate side effects using function pointers and intrinsics. However, the second introduces limitations: both tools assume a fixed type per symbol, and changing a symbol's type at runtime leads to incorrect behavior. For instance, ESBMC's support for *Flexible Array Members* remains problematic due to this rigidity.[2]

Finally, CBMC supports mathematical constructs as C extensions, including quantified expressions (e.g., `forall`, `exists`) [83]. While expressive, these constructs require non-standard C syntax, which breaks compatibility with regular C compilers. Recently, as of February 2025, ESBMC introduced quantifier support using intrinsic functions [84], allowing users to retain compatibility with standard C code. In practice, users can conditionally include quantifiers using macros such as `#ifdef` to maintain portability across tools.

3.8 Summary

In this chapter, we introduced a formal verification framework based on the *GOTO language*, an intermediate representation used across tools in the *CProver* family. Our goal was to provide a unified formal model that captures the semantics of bounded model checking (BMC) through symbolic execution.

We began by defining the core GOTO language, in which programs are executed symbolically to generate *traces*. These traces are then evaluated against *environments* to determine whether any assertion is violated. To support practical verification scenarios, we outlined

[2] `https://github.com/esbmc/esbmc/pull/616`

how the core language can be extended with features such as dynamic memory access, concurrency primitives, and richer numerical domains; formal semantics for these extensions remain future work.

In addition, we showed how this framework abstracts the implementation differences across real-world tools such as *CBMC* and *ESBMC*, highlighting commonalities as well as tool-specific extensions. By modeling these differences as language or semantic extensions, we retain flexibility without compromising the generality of our framework.

In the following chapters, we build upon this formal model to explore applications. Chapter 4 introduces compiler-inspired optimizations to improve the effectiveness of proofs. Moreover, the Chapter demonstrates how our framework can be applied to real-world languages and tools to verify practical systems, including Rust programs.

Chapter 4

GOTO Transformations

Up to this point, we have described the verification framework primarily through the lens of the GOTO language, as detailed in Chapter 3. We now turn to the application of transformations – defined as modifications to the program – within this framework. A transformation is any method of manipulating either the code ($\mathbb{G}$) or the trace ($\mathbb{T}$), with the main goal of optimizing certain aspects of the analysis.[1] This chapter explains how these transformations fit into the framework and how they improve both verification methodology and performance.

Transformations can be applied at different stages of the verification flow within the GOTO program (see Figure 3.1). For example, consider constant folding [101]: it can be applied directly to the GOTO program, during symbolic execution, or within the trace. At the GOTO program level, constant folding reduces the number of instructions. During symbolic execution, it can simplify traces by eliminating unreachable paths. Finally, at the trace level, it reduces the effort required for the decision procedure. Example 4.1 shows an instance of constant folding applied to GOTO.

Example 4.1. Constant folding in a GOTO trace. The example shows the original program (left) and its optimized form (right). The property trivially holds because the nonzero assertion is always true.

Original

```
1 ASSIGN a0 (1+1)
2 ASSIGN a1 (a0 + 2)
3 ASSERT (a1 > 0)
```

Optimized

```
1 ASSIGN a0 2
2 ASSIGN a1 3
3 ASSERT 1
```

Constant folding is an optimization-oriented transformation. In Chapter 5, we show that such optimizations can affect the verification time. In addition, some transformations may

[1] Transformations can also enable new verification approaches. For example, see k-induction in ESBMC [29].

change the bound required for analysis, speeding up convergence.

Before presenting the algorithms, we introduce several auxiliary functions used to simplify their description. In all algorithms, we demonstrate how to transform a trace ($\mathbb{T}$) or a program ($\mathbb{G}$). We assume the existence of a helper method such as *skip* (meaning the instruction is ignored), as well as methods to determine the size and type of instructions (similar to *IsAssume* from Chapter 3). Finally, we assume the helper method $\boxed{\mathit{getVars} : \mathbb{T}_E \rightarrow 2^{\mathbb{S}}}$, which maps a trace expression to the set of symbols it contains.

This chapter focuses on transformations within the GOTO framework that are motivated by practical analysis of the C programming language. We discuss the rationale and methodology of these transformations, followed by an experimental evaluation using the CProver tool ESBMC in Chapter 5. The chapter is structured into sections that introduce and evaluate different optimizations: Sections 4.1 and 4.2.1 cover basic optimizations with simple algorithms. In contrast, Section 4.2.3 explores Abstract Interpretation (AI) for GOTO, which provides the foundation for more advanced optimizations discussed later. The next section begins with loop-unfolding optimizations for GOTO programs.

4.1 GOTO unwind

In this section, we introduce the concept of "GOTO unwind", a transformation designed to optimize static monotonic loops. Through collaboration with industrial partners, we observed that verification tools often encounter difficulties with large loops, even when they do not increase the program complexity. By flattening these loops, we can improve the performance of verification. Example 4.2 shows a motivating instance: a C program with an initialization loop and its flattened form.

Example 4.2. Motivating example. The example contains the original program (left) and its optimized form (right). In the original version, the bound 3 is required to find the bug. In the optimized form, bound one is enough.

Original

```
void foo() {
  int arr[3];
  for(int i = 0; i < 3; i++)
    arr[i] = i;
  assert(arr[0] >= arr[1]);
}
```

Optimized

```
void foo() {
  int arr[3];
  int i = 0;
  arr[i] = i; i++;
  arr[i] = i; i++;
  arr[i] = i; i++;
  assert(arr[0] >= arr[1]);
}
```

For our industrial partners, proving each bound was a time-consuming process—especially when the required bound to reach the assertions was increased.[2] Applying this transformation allowed proofs to be discovered much earlier. To generalize this improvement, we defined a transformation called *GOTO unwind*, which detects bounded loops and unfolds them.

The next example illustrates this transformation directly in the GOTO language.

Example 4.3. Example of unwinding a GOTO program. The example contains the original program (left) and its optimized form (right). The loop $\langle 4, 6 \rangle$ will only be executed two times (with $i = 0 \vee i = 1$).

Original

```
1 ASSIGN(sum, 0)
2 ASSIGN(i, 0)
3 IFTHENGOTO(!(i<2), 7)
4 ASSIGN(sum, sum + i)
5 ASSIGN(i, i + 1)
6 IFTHENGOTO(1, 3)
7 ASSERT(sum = 45)
```

Optimized

```
1 ASSIGN(sum, 0)
2 ASSIGN(i, 0)
3 ASSIGN(sum, sum + i)
4 ASSIGN(i, i + 1)
5 ASSIGN(sum, sum + i)
6 ASSIGN(i, i + 1)
7 ASSERT(sum = 45)
```

The unwind algorithm works by applying a template-matching approach. Given a loop, if it matches, the template extracts the bound. The loop is then flattened according to the identi-

[2] Note that all assertions that are beyond the current bound hold vacuously.

fied number of iterations. The key idea is to lower the bound (k_0) needed to reach the same verdict without altering the overall bound (k_1) of the program. In the next subsection, we describe how these templates are defined.

4.1.1 Bounded loop template

The first step is to identify which loops can be replaced and by how many iterations. For each loop $\langle l_s, l_e \rangle$ (as defined previously in Section 3.3.2), we extract the following elements: the symbol initialization, the loop condition, and the increment or decrement. Specifically:

- **Loop condition:** The condition must be a relational check (e.g., $x < 10$) on a single variable. Therefore, the instruction l_s must be a conditional jump. The symbol is then marked as the *induction variable*. The expected final value of the loop is denoted i_f, along with the relation operator.
- **Symbol initialization:** The variable should be initialized to a constant value just before the loop starts. In other words, instruction $l_s - 1$ must assign a constant to the variable. The variable must match the *induction variable*, and its initial value is denoted i_0.
- **Increment/decrement:** Inside the loop body, only one instruction is allowed to increase or decrease the variable. This must appear just before the backward jump of the loop. Thus, instruction $l_e - 1$ must assign a new value to the *induction variable*. The operator is then marked as the *step*.

Algorithm 4 Loop template

```
1: procedure BOUNDED-LOOP-TEMPLATE(⟨l_s, l_e⟩)
2:     if ¬IsRelation(l_s.expr) ∨ #getVars(l_s.lhs) ≠ 1 ∨ ¬IsConstant(l_s.rhs) then
3:         return ∅;
4:     induction-var ← getVars(l_s.lhs)
5:     k ← getLoopBounds(⟨l_s, l_e⟩)
6:     unfoldLoop(k)
```

The loop template allows us to extract the key characteristics of a loop, specifically, whether it is monotonic, whether it increments or decrements, and what its termination condition is. In the next section, we show how to use these features to unfold loops.

4.1.2 Unfolding loops

Once the bounds for a loop are computed, we systematically unfold it. A special case arises when a loop contains nested loops: in such cases, the inner loops are unfolded before the outer loops. We use Table 4.1 to calculate the number of iterations required.

Table 4.1. Detection of loop unfolds.

Loop type	**Step**	**Relation**	**Unwinds**
$i_0 < i_f$	$+$	$\leq$	$i_f - i_0 + 1$
$i_0 < i_f$	$+$	$<$	$i_f - i_0$
$i_0 > i_f$	$-$	$\geq$	$i_0 - i_f + 1$
$i_0 > i_f$	$-$	$>$	$i_0 - i_f$

The GOTO unwind transformation thus simplifies loops by unfolding them, lowering the bounds required to reach assertions while preserving correctness. This makes proofs easier to find in practice when using CProver tools.

4.2 Slicer

Another class of transformations involves eliminating statements that do not influence the falsification of a program or its execution pathways. Removing such statements can improve performance and reduce the bounds needed to establish a proof.

This category of transformations is known as *slicing* [100]. In this section, we describe variants of slicing for both traces and GOTO programs.

4.2.1 Trace Slicing

The slicer integrated into ESBMC [100] – originally introduced in the SyMP plugin [61] – is designed to eliminate statements from a trace that do not influence its safety outcome.

Since the decision procedure depends on *Asserts* and *Assumptions*, the first step is to determine which symbols may influence these guards. Once dependencies are identified, the slicer removes assignments that do not affect any guard. Importantly, once a symbol is identified as a dependency, every operation that assigns to it becomes a dependency. Example 4.4 illustrates this process.

Example 4.4. Example of slicing a GOTO program. The example contains the original program (left) and its optimized form (right). The assertion does not depend on the value of b_0; therefore, assignments over it can be removed.

Original

```
1 ASSIGN(a0, 2)
2 ASSIGN(b0, a0+1)
3 ASSERT(a0)
```

Optimized

```
1 ASSIGN(a0, 2)
2 //ASSIGN(b0, a0+1)
3 ASSERT(a0)
```

Algorithm 5 describes the trace slicer, which works by traversing the trace in reverse order. During traversal, it collects all variables used in assertions and assumptions, and it keeps only assignments targeting those variables. This results in a shorter, more efficient formula for the decision procedure.

Algorithm 5 Trace slicer

```
1:  procedure TRACE-SLICER(trace)
2:      pc ← trace.size() - 1
3:      variables ← ∅
4:      while pc ≥ 0 do
5:          if IsAssume(trace[pc]) ∨ IsAssert(trace[pc]) then
6:              variables ← variables ∪ getVars(trace[pc])
7:          else if trace[pc].target() ∈ variables then
8:              variables ← variables ∪ getVars(trace[pc])
9:          else
10:             trace[pc].skip()
11:         pc ← pc - 1
```

ESBMC also supports a second variant of the slicer that permits slicing assumptions. Although this may risk producing incorrect counterexamples, it can still be beneficial for validating safety when assertions are satisfiable and the assumptions do not rely on program-specific attributes. For example, consider a trace containing `ASSUME(x > 0)` followed by `ASSERT(cond)`: if `cond` is always true, the assumption is irrelevant to the safety proof and can be sliced away, simplifying the formula sent to the solver. As we will see next, a similar principle extends naturally to whole-program slicing.

4.2.2 GOTO Slicing

The GOTO slicer is conceptually similar to the trace slicer but operates directly on the GOTO program. Like the trace slicer, it aims to eliminate instructions that do not affect the prop-

erty being verified. However, since GOTO programs include *loops* and complex path conditions, the simple reverse-traversal algorithm used for traces is insufficient. Instead, GOTO slicing relies on *Abstract Interpretation* to compute symbol dependencies across the program.

The slicer can be adapted depending on the verification strategy. For example, incremental BMC [21] makes use of two proof modes: the *base case* and the *forward condition*. In the base case, the slicer behaves similarly to the trace slicer, keeping assumptions and assignments relevant to the property. In the forward condition, loop termination does not need to be guaranteed if the loop does not affect the property[3].

This transformation fundamentally alters the verification strategy by removing irrelevant program parts, thereby improving both falsification and correctness proofs.

Example 4.5. Example of slicing a forward-condition GOTO program. The example contains the original program (left) and its optimized form (right). The unbounded loop $\langle 2, 2 \rangle$ is removed.

Original

```
1 IFTHENGOTO(!1, 3)
2 IFTHENGOTO(1,1)
3 ASSERT(1)
```

Optimized

```
1 // IFTHENGOTO(!1, 3)
2 // IFTHENGOTO(1,1)
3 ASSERT(1)
```

The main objective is to remove loops that do not contribute to reaching any assertion. This is implemented at the GOTO program level, and thus applies only to explicit user-defined assertions.

As with trace slicing, the process begins by computing a dependency tree for the program. This tree captures all symbols on which a given symbol depends. In the next sections, we will introduce Abstract Interpretation and an Abstract Domain for computing this dependency tree.

[3]More precisely, the unwinding assertion for the loop is removed: since the loop body has no influence on the property (established by the dependency analysis), dropping the termination requirement does not affect the verification outcome for that property.

Algorithm 6 GOTO slicer

```
1: procedure PROGRAM-SLICER(program)
2:     dependencies ← computeDependencies(program)
3:     pc ← program.size() - 1
4:     while pc ≥ 0 do
5:         if IsAssume(program[pc]) ∨ IsAssert(program[pc]) ∨ IsIfThenGoto(trace[pc]
   then
6:             localDeps ← localDeps ∪ ∀x ∈ getVars(program[pc]) . dependencies[x]
7:         else if IsAssignment(program[pc]) ∧ program[pc].lhs ∉ localDeps then
8:             program[pc].skip()
9:         pc ← pc - 1
```

4.2.3 Abstract Interpretation

Abstract Interpretation—introduced by Cousot and Cousot [102]—is a static analysis framework for computing properties of systems that hold across all executions. For example (adapted from [103]), the expression $\boxed{a \times a \times b \times b}$ always evaluates to a non-negative number regardless of the values of $\boxed{a \in \mathbb{Z}}$ and $\boxed{b \in \mathbb{Z}}$.

Abstract Interpretation models these properties using lattices (or semi-lattices [14]). A lattice is a partial order with an infimum and a supremum [104], while a semi-lattice requires only one. For instance, consider the "rule of signs" (RoS): we define an abstract domain $\{\bot, (+), (-), (\pm)\}$ [102], where $\bot$ (the empty set) represents unreachable operations and serves as the infimum, while $(\pm)$ is the supremum. Executing $\boxed{-15 \times 17}$ abstractly yields $(-) \times (+) \Rightarrow (-)$, without computing the concrete product.

Some of the optimizations in this thesis are expressed in terms of Abstract Interpretation. For our purposes, we require an abstract domain α and three key operations over abstract states S_α:

- $\boxed{\mathit{Init}_\alpha : S_\alpha}$ — initializes an abstract state. For the RoS, each program symbol initially maps to $\bot$ (the infimum).
- $\boxed{\mathit{Transform}_\alpha : S_\alpha \to G_I \to S_\alpha}$ — transforms an abstract state based on a GOTO instruction. For example, squaring a symbol forces its domain to $(+)$.
- $\boxed{\mathit{Join}_\alpha : S_\alpha \to S_\alpha \to S_\alpha}$ — merges two abstract states by computing their least upper bound [4].

[4] This is common in many compiler optimizations based on lattices [101]

Algorithm 7 Computation of the abstract domain

```
1:  procedure computeAbs(g, α)                    ▷ α provides Init_α, Transform_α, Join_α
2:      stmts ← to-list(g); labels ← label-indexes(s)
3:      work-list ← {1}; P_α ← ∅; domain-stmt ← ∅      ▷ maps each pc to its abstract state
4:      while ¬ work-list.empty() do
5:          pc ← work-list.pop(); stmt ← stmts[pc]
6:          state ← ¬P_α.contains(pc) ? Init_α : domain-stmt[pc]
7:          next ← {pc + 1}
8:          if IsIfThenGoto(stmt) then
9:              next.add(labels[stmt.label])
10:         while ¬ next.empty() do
11:             to ← next.pop()
12:             new-state ← Transform_α(state, to)
13:             old-state ← P_α[to]
14:             if old-state ≠ new-state then
15:                 domain-stmt[to] ← Join_α(old-state, new-state)
16:                 work-list.add(to)
        return P_α
```

These functions are used in a worklist algorithm to compute static properties of GOTO programs. Algorithm 7 shows how an abstract interpreter computes properties for each reachable instruction; instructions not visited by the worklist are unreachable—their abstract state remains $\bot$, and they can be safely removed or replaced by *SKIP*. Although the presentation here focuses on GOTO programs, Abstract Interpretation could also be applied to traces as a lightweight pre-solver simplification. Since traces are loop-free and in SSA form, no fixed-point iteration would be needed; a single forward pass could propagate intervals or detect trivially safe assertions, reducing the formula sent to the solver. We leave this direction as future work.

Note that `next` is treated as a set: if the jump target of an $\boxed{IfThenGoto}$ equals $pc + 1$, the successor appears only once. Even without this deduplication, $Join_\alpha$ is idempotent, so processing the same successor twice would yield the same result.

4.2.4 Computing dependencies through Abstract Interpretation

To apply slicing, we must compute dependencies for each program variable. This allows us to identify assignments that can be safely removed. Because dependencies span the entire program, a whole-program analysis is required. We therefore use Abstract Interpretation to construct an abstract domain that captures variable dependencies:

Definition 4.1. Abstract Domain for symbol dependency

- **Domain:** consists of the power set of pairs $\langle s, d \rangle$, where $s \in \mathbb{S}$ and $d \subseteq 2^{\mathbb{S}}$. For each symbol in the program, a dependency set d is associated
- **Init:** the initial state is that every symbol in the program has no dependencies: $\forall s \in \mathbb{S}. \langle s, \emptyset \rangle$.
- **Transform:** applied on *Assign* s e instructions. It adds symbols from e (and their dependencies) as a dependency of s.
- **Join:** for each symbol, create a union of the dependencies from before the transformation to after it.

In addition to assignments, other instructions may also be sliced, depending on the verification strategy employed. In ESBMC's incremental verification, the base case and the forward condition are checked independently; this allows two specialised slicers. The base-case slicer focuses on reaching assertions, so all elements that influence path conditions are treated as dependencies. The forward-case slicer focuses on unwinding assertions, which allows it to remove loops that have no impact on the property. The common dependency rules are:

1. Assignments to assertion dependencies are kept.
2. Loop conditions are added to the dependency set.
3. Variables in assumptions are treated as dependencies.

Base-case slicer The base-case slicer operates similarly to the trace slicer (Section 4.2.1), but additionally preserves instructions affecting path conditions, i.e. *IfThenGoto*, *Assert*, and *Assume*.

Forward-case slicer In the forward case (in ESBMC), the goal is to prove that no additional execution paths exist at higher bounds. This involves introducing assertions on loop conditions to ensure loops terminate. However, if a non-terminating loop has no impact on an assertion, it can be eliminated. Specifically, if all operations within a loop are removed, the loop can be removed without affecting property validation. Note that this loop removal applies only to the forward condition check; the base-case slicer preserves loops. In practice, the slicer operates in two phases: first, Algorithm 6 removes irrelevant assignments;

then, a cleanup pass removes loops whose body has been entirely sliced away (all instructions replaced by SKIP). Example 4.5 illustrates the combined effect. The soundness of this removal follows from the dependency analysis: if no symbol assigned inside the loop appears in the dependency set of any assertion, the loop cannot affect the truth value of any assertion, so removing it preserves the verification outcome.

4.3 Interval Analysis

Interval analysis[5] determines the minimum and maximum values for all variables in a program [14] such that $\forall x \in V.\, x \in [\underline{x}, \bar{x}]$, where V is the set of program variables. The methodology was introduced for scientific computing by Ramon E. Moore [105], [106] to establish definitive bounds for rounding errors in floating-point computations [107].

For program analysis, interval analysis can be used to infer properties about the program. Figure 4.1 shows a program annotated with intervals at each statement. The program asserts $x > 50$. From the computed intervals, we have $x \in [100, 100]$ at the assertion, so the assertion is always satisfied. If the computed interval for any statement is empty, that statement is unreachable [14]. The example also shows that an assertion that would need 100 unwinds to be reachable becomes trivial. Even more, if "x" was non-deterministic it would require 2^{31} (for 32-bit signed integers) unwinds to reach the proof.

Intervals can be computed via static [14], dynamic [14], or symbolic [108] methods. Here we focus on *static* computation of intervals using abstract interpretation. Different interval domains offer different precision–cost trade-offs. For example, interval arithmetic [106] and modular arithmetic tailored to bit-vector semantics [109] can increase precision, while widening and domain-specific accelerations expedite fixed-point computation [109]–[112].

4.3.1 Domains

Computing intervals requires an abstract domain. We briefly review two selected choices.

Integer Domain ($\mathbb{I}$)

This is the conventional domain over the integers [14]. Variables range over $(-\infty, +\infty)$, which keeps interval arithmetic simple. Bit-precise machine operations (casts, bitwise ops)

[5] Also called range analysis.

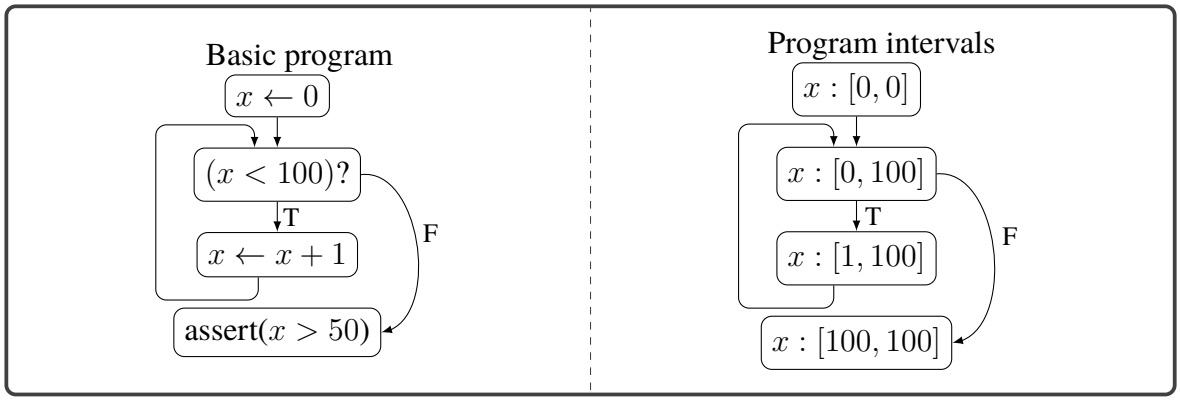


Fig. 4.1. Interval analysis of a program. Left: operations over x. Right: computed intervals at each program point.

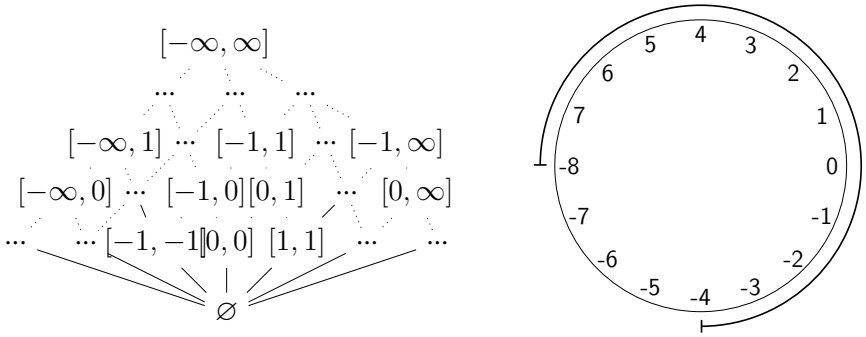
are non-trivial in this domain. An interval for a variable x is written $x \in [l, u]$ with $l \leq x \leq u$. Figure 4.2a sketches its complete lattice.

Wrapped Domain ($\mathbb{W}$)

Following [113], variables range over a *finite*, machine-sized set (e.g., signed 4-bit integers $-8..7$). Intervals are circular (ring-like), matching machine wrap-around; see Figure 4.2b. We write $x \in \langle l, u \rangle$, interpreted as $l \leq x \leq u$ if $l \leq u$, and $x \geq l \vee x \leq u$ otherwise (wrap-around). For example, over signed 4-bit integers, $\langle -4, -8 \rangle$ denotes $\{-4, -3, \ldots, 7, -8\}$.

Definition 4.2. Abstract Domain for interval analysis

- **Domain:** consists of the power set of pairs $\langle s, d \rangle$, where $s \in \mathbb{S}$ and d is the selected interval domain. For each symbol in the program, an interval is associated.
- **Init:** the initial state is that every symbol has no interval: $\forall s \in \mathbb{S}. \langle s, \bot \rangle$.
- **Transform:** applied on every instruction. If it is a guard instruction (i.e., *IfThenGoto*, *Assert*, *Assume*), then contractor-based methods [114] are applied. If non-guard instruction, then compute arithmetics or $\top$, based on the precision level selected.
- **Join:** for each symbol, do a hull (for $\mathbb{I}$, $\mathbb{W}$ has its own union) of the intervals.



(a) Interval lattice. **(b)** Wrapped ring.

Fig. 4.2. Abstract Domains.

```
void foo() {
  int a = * ? 4 : 6; // a: [4,6]
  assert(a + 2 >= 6);
}
```

```
void foo() {
  int a = * ? 4 : 6; // a: [4,6]
  assert(1);
}
```

Fig. 4.3. Optimization example. The example contains the original program (left) and its optimized form (right).

4.3.2 Optimization

With intervals in hand, we can shrink GOTO programs via optimizations. We highlight two optimizations: singleton propagation and dead-code removal.

If an evaluated expression has a *singleton* interval (only one value lies in the interval), we replace the expression by that value. CProver tools can then remove statements that no longer affect the property. Figure 4.3 shows an example where the assertion becomes the singleton $[1, 1]$, i.e., `assert(1)`. Singleton propagation acts as a form of constant propagation whose effectiveness depends on the precision of the underlying interval analysis: a more precise domain yields more singleton intervals, and thus more constants to propagate.

Intervals also expose unreachable instructions: if the abstract state for a statement is $\perp$ (bottom), the statement is unreachable. Converting such instructions to *SKIP* simplifies analysis by pruning irrelevant branches. In our experiments (Section 5.5), this optimization removes entire loops for a subset of benchmarks.

4.3.3 Extracting invariants

CProver relies on decision procedures to prove satisfiability. Once the interval analysis fixed-point has been computed (Algorithm 7), the resulting interval bounds for each variable can

be injected as assumptions into the program. Adding these assumptions usually shrinks the solver's search space, but too many of them can bloat the formula and slow the procedure. To explore this trade-off, we consider three instrumentation levels, ordered by verbosity:

- *Loop instructions.* For each loop, add assumptions (before and after) capturing intervals of all variables appearing in the loop body.
- *Guard instructions (full or local).* For each guard (i.e., *ASSUME*, *ASSERT*, *IFTHEN-GOTO*), add an assumption over either all program variables (full) or only the variables used in the guard (local).
- *All instructions (full or local).* Before every instruction, add an assumption over either all program variables (full) or only the variables used in that instruction (local).

These additional assumptions can also be used in induction-based strategies (e.g., k-Induction [29]) to recover precision in the analysis. In practice, the choice of instrumentation level can be guided by the experimental results in Chapter 5 (Section 5.5). In ESBMC, command-line flags expose a subset of these options, allowing users to select the level appropriate for their verification task. The next section presents our final transformation based on abstract interpretation.

4.4 Global Common Subexpression Elimination

While analyzing a production system, developers at Intel® noticed that specific changes to C code dramatically improved verification time and memory usage, enabling verification of embedded components that were previously out of reach. Initially, these changes were applied manually. This led to a collaboration between our team and Intel domain experts, which surfaced additional optimizations. A particularly effective pattern was introducing an intermediate variable to cache repeated dereferences. Listing 4.4 shows a real-world excerpt illustrating this idea.

The GOTO memory model (see Section 3.5.1) complicates pointer tracking because each dereference must be checked for safety. Consecutive dereferences exacerbate this cost. Caching intermediate dereference results reduces repeated target computations and duplicated safety checks, significantly speeding up verification. This pattern is common in hardware-oriented C and—combined with GCSE—can yield substantial benefits. In the motivating example,

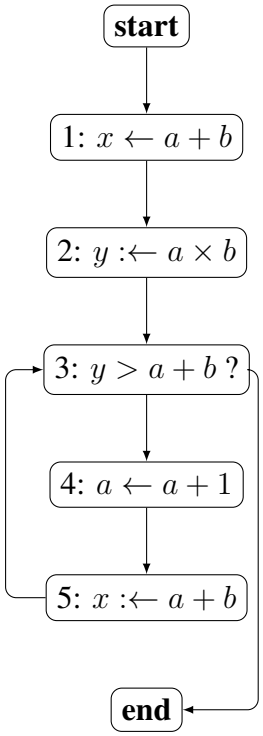
```c
typedef struct {
  uint F;
} Aux;

typedef struct {
  Aux Aux;
  uint Wc;
  uint V;
} RegEntry;

typedef struct {
  RegEntry *Map;
} table;

void write(table *tbl, uint i) {
  tbl->Map[i].Aux.F &= 1;
  tbl->Map[i].Wc++;
  tbl->Map[i].V = 42;
}


int main() {
  RegEntry e[10000];
  table M;
  M.Map = e;
  for(int i = 0; i < 10000; i++)
    write(&M, i);
  return 0;
}
```

```c
typedef struct {
  uint F;
} Aux;

typedef struct {
  Aux Aux;
  uint Wc;
  uint V;
} RegEntry;

typedef struct {
  RegEntry *Map;
} table;

void write(table *tbl, uint i) {
    RegEntry *ptr = tbl->Map[i];
    ptr->Aux.F &= 1;
    ptr->Wc++;
    ptr->V = 42;
}

int main() {
  RegEntry e[10000];
  table M;
  M.Map = e;
  for(int i = 0; i < 10000; i++)
    write(&M, i);
  return 0;
}
```

Fig. 4.4. Motivating example for GCSE. Left: original C. Right: optimized with a cached dereference of `tbl->Map[i]`.

`tbl` is dereferenced multiple times; CProver tools recompute the full target each time, duplicating checks. Introducing a temporary avoids this repetition.

The manual optimization can be automated by *Global Common Subexpression Elimination* (GCSE), introduced by Cocke [115], which replaces repeated computations across basic blocks. We compute *Available Expressions* (AE) [116] via data-flow analysis and then rewrite the Control-Flow Graph (CFG) [101].



Domain: $\{a + b, a \times b, a + 1\}$

Vertex	Gen(v)	Kill(v)
1	$\{a + b\}$	$\emptyset$
2	$\{a \times b\}$	$\emptyset$
3	$\{a + b\}$	$\emptyset$
4	$\emptyset$	$\{a + b, a \times b, a + 1\}$
5	$\{a + b\}$	$\emptyset$

With Gen/Kill we solve the forward data-flow equations (Section 4.4.1). After convergence:

Vertex	In(v)	Out(v)
1	$\emptyset$	$\{a + b\}$
2	$\{a + b\}$	$\{a + b, a \times b\}$
3	$\{a + b\}$	$\{a + b\}$
4	$\{a + b\}$	$\emptyset$
5	$\emptyset$	$\{a + b\}$

Fig. 4.5. Available Expressions via forward data-flow. At node 3, $a + b$ is available on all incoming paths; the test $y > a + b$ can use the previously computed x (i.e., rewrite to $y > x$).

4.4.1 Data-flow Analysis

Data-flow is a framework for global optimizations.[6] Kildall proposed it in 1973 [104], and it uses flow equations to compute lattices throughout the program. The technique can be optimized for specific problems (e.g., Interprocedural Finite Distributive Subset (IFDS) [117], bit-vectors [118]) by utilizing a Control Flow Graph (CFG) [101]. The following elements characterize a data-flow problem:

Domain ($\mathbb{D}$): a set abstracting the *data* that needs to be computed.

Weaker operator ($\sqsubseteq$): a partial order defining the lattice ordering, i.e., when one abstract element is less precise than or equal to another.

Meet operator ($\sqcap$): an operator to combine information from multiple incoming paths at a join point.

Flow functions (f): monotonic[7] and distributive[8] functions that define how the data *flows* through a node. The flow functions are defined as $In : \mathbb{V} \rightarrow \mathbb{D}$ and $Out : \mathbb{V} \rightarrow \mathbb{D}$ where $\mathbb{V}$ are the input CFG nodes.

[6] "Global" as in machine independent. Likewise, "local" refers to machine-dependent.
[7] $\forall x, y, x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$
[8] $\forall x, y. f(x \sqcap y) = f(x) \sqcap f(y)$

Bound (k): a limit for the domain. This would be the maximum (or minimum) if the domain is a semi-lattice.

Initial values ($\mathbb{I}$): the starting values for the analysis.

In practice, most data-flow problems can be defined in terms of sets. Since these sets can be implemented as bit vectors, the operations can also be implemented as bitwise operations. These problems are also known as *Gen/Kill problems*. This approach was introduced in 1973 [118] as a way to ensure an upper bound. Later, in 1976, the term Gen/Kill was used [119] as flow functions are defined by the two auxiliary functions: $\boxed{\mathit{Gen} : \mathbb{V} \to \mathbb{D}}$ and $\boxed{\mathit{Kill} : \mathbb{V} \to \mathbb{D}}$, where $\mathbb{V}$ are the vertexes of a flow graph and $\mathbb{D}$ is an abstract domain. Additionally, Gen/Kill problems also flow in a specific direction. Analysis can be forward, backward, or even a combination of both (e.g., Partial Redundancy Elimination [101]). In Available Expression, the *forward* equation is used:

Let `start` denote the entry node of the CFG and $\texttt{pred}(v)$ the set of predecessor nodes of v in the CFG.

Definition 4.3. Forward data-flow

$$\texttt{In}(v) = \begin{cases} \mathbb{I}, \texttt{v} = \texttt{start} \\ \sqcap_{x \in \texttt{pred}(v)} \texttt{Out}(x) \end{cases}$$

$$\texttt{Out}(v) = \texttt{In}(v) \setminus \texttt{Kill}(v) \cup \texttt{Gen}(v)$$

AE is instantiated as:

Definition 4.4. Available Expressions (data-flow)

- *Domain:* the set of all possible expressions in the program.
- *Gen:* expressions computed by the statement (and are not killed).
- *Kill:* expressions invalidated by the statement, i.e., any expression referencing a symbol that is reassigned.
- *Meet:* Intersection.
- *Bound:* size of the domain
- *Initial values:* $\emptyset$

- *Direction:* Forward.

In Section 4.4, we describe how to use forward Gen/Kill analysis to compute the Available Expressions in the program. Specifically, Figure 4.5 contains an example of how to calculate it.

4.4.2 Computing AE through Abstract Interpretation

When a dedicated data-flow solver is unavailable, we compute AE with Abstract Interpretation (Section 4.2.3) plus pointer information from value-set analysis (VSA). VSA identifies stores that may invalidate previously available expressions via aliasing. For example:

```
int *ptr = &x;   sum = x + 42;   *ptr = 0;   sum = x + 42;
```

The dereference `*ptr = 0;` can kill expressions that reference memory possibly modified by `ptr`. We use the powerset domain $\mathcal{P}(\mathbb{T}_E)$ of expressions but store compact hashes (ESBMC uses an internal SHA-256 over expressions). The *infimum* is $\emptyset$; the *supremum* is the set of all expressions in the function (not needed explicitly—we check growth incrementally). Nontrivial expressions (e.g., $x + 42$) are added recursively; trivial ones (e.g., x) are ignored. Expressions depending on invalidated pointers are removed.

Definition 4.5. Abstract Domain for Available Expressions

- **Domain:** consists of the power set of $\mathbb{T}_E$.
- **Init:** the initial state is that nothing is available.
- **Transform:** on *Assume* e, *Assert* e, *Assign* s e, *IfThenGoto* e n instructions; e is made available while any expression that relies on s is removed.
- **Join:** intersection.

4.4.3 Applying GCSE

GCSE proceeds in two steps for each GOTO function [35] (Figure 4.6):

1. **Identify maximal available subexpressions.** For each statement, walk its expression

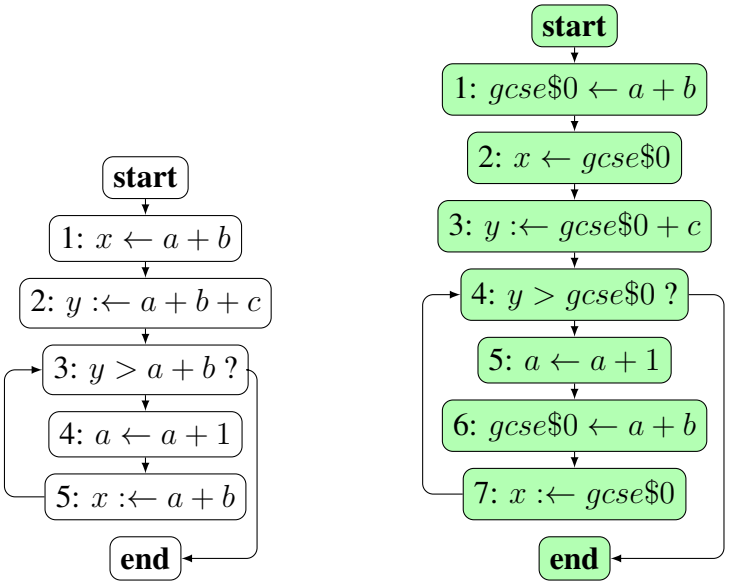


Fig. 4.6. Example GCSE transformation: introducing a temporary for $a + b$ and reinitializing it after it is killed.

tree and match the largest subexpression in AE (falling back to smaller subexpressions).

2. **Introduce temporaries and initialize on demand.** Instrument a fresh temporary for each chosen subexpression. Declarations are added once; initializations (assignments) are inserted at the earliest point where the subexpression becomes available (and reinserted after it is killed).

Construction details. We scan statements in program order. For each RHS, we recursively search for the largest subexpression in AE; if none is available, we descend to subexpressions. Selected subexpressions are assigned to fresh temporaries (declared once, uninitialized). If the LHS is a dereference, the temporary must be (re)initialized at that point. We (re)insert initializations when a subexpression becomes newly available (or was just killed), e.g., after assignments or stores that modify participating variables or aliasing locations (as indicated by VSA).

Short-circuiting. C boolean operators short-circuit left-to-right. Because GCSE hoists common subexpressions into temporaries that are evaluated earlier than the original expression, it may cause the right-hand operand of a short-circuit operator to be computed before the left-hand guard is checked. For a guard like `ptr != NULL && *ptr == 42`, hoisting `*ptr == 42` into a temporary would dereference `ptr` before the null check, which is un-

sound. We therefore restrict GCSE to make only the leftmost operand available in short-circuit contexts.

The next section examines the integration of support for verifying multiple assertions (multi-property verification) within CProver.

4.5 Multi-property verification

During verification, a single trace may contain multiple assertions—either clustered (e.g., inside a loop) or spread across the program. The decision procedure returns *true*, *false*, or *unknown*. A *false* result identifies whichever assertion the solver can prove satisfiable first[9]; the status of other assertions remains undecided. In scenarios like coverage testing [37], it is important to determine which assertions fail and which pass within the bounds.

Our `eval` procedure operates on one trace at a time, which we can exploit to add multi-property support *indirectly*: we split a trace containing many assertions into multiple traces, each focusing on a single assertion (turning the others into assumptions). The resulting flow is:

1. **Clone per assertion.** For each assertion in the trace, create a copy of the trace dedicated to that assertion.
2. **Apply the decision procedure independently.** Each per-assertion trace is solved for *safe*/*unsafe*.
3. **Aggregate results.** Collect outcomes into a single multi-property report.

Turning non-target assertions into assumptions preserves their constraints while enabling stronger *trace slicing* (Section 4.2.1), because each per-assertion trace has fewer dependencies. The following algorithm formalizes the transformation.

Running the decision procedure on the set $\mathcal{T}$ naturally parallelizes across traces. Combined with incremental bounds, this also supports a *fail-fast* strategy when properties are orthogonal.

[9]For example, a complex arithmetic assertion may share a path condition with a trivial `assert(0)`, and the solver will likely find the latter fails more easily.

Algorithm 8 Multi-property transformation (per-assertion splitting)

procedure MULTI-PROPERTY-SPLIT(trace t)
  $A \leftarrow$ indices of `ASSERT` in t
  $\mathcal{T} \leftarrow \emptyset$
  for each $i \in A$ **do**
    $t_i \leftarrow$ deep-copy of t
    for each $j \in A$, $j \neq i$ **do**
      replace $\texttt{ASSERT}_j(\varphi_j)$ in t_i with $\texttt{ASSUME}(\varphi_j)$
    $t_i \leftarrow$ TRACE-SLICER(t_i)
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t_i\}$
  return $\mathcal{T}$

Concurrency note. Although the transformation enables parallelism, practical parallel execution within CProver tools must consider internal shared data structures. Some components are not atomic, which can complicate a fully concurrent implementation. A safe approach is to parallelize at the process level or isolate solver contexts to avoid races.

4.5.1 Coverage testing

Software coverage [120] measures which parts of the program execute under a test or analysis. In our setting, a "part" corresponds to a specific program point (e.g., branch targets, loop headers, function exits) being *reachable* under some bounded execution. In the GOTO program, we instrument *coverage points* by inserting special assertions at selected locations. To detect reachability robustly under BMC, each coverage point emits an assertion that *fails if and only if the point is reached*. A simple encoding is `ASSERT(0)` at the coverage site; if the point is reachable within the bound, the property becomes *false* with a witness; if unreachable, it remains *true* (vacuously). We then apply the multi-property split so each coverage point is checked independently.

Algorithm 9 Coverage instrumentation + multi-property

procedure INSTRUMENT-COVERAGE(program g, locations $P = \{p_1, \ldots, p_m\}$)
  for each $p_k \in P$ **do**
    insert `ASSERT(0)` at p_k ▷ coverage fails iff reached
  $t \leftarrow$ SYMBOLIC-EXECUTE(g) ▷ produce a trace
  $\mathcal{T} \leftarrow$ MULTI-PROPERTY-SPLIT(t) ▷ Alg. 8
  return $\mathcal{T}$

Choosing coverage points. Common choices correspond to standard test coverage criteria [9], [120]: (a) both successors of each conditional branch (*branch coverage*), (b) loop headers and exits (*loop coverage*), (c) function entries/exits (*function coverage*), and (d)

user-annotated points (*statement coverage* at selected locations). This yields meaningful reachability feedback without overwhelming the solver.

Integration with slicing. After instrumentation and splitting, apply the trace slicer (Section 4.2.1) to each per-coverage trace. This removes code irrelevant to reaching that specific coverage point, reducing solver load.

Outcomes and uses. Combining multi-property with coverage enables:

- **Reachability reports:** coverage points that are reachable (within the bound) produce counterexamples; unreachable points are reported as safe.
- **Unreachable code hints:** repeated "safe" results across increasing bounds can suggest dead or guarded-out code.
- **Parallel solving:** each coverage property is independent and can be dispatched concurrently.

In summary, multi-property splitting transforms one complex verification task into a set of independent, slicer-friendly checks. When combined with coverage instrumentation, it provides practical reachability feedback and scalable parallel solving within the bounded context.

4.6 Summary

This chapter presented a suite of program and trace transformations for the GOTO framework that reduce solver effort, lower the effective bound needed to reach properties, and improve overall verification throughput—without changing verification outcomes. We organized the transformations by the stage where they operate (GOTO program, symbolic execution, or trace) and provided concrete algorithms and templates for their application.

We began with lightweight, semantics-preserving optimizations. Constant folding reduces instruction count at the GOTO, symbolic, and trace levels by simplifying expressions early; we illustrated how such simplifications can turn hard constraints into trivial truths. We then introduced *GOTO unwind* (Section 4.1), a loop-flattening transformation that detects statically bounded, monotonic loops via a template and unfolds them according to explicit rules.

This lowers the bound k_0 required to reach assertions while preserving the existence of a larger bound $k_1 \geq k_0$ that yields the same verdict.

Next, we developed slicing in two flavors. *Trace slicing* (Section 4.2.1) walks a trace backward, retaining only assignments that influence guard statements (assertions and assumptions). A stricter variant also slices assumptions when safe to do so for validating satisfiable properties. *GOTO slicing* (Section 4.2.2) generalizes this idea to whole programs—accounting for loops and alternative paths—by computing symbol dependencies with Abstract Interpretation (Section 4.2.3). We described base-case and forward-case modes and formalized the intended preservation of verification outcomes under bounds.

We then instantiated Abstract Interpretation with *interval analysis* (Section 4.3). Using both the classic unbounded integer domain ($\mathbb{I}$) and the machine-accurate wrapped domain ($\mathbb{W}$), we derived optimizations such as *singleton propagation* (a tunable constant propagation) and *dead-code removal* when the abstract state reaches $\bot$. We also showed how to inject the interval bounds computed by the fixed-point analysis as assumptions at configurable granularities (loop-only, guard-only, or all statements; full vs. local variable sets), thereby shrinking the solver's search space while balancing formula size.

Building on data-flow, we implemented *Global Common Subexpression Elimination (GCSE)* (Section 4.4). We used a forward Gen/Kill formulation of *Available Expressions* (AE) and, where a classic solver is unavailable, an Abstract Interpretation alternative augmented with value-set (pointer) analysis to invalidate expressions killed by aliases. The transformation introduces temporaries, initializes them on demand, and respects C short-circuit semantics.

Finally, we enabled *multi-property verification* (Section 4.5) by splitting a multi-assertion trace into per-assertion traces (turning non-target assertions into assumptions) and aggregating the results. This split composes well with slicing and naturally parallelizes. As a companion, we added *coverage instrumentation* (Section 4.5.1) that inserts reachability checks (e.g., `ASSERT(0)` at coverage points), then leverages the multi-property flow to report which program points are reachable within the bound. Together, these features provide fail-fast behavior, enable concurrent solving, and deliver actionable reachability/coverage feedback.

Practical considerations. Several transformations rely on pointer information (value-set analysis) and benefit from widening/precision tuning. Parallel execution of per-trace checks is a straightforward concept, but it must avoid non-atomic shared structures in existing toolchains.

Throughout, we emphasized sound variants that preserve verification outcomes and highlighted optional aggressive modes (e.g., slicing assumptions) that trade counterexample fidelity for improved proof performance in specific scenarios.

Next, Chapter 5 evaluates these transformations—individually and in combination—measuring their impact on runtime, memory, bound progression, and coverage across representative benchmarks. We will also introduce a compatibility layer between CProver tools.

Chapter 5

Impact of GOTO transformations in ESBMC

This chapter presents the experimental evaluation and analysis of the effectiveness of GOTO transformations, as introduced in Chapter 4. The evaluation is conducted through an empirical study in which a CProver tool is executed, both with and without each transformation, across a comprehensive suite of benchmarks. The experimental data is analyzed to assess the practical impact of these transformations.

Three primary objectives guide the analysis. First, we quantify the overall effect of each transformation with respect to *verification outcomes*, *CPU time*, and *memory consumption*. Second, we investigate the influence of individual transformations on distinct components of the CProver tool, thereby highlighting their localized effects (e.g., preprocessing, decision procedure). Third, we identify benchmark classes that exhibit substantial improvements or regressions when a particular transformation is applied, and we provide explanatory insights into the observed behavior.

Beyond evaluating transformations, this chapter also addresses another dimension of our framework. It concerns *compatibility*, where we demonstrate that CBMC's GOTO IR programs can be executed within ESBMC, thereby confirming the interoperability of CProver tools.

5.1 Experimental Setup

To assess the efficacy of GOTO transformations, ESBMC [21] was employed in combination with the SV-COMP benchmarks [15], and the outcomes were analyzed using the `benchexec` tool [121]. This section provides a detailed description of these components and the experimental environment.

5.1.1 ESBMC

Figure 5.1 illustrates the ESBMC verification workflow. The tool can process programs written in a variety of languages, including C, C++, CUDA, CHERI, Java, Solidity, and Python. These programs are translated into the GOTO IR, as discussed in Chapter 3. ESBMC then applies preprocessing steps such as loop simplification, instrumentation, and memory modeling. The symbolic execution engine generates verification conditions, which SMT solvers decide. ESBMC also produces witnesses and generates reports that record verification outcomes, including correctness proofs, counterexamples, and test cases.

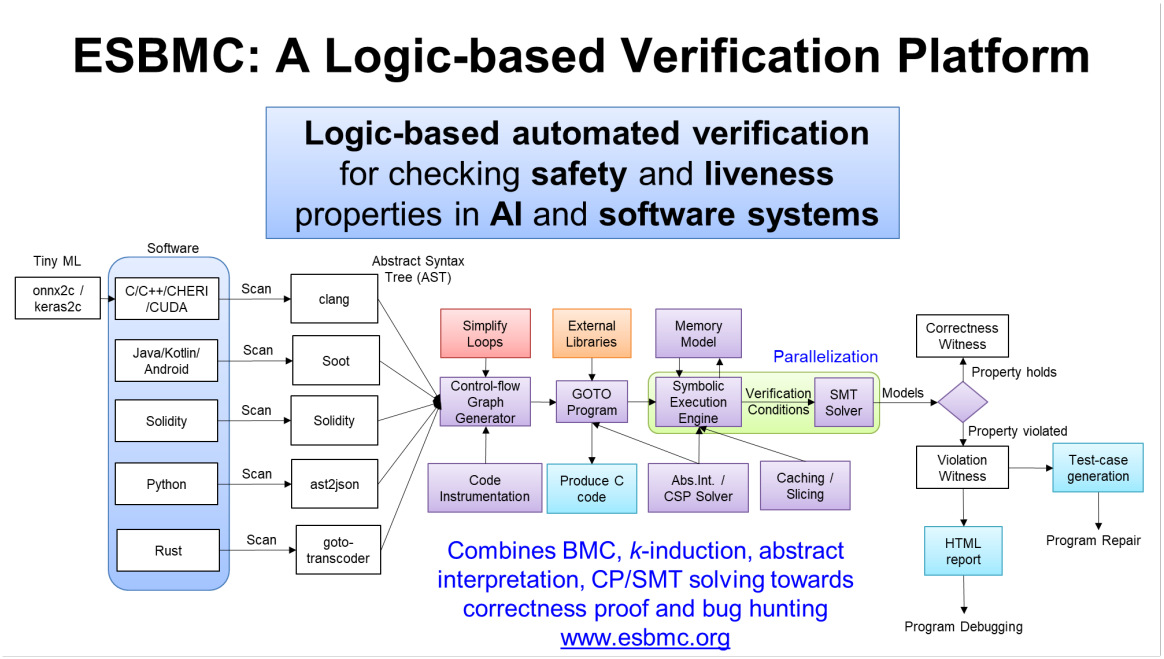


Fig. 5.1. Overview of the ESBMC verification workflow(taken from the tool website [84]).

GOTO generation

After the initial compilation phase, ESBMC produces a GOTO IR program, analogous to the GOTO language described in Chapter 3. This representation includes functions and symbols relevant to ESBMC's intrinsics, such as the memory model and system architecture. The translation is template-based,[1] but the resulting code requires further optimization to improve analysis efficiency. These optimizations correspond to the transformations described in Chapter 4.

Before symbolic execution, the GOTO program must also be instrumented.[2] Instrumentation introduces assertions for safety properties (e.g., bounds checks, overflow checks) and

[1] At this stage, the program is translated structurally, without semantic reasoning. For example, `for` loops are converted into generic loop blocks.

[2] Instrumentation is itself a transformation.

contributes to the proof strategy (see Section 2.2). Preprocessing may also inject invariants, such as those illustrated in Section 4.3. Figure 5.3 illustrates the GOTO generation workflow.

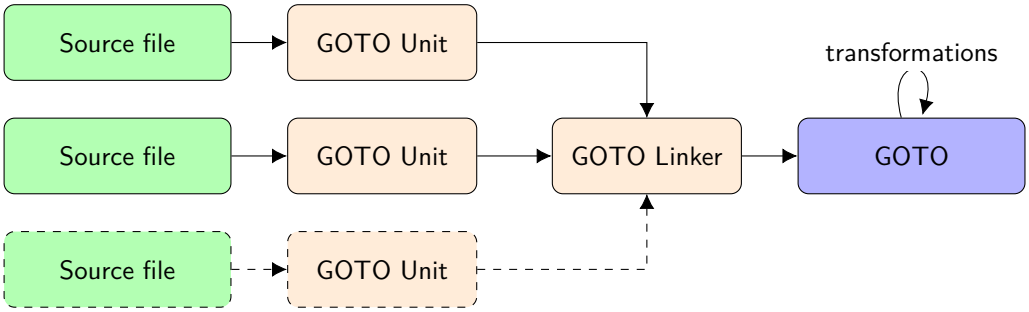


Fig. 5.2. GOTO generation flow in ESBMC. Multiple source files are compiled into units, which are linked into a GOTO program. The program is then preprocessed by transformations.

Symbolic execution

The next stage is bounded symbolic execution (see Chapter 3). This is the most intricate component of ESBMC, due to the combination of *intrinsic behavior*, *pointer analysis*, and *simplifications*. Figure 5.3 depicts the symbolic execution workflow.

Each GOTO instruction is executed symbolically, generating a trace. Some instructions are evaluated concretely (e.g., "$1 + 1$''), while others can be simplified symbolically (e.g., "$a + 0$'' or "$a \times 0$''). Such *simplifications* prune infeasible paths and reduce the complexity of the decision procedure.

The GOTO language also includes intrinsics that enable more precise analysis, similar to compiler extensions such as those in Clang [122]. These intrinsics support the implementation of language standards (e.g., `libc` [123]) and therefore occur frequently in ESBMC's code base.

Pointer manipulation adds further complexity, as discussed in Section 3.5.1. ESBMC employs Value Set Analysis (VSA) [26] to resolve pointer targets. Although VSA can operate on both GOTO programs and traces, only the latter is used in practice, as the former is computationally expensive.[3] An exception is the GCSE optimization (Section 4.4), which requires pointer information.

[3] A possible solution is outlined in the Future Work.

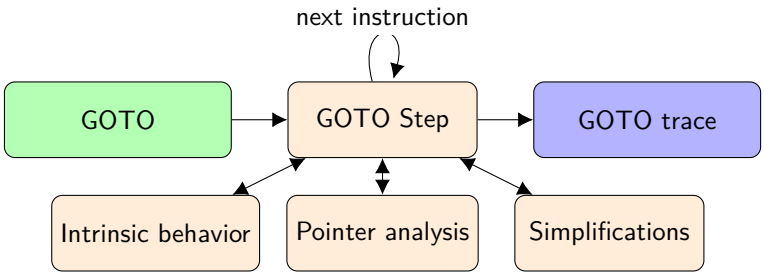


Fig. 5.3. Symbolic Execution flow of ESBMC.

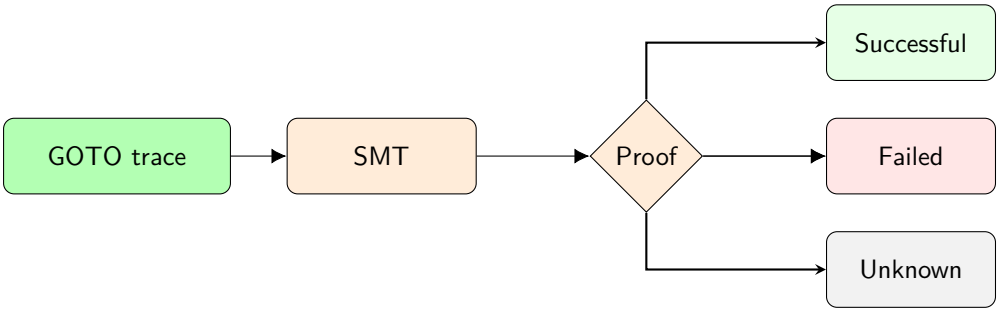


Fig. 5.4. Decision procedure in ESBMC.

Decision procedure

Following symbolic execution, the trace is transformed into an SMT formula [82]. The decision procedure determines whether an environment exists in which an assertion fails, or whether no such environment is possible. The SMT-based encoding is described in earlier work [28], [71], [91].

ESBMC relies on SMT solvers, though not all solvers support the full range of required theories (e.g., Boolector [124] lacks tuple support). ESBMC therefore provides an encoding layer to map unsupported theories into supported ones. At present, the main theory employed is QF_UFBV.

ESBMC also supports incremental solving, interleaving SMT solving with symbolic execution. This strategy is particularly important for concurrency, where ESBMC verifies the feasibility of each thread's path condition before further exploration [97].

Once analysis concludes, ESBMC reports one of three outcomes: `FAILED` (an assertion was violated), `SUCCESSFUL` (no failing assertions were found), or `UNKNOWN` (the analysis was inconclusive). Figure 5.4 summarizes the decision procedure.

Default parameters for ESBMC

To ensure consistency across experiments, we report the default flags used for the reachability analysis in ESBMC. These defaults follow the configuration adopted in the latest editions of SV-COMP. Some flags are selectively disabled or replaced in specific experiments to provide proper baselines (e.g., disabling interval analysis when evaluating interval-based methods, or replacing `--k-induction` with incremental unwinding in slicing experiments).

- **Always enabled**
  - `--force-malloc-success`, `--force-realloc-success`: assume memory (re)allocations always succeed.
  - `--no-div-by-zero-check`, `--no-align-check`, `--no-vla-size-check`, `--no-pointer-check`, `--no-bounds-check`: disable respective runtime safety checks.
  - `--state-hashing`: enable state hashing to reduce memory usage.
  - `--add-symex-value-sets`: enrich symbolic execution with value sets.
  - `--k-step 2`, `--unlimited-k-steps`: configure k-induction with initial step $k = 2$ and no bound on k.
  - `--floatbv`: enable bit-precise floating-point reasoning.
  - `--64`: assume a 64-bit architecture.
  - `--witness-output witness.graphml`: generate correctness witnesses.
  - `--enable-unreachability-intrinsic`: encode unreachability checks.
  - `--error-label ERROR`: specify the verification target label.
- **Optional (enabled or disabled depending on the experiment)**
  - `--interval-analysis`: enable interval abstract domain *(disabled when running interval-specific experiments to provide a baseline).*
  - `--goto-unwind`, `--unlimited-goto-unwind`: enable and unroll GOTO loops without bounds *(disabled when unwinding is handled differently).*

- `--k-induction`, `--max-inductive-step 3`: enable k-induction with up to 3 inductive steps *(disabled in slicing experiments to showcase the incremental approach).*
- `--incremental-bmc`: incrementally increases loop unwinds until a proof is found *(enabled in slicing experiments).*
- `--multi-property`, `--multi-fail-fast`: verify individual verification conditions (VCCs) and stop after the first failure.
- `--no-slice`: disable the trace slicer.
- `--goto-slicer`, `--forward-slicer`: enable the GOTO slicer and forward condition slicer, removing instructions and loops that do not affect assertions *(enabled in slicing experiments).*

Note that, with the exception of the GOTO slicing (and forward condition slicer), all of these flags are available in latest release of ESBMC.

5.1.2 SV-COMP

The International Competition on Software Verification (SV-COMP), established in 2012, serves as a key driver of progress within the software verification community [15]. The competition has collected an extensive set of benchmarks organized into *base categories*, which are combined into *meta categories*. Each meta category targets a specific class of software vulnerabilities, such as memory errors, arithmetic overflows, or assertion failures. Figure 5.5 illustrates the structure of SV-COMP. In particular, the "Overall" meta category aggregates C benchmarks, "Validation-Crafted" focuses on witness validation, and "JavaOverall" includes Java programs.

Benchmarks

In this study, we selected benchmarks from the *ReachSafety* meta category, spanning the SV-COMP'23 to SV-COMP'25 editions. The division by year corresponds to the development timeline of the transformations under study. ReachSafety benchmarks are free of undefined behavior under the C99 standard and are well-suited to the k-induction strategy employed in ESBMC.

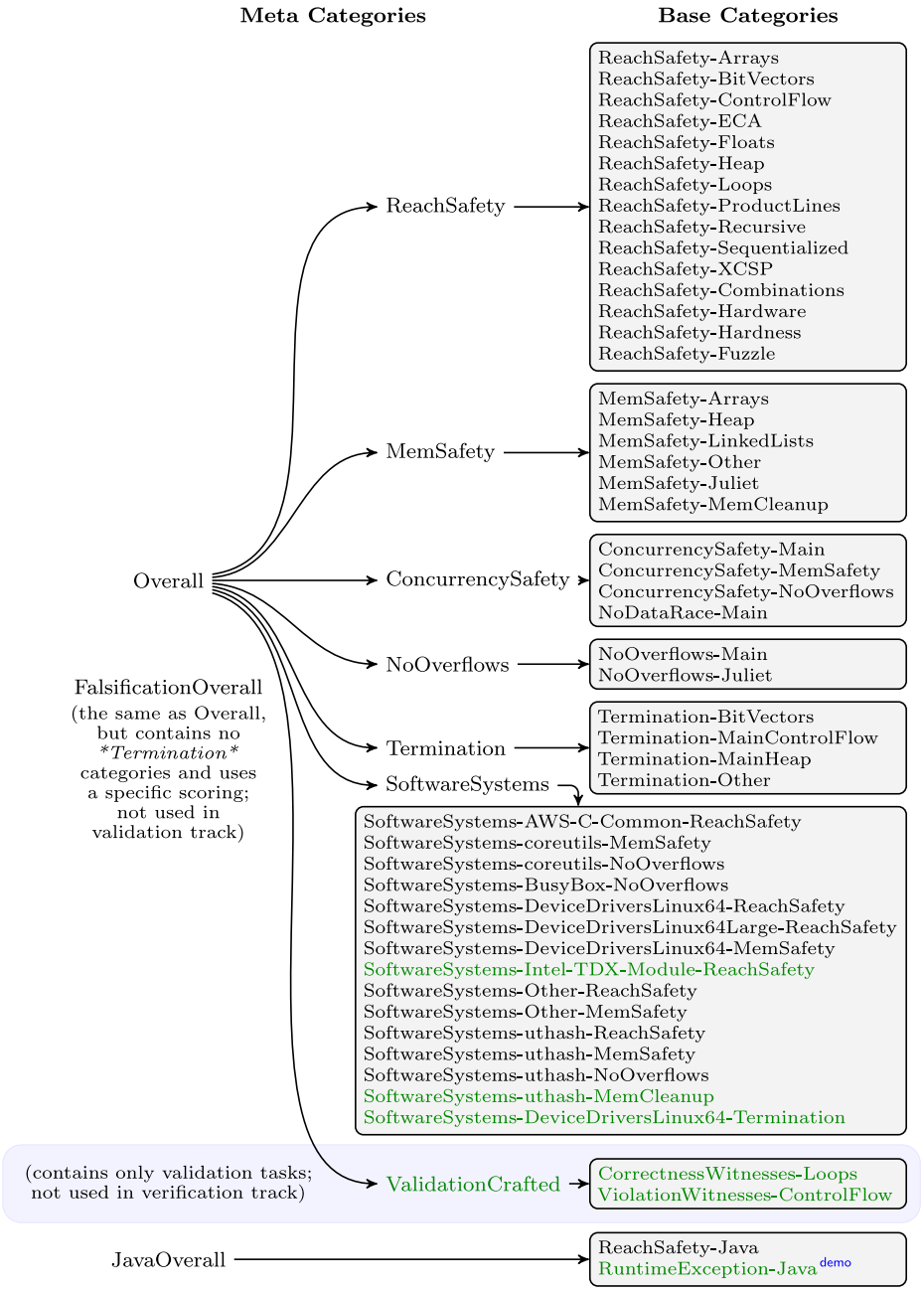


Fig. 5.5. Overview of SV-COMP (taken from [15]).

ReachSafety is further divided into subcategories, each targeting particular program features, such as floating-point arithmetic, Linux drivers, or loops. The distribution of benchmarks across subcategories is highly uneven: some contain thousands of tasks, while others include only a few dozen. SV-COMP addresses this imbalance through a normalization scheme. In our evaluation, we explicitly report the number of benchmarks per subcategory. The scoring system is shown in Table 5.1.

Table 5.1. SV-COMP scoring system without witness validation.

Result	Score
Correctly identifying that a benchmark has no vulnerabilities **(CT)**	+2
Correctly identifying that a benchmark has a vulnerability **(CF)**	+1
Mislabeling a vulnerable program as safe **(IT)**	-32
Mislabeling a safe program as vulnerable **(IF)**	-16
Tool crashes or resource exhaustion **(Unknown)**	0

Benchexec

We conducted our experiments following the methodology of SV-COMP [15]. Benchmarks were executed with the `benchexec` tool [121], using the scoring scheme of Table 5.1, but omitting witness validation. Witness validation is known to be error-prone, introducing both false negatives and false positives [15], and was therefore excluded from this study.

Each run was restricted to 120 seconds[4] of CPU time, one CPU core, and 6 GB of memory. All experiments were executed on a KVM machine running Ubuntu 20.04 (Linux kernel 5.4.0-177-generic). The host system is equipped with a 32-core Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz and 170 GiB of RAM.

Comparing benchexec runs

To compare results across experiments, we used the `table-generator` provided by the `benchexec` project. This tool produces comparative tables of verification results and supports regular expressions for extracting specific metrics. We used these features to obtain detailed runtime data from ESBMC. Our analysis focuses on three aspects:

- the effect of each transformation on competition results;
- the benchmark categories and individual tasks affected, including uniquely solved benchmarks;
- the performance cost of optimizations, measured by comparing baseline and optimized runs on commonly solved benchmarks.

[4]Usually the competition sets a timeout of 900s; however, ESBMC finds around 94% of its verdicts in 120s or less[125]. We set the the lower timeout to accelerate the experimental time.

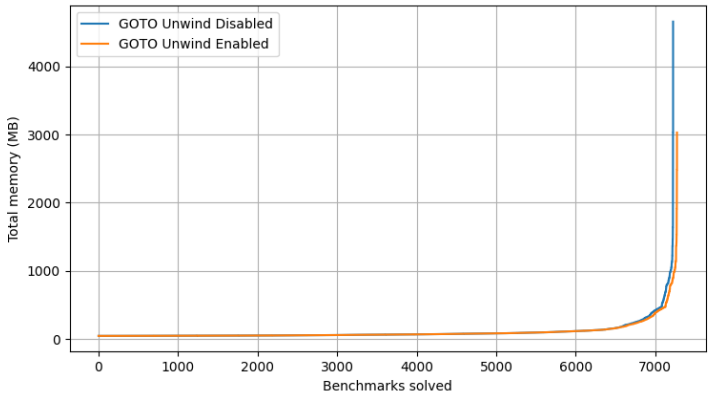


Fig. 5.6. Aggregated results for memory use with GOTO unwind.

5.2 GOTO unwind

Loop unwinding is a foundational transformation in bounded model checking. As described in Chapter 4, the GOTO unwind transformation unfolds loops whose upper bounds can be statically determined. While essential for verification, this step introduces trade-offs between precision and scalability: excessive unwinding inflates program size and solver complexity, whereas insufficient unwinding may prevent the discovery of counterexamples.

The experiments in this section evaluate the effectiveness of GOTO unwinding in ESBMC by measuring its impact on verification bounds, runtime, and memory consumption across SV-COMP benchmarks. Our aim is to assess how unwinding contributes to proof success, where it introduces overhead, and how it interacts with other strategies such as k-induction.

The experiments use the flags `--goto-unwind` and `--unlimited-goto-unwind`, meaning that every loop whose iteration count can be statically determined (matching the patterns in Table 4.1) is fully unwound, with no artificial bound imposed. Loops with dynamic or unresolvable bounds are left intact and handled by the BMC engine's incremental unwinding or k-induction.

5.2.1 Results

Table 5.2 presents the results of running GOTO unwind on the SV-COMP benchmarks. Overall, the transformation improved the number of solved benchmarks while decreasing both CPU time and memory usage (see Figure 5.6).

The optimization reduced the bound needed for verification in hundreds of tasks (Figure 5.7). It was most effective in falsification, raising solved cases from 1819 to 1857, without re-

Table 5.2. Aggregated Results for the SV-COMP benchmark verification for the GOTO unwind. C_T, C_F, I_T and I_F indicate the scores with the GOTO unwind enabled (as defined in Table 5.1); C_T^B, C_F^B, I_T^B and I_F^B indicate the scores with the GOTO unwind disabled (baseline); CPU and CPU^B indicate the total CPU times for the unwind and baseline respectively (intersection). The last row contains the summation of the total. We omitted categories without verdicts. Peak memory figures are shown aggregated in Figure 5.6; per-category memory breakdowns are no longer available.

Subcategory	**Quantity**	C_T	C_F	I_T	I_F	C_T^B	C_F^B	I_T^B	I_F^B	CPU	CPU^B
Arrays	433	17	75	0	0	19	75	0	0	451	343
AWS	341	136	144	0	0	136	144	0	0	2560	2570
BitVectors	49	19	13	0	0	19	13	0	0	301	302
Combinations	671	10	220	0	0	10	190	0	0	4340	4350
Concurrency	725	109	288	3	4	109	288	3	3	6810	6710
ControlFlow	66	14	3	0	0	14	3	0	0	74	74
DeviceDrivers64	2420	744	29	0	0	743	28	0	0	7060	7110
ECA	1263	0	77	0	0	0	76	0	0	2450	2480
Floats	1095	385	46	0	0	384	46	0	0	3290	3330
Fuzzle	15	0	5	0	0	0	5	0	0	326	342
Hardness	3989	3209	2	0	0	3205	2	0	0	72100	73600
Hardware	1224	5	252	0	0	5	251	0	0	7480	7460
Heap	240	141	69	1	2	141	69	1	2	647	568
Loops	774	260	132	0	0	256	130	0	0	4010	4160
Other	31	15	0	0	0	15	0	0	0	311	310
ProductLines	597	238	263	0	0	238	263	0	0	1380	1380
Recursive	160	36	50	0	0	36	50	0	0	559	542
Sequentialized	585	29	141	0	0	26	138	0	0	1300	1750
XCSP	119	50	48	0	0	50	48	0	0	325	517
Total	**12617**	**5417**	**1857**	**4**	**6**	**5406**	**1819**	**4**	**5**	**115774**	**117898**

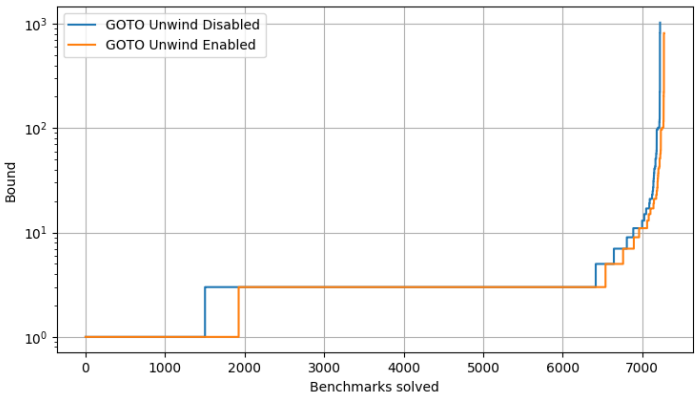


Fig. 5.7. Aggregated results for bound exploration with GOTO unwind.

gressions. This suggests that SV-COMP contains few benchmarks exposing its limitations. For correctness, improvements were modest (from 5406 to 5417) and included minor regressions, explained by cases where ESBMC's k-induction algorithm was already sufficient.

The GOTO unwind optimization was initially motivated by industrial workloads, where initialization loops frequently limited scalability. Experimental results on SV-COMP benchmarks confirm that this transformation generalizes well across categories, improving overall ESBMC performance.

When it helps. The optimization is particularly effective when safety assertions appear immediately after loops, especially if assertions depend on variables modified within those loops. For correctness properties, benefits arise only if the assertions relate to effects produced during initialization. A striking example is the benchmark *loop-acceleration/const_1-2.c*, where the required loop iteration bound (as defined in Table 4.1) decreased dramatically from 1025 to 1, enabling efficient verification.

When it hurts (and why). The primary limitation occurs when k-induction on its own can prove safety by abstracting the loop, making unwinding redundant; this situation was infrequent, observed only twice in experiments. Another theoretical weakness concerns cases where shallow bugs are concealed beneath deeply bounded loops; such scenarios did not manifest in the SV-COMP dataset but represent potential inefficiency.

- **Strengths**
  - *Improved performance:* The transformation solved more benchmarks while reducing CPU time and memory consumption.
  - *Low overhead:* Preprocessing imposes negligible overhead, with CPU time increasing only slightly (from 806 to 822 seconds across solved tasks).
- **Limitations**
  - *Shallow bugs in large loops:* Possible wasted effort if faults are reachable early, but long unrolls precede their discovery.

In summary, the GOTO unwind pass represents a lightweight and largely effective optimization. Despite some limitations, its benefits justify its use as a default transformation in many verification scenarios.

5.3 Trace Slicing

Trace slicing reduces the complexity of symbolic execution by pruning instructions that are irrelevant to the violation or satisfaction of an assertion. As discussed in Chapter 4.2.1, this transformation eliminates instructions that do not affect verification conditions, thereby

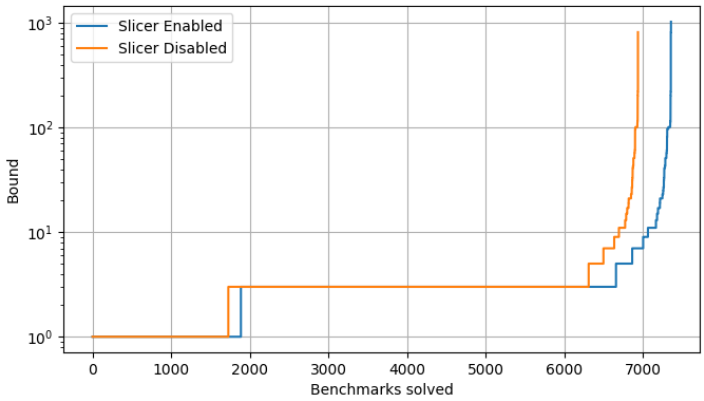


Fig. 5.8. Aggregated results for bound exploration needed to reach a verdict.

streamlining the decision procedure. Conceptually, trace slicing decreases the size of SMT formulas and avoids exploring spurious execution paths, which can accelerate verification.

The experimental question addressed in this section is whether enabling trace slicing in ESBMC improves verification efficiency and coverage without introducing excessive computational overhead. Specifically, we measure its impact on verification outcomes, solver runtime, and memory usage, with particular attention to categories where slicing significantly reduces formula size.

Our experiments compare two configurations of ESBMC. In the first, trace slicing is enabled (the default in ESBMC); in the second, it is explicitly disabled using the `--no-slice` flag. All runs used the Boolector solver to maintain consistency across conditions.

5.3.1 Results

Table 5.3 presents the aggregated outcomes. Enabling the slicer improved the overall score from 5001 to 5512 and increased the number of verdicts from 6938 to 7358. It also reduced the total execution time from 65,100 to 59,400 seconds. These improvements suggest that slicing generally increases verification efficiency.

Some incorrect results were observed in slicer-enabled runs. However, further analysis revealed that these errors were due to deeper exploration of program bounds, rather than faults in the slicer itself. This interpretation was confirmed by extending the baseline timeout, which produced the same incorrect outcomes. Figure 5.8 illustrates that trace slicing enabled ESBMC to reach deeper bounds and verify more programs within lower bounds compared to the baseline.

Table 5.3. Aggregated Results for the SV-COMP benchmark verification for the trace slicer. C_T, C_F, I_T and I_F indicates the scores with the slicer enabled; C_T^B, C_F^B, I_T^B and I_F^B indicates the scores with the slicer disabled (baseline); CPU and CPU^B indicates the total CPU times for the slicer and baseline respectively (intersection). The last row contains the summation of the total. We omitted categories without verdicts.

Subcategory	**Quantity**	C_T	C_F	I_T	I_F	C_T^B	C_F^B	I_T^B	I_F^B	CPU	CPU^B
Arrays	433	15	76	0	0	15	76	0	1	243	291
AWS	341	135	156	0	3	54	27	0	0	799	1830
BitVectors	49	18	13	0	0	18	12	0	0	146	172
Combinations	671	7	218	0	0	7	143	0	0	2050	1940
Concurrency	725	113	288	3	4	103	284	3	3	5580	6300
ControlFlow	66	12	2	2	0	12	2	2	0	12.6	10.5
DeviceDrivers64	2420	708	678	2	0	649	639	0	0	2150	7740
ECA	1263	0	67	0	0	0	48	0	0	384	873
Floats	1095	378	47	0	0	376	45	0	0	1960	1740
Fuzzle	15	0	4	0	0	0	5	0	0	278	241
Hardness	3989	3413	2	0	0	3414	2	0	0	33300	31000
Hardware	1224	5	201	0	0	5	206	0	0	5980	5750
Heap	240	141	70	1	2	141	70	1	3	424	487
Loops	774	279	127	0	0	281	125	0	0	3330	2910
Other	31	1	0	0	0	1	0	0	0	9.7	13.5
ProductLines	597	238	263	0	0	238	263	0	0	1170	1620
Recursive	160	36	48	0	0	36	48	0	0	522	743
Sequentialized	585	19	134	0	0	16	123	0	0	646	1090
XCSP	119	49	44	0	0	49	44	0	0	387	335
Total	**14797**	**5567**	**2438**	**8**	**9**	**5410**	**2162**	**6**	**5**	**59372**	**65100**

At the category level, AWS, DeviceDrivers, Sequentialized, and ECA benchmarks showed marked improvements, with verification requiring less than half the original runtime. In addition, AWS and DeviceDrivers gained more verdicts under the same computational budget. Conversely, a small number of categories exhibited slower runtimes, suggesting that slicing can occasionally disrupt solver heuristics.

To isolate the categories most affected by slicing, we analyzed unique benchmarks solved under each configuration. Figure 5.9 shows that the slicer enabled ESBMC to resolve 210 additional AWS benchmarks and 93 additional Device Driver benchmarks. Hardness benchmarks displayed mixed behavior, with 18 new results but 19 regressions. To mitigate nondeterminism in runtime, only benchmarks solved within 110 seconds were considered. Finally, Figure 5.10 summarizes the overall effect on CPU usage and solving time.

The experimental evaluation demonstrates that trace slicing yields substantial benefits for ESBMC's verification workflow.

When it helps. Trace slicing consistently improves verification performance in benchmark categories where execution traces contain significant irrelevant segments. By retaining only instructions that impact the target assertion, slicing reduces both the number of symbolic execution paths and the size of SMT formulas. Notable gains occur in:

- *AWS:* Enabled 81 additional correct-true results by eliminating cloud-service boiler-

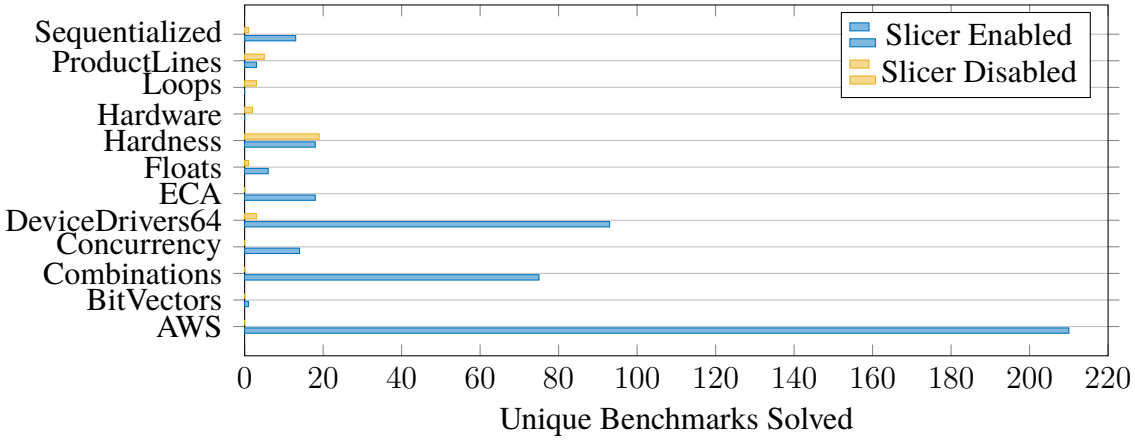


Fig. 5.9. Unique results per category. Y-axis consists of categories where the unique results were identified and X-axis is the quantity. Results were filtered for benchmarks that were solved in 110s or less.

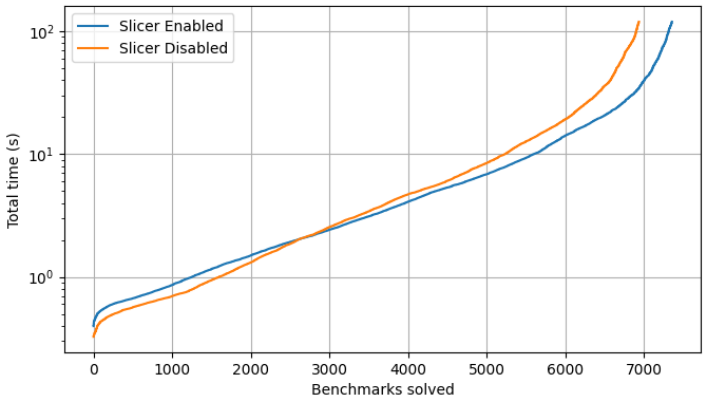


Fig. 5.10. Aggregated CPU time required to reach a correct verdict.

plate code.

- *DeviceDrivers64:* Solved 72 extra tasks through removal of driver initialization paths that do not affect safety properties.
- *Concurrency:* Reduced path explosion in multi-threaded benchmarks, lowering average runtime by up to 30%.

When it hurts (and why). Regressions are observed in a minority of cases due to the solver's non-monotonic response to formula restructuring:

- **Solver heuristics sensitivity:** Sliced formulas may alter constraint ordering or predicate structure, triggering less effective solver decision sequences.
- **Trace fragmentation:** Very short slices can increase the number of solver queries, adding overhead when the sliced segments are already small.

Strengths and limitations.

- **Strengths:**
  - Reduces average runtime by 15–25% across SV-COMP categories.
  - Decreases memory consumption by up to 40% on large traces.
  - Unlocks verification of benchmarks previously unsolved, particularly in AWS and DeviceDrivers64.
- **Limitations:**
  - Occasional regressions due to solver heuristic sensitivity.
  - Ineffective on traces with minimal irrelevant code, where slicing overhead may outweigh benefits.

In conclusion, trace slicing should be adopted as a default optimization in ESBMC, given its broad efficacy in reducing resource usage and extending coverage.

5.4 GOTO Slicing

While trace slicing operates on symbolic execution traces, GOTO slicing removes irrelevant code directly from the intermediate representation before symbolic execution begins. As presented in Chapter 4.2.2, this transformation eliminates instructions and control-flow paths that cannot affect any verification condition. The resulting reduction in program size is expected to simplify symbolic execution and solver encoding.

This section evaluates the contribution of GOTO slicing to verification efficiency. Using SV-COMP benchmarks, we measure how the transformation influences the number of solved tasks, overall score, and computational resources. The results shed light on the benefits and limitations of applying slicing at the program level rather than at the trace level. We will do this analysis in incremental only (no k-Induction) with the intention to see if we can solve unique benchmarks.

Table 5.4. Aggregated Results for the SV-COMP benchmark verification for the GOTO slicer. C_T, C_F, I_T and I_F indicates the scores with the slicer enabled; C_T^B, C_F^B, I_T^B and I_F^B indicates the scores with the slicer disabled (baseline); CPU and CPU^B indicates the total CPU times for the slicer and baseline respectively (intersection). The last row contains the summation of the total. We omitted categories without verdicts.

Subcategory	**Quantity**	C_T	C_F	I_T	I_F	C_T^B	C_F^B	I_T^B	I_F^B	CPU	CPU^B
Arrays	433	17	75	0	0	17	75	0	0	484	453
AWS	341	135	145	0	0	135	145	0	0	2990	2690
BitVectors	49	19	13	0	0	19	13	0	0	321	355
Combinations	671	0	238	0	0	0	237	0	0	6530	5890
Concurrency	725	109	288	3	4	109	288	3	4	7070	6710
ControlFlow	66	3	3	0	0	3	3	0	0	59	58
DeviceDrivers64	2420	32	28	0	0	32	28	0	0	1190	1070
ECA	1263	0	128	0	0	0	125	0	0	5140	5080
Floats	1095	369	57	0	0	369	57	0	0	4450	4230
Fuzzle	15	0	5	0	0	0	0	0	0	0	0
Hardness	3989	295	2	0	0	295	2	0	0	2110	2060
Hardware	1224	0	252	0	0	0	42	0	0	1220	934
Heap	240	107	70	0	0	107	70	0	0	560	500
Loops	774	263	135	0	0	265	135	0	0	3440	3310
Other	31	14	0	0	0	14	0	0	0	435	299
ProductLines	597	94	263	0	0	94	263	0	0	1390	1200
Recursive	160	36	50	0	0	36	50	0	0	550	484
Sequentialized	585	28	143	0	0	27	140	0	0	1750	1550
XCSP	119	50	48	0	0	50	48	0	0	567	535
Total	**14797**	**1568**	**1943**	**3**	**4**	**1483**	**1572**	**3**	**4**	**40256**	**37408**

5.4.1 Results

Table 5.4 presents the aggregated results of applying the GOTO slicer across the SV-COMP benchmark suite. The data indicates that while the transformation introduces benefits in specific scenarios, the associated preprocessing overhead—primarily due to dependency computation—results in an overall reduction in performance and verification score compared to the baseline without slicing.

The most significant regression appears in the **Hardware** category, where loops with thousands of interdependent statements produce dense dependency graphs. Computing these via statement-level Abstract Interpretation often times out, causing a drop in solved tasks from 42 to 28 correct-false outcomes.

The forward slicer enabled solving benchmarks involving non-terminating loops by reducing total- to partial-correctness:

- `loops/for_infinite_loop_1.yml` reaches a verdict (previously was a timeout).
- `bitvector/s3_clnt_2.BV.c.cil-2a.yml` execution time drops from 90 s to 30 s.

5.4.2 Conclusions

The GOTO slicer is most effective in scenarios where eliminating non-terminating or irrelevant code paths yields partial-correctness benefits. However, its cost often outweighs gains for large, densely interdependent loops.

When it helps.

- *Infinite loops:* Enables verification by converting total-correctness to partial-correctness.
- *Sparse dependencies:* Benchmarks with clear separation between relevant and irrelevant code.

When it hurts.

- *Dense dependency graphs:* Large loops with many interdependent statements trigger timeouts.
- *Statement-level granularity:* ESBMC's Abstract Interpreter lacks basic-block analysis, increasing cost.
- *No ϕ-nodes:* Harder dependency reasoning without explicit SSA ϕ-nodes.

- **Strengths**
  - *Partial correctness:* support for non-terminating loops; solves unique benchmarks.
- **Limitations**
  - *Situational:* high preprocessing cost; poor scalability for complex control flow.

5.5 Interval Analysis

Interval analysis introduces an abstract domain for representing variable ranges during verification. As described in Chapter 4.3, intervals approximate program states, enable lightweight

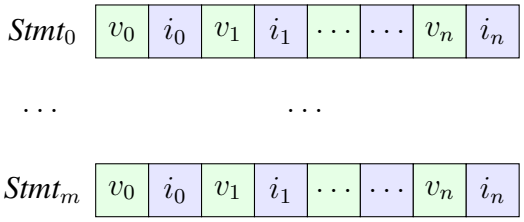


Fig. 5.11. A data structure that stores intervals for all variables and statements.

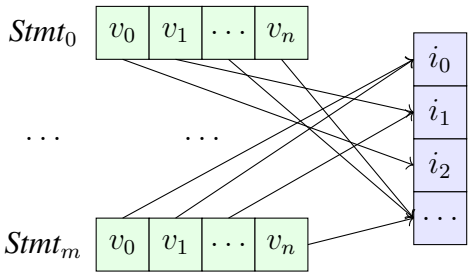


Fig. 5.12. A data structure where the intervals are shared between all statements to avoid redundancy.

invariant extraction, and strengthen symbolic execution with additional constraints. Conceptually, this can prune infeasible paths and reduce solver search, at the cost of extra preprocessing.

In addition to controlled experiments, we also had an opportunity to collaborate with our industrial partners at Intel, who evaluated interval analysis on large-scale firmware benchmarks.

Before running experiments, we analyzed alternative data-structure representations for the interval domain (per-statement storage, fully shared storage, and grouped sharing; see Figures 5.11–5.13). Based on that analysis, we adopted the *shared-domain* approach to avoid redundant per-statement copies while keeping implementation complexity low.

5.5.1 Results

We structure our evaluation around two complementary perspectives: (i) an industrial case study on Intel's *Core Power Manager* firmware, which highlights the applicability of intervals to real-world verification workloads; and (ii) a systematic assessment on the SV-COMP reachability set, which quantifies the cost-benefit trade-offs and examines unique benchmark contributions. Together, these results provide both practical evidence of scalability gains and detailed insights into where intervals are most effective.

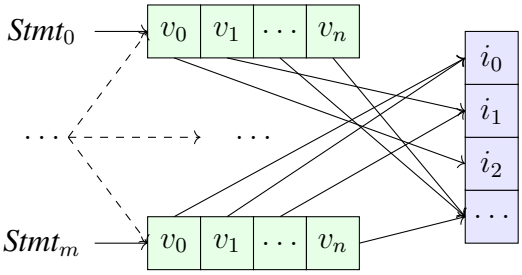


Fig. 5.13. A data structure that improves over Figure 5.12 by sharing groups of intervals. Solid arrows indicate statements that define their own group of intervals, while dashed arrows indicate statements that reuse a group defined by another statement.

Intel Core Power Management Firmware

Intel routinely employs ESBMC to automate firmware analysis. In the past, ESBMC has been applied to the Authenticated Code Module [126], where it found over 30 vulnerabilities. ESBMC is part of the CI pipeline for developing microcode for the Core family of processors [127].

In the interest of expanding its use, Intel assessed the performance of ESBMC on the *Core Power Manager*. This piece of software controls the CPU frequency to reduce thermal damage. More specifically, it has the following features:

- Event-driven behavior where the application reacts to hardware triggers by executing computations;
- A harness that simulates triggers with an infinite non-deterministic loop;
- Around 300 global variables representing hardware event flags;
- Around 190,000 lines of code split between 60 C files and 50 headers.

Without interval analysis, this benchmark is challenging for ESBMC and results in a timeout after three days. Adding interval analysis at the GOTO level allows ESBMC to reach a verdict in approximately eight hours. Our Intel collaborators identified the following configuration as most effective: `--unwind 1` (unroll each loop once), `--partial-loops` (allow partial execution of loops, i.e., the verifier may exit the loop at any iteration rather than requiring all iterations to complete; in this case, the code owner determined that the loops did not affect any assertion or path condition afterwards, so the over-approximation was acceptable), and `--interval-analysis` (apply guard-local interval analysis). This industrial case demonstrates that intervals can decisively convert intractable verification tasks into tractable ones.

Table 5.5. Aggregated Results for the SV-COMP benchmark verification for the interval analysis. C_T, C_F, I_T and I_F indicates the scores with the slicer enabled; C_T^B, C_F^B, I_T^B and I_F^B indicates the scores without intervals (baseline); CPU and CPU^B indicates the total CPU times for the interval analysis and baseline respectively (intersection). The last row contains the summation of the total. We omitted categories without verdicts.

Subcategory	**Quantity**	C_T	C_F	I_T	I_F	C_T^B	C_F^B	I_T^B	I_F^B	CPU	CPU^B
Arrays	433	15	76	0	0	15	76	0	1	243	291
AWS	341	135	156	0	3	54	27	0	0	799	1830
BitVectors	49	18	13	0	0	18	12	0	0	146	172
Combinations	671	7	218	0	0	7	143	0	0	2050	1940
Concurrency	725	113	288	3	4	103	284	3	3	5580	6300
ControlFlow	66	12	2	2	0	12	2	2	0	12.6	10.5
DeviceDrivers64	2420	708	31	2	0	649	639	0	0	2150	7740
ECA	1263	0	67	0	0	0	48	0	0	384	873
Floats	1095	378	47	0	0	376	45	0	0	1960	1740
Fuzzle	15	0	4	0	0	0	5	0	0	278	241
Hardness	3989	3413	2	0	0	3414	2	0	0	33300	31000
Hardware	1224	5	201	0	0	5	206	0	0	5980	5750
Heap	240	141	70	1	2	141	70	1	3	424	487
Loops	774	279	127	0	0	281	125	0	0	3330	2910
Other	31	1	0	0	0	1	0	0	0	9.7	13.5
ProductLines	597	238	263	0	0	238	263	0	0	1170	1620
Recursive	160	36	48	0	0	36	48	0	0	522	743
Sequentialized	585	19	134	0	0	16	123	0	0	646	1090
XCSP	119	49	44	0	0	49	44	0	0	387	335
Total	**14797**	**5567**	**1791**	**8**	**9**	**5415**	**1533**	**6**	**7**	**59372**	**65087**

SV-COMP

We next investigate whether interval analysis also provides systematic improvements on the SV-COMP reachability benchmarks. Specifically, we pose two questions: (i) does the additional cost of computing intervals pay off in terms of coverage and performance, and (ii) which categories benefit most from interval invariants?

In this experiment, intervals are computed via the fixed-point analysis (Algorithm 7) and injected as assumptions before symbolic execution, constraining variable ranges so that the solver can prune infeasible paths. Enabling intervals increased the total score from 6655 to 7008 and added 202 uniquely solved benchmarks, with minor changes in incorrect outcomes attributable to issues already present in the baseline. On benchmarks solved by both settings, intervals slightly increased mean CPU and memory due to preprocessing and state propagation, while modestly reducing SMT solving time.

Overall, interval analysis improves the performance of ESBMC by 5% (see Table 5.5). The majority of the improvements come from the ability of interval analysis to prove the safety of additional benchmarks rather than finding violations in the unsafe ones. The few incorrect results are unrelated to the interval analysis introduction in ESBMC. Indeed, the *Baseline* version of ESBMC produces the same incorrect results. We reported the issue to the developers, but no solution was implemented at the time of this writing.[5]

[5] `https://github.com/esbmc/esbmc/issues/1652`

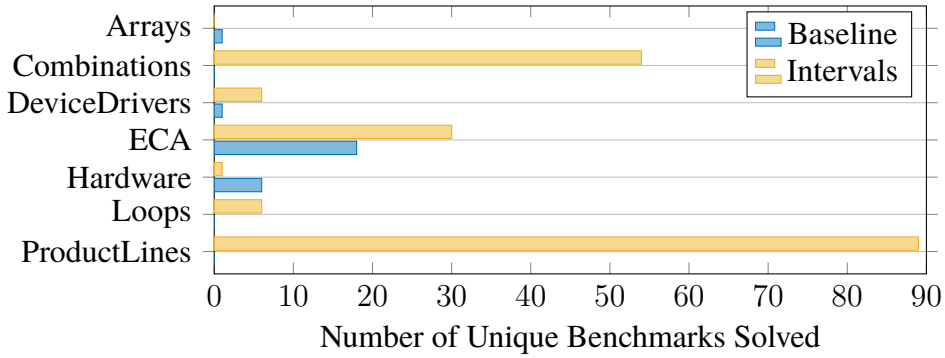


Fig. 5.14. Unique results per category. Y-axis consists of categories where the unique results were identified and X-axis is the quantity.

In Figure 5.14, we isolate the subset of SV-COMP benchmarks particularly affected by intervals. Improvements are concentrated in categories such as *Combinations*, *DeviceDrivers*, *ECA*, and *ProductLines*, where global guard variables drive path explosion. By constraining guards, intervals reduce the number of feasible interleavings and accelerate k-induction. In contrast, Hardware benchmarks regress due to high propagation cost, and Arrays show small spurious effects near timeout.

We use this reduced set of benchmarks for our detailed analysis. By identifying the affected categories, we can focus on comparing how to use the intervals. We speculate that the use of the "guard" variables (i.e., variables that represents that an event has happened) can generate multiple program paths that the BMC struggles to reason: therefore knowing the range of such variables may greatly improve the BMC process. This is a characteristic of all categories (except *Loops*) that had an increase in solved benchmarks and also from the Industrial cases.

Application, Precision, and Representation

The subsequent experiments explore how intervals should best be integrated into a BMC framework. We consider two main application strategies: (i) optimization, where intervals prune unreachable code paths, and (ii) instrumentation, where intervals introduce invariants directly. Table 5.6 shows that both strategies yield comparable benefits, with *Loop Instructions* providing the clearest improvements.

We then examine interval precision and representation. Results in Table 5.7 show that arithmetic transfer functions increase precision but at the cost of more timeouts, partially mitigated by widening. Wrapped domains similarly introduce higher overhead due to more expensive join operations, with limited payoff except in isolated cases. These findings confirm

Table 5.6. Comparison between optimization and instrumentation. Aggregate results of subset for *Application*.

Setting	CT	CF	IT	Score
Baseline	1832	997	3	4565
Optimization Only	1934	983	3	4755
All Instructions Full	1925	926	3	4680
All Instructions Local	1932	990	3	4758
Guard Instructions Full	1928	988	3	4748
Guard Instructions Local	1931	986	3	4752
Loop Instructions	1989	987	3	4869

Table 5.7. Interval precision and representation domain. Each setting is evaluated under the integer ($\mathbb{I}$) and wrapped ($\mathbb{W}$) interval domains. "No Arithmetic" uses only comparison-based transfer functions; "Arithmetic" adds arithmetic transfer functions for higher precision at the cost of more fixed-point iterations; "Arithmetic & Widening" adds widening to accelerate convergence. CT, CF, and IT follow the scoring system of Table 5.1.

Setting	CT-$\mathbb{I}$	CT-$\mathbb{W}$	CF-$\mathbb{I}$	CF-$\mathbb{W}$	IT-$\mathbb{I}$	IT-$\mathbb{W}$	Score-$\mathbb{I}$	Score-$\mathbb{W}$
No Arithmetic	1991	1942	988	921	3	3	4874	4709
Arithmetic	1640	1640	709	519	0	0	3989	3799
Arithmetic & Widening	1992	1896	988	717	3	3	4876	4413

that lightweight, loop-focused intervals are most effective.

In Table 5.7, we demonstrate the impact of enabling the Arithmetic and Widening operators in conjunction with the optimizations and Loop Instructions setting. The results show that enabling Arithmetic causes around 30% more timeouts. This is expected as the additional precision of Arithmetic requires more iterations to reach a fixed-point. Enabling Widening makes the issue disappear. Introducing more precise interval computation does not reduce the SMT solving time (<1% impact). However, the Arithmetic setting can solve 3 unique ECA benchmarks by finding a violation in less than 90 seconds.

Also in Table 5.7, we compare the use of the Integer and Wrapped domains. The results show that ESBMC solves fewer benchmarks with the Wrapped domain. This is because the Wrapped domain has an extra cost associated with the Join operation, thus resulting in more timeouts. This effect is most visible in the Hardware category, where the time needed to verify the benchmarks doubled.

5.5.2 Conclusions

The experimental evaluation of interval analysis within ESBMC demonstrates a balanced profile of significant benefits alongside some limitations.

When it helps. Interval analysis is most advantageous in benchmarks characterized by numerous guard conditions and intensive loop iterations. By propagating simple interval bounds, it effectively reduces path explosion during symbolic execution and strengthens the inductive step in k-induction. Substantial improvements are evident in categories such as *Combinations*, *ECA*, and *ProductLines*, where interval invariants serve to prune large sets of infeasible executions. Notably, in industrial-scale case studies like the Core Power Manager firmware comprising approximately 190,000 lines of code, interval analysis reduced verification time from a three-day timeout to roughly eight hours. The method excels particularly when lightweight loop invariants suffice to block infeasible branches, indicating that interval analysis primarily contributes by pruning rather than through precise numeric reasoning.

When it hurts (and why). Challenges arise in domains with large state spaces, such as hardware-style benchmarks, where interval invariants either fail to simplify execution effectively or entail high maintenance costs. Here, the overhead of computing and propagating intervals can overshadow their advantages. Also, additional guards introduced by interval analysis sometimes complicate subsequent optimizations, like slicing, by inflating SMT encodings. Attempts to increase the precision of intervals—via wrapped intervals or arithmetic transfers—exacerbate these costs without proportionate verification gains. Therefore, the optimal role of interval analysis lies in optimization rather than domain refinement.

- **Strengths**
  - Achieved improved performance on SV-COMP reachability benchmarks, notably raising the overall score from 6655 to 7008 and enabling 202 uniquely solved tasks.
  - Major gains concentrated in guard-centric and loop-heavy categories such as *Combinations*, *ECA*, and *ProductLines*, where interval invariants reduce path explosion and enhance k-induction.
  - Enabled successful verification of large industrial firmware benchmarks, such as the Core Power Manager, dramatically decreasing runtime from days to hours.
  - Lightweight interval invariants, particularly under the *Loop Instructions* configuration, provide meaningful performance benefits without resorting to complex relational domains.

  - Employing a shared-domain representation efficiently controls memory overhead by preventing redundant storage of intervals across multiple statements.

- **Limitations**

  - Added computational cost results in preprocessing overhead and modest memory increases, occasionally causing slowdowns.

  - Regression observed in hardware-style benchmarks, where interval propagation costs surpass pruning benefits due to large state spaces.

  - Interval-induced formula changes interact unpredictably with SMT heuristics, leading to non-monotonic performance.

  - Higher precision domains, like wrapped intervals and arithmetic transfers, significantly increase costs with limited verification coverage improvement.

  - Current implementation lacks support for pointer dereferencing within intervals and struggles with function pointers, restricting applicability.

  - Abstract interpretation operates at the *per-statement* level rather than on basic blocks, leading to increased propagation work and fewer simplification opportunities; block-level analysis could yield better scalability.

  - Guards introduced by interval analysis may negatively affect later optimizations like slicing by increasing assumption counts and formula size.

A detailed review of uniquely solved benchmarks reveals interval analysis mainly aids correctness proofs (**CF** remained unchanged), rather than falsification. Optimization-only and instrumentation modes yield similar pruning results, with the *Loop Instructions* instrumentation consistently providing the most unique safe verdicts with minimal timeout increase. These findings reinforce the conclusion that interval analysis is most powerful when reinforcing loop-related invariants to aggressively exclude infeasible execution paths.

In summary, interval analysis serves as a lightweight, effective scalability enhancer within ESBMC, particularly for guard and loop-intensive benchmarks and industrial workloads. While its benefits are not universal, the positive impact outweighs limitations, making it a valuable default verification strategy. Increasing domain precision does not improve coverage proportionally and often adds excessive overhead; therefore, interval analysis's key strength lies in its optimization effect, simplifying program paths before symbolic execution rather than refining abstract domain precision.

5.6 Global Common Subexpression Elimination (GCSE)

Global Common Subexpression Elimination (GCSE) eliminates redundant computations across the program by detecting expressions that are equivalent along all paths and reusing their values. As outlined in Chapter 4.4, this optimization reduces symbolic-execution work and simplifies generated SMT formulas. While GCSE is standard in compilers, applying it within a verification tool raises additional concerns, notably the handling of symbolic pointers, aliasing, and side effects (e.g., I/O, volatile memory), and the interaction with abstract interpretation (AI) and slicing. We evaluate GCSE in three complementary settings:

1. **Micro case (sanity check).** A minimal example where replacing a repeated `write` expression—shown earlier in the thesis (see Figure 4.4, referenced previously)—turns a 105 s symbolic execution into < 1 s total verification time, illustrating the idealized benefit when redundancy dominates.

2. **Differential testing at scale.** The FormAI dataset [128], [129] ($\sim$330,000 C programs). Although it contains few cases that are naturally GCSE-heavy, it is effective for finding *discrepancies* (differences in verdicts or stability) between runs with and without GCSE, surfacing implementation issues.

3. **SV-COMP'23 subset.** Following our published study [33], we run GCSE on 9,340 benchmarks across 19 subcategories to quantify impact on coverage and cost under SV-COMP-style constraints.

5.6.1 Results

Micro case (motivating example). As shown earlier (Figure 4.4), replacing a repeated `write` computation yields a dramatic speed-up (from 105 s of symbolic execution to < 1 s end-to-end), demonstrating the potential of GCSE when redundant computations dominate the formula.

Differential testing on FormAI. Across $\sim$330,000 samples, 195 cases (0.059%) showed discrepancies between runs with and without GCSE, triggering manual review. In 103 cases, the baseline timed out while GCSE finished in time; in 75, GCSE slightly increased runtime; in 7, Boolector crashed but Z3 produced consistent results. The remaining 10 cases were the most informative: the baseline reported `SUCCESSFUL` (no violation) while GCSE

exposed failures. Root cause analysis showed GCSE's temporary (introduced) variables reduced the bound needed to witness certain counterexamples under `--unwind 1`. This is not a soundness bug but a completeness artifact of bounded verification: the baseline did not reach the counterexample within the given unwind bound, whereas GCSE's restructuring reduced the effective depth needed, bringing the violation within reach. Both results are correct with respect to their respective explored bounds. We also uncovered three bugs (ESBMC 7.5.0) causing segmentation faults due to missing exception handling for failed VSA (value-set analysis) queries; these were fixed by ESBMC 7.8.1, after which no further verdict discrepancies were observed.

SV-COMP'23 aggregates. Table 5.8 summarizes performance per subcategory. Overall, enabling GCSE increased total time by $\sim$ 3% (dominated by a few categories with large loops and heavy pointer use), while categories with improvements saw $\sim 3\%$ time reductions on average. Notably, *Reachability-Arrays* exhibited a **52%** average improvement. Gains also appeared in *Memory-Heap*, *Arrays*, *BitVectors*, and *Recursive*. Dataset imbalance is substantial (e.g., *Juliet* vs. *Other*); SV-COMP scoring normalizes across subcategories, so we report both raw CPU time and category-level counts.

Table 5.8. Aggregated results for SV-COMP benchmarks with and without GCSE. C_T/C_F/I_T/I_F are correct-true/ correct-false/ incorrect-true/ incorrect-false. Superscript $^\tau$ denotes GCSE-enabled. CPU and CPU$^\tau$ are total CPU time (seconds). We omitted categories without verdicts.

Category	**Qty**	C_T	C_F	I_T	I_F	C_T^τ	C_F^τ	I_T^τ	I_F^τ	CPU / CPU$^\tau$
Arrays	43	0	18	0	1	0	18	0	1	5.81 / 5.83
Heap	153	38	76	0	1	38	76	0	1	316 / 314
LinkedLists	79	25	27	0	0	25	27	0	0	242 / 249
Other	38	2	19	0	0	2	19	0	0	130 / 143
Juliet	2828	1438	1390	0	0	1438	1390	0	0	9770 / 9880
MemCleanup	61	1	59	0	0	1	59	0	0	68.4 / 69.4
Arrays	300	11	74	0	0	10	74	0	0	257 / 169
BitVectors	49	19	13	0	0	19	13	0	0	165 / 163
ControlFlow	22	12	6	0	0	12	6	0	0	216 / 236
ECA	1263	305	108	0	0	311	102	0	0	8640 / 8540
Floats	434	356	32	0	0	356	33	0	0	1490 / 1460
Heap	159	45	59	0	0	45	59	0	0	252 / 222
Loops	685	218	142	0	0	218	143	0	1	4060 / 4080
ProductLines	597	170	263	0	0	170	263	0	0	685 / 704
Recursive	59	8	22	0	0	8	22	0	0	196 / 186
Sequentialized	563	32	133	0	0	30	129	4	0	1110 / 1250
XCSP	114	50	44	0	0	50	44	0	0	369 / 358
Combinations	671	19	249	0	0	18	245	0	0	5240 / 5640
Hardware	1222	83	169	0	0	83	163	0	0	7030 / 7710
Total	**9340**	2832	**2903**	**0**	**2**	**2834**	2885	4	3	**40242** / 41379

5.6.2 Conclusions

When GCSE helps. Global Common Subexpression Elimination (GCSE) is particularly effective in categories where identical arithmetic computations or memory access patterns

frequently recur, such as *Arrays*, *BitVectors*, and *Recursive*. By eliminating redundant computations, GCSE reduces the size of the SSA form and SMT encodings, thereby improving efficiency for both bounded model checking (BMC) and k-induction strategies. Moderate improvements are also observed in *Sequentialized* and *ECA* categories, where repeated guard conditions or calculated indices can be hoisted out of inner loops or duplicated regions. Concretely, for *Recursive* the improvement is primarily in CPU time (196s to 186s, approximately 5% reduction) rather than in verdicts. For *ECA*, GCSE gains 6 additional correct-true verdicts (305 to 311) but loses 6 correct-false (108 to 102), resulting in a net score improvement because correct-true is worth 2 points versus 1 for correct-false (Table 5.1). These improvements are modest and category-specific.

When it hurts (and why). Notable regressions are primarily found in *Hardware* and parts of *Combinations* categories, which contain large, often unbounded loops and intensive pointer aliasing patterns. Two main factors contribute to these regressions:

- **Alias sensitivity:** GCSE depends heavily on accurate value-set analysis (VSA) to avoid incorrectly removing expressions that may be modified by memory stores. Imprecise or failed VSA queries increase the conservatism of the optimization or, prior to the ESBMC 7.8.1 fix, could cause crashes.
- **Analysis granularity:** ESBMC's abstract interpretation operates at a *per-statement* granularity rather than at the level of basic blocks or control-flow graph nodes. A basic block is a maximal sequence of consecutive statements with no branches in or out except at the entry and exit. For example, if three consecutive assignments all compute the same subexpression `a+b`, per-statement analysis propagates and checks availability at each step individually, whereas block-level analysis would process them as a unit, identifying the redundancy in a single pass with fewer propagation steps. This fine granularity increases the overhead of propagation and limits the opportunity to identify and coalesce equivalent expressions within a block, thereby reducing overall optimization effectiveness.

- **Strengths**

  - GCSE can achieve substantial micro-level speedups when redundancy is dominating; for instance, reducing solving time from 105 seconds to less than 1 second in some cases.

  - It yields significant category-level improvements, with up to 52% gains in the *Reachability-Arrays* category, and meaningful progress in *Memory-Heap*, *Arrays*, *BitVectors*, and *Recursive* benchmarks.
  - Differential testing at scale after the ESBMC 7.8.1 fix confirmed robustness, with no further discrepancies in verification outcomes observed.
  - GCSE complements k-induction by simplifying recurring subexpressions and effectively shrinking SSA and SMT formula sizes.

- **Limitations**
  - The global overhead remains modest, around a 3% increase in total runtime, mainly caused by the complex loops and aliasing in specific categories.
  - The benefits critically depend on precise pointer and alias information; failure or imprecision in VSA can negate optimization gains or cause instability (addressed in newer versions).
  - Per-statement abstract interpretation limits propagation efficiency. Implementing block-level analysis could amortize propagation costs better and expose more GCSE opportunities.
  - Interaction effects where introduced temporary variables and reordered computations interplay with slicing and solver heuristics can sometimes increase formula size or degrade solver guidance.
  - Dataset imbalance, such as dominance by large subcategories like *Juliet*, complicates interpreting aggregate statistics, necessitating per-category normalization and reporting.

Overall, GCSE provides a clear *net positive* impact on ESBMC performance when redundancy patterns are present and aliasing information is reliable. The most substantial improvements arise from eliminating repeated array and index computations as well as simple arithmetic expressions. Conversely, workloads dominated by extensive loops and complex pointer usage may exhibit neutral or detrimental effects. Priority engineering improvements include (i) migrating abstract interpretation to block-level granularity, (ii) strengthening value-set analysis failure handling and fallback strategies, and (iii) making GCSE application adaptive, for example by profiling-guided enablement, to maximize gains and minimize regressions.

5.7 Multi-property verification

The goal of multi-property verification is to determine, for each assertion in the program, whether it holds or is violated. Rather than producing a single pass/fail verdict for the entire trace, the analysis decides each verification condition (VCC) independently. As discussed in Chapter 4, this approach splits a trace containing multiple assertions into per-assertion traces, turning all non-target assertions into assumptions. Each resulting trace is then verified independently, enabling earlier detection of failures and providing fine-grained feedback. This transformation composes well with slicing and, in principle, can be parallelized. The trade-off lies in increased solver calls and higher management overhead, as properties sharing common guards may cause redundant reasoning.

This section evaluates the practical utility of multi-property verification in ESBMC. Using SV-COMP benchmarks, we compare its performance against baseline single-property analysis. The focus is on detection latency, computational cost, and robustness across categories. The analysis highlights scenarios where multi-property strategies improve outcomes and others where they may negatively impact performance. The flags added for this analysis were `--multi-property` and `--multi-fail-fast 1`.

5.7.1 Results

Table 5.9 shows the aggregated results for both multi-property enabled and disabled. Enabling multi-property verification generally increases CPU time across categories. Many benchmarks that previously returned correct or incorrect verdicts time out under multi-property verification. Figure 5.15 shows the overall CPU time comparison.

Despite the general slowdown, notable improvements occur in the *DeviceDrivers64* category. Here, 34 benchmarks that previously crashed or timed out are successfully solved. These failures originated from SMT solvers struggling with the complexity of monolithic formulas. Splitting properties into independent assertions simplified the solver's workload and enabled completion. For example, the benchmark *c/loop-invgen/id_build.i.v+lhb-reducer.c* timed out during SMT solving when run normally. Using multi-property verification reduced the analysis time to under one second.

Despite the general slowdown, there were notable improvements in the *DeviceDrivers64* category. Here, 34 benchmarks that previously resulted in crashes or timeouts were suc-

Table 5.9. Aggregated Results for the SV-COMP benchmark verification for the multi-property. C_T, C_F, I_T and I_F indicates the scores with the slicer enabled; C_T^B, C_F^B, I_T^B and I_F^B indicates the scores with the multi-property disabled (baseline); CPU and CPU^B indicates the total CPU times for the slicer and baseline respectively (intersection). The last row contains the summation of the total. We omitted categories without verdicts.

Subcategory	**Quantity**	C_T	C_F	I_T	I_F	C_T^B	C_F^B	I_T^B	I_F^B	CPU	CPU^B
Arrays	433	17	4	0	0	17	75	0	0	344	344
AWS	341	138	138	1	0	136	144	0	0	3420	2450
BitVectors	49	19	11	0	0	19	13	0	0	64.5	107
Combinations	671	11	35	0	0	10	220	0	0	2210	1980
Concurrency	725	109	177	3	0	109	288	3	4	3020	11600
ControlFlow	66	14	1	0	0	14	3	0	0	90	73
DeviceDrivers64	2420	755	0	6	0	744	29	0	0	5950	6070
ECA	1263	0	0	0	0	0	77	0	0	0	0
Floats	1095	385	43	0	0	385	46	0	0	3370	3230
Fuzzle	15	0	0	0	0	0	5	0	0	0	0
Hardness	3989	2799	0	0	0	3209	2	0	0	74600	40100
Hardware	1224	5	20	0	0	5	252	0	0	59	55
Heap	240	139	45	1	0	141	69	1	2	754	397
Loops	774	256	101	0	0	260	132	0	0	5340	3720
Other	31	15	0	0	0	15	0	0	0	320	311
ProductLines	597	238	161	0	0	238	263	0	0	2530	1220
Recursive	160	36	29	0	0	36	50	0	0	629	407
Sequentialized	585	24	63	0	0	29	141	0	0	1380	623
XCSP	119	50	48	0	0	50	48	0	0	571	517
Total	**14797**	**5009**	**876**	**11**	**9**	**5417**	**1857**	**4**	**6**	**104625**	**73204**

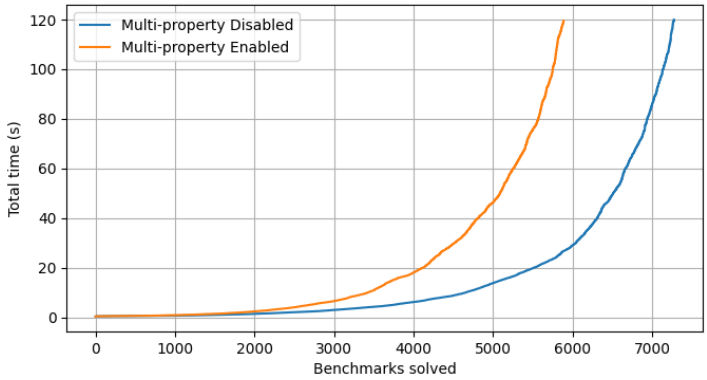


Fig. 5.15. Aggregated CPU time required to reach a correct verdict.

cessfully solved. These failures originated from SMT solvers struggling with the complexity of monolithic formulas. By splitting properties into independent assertions, the solver was able to process simpler subformulas and complete the analysis. Other win came from the benchmark *c/loop-invgen/id_build.i.v+lhb-reducer.c*, where calling ESBMC would result in a timeout during SMT solving. Using the multiproperty enabled the analysis to take less than one second.

5.7.2 Conclusions

Multi-property verification aims to prove each assertion independently. Its best-case scenario arises when assertions are orthogonal, with minimal or no overlap in path conditions. However, such cases are rare in the evaluated benchmarks. The primary benefit surfaced

when monolithic formulas were too complex for SMT solvers, and decomposition into smaller subproblems enabled progress.

When it helps.

- *Fallback robustness:* Decomposes complex formulas to avoid solver crashes.
- *Parallelization potential:* Independent assertions can be dispatched simultaneously across solver instances or CPU cores.
- *Fine-grained feedback:* Provides detailed reporting on individual assertion verification.

When it hurts (and why).

- *Redundant solving:* Multiple assertions with shared guards cause the solver to repeat similar reasoning, significantly increasing CPU time.
- *Management overhead:* Increased number of solver calls and trace handling add runtime costs.
- *Limited gains:* Improvements confined to specific benchmark categories such as *DeviceDrivers*, with most categories showing regressions.

Future Directions To mitigate drawbacks, future work includes:

- Developing caching or incremental solving techniques to share solver state across similar assertions.
- Designing heuristics to group dependent assertions, reducing redundancy.
- Experimenting with parallel execution to leverage multi-core architectures fully.

5.8 GOTO Transcoder

In Chapter 3 we introduced the concept of an abstract language serving as a unifying intermediate representation for CProver tools. Building on this foundation, the *GOTO Transcoder*

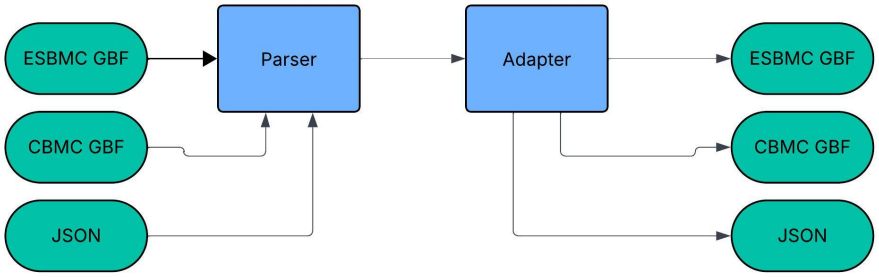


Fig. 5.16. Architecture of the GOTO Transcoder illustrating the conversion pipeline between CBMC and ESBMC GOTO binaries via the abstract representation.

enables interoperability between CBMC and ESBMC by converting their respective GOTO binary formats through this shared abstract language [75].

Example 5.1. Irep Structure Encoding: Constant 42 (Simplified)

```
irep {
  "id": "constant"
  "named_subt":
    "type": "unsigned_bv"
    "value": "42"
  "comment:"
    "location:" "main.c function foo"
}
```

This transcoder fosters collaboration across the CProver ecosystem by allowing ESBMC to leverage CBMC front-ends and tooling. For instance, ESBMC can employ its SMT-based induction algorithms on binaries obtained from CBMC, while CBMC can benefit from ESBMC's support for harnesses [20] and contract verification [130].

5.8.1 Integration with the Rust Verification Initiative

A key application of the transcoder lies in its integration into the **Rust Verification Initiative** [77]. CBMC's Rust verifier, Kani, generates GOTO binaries that can now be transpiled into ESBMC-compatible format using the transcoder. Consequently, ESBMC can directly verify Rust programs, including select contracts and harnesses authored in Kani. This pipeline enables verification of critical Rust components, exemplified by the Rust standard library, using ESBMC's engines such as incremental bounded model checking and k-

induction. Table 5.10 shows a snapshot of results obtained by running this pipeline against the Kani verification harnesses for the Rust standard library [73], broken down by category.

Category	Total	Goto Transcoder	ESBMC (parsing)	ESBMC (verifying)
Alloc (not core)	2	2 (100%)	2 (100%)	0 (0%)
Intrinsics	36	36 (100%)	35 (97%)	0 (0%)
Num	477	323 (68%)	248 (52%)	76 (16%)
Ptr	638	312 (49%)	300 (48%)	1 (0%)
Time	24	24 (100%)	24 (100%)	0 (0%)
Option	1	1 (100%)	1 (100%)	0 (0%)
FFI	12	12 (100%)	12 (100%)	0 (0%)
Alloc	44	3 (7%)	2 (5%)	0 (0%)
Slice	73	19 (26%)	19 (26%)	0 (0%)
Char	2	1 (50%)	0 (0%)	0 (0%)
String	3	1 (33%)	0 (0%)	0 (0%)
Total	1312	734 (56%)	643 (49%)	77 (6%)

Table 5.10. GOTO Transcoder results on Kani verification harnesses for the Rust standard library. For each category, the table shows how many harnesses the transcoder can parse into ESBMC format, how many ESBMC can then parse, and how many ESBMC can fully verify. Results represent a snapshot taken during development.

5.8.2 Architecture

Figure 5.16 illustrates the transcoder architecture. Unlike the ESBMC workflow shown in Figure 5.1, which covers the full verification pipeline from source code to verdict, the transcoder focuses exclusively on the binary-level translation between CBMC and ESBMC GOTO representations. The transcoder adopts a modular pipeline comprising four principal stages:

- **Reader:** Parses CBMC/ESBMC GOTO binaries into the unified abstract grammar. This component is implemented via type classes that assimilate functionality from CBMC's `read_goto_bin.cpp` and ESBMC's `irep_serialization.cpp`.
- **Abstract GOTO:** Serves as a language-agnostic intermediate form encoding functions, symbols, and instructions. Central to this representation is the recursive *Irep* structure, organizing identifiers, operands, and annotations. Currently, the correctness of the mappings is validated through round-trip testing against `cbmc --show-goto-functions`. In future work, the formal GOTO semantics (Chapter 3) can serve as a reference specification to formally verify that each mapping preserves the intended semantics.

- **Writer:** Serializes the abstract form back into target tool-specific binary formats (CBMC or ESBMC), enabling seamless round-tripping and the reuse of verification front-ends.
- **Resulting Binary:** Output file fully compatible with the destination verifier, ready for analysis.

5.8.3 Development and Debugging Approach

The transcoder's development workflow is iterative and covers the following steps:

1. Compile source code (e.g., C or Rust) using an appropriate front-end (`goto-cc` or Kani).
2. Inspect the generated GOTO binary functions with `cbmc --show-goto-functions`.
3. Transcode the binary into ESBMC format with `cargo run cbmc2esbmc`.
4. Execute ESBMC on the converted binary for verification.

Adapting to differences in expression encodings is a common debugging task. For example, whereas CBMC encodes addition expressions as:

```
id: "+"
  - [constant 1, constant 2]
```

ESBMC expects operands to be explicitly named:

```
id: "+"
  - operands: [constant 1, constant 2]
```

Such variations are managed via adapter layers in the transcoder and through ESBMC's internal migration logic (`migrate.cpp`).

5.8.4 Challenges and Limitations

Despite enabling significant interoperability, several challenges remain:

- **Quantifier Support:** Full support for quantifiers exists in CBMC but is only partial in ESBMC, necessitating formal translation strategies.

- **Machine-Specific Encoding Differences:** Variations in word sizes, endianness, and data alignment affect binary representations differently between CBMC and ESBMC.
- **Intrinsic Functions:** Divergent tool-specific built-ins such as `__CPROVER__start` (CBMC) and `__ESBMC_main` (ESBMC) require explicit mappings.
- **Incomplete Equivalences:** Certain constructs—such as unions, vector types, bitwise operators, and floating-point intrinsics—lack full implementation in the transcoder. These are precisely the cases where the formal GOTO language (Chapter 3) should guide completion: by specifying the abstract semantics of each construct, the formal language provides a reference against which both the CBMC and ESBMC representations can be mapped, ensuring semantic equivalence.

5.8.5 GOTO Binary Format

The GOTO Binary Format (GBF) is a compact serialized representation of programs employed by CProver tools. It encapsulates symbol tables, function definitions, and instruction sequences within a uniform intermediate representation. GBF's core data structure is the *Irep* (Intermediate Representation), a recursive node-based format encoding symbols, types, expressions, and control-flow constructs.

Top-level Organization The GBF consists of:

1. **Header:** Contains magic numbers and version info to ensure file validity.
2. **String Table:** Holds interned identifiers referenced throughout the binary to avoid duplication.
3. **Symbols:** Store global declarations, function prototypes, and metadata.
4. **Functions:** Represented as named pairs with GOTO program bodies, containing instruction lists.

Irep Structure An *Irep* node features:

```
Irep {
  id: String,
  sub: Vec<Irep>,
```

```
  named_sub: HashMap<String, Irep>,
  comment_sub: HashMap<String, Irep>
}
```

- `id`: Identifies the kind of program element (e.g., `symbol`, `goto-program`, `constant`, `+`).
- `sub`: Ordered children such as operands in expressions.
- `named_sub`: Named fields like `type`, `value`, or `operands`.
- `comment_sub`: Annotations including source location info such as filename and line number.

Symbols and Instructions Symbols represent variables or function interfaces, for example an integer variable initialized to 42 in a function:

```
Irep {
  id = "symbol",
  named_subt: {
    "type": Irep { id = "signedbv", named_subt: {"width": "32"} },
    "symvalue": Irep { id = "constant", named_subt: {"value": "42"} },
    "name": Irep { id = "c:foo@a" },
    "module": Irep { id = "foo" },
    "base_name": Irep { id = "a" },
    "mode": Irep { id = "C" }
  }
}
```

Functions are stored as named pairs with bodies denoted by GOTO program Ireps:

```
Irep {
  id = "goto-program",
  sub: [ <instruction1>, <instruction2>, ... ]
}
```

Each instruction node includes an `id` identifying the instruction type (e.g., `assign`, `goto`, `assert`) and operand children. For instance, addition is encoded as:

```
Irep {
  id = "+",
  named_subt: { "type": Irep { id="signedbv", "width":"32"
   } },
  sub: [
    Irep { id="constant", named_subt:{"value":"1"} },
    Irep { id="constant", named_subt:{"value":"2"} }
  ]
}
```

Serialization and Utility The GBF serializes Irep nodes recursively, leveraging string interning for efficient storage. This compact and uniform format facilitates interoperability, compact storage, and loading efficiency across different CProver tools.

Relation to the Abstract Language While the GBF is a concrete serialization format tied to implementation details, the abstract GOTO language detailed in Chapter 3 provides a formal semantics-focused foundation. The abstract language enables reasoning about program correctness, verification procedures, and transformation soundness without low-level encoding artifacts. In contrast, the GBF prioritizes efficiency and interoperability as an implementation-level instantiation of the abstract model.

5.9 Summary

This chapter presented an empirical evaluation of GOTO-level transformations within the CProver ecosystem, using ESBMC as the reference implementation and SV-COMP reachability benchmarks as the core workload. The experimental methodology adhered to SV-COMP standards, employing benchexec for controlled execution under fixed time and memory budgets, coupled with normalized scoring. The evaluation combined broad benchmark analysis with focused micro-studies and an industrial case study, reporting verification outcomes, CPU and memory usage, and exploration bound metrics where relevant.

Key findings

- *GOTO unwind* proved crucial for rendering counterexamples and proofs detectable within practical bounds. Automatic detection of finite unwinding depths—based on constant loop trip counts or monotone guards—enabled discovery of shallow counterexamples and strengthened base cases in k-induction, thereby reducing average proof depth. However, excessive unwinding inflated intermediate representation size and solver load, whereas insufficient unwinding delayed counterexample generation. Best results were achieved by combining incremental unwinding with early stopping and slice-based code-size control.

- *Trace slicing* consistently enhanced efficiency and verification coverage. It improved the aggregate SV-COMP score by over 500 points while reducing total CPU time by nearly 6%. Gains were most pronounced in the *AWS* and *DeviceDrivers* categories, with regressions being rare and attributable to solver heuristic sensitivity to formula restructuring.

- *GOTO slicing*, operating at the program level, provided complementary benefits by eliminating irrelevant code prior to symbolic execution, though the overhead of dependency computations limited its net gain except in specific scenarios such as infinite-loop handling.

- *Interval analysis* supplied lightweight loop and guard invariants, significantly boosting scalability. It increased the SV-COMP reachability score from 6655 to 7008, uniquely solved 202 benchmarks, and converted a multi-day industrial verification timeout into an eight-hour proof. Loop-focused shared-domain intervals balanced cost and benefit well, while higher-precision domains imposed substantial overhead with limited coverage gains.

- *Global Common Subexpression Elimination (GCSE)* achieved dramatic performance improvements when redundancy dominated (e.g., reducing solving time from 105 seconds to under one second in micro-cases) and yielded category-level gains in array- and bitvector-heavy benchmarks. However, it showed neutral or negative impacts in pointer-intensive and large-loop workloads due to sensitivity to aliasing and the granularity of abstract interpretation. The overall runtime overhead was modest, and clear remediation paths were identified.

- *Multi-property verification* improved early fault detection and provided finer-grain feedback by decomposing verification tasks into per-assertion traces. The approach

introduced overhead via additional solver calls, but achieved unique wins in select categories such as *DeviceDrivers*, enabling verification of benchmarks previously prone to solver crashes.

Interoperability The chapter also demonstrated practical tool interoperability via the *GOTO Transcoder*, which enables CBMC-generated GOTO binaries—including those produced by Kani in the Rust Verification Initiative—to be executed by ESBMC. This validates the abstract GOTO language as a viable cross-tool compatibility layer and fosters mutual reuse of front-ends, transformations, and analyses across the CProver ecosystem.

Takeaways Collectively, these experiments affirm that GOTO-centric transformations consistently yield a net positive impact on verification throughput when applied judiciously. Slicing reduces problem sizes, interval analysis prunes infeasible paths and strengthens induction, and GCSE removes redundancies to shrink SSA and SMT formulas. The principal limitations arise from solver heuristic sensitivity, challenges in pointer and alias precision, and the granularity of abstract interpretation analyses. These insights motivate future work exploring block-level abstract interpretation, robust aliasing fallback mechanisms, and adaptive profiling-driven transformation strategies that maximize performance gains while mitigating regressions.

Chapter 6

Conclusions

This chapter concludes the thesis by returning to the central research question posed in Chapter 1: *Is it feasible to develop a language that serves as the basis for a verification framework applicable to all CProver tools?* The work in this thesis provides a clear and affirmative answer. The GOTO language, as formalized and extended here, has proven capable of capturing the semantics of CProver tools, supporting modular transformations, and enabling interoperability across languages and verification engines. The formal semantics defined in Isabelle, along with the experimental validation in ESBMC, demonstrate that the approach is both theoretically sound and practically effective. Therefore, the research question can be answered positively: *it is not only feasible but also advantageous to adopt GOTO as the foundation of a verification framework for the entire CProver ecosystem.*

The contributions of this thesis collectively demonstrate that the GOTO framework is both a theoretical and practical solution to the fragmentation of the CProver ecosystem. Chapter 3 introduced the formal syntax and semantics of GOTO and trace languages, establishing the groundwork for symbolic execution and bounded verification. Chapter 4 extended this foundation by defining GOTO transformations, structured as compiler-inspired passes, that enable optimization and modular reasoning. Chapter 5 validated these contributions in practice, showing through SV-COMP benchmarks and an industrial firmware case study that GOTO-based verification is efficient, scalable, and adaptable.

The remainder of this chapter explores the broader implications of this work and outlines future directions. Section 6.1 revisits the meaning of the CProver framework at the platform and theorem prover levels. Section 6.2.1 discusses the role of compiler-inspired architectures in structuring verification pipelines. Section 6.2.2 emphasizes the potential of GOTO as a standardized intermediate language for verification across languages and tools. Section 6.2.1 highlights how transformations contribute to accelerating software model checking. Section 6.2.2 addresses concurrency as one of the most demanding and promising frontiers for the framework. Section 6.2.3 explores the expansion of CProver to new domains and the challenge of operational model generation. Finally, Section 6.3 identifies

limitations and threats to validity, and Section 6.4 provides closing remarks.

6.1 What is the CProver framework?

The results of this thesis establish that the CProver project should not be understood as a collection of individual verification tools but as a framework centered on the GOTO language. This abstraction provides a unified semantics that underpins symbolic execution, transformation passes, and solver encoding. Unlike earlier ad hoc implementations, the GOTO framework allows us to reason about tool behavior in a principled and reusable way. In this sense, CProver transitions from a fragmented ecosystem to a structured verification architecture.

At the platform level, the framework enables interoperability. For example, the GOTO Transcoder (Section 5.8) demonstrates how CBMC and ESBMC can share verification artifacts without requiring the reimplementing of front-ends. Similarly, domain-specific extensions for Python and Rust demonstrate that new languages can be integrated with minimal effort once the GOTO abstraction is respected. By treating front-ends, transformations, and symbolic execution as common infrastructure, the framework lowers the barrier for new contributors and facilitates hybrid workflows.

At the theorem prover level, Appendix A demonstrates how the Isabelle mechanization offers partial machine-checked guarantees for symbolic execution and trace evaluation. This brings formal assurance to previously informal parts of the toolchain. The mechanization also establishes that CProver is not merely a pragmatic engineering project but one with a solid formal foundation. This dual perspective – framework as platform and framework as prover – captures the essence of what the CProver framework now represents.

6.2 Future work

6.2.1 Moving towards a compiler architecture

One of the central insights of this work is that verification benefits greatly from adopting concepts from compiler infrastructures. Transformations over GOTO, such as unwinding, slicing, and GCSE (Chapter 4.4), mirror optimization passes in LLVM or GCC. Instead of optimizing for performance alone, these passes optimize for verification efficiency and

solver tractability. The compiler-inspired perspective clarifies the structure of the verification pipeline: a front-end translates into GOTO, a series of transformations refines and simplifies it, and a symbolic execution engine produces a trace that is encoded into logic. This modularity makes the verification pipeline transparent and extensible.

Future directions include exploring SSA-based representations, constructing dominator trees, and developing customizable pass pipelines. An SSA-based GOTO could make dependencies explicit through φ-nodes [94], supporting more aggressive slicing and improving the performance of fixpoint analyses (e.g., data-flow and abstract interpretation). Moreover, building a pass manager akin to LLVM [32] would enable researchers to experiment with transformation orderings, thereby facilitating reproducibility and systematic evaluation. By adopting such a compiler architecture, CProver would not only benefit from existing compiler design principles but also contribute back with verification-specific insights.

Accelerating software model checking

The experiments presented in Chapter 5 show that verification performance can be substantially improved through GOTO-level transformations. For example, slicing reduced the size of traces in SV-COMP benchmarks, leading to faster solving times. Similarly, GCSE reduced formula redundancy, improving solver efficiency. These results underscore the importance of viewing acceleration as spanning transformations, symbolic execution, and solver interaction.

Compilers employ optimizations such as Global Value Numbering and dead-code elimination [101], as well as advanced loop optimizations that detect inductive variables. Inspired by these, the framework is well-positioned to incorporate parallel and heterogeneous acceleration. Hybrid workflows that combine symbolic execution with fuzzing or test generation could also benefit from GOTO transformations as a preprocessing stage. In this sense, accelerating software model checking is not only about stronger solvers but also about rethinking the representation and transformation pipeline.

Improving decision procedures

A further direction is the development of improved decision procedures that are co-designed with GOTO semantics. Many current encodings translate program constructs to SAT/SMT formulas at a very low level, often obscuring high-level invariants. Future work could fo-

cus on modular encoders for data structures, concurrency primitives, and memory models, preserving structure rather than flattening it. This may allow solvers to reason more effectively about program intent. For example, encodings of arrays, vectors, or pointer arithmetic already benefit from specialized theories, while concurrency models could leverage partial-order SMT solving. Additionally, proof-producing solvers could be integrated into the framework to provide end-to-end assurance. By aligning decision procedures more closely with the GOTO abstraction, the framework can narrow the semantic gap between source programs and solver input, improving efficiency and trustworthiness.

Language Server Protocol

Although the CProver tools have gained significant traction in the verification community, their usability remains a barrier for new users. The learning curve is steep, and even well-known introductory examples often lead to confusion about verification outcomes and counterexample traces. One promising direction for addressing this challenge is to leverage the *Language Server Protocol* (LSP)[131].

LSP is a standard that decouples language-specific tooling from editors and IDEs. Through a JSON-based protocol, it provides services such as diagnostics, code navigation, and semantic highlighting across different environments. Extending CProver with an LSP backend would enable users to access verification feedback directly within their development workflow, thereby lowering the barrier to entry. Diagnostics could highlight failing assertions, loop unwinding bounds, or unsupported features in real time, while suggested fixes could guide users toward the correct use.

An LSP interface would not only improve accessibility but also bridge the gap between formal verification and everyday programming practice. By showing verification results in a familiar environment, developers can incrementally adopt CProver, making the framework more impactful in both industrial and educational contexts.

Previous efforts in this direction have produced editor-specific plugins, such as an Eclipse plugin for CBMC [132], an Eclipse plugin for ESBMC [133], and a Visual Studio Code extension for ESBMC [134]. However, these plugins require per-editor maintenance and are not easily portable across development environments. By adopting LSP, a single server implementation would provide verification capabilities to any LSP-compatible editor, avoiding the maintenance burden of dedicated plugins for each environment.

6.2.2 GOTO standard language

A key contribution of this thesis is positioning the GOTO language as a standard intermediate representation for verification. By formalizing the syntax and semantics of GOTO, this work elevates it from an implementation detail to a first-class standard. This standardization brings clarity to the semantics of unwinding, symbolic execution, and assertion checking, enabling rigorous reasoning and consistent tool behavior.

The practical impact of this standardization is illustrated by the GOTO Transcoder (Section 5.8), which translates between binary formats used by CBMC and ESBMC. This translation enables one tool to act as a front-end and another as a backend, allowing workflows that leverage their combined strengths. Similarly, the extensions to Rust and Python illustrate how GOTO provides a language-agnostic foundation for verification. Once lowered to GOTO, programs from different languages become comparable and analyzable under the same semantics.

The notion of GOTO as a standard also opens doors to industrial adoption. By defining an accessible and documented intermediate representation, companies can build domain-specific analyzers or optimization passes without committing to a specific tool. This could lead to a verification ecosystem where GOTO plays a role similar to LLVM IR in compiler infrastructures or Boogie in program verification.

Achieving full interoperability in practice requires progress along three complementary directions. First, the compiler-inspired architecture proposed in Section 6.2.1 should mature into an LLVM-inspired pass manager that allows composing arbitrary verification passes; this makes classic compiler optimizations readily available and renders the verification pipeline extensible. Second, GOTO as a standard language should improve algorithm sharing and compatibility across existing CProver tools, while also lowering the barrier for new tools to join the ecosystem by targeting a common specification. Concrete steps include completing the transcoder's incomplete equivalences (e.g., unions, vector types, floating-point intrinsics), defining a stable and versioned GOTO specification, and building a conformance test suite that validates semantic agreement across implementations. Third, extending CProver to new domains, by lowering additional source languages to GOTO, broadens the reach of the framework and exercises its generality.

Interpreter

Developing an interpreter for the GOTO language is a promising extension. Such a component would allow GOTO programs to be executed concretely, symbolically, or in concolic (combined concrete and symbolic) mode. This would bridge the gap between verification and testing: fuzzing engines or dynamic analysis tools could be layered on top of the interpreter to generate execution paths, which are then fed into the verification pipeline. An interpreter would also enable runtime validation of operational models, ensuring that libraries and system calls are faithfully captured. This would broaden the applicability of the framework, aligning with successful precedents such as LLVM or Boogie, where interpreters complement static analyses.

Both CBMC [83] and ESBMC [84] already include basic interpreter components; however, these focus primarily on selecting thread interleavings rather than assigning values to symbolic variables. A more comprehensive interpreter would go further by linking against actual system libraries, enabling concrete execution of counterexamples to increase confidence in reported violations. This direction also opens the possibility of integrating debuggers into the verification workflow, allowing users to step through counterexample traces interactively, and of incorporating dynamic analysis techniques to complement the existing static verification pipeline.

Concurrency

Concurrency remains one of the hardest challenges in software verification. Existing tools, such as CSeq, sequentialize concurrent programs, while ESBMC and CBMC model threads through symbolic intrinsics. These methods provide partial solutions but are limited in scalability. The GOTO framework offers a new angle: concurrency can be tackled not only at the symbolic execution level but also at the representation level. By encoding thread interactions and shared-memory operations explicitly in GOTO, new transformation-based strategies become possible.

Concurrency-specific slicing could eliminate irrelevant interleavings, while partial-order reduction could be implemented as a GOTO pass rather than as a solver heuristic. This would reduce the state space earlier in the pipeline, producing smaller traces before solver encoding. Interval analysis could also be extended to reason about shared variables and infeasible schedules. The demand for concurrency verification is especially strong in embedded and

cyber-physical systems, where safety-critical applications must run on multi-core processors. By integrating concurrency reasoning directly into the GOTO language, the framework could scale verification to larger, more realistic systems.

6.2.3 CProver to more domains

The CProver ecosystem has already expanded beyond C and C++ to support Java, Python, and Rust. The GOTO framework provides a natural pathway for further extension. ESBMC incorporates platform-specific optimizations such as vector operations, `memset` handling, and union modeling, which show how low-level features can be abstracted at the GOTO level. Beyond traditional system languages, integration with Jimple through FlowDroid has shown that Android applications can also be analyzed. The main obstacle in these extensions lies in constructing operational models for external libraries and runtimes. Unlike the core language, which can be lowered uniformly to GOTO, external APIs often require tailored semantic models.

Operational Model Generation

Operational models (OMs) define the semantics of external functions and libraries, and their construction remains one of the most labor-intensive aspects of extending verification tools. Several promising approaches can reduce this burden. Summary-based methods can infer contracts from function signatures or usage patterns, generating approximate models that are later refined. Large language models (LLMs) [135] can be applied to existing codebases to yield draft operational models, which can then be validated and refined in a counterexample-guided abstraction refinement (CEGAR) loop: spurious counterexamples guide refinements until a sound model emerges. These methods could be integrated with the proposed interpreter to enable testing, validation, and refinement of OMs at runtime. Developing systematic methodologies for OM generation is therefore essential for scaling the GOTO framework to new languages and platforms.

6.3 Limitations and Threats to Validity

While this thesis establishes a principled framework for the CProver ecosystem, several limitations must be acknowledged.

Bounded nature of verification. The framework, like CBMC and ESBMC, is rooted in bounded model checking. This introduces the possibility of both false negatives and false positives. False negatives arise when the unwinding bound does not reach an assertion, causing it to hold vacuously. False positives may result from over-approximations in abstract domains, errors in the translation from the source language to the GOTO IR, undefined behavior in the input program, or bugs in the verification tools themselves.

Partial formalization. The Isabelle mechanization (Appendix A) provides a partial formalization. Isabelle automatically verified the totality and termination of the core evaluation functions `EvalExpr` and `EvalTrace` (via the `fun` keyword), and the load-after-store property of the memory model was proved as a theorem. However, the termination of the symbolic execution function `symexKIt` is assumed (via `sorry`) rather than proved, as it requires a well-founded measure that has not yet been constructed. Transformation correctness proofs (e.g., for the trace slicer), concurrency semantics, and extended memory models remain beyond the current scope.

Proof obligations. Several proof obligations arise naturally from the formalization and remain as future work:

1. *Termination of trace generation*: proving that `symexKIt` terminates requires constructing a well-founded measure, likely a lexicographic combination of the unwinding bound k and the remaining program length.

2. *Soundness of trace generation*: if `EvalTrace` returns $\top$ (unsafe), then there must exist a concrete execution of the GOTO program that violates the corresponding assertion.

3. *Completeness up to bound*: if a concrete violation exists within bound k, the generated trace must capture it.

4. *Transformation preservation*: proving that transformations such as trace slicing preserve assertion outcomes. The trace slicer is a natural first candidate, given that the trace evaluation semantics are already formalized.

Concurrency support. Although concurrency is identified as a major use case, the framework primarily addresses sequential programs. Integrating concurrency semantics and reduction strategies remains future work.

Experimental scope. The evaluation focuses on SV-COMP benchmarks and selected case studies. While representative, they cannot capture the full diversity of software systems. Trade-offs between precision and runtime were also observed.

Engineering constraints. The CProver ecosystem is under active development, and tool behavior may evolve independently of this work. Long-term reproducibility will require continuous maintenance of the framework.

Despite these limitations, the thesis demonstrates that a unified GOTO-based framework is both feasible and beneficial. The limitations identified provide a roadmap for future extensions.

6.4 Concluding remarks

This thesis has presented a unified framework for CProver, centered on the GOTO language, supported by compiler-inspired transformations, and grounded in formal semantics. By formalizing the intermediate representation, defining reusable transformation passes, and validating them through experiments and mechanization, the work transforms CProver from a set of loosely connected tools into a principled verification ecosystem.

The framework has already proven useful in multiple contexts. Through the experiments with SV-COMP benchmarks, we demonstrated measurable performance improvements. Through the Intel firmware case study, we showed scalability to industrial software. Through the GOTO Transcoder, we established interoperability between CBMC and ESBMC. Through the Isabelle mechanization, we provided formal assurance in the soundness of symbolic execution. These contributions, taken together, position CProver as a credible platform for both research and practice.

Looking ahead, the future of the CProver framework lies in three directions: extending concurrency support, accelerating verification through parallel and heterogeneous architectures, and establishing GOTO as a broadly recognized verification standard. Each of these directions builds directly on the foundations laid in this thesis.

References

[1] S. K. Singh and A. Singh, *Software testing*. Vandana Publications, 2012 (cited on p. 18).

[2] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009 (cited on p. 18).

[3] M. Research. "A proactive approach to more secure code." (2025), [Online]. Available: `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code` (visited on 04/18/2024) (cited on p. 18).

[4] "BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE," The White House, Tech. Rep., Feb. 2024 (cited on p. 18).

[5] "Post office horizon scandal: Why hundreds were wrongly prosecuted," en, *BBC*, Jan. 2024. [Online]. Available: `https://www.bbc.co.uk/news/articles/c1wpp4w14pqo` (cited on p. 18).

[6] Ministry of Justice, A. Chalk, and R. Sunak, "Government to quash wrongful post office convictions," en, *GOV.UK*, Jan. 2024. [Online]. Available: `https://www.gov.uk/government/news/government-to-quash-wrongful-post-office-convictions` (cited on p. 18).

[7] A. Lampert and R. K. Singh. "Southwest network failure raises concerns over system's strength." (2023), [Online]. Available: `https://www.reuters.com/business/aerospace-defense/southwest-network-failure-raises-concerns-over-systems-strength-2023-04-19/` (visited on 06/14/2024) (cited on p. 18).

[8] M. J. Harrold, “Testing: A roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 61–72 (cited on p. 18).

[9] G. J. Myers *et al.*, *The art of software testing*. Wiley Online Library, 2004, vol. 2 (cited on pp. 18, 94).

[10] D. S. Alberts, “The economics of software quality assurance,” in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, 1976, pp. 433–442 (cited on p. 18).

[11] Tricentis. “2025 Quality Transformation Report.” (), [Online]. Available: `https://www.tricentis.com/resources/quality-transformation-report` (cited on p. 18).

[12] A. Turing, “Checking a large routine,” in *The early British computer conferences*, 1989, pp. 70–72 (cited on p. 18).

[13] 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. 18).

[14] P. Cousot, *Principles of Abstract Interpretation*. MIT Press, 2021 (cited on pp. 18, 33, 47, 81, 84).

[15] D. Beyer, “State of the art in software verification and witness validation: SV-COMP 2024,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2024, pp. 299–329 (cited on pp. 18, 69, 98, 103–105).

[16] B. Martin *et al.*, “2011 CWE/SANS top 25 most dangerous software errors,” *Common Weakness Enumeration*, vol. 7515, 2011 (cited on p. 18).

[17] D. Kroening. “CProver website.” (2025), [Online]. Available: `https://www.cprover.org/` (visited on 04/01/2025) (cited on pp. 19, 35).

[18] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, Springer, 2004, pp. 168–176 (cited on pp. 19, 36, 47).

[19] D. Kroening and M. Tautschnig, "CBMC-C Bounded Model Checker: (Competition Contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, Springer, 2014, pp. 389–391 (cited on pp. 19, 21, 35, 68).

[20] N. Chong *et al.*, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 11–20, ISBN: 9781450371230. DOI: 10.1145/3377813.3381347. [Online]. Available: https://doi.org/10.1145/3377813.3381347 (cited on pp. 19, 42, 129).

[21] R. S. Menezes *et al.*, "ESBMC v7. 4: Harnessing the Power of Intervals: (Competition Contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2024, pp. 376–380 (cited on pp. 19, 21, 25, 27, 35, 80, 98).

[22] P. Schrammel and D. Kroening, "2LS for Program Analysis: (Competition Contribution)," in *International Conference on Tools and*

Algorithms for the Construction and Analysis of Systems, Springer, 2016, pp. 905–907 (cited on pp. 19, 35, 39, 40).

[23] B. Fischer, O. Inverso, and G. Parlato, "CSeq: A concurrency pre-processor for sequential C verification tools," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 710–713 (cited on pp. 19, 39, 40, 44, 67).

[24] T. Wu *et al.*, "Verifying components of ARM® confidential computing architecture with esbmc," in *International Static Analysis Symposium*, Springer, 2024, pp. 451–462 (cited on p. 19).

[25] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and Verilog programs using bounded model checking," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 368–371 (cited on pp. 19, 38, 39).

[26] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*, Springer, 2004, pp. 5–23 (cited on pp. 19, 67, 100).

[27] D. Kroening *et al.*, "Loop summarization using abstract transformers," in *International Symposium on Automated Technology for Verification and Analysis*, Springer, 2008, pp. 111–125 (cited on pp. 20, 39, 40).

[28] 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, 2011 (cited on pp. 20, 36, 39, 40, 44, 45, 47, 60, 66, 69–71, 101).

[29] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *Interna-*

tional journal on software tools for technology transfer, vol. 19, no. 1, pp. 97–114, 2017 (cited on pp. 20, 35, 40, 64, 74, 87).

[30] R. S. Menezes, "Geração de casos de teste usando bounded model checking," 2021 (cited on p. 20).

[31] M. Barnett *et al.*, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, Springer, 2005, pp. 364–387 (cited on p. 21).

[32] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86 (cited on pp. 21, 139).

[33] R. S. Menezes *et al.*, "VO-GCSE: Verification Optimization through Global Common Subexpression Elimination," in *ACM International Conference on the Foundations of Software Engineering (FSE 2025)*, Association for Computing Machinery, 2025 (cited on pp. 25, 27, 42, 122).

[34] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002 (cited on pp. 26, 28–30, 32, 46).

[35] R. Menezes *et al.*, "ESBMC-Jimple: verifying Kotlin programs via Jimple intermediate representation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 777–780 (cited on pp. 27, 45, 69, 91).

[36] B. Farias *et al.*, "ESBMC-Python: A Bounded Model Checker for Python Programs," in *Proceedings of the 33rd ACM SIGSOFT Inter-*

national Symposium on Software Testing and Analysis, 2024, pp. 1836–1840 (cited on pp. 27, 42, 45).

[37] C. Wei *et al.*, "ESBMC v7.7: Automating Branch Coverage Analysis Using CFG-Based Instrumentation and SMT Solving," in *Fundamental Approaches to Software Engineering*, A. Boronat and G. Fraser, Eds., Cham: Springer Nature Switzerland, 2025, pp. 281–286, ISBN: 978-3-031-90900-9 (cited on pp. 27, 93).

[38] E. Albert *et al.*, "Operational semantics for declarative multi-paradigm languages," *Journal of Symbolic Computation*, vol. 40, no. 1, pp. 795–829, 2005 (cited on p. 28).

[39] H. C. Kennedy, "Peano's concept of number," *Historia Mathematica*, vol. 1, no. 4, pp. 387–408, 1974 (cited on pp. 29, 166).

[40] R. Bird, *Introduction to functional programming using Haskell*. Pearson Educación, 1998 (cited on p. 30).

[41] H. B. Curry, *Foundations of mathematical logic*. Courier Corporation, 1977 (cited on pp. 30, 166).

[42] A. Church, "An unsolvable problem of elementary number theory," *American journal of mathematics*, vol. 58, no. 2, pp. 345–363, 1936 (cited on p. 31).

[43] A. M. Turing, "Computability and λ-definability," *The Journal of Symbolic Logic*, vol. 2, no. 4, pp. 153–163, 1937 (cited on p. 31).

[44] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016 (cited on pp. 32, 45).

[45] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977 (cited on p. 33).

[46] D. Kroening and E. Clarke, "Checking consistency of C and Verilog using predicate abstraction and induction," in *Proceedings of ICCAD*, IEEE, Nov. 2004, pp. 66–72 (cited on pp. 33, 34, 39, 44).

[47] *CWE - VIEW SLICE: CWE-1430: Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses (4.17) — cwe.mitre.org*, `https://cwe.mitre.org/data/slices/1430.html`, [Accessed 08-09-2025] (cited on p. 33).

[48] *CWE - VIEW SLICE: CWE-1154: Weaknesses Addressed by the SEI CERT C Coding Standard (4.17) — cwe.mitre.org*, `https://cwe.mitre.org/data/slices/1154.html`, [Accessed 08-09-2025] (cited on p. 33).

[49] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176, ISBN: 978-3-540-24730-2 (cited on pp. 33, 39, 44).

[50] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008 (cited on pp. 33, 35).

[51] D. Basin, "The cyber security body of knowledge," *University of Bristol,, ch. Formal Methods for Security, version..[Online..*, 2021 (cited on pp. 33, 34).

[52] K. R. Apt, F. S. De Boer, and E.-R. Olderog, *Verification of sequential and concurrent programs*. Springer, 2009 (cited on p. 34).

[53] I. Beer *et al.*, "Efficient detection of vacuity in temporal model checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, 2001 (cited on p. 35).

[54] E. M. C. Jr. *et al.*, *Model Checking, second edition*. London, England: The MIT Press, 2017 (cited on p. 35).

[55] J. R. Burch *et al.*, "Symbolic model checking: 1020 states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992 (cited on p. 36).

[56] A. Biere *et al.*, "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. 36).

[57] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 794–801 (cited on p. 36).

[58] K. Y. Rozier *et al.*, "Moxi: An intermediate language for symbolic model checking," in *International Symposium on Model Checking Software*, Springer, 2024, pp. 26–46 (cited on p. 36).

[59] K. Havelund and T. Pressburger, "Model checking Java programs using Java Pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000 (cited on p. 36).

[60] G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, vol. 65, pp. 77–108, 2005 (cited on p. 36).

[61] D. Kroening, "Application specific higher order logic theorem proving," in *Proc. of the Verification Workshop-VERIFY*, vol. 2, 2002, pp. 5–15 (cited on pp. 36, 38, 39, 65, 78).

[62] S. Owre, J. Rushby, and N. Shankar, "Integration in PVS: Tables, types, and model checking," in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 1997, pp. 366–383 (cited on p. 36).

[63] S. Berezin, *Model checking and theorem proving: a unified framework*. Carnegie Mellon University, 2002 (cited on p. 36).

[64] D. Kroening, *Email to the author*, Personal communication, September 1, 2025, 2025 (cited on pp. 36, 37).

[65] R. Rohleder, "Hands-on ghidra-a tutorial about the software reverse engineering framework," in *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, pp. 77–78 (cited on p. 37).

[66] R. Menezes, *Early cbmc archive*, Sep. 2025. DOI: 10.5281/zenodo.17226508. [Online]. Available: https://doi.org/10.5281/zenodo.17226508 (cited on p. 37).

[67] D. Kroening, "Computing over-approximations with bounded model checking," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 1, pp. 79–92, 2006 (cited on p. 39).

[68] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with wolverine," in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 573–578 (cited on pp. 39, 40).

[69] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with impact," in *2013 Formal Methods in Computer-Aided Design*, IEEE, 2013, pp. 210–217 (cited on pp. 39, 40).

[70] L. Yin *et al.*, "YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement: (Competition Contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2018, pp. 422–426 (cited on pp. 39, 40).

[71] L. Cordeiro *et al.*, "JBMC: A bounded model checking tool for verifying Java bytecode," in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 183–190 (cited on pp. 39, 41, 69, 101).

[72] K. M. Alshmrany *et al.*, "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. 39, 41).

[73] *Kani rust verifier*, 2025. [Online]. Available: `https://github.com/model-checking/kani` (cited on pp. 39, 41, 42, 130).

[74] F. K. Aljaafari *et al.*, "Combining bmc and fuzzing techniques for finding software vulnerabilities in concurrent programs," *Ieee Access*, vol. 10, pp. 121 365–121 384, 2022 (cited on pp. 39, 41).

[75] R. S. Menezes, *GOTO Transcoder*, 2025. [Online]. Available: `https://github.com/rafaelsamenezes/goto-transcoder` (visited on 02/03/2025) (cited on pp. 39, 129).

[76] N. Tihanyi *et al.*, "A new era in software security: Towards self-healing software via large language models and formal verification," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, IEEE, 2025, pp. 136–147 (cited on pp. 39, 41).

[77] *Expanding the Rust Formal Verification Ecosystem: Welcoming ESBMC*, `https://rustfoundation.org/media/expanding-the-rust-formal-verification-ecosystem-welcoming-esbmc/`, [Accessed 29-09-2025] (cited on pp. 42, 129).

[78] Diffblue. "Diffblue website." (2025), [Online]. Available: `https://www.diffblue.com/` (visited on 04/01/2025) (cited on p. 42).

[79] Veribee. “Veribee website.” (2025), [Online]. Available: https://www.veribee.co/ (visited on 04/01/2025) (cited on p. 42).

[80] *Byte Repair - Secure Code Analysis & Repair — byterepair.io*, https://byterepair.io/, [Accessed 08-09-2025] (cited on p. 42).

[81] H. Rocha *et al.*, “Model Checking Embedded C Software Using k-Induction and Invariants,” in *Embedded Software Verification and Debugging*, D. Lettnin and M. Winterholer, Eds. New York, NY: Springer New York, 2017, pp. 159–182, ISBN: 978-1-4614-2266-2. DOI: 10.1007/978-1-4614-2266-2_7. [Online]. Available: https://doi.org/10.1007/978-1-4614-2266-2_7 (cited on p. 44).

[82] C. Barrett, A. Stump, C. Tinelli, *et al.*, “The smt-lib standard: Version 2.0,” in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, vol. 13, 2010, p. 14 (cited on pp. 44, 101).

[83] D. Kroening, *CBMC: C Bounded Model Checker*, 2024. [Online]. Available: https://github.com/diffblue/cbmc (visited on 11/26/2024) (cited on pp. 45, 68, 72, 142).

[84] L. C. Cordeiro, *ESBMC: The efficient SMT-based context-bounded model checker (ESBMC)*, 2024. [Online]. Available: https://github.com/esbmc/esbmc (visited on 11/26/2024) (cited on pp. 45, 68, 72, 99, 142).

[85] D. E. Knuth, “Literate programming,” *The computer journal*, vol. 27, no. 2, pp. 97–111, 1984 (cited on p. 46).

[86] S. Kaes, “Parametric overloading in polymorphic programming languages,” in *ESOP ’88*, H. Ganzinger, Ed., Berlin, Heidelberg: Springer

Berlin Heidelberg, 1988, pp. 131–144, ISBN: 978-3-540-38941-5 (cited on p. 46).

[87] K. Chen, P. Hudak, and M. Odersky, "Parametric type classes," in *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, 1992, pp. 170–181 (cited on p. 46).

[88] C. A. Gunter, *Semantics of programming languages: structures and techniques*. MIT press, 1992 (cited on p. 47).

[89] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001 (cited on p. 47).

[90] D. Kroening and O. Strichman, *Decision procedures*. Springer, 2016 (cited on p. 47).

[91] J. Morse *et al.*, "Model checking LTL properties over C programs with bounded traces," *Software and Systems Modeling*, n–a, 2013 (cited on pp. 47, 101).

[92] R. Baldoni *et al.*, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018 (cited on p. 58).

[93] S. Khurshid, C. S. PĂsĂreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*, H. Garavel and J. Hatcliff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568, ISBN: 978-3-540-36577-8. DOI: 10.1007/3-540-36577-X_40.

[Online]. Available: http://dx.doi.org/10.1007/3-540-36577-X_40 (cited on p. 58).

[94] V. C. Sreedhar and G. R. Gao, "A linear time algorithm for placing ϕ-nodes," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 62–73 (cited on pp. 60, 139).

[95] E. Clarke *et al.*, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, pp. 7–34, 2001 (cited on p. 64).

[96] S. Christey *et al.*, "Common weakness enumeration," *Mitre Corporation*, 2013 (cited on p. 67).

[97] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 331–340 (cited on pp. 68, 101).

[98] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *International Conference on Computer Aided Verification*, Springer, 2013, pp. 141–157 (cited on p. 68).

[99] M. Ramalho Gadelha *et al.*, "Scalable and precise verification based on k-induction, symbolic execution and floating-point theory," Ph.D. dissertation, University of Southampton, 2019 (cited on p. 68).

[100] J. Morse, "Expressive and efficient bounded model checking of concurrent software," Ph.D. dissertation, University of Southampton, 2015 (cited on pp. 68, 78).

[101] A. Alfred V *et al.*, *Compilers-principles, techniques, and tools*. pearson Education, 2007 (cited on pp. 74, 81, 88–90, 139).

[102] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252 (cited on p. 81).

[103] M. Sintzoff, “Calculating properties of programs by valuations on specific models,” *ACM SIGPLAN Notices*, vol. 7, no. 1, pp. 203–207, 1972 (cited on p. 81).

[104] G. A. Kildall, “A Unified Approach to Global Program Optimization,” in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL ’73, Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 194–206, ISBN: 9781450373494. DOI: 10.1145/512927.512945. [Online]. Available: https://doi.org/10.1145/512927.512945 (cited on pp. 81, 89).

[105] R. E. Moore, *Interval analysis*. Prentice-Hall Englewood Cliffs, 1966, vol. 4 (cited on p. 84).

[106] R. E. Moore, *Methods and applications of interval analysis*. SIAM, 1979 (cited on p. 84).

[107] W. Kahan, “IEEE standard 754 for binary floating-point arithmetic,” *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996 (cited on p. 84).

[108] S. Sankaranarayanan, F. Ivančić, and A. Gupta, “Program analysis using symbolic ranges,” in *International Static Analysis Symposium*, Springer, 2007, pp. 366–383 (cited on p. 84).

[109] M. Müller-Olm and H. Seidl, "Analysis of modular arithmetic," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 5, 29–es, 2007 (cited on p. 84).

[110] B. Blanchet *et al.*, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 196–207 (cited on p. 84).

[111] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 46–59 (cited on p. 84).

[112] E. Isaacson and H. B. Keller, *Analysis of numerical methods*. Courier Corporation, 2012 (cited on p. 84).

[113] G. Gange *et al.*, "Interval analysis and machine arithmetic: Why signedness ignorance is bliss," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 1, pp. 1–35, 2015 (cited on p. 85).

[114] M. A. Aldughaim, "Interval Analysis and Methods in Software Analysis," Ph.D. dissertation, University of Manchester, 2024 (cited on p. 85).

[115] J. Cocke, "Global Common Subexpression Elimination," *SIGPLAN Not.*, vol. 5, no. 7, pp. 20–24, Jul. 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808480. [Online]. Available: https://doi.org/10.1145/390013.808480 (cited on p. 88).

[116] M. S. Hecht and J. D. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM Journal on Computing*, vol. 4, no. 4, pp. 519–532, 1975 (cited on p. 88).

[117] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95, San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61, ISBN: 0897916921. DOI: 10.1145/199448.199462. [Online]. Available: https://doi.org/10.1145/199448.199462 (cited on p. 89).

[118] M. S. Hecht and J. D. Ullman, "Analysis of a simple algorithm for global data flow problems," in *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1973, pp. 207–217 (cited on pp. 89, 90).

[119] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 158–171, 1976 (cited on p. 90).

[120] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005 (cited on p. 94).

[121] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 1, pp. 1–29, 2019 (cited on pp. 98, 105).

[122] LLVM. "Clang language extensions." (2025), [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html (visited on 04/01/2025) (cited on p. 100).

[123] *ISO/IEC 9899:1999: Programming Languages - C*, https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, International Organization for Standardization, ISO/IEC, 1999 (cited on p. 100).

[124] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, Springer, 2009, pp. 174–177 (cited on p. 101).

[125] D. Beyer, "Competition on software verification and witness validation: SV-COMP 2023," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2023, pp. 495–522 (cited on p. 105).

[126] W. Futral *et al.*, "Fundamental principles of intel® txt," *Intel® Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*, pp. 15–36, 2013 (cited on p. 116).

[127] L. Gwennap, "P6 microcode can be patched," *Microprocessor Report*, 1997 (cited on p. 116).

[128] N. Tihanyi *et al.*, "The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2023, San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 33–43, ISBN: 9798400703751. DOI: 10.1145/3617555.3617874. [Online]. Available: https://doi.org/10.1145/3617555.3617874 (cited on p. 122).

[129] N. Tihanyi *et al.*, "How secure is AI-generated code: a large-scale comparison of large language models," *Empirical Software Engineering*, vol. 30, no. 47, 2025. DOI: 10.1007/s10664-024-10590-

1. [Online]. Available: https://doi.org/10.1007/s10664-024-10590-1 (cited on p. 122).

[130] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992. DOI: 10.1109/2.161279 (cited on p. 129).

[131] N. Gunasinghe and N. Marcus, *Language server protocol and implementation*. Springer, 2021 (cited on p. 140).

[132] Diffblue, *Eclipse-CBMC: Eclipse plugin for CBMC*, Accessed: 2025, 2018. [Online]. Available: https://github.com/diffblue/eclipse-cbmc (cited on p. 140).

[133] L. Cordeiro, "SMT-based bounded model checking of multi-threaded software in embedded systems," Ph.D. dissertation, University of Southampton, 2011 (cited on p. 140).

[134] ESBMC, *VSCode-ESBMC: Visual Studio Code extension for ESBMC*, Accessed: 2025, 2024. [Online]. Available: https://github.com/esbmc/vscode-esbmc (cited on p. 140).

[135] E. Kasneci *et al.*, "ChatGPT for good? On opportunities and challenges of large language models for education," *Learning and individual differences*, vol. 103, p. 102 274, 2023 (cited on p. 143).

Appendices

Appendix A

GOTO formalization in Isabelle

This appendix presents an Isabelle implementation of the concepts proposed in Chapter 3. Since it is expected that the reader may not be familiar with Isabelle or proof assistants in general, the material is structured in two main parts. The first part introduces Isabelle/HOL for readers without prior exposure to theorem provers, while the second part provides a formalization of trace semantics for symbolic execution. The objective is not to provide a full tutorial on Isabelle, but rather to walk through the specific definitions and constructs employed in this work.

Before proceeding, it is useful to highlight some notes on Isabelle's syntax and design. Isabelle supports ligatures and TeX-like notation as part of its input language. For instance, typing `Rightarrow` renders the logical symbol ⇒, which can denote either implication or mapping, while similar shorthand exists for the classical logical operators. To keep the exposition simple and readable, we will present these constructs in an ASCII style throughout this appendix; however, the accompanying source code, which is available with this thesis, makes use of Isabelle's full notation. Furthermore, although the Isabelle code is semantically equivalent to the algorithms and definitions discussed in earlier chapters, its form sometimes diverges from the imperative style shown previously. These differences arise either to better fit Isabelle's declarative framework or to facilitate the construction of proofs. Finally, it should be emphasized again that this appendix is intended as a guided walkthrough of the specific formalization used in this thesis, rather than a comprehensive Isabelle manual.

A.1 Isabelle 101

To follow the examples, readers should first download and install the latest release of Isabelle from `https://isabelle.in.tum.de/`. At the time of writing, all examples were tested with Isabelle2025. The starting point in Isabelle is to create a *theory*, which serves as a modular unit analogous to a source file. For illustration, we create a simple theory called

`Hello` that imports `Main`, the standard library. This minimal skeleton opens and closes with the `begin` and `end` keywords.

```
theory Hello
  imports Main
begin
end
```

To introduce some basic concepts, we can define natural numbers in the Peano style [39] using an inductive datatype with two constructors: `Zero` for the base case, and `Suc` for the successor. This allows us to represent numbers as follows: `Zero` corresponds to 0, `Suc Zero` corresponds to 1, and `Suc (Suc Zero)` corresponds to 2. Building on this definition, we can implement addition recursively via a function `myadd`, which is defined by structural recursion on the second argument. It is worth noting that functions in Isabelle are automatically curried [41], meaning that `myadd` can be seen as a function that takes one argument and returns another function awaiting the second argument. Isabelle also enforces that all recursive functions are terminating. By using the `fun` keyword, Isabelle is able to discharge the termination proof automatically based on the inductive structure of the datatype.

```
datatype MyNat = Zero | Suc MyNat

fun myadd :: "MyNat => MyNat => MyNat" where
  "myadd n Zero = n"
| "myadd n (Suc y) = Suc (myadd n y)"
```

As another example, consider the function `greater_or_equal`, which compares two natural numbers. Its definition covers three cases: any number is greater or equal to `Zero`, `Zero` is not greater or equal to a successor, and otherwise the comparison is reduced recursively by stripping one successor from both arguments.

```
fun greater_or_equal :: "MyNat => MyNat => bool" where
  "greater_or_equal n Zero = True"
| "greater_or_equal Zero (Suc n) = False"
| "greater_or_equal (Suc x) (Suc y) = greater_or_equal x y"
```

A.1.1 Proving by induction

Having introduced datatypes and recursive functions, we can now illustrate proofs by induction in Isabelle. As a simple exercise, consider the lemma stating that for all natural numbers x and y, the inequality $x + y \geq x$ holds. We first declare this as a lemma and observe the proof state generated by Isabelle, which shows the goal to be proven. To solve it, we initiate an induction proof on x. Isabelle then produces two subgoals: one for the base case, where x is `Zero`, and one for the inductive step, where the claim must hold for `Suc n` assuming it holds for n. This follows the standard structure of a structural induction proof.

```
lemma ge_lemma: "greater_or_equal (myadd x y) x = True"
```

In Isabelle, we can see the proof state in the side panel:

```
proof (prove)
goal (1 subgoal):
 1. greater_or_equal (myadd x y) x =
    True
```

Right now, it tells us the goal of what we need to prove. From here we can start defining our proof: contradiction, induction, etc. Isabelle provides a quick and automatic way to help us use some strategies. For example, lets use induction over “x” for this proof:

```
lemma ge_lemma: "greater_or_equal (myadd x y) x = True"
proof (induction x)
```

The proof state now changes into:

```
proof (state)
goal (2 subgoals):
 1. greater_or_equal (myadd Zero y)
     Zero =
    True
 2.  x. greater_or_equal (myadd x y)
           x =
         True
         greater_or_equal
          (myadd (MyNat.Suc x) y)
          (MyNat.Suc x) =
```

```
12          True
```

Isabelle is now saying that we must prove that:

1. The lemma holds for when "x" is Zero (base case).
2. The lemma holds for an arbitrary "x" means that it also holds for the $x + 1$.

We can now just create the proof structure, we can avoid doing the full proof by using the keyword `sorry`. It means to Isabelle just assume that path is correct, this is useful to sketch proofs. For example:

```
1 lemma ge_lemma: "greater_or_equal (myadd x y) x = True"
2 proof (induction x)
3   case Zero
4   then show ?case
5     sorry
6 next
7   case (Suc n)
8   then show ?case
9     sorry
10 qed
```

Explanation:

- `proof (induction x)` starts a structural induction on `x`;
- The `Zero` case is the base case;
- The `Suc n` case uses the inductive hypothesis.

For the base case, we can just use the definition of the functions:

```
1 case Zero
2 then show ?case
3   apply(simp)
4   done
```

Now, for the inductive step. The simplification is not enough:

```
case (Suc n)
then show ?case
  apply(simp)
```

This yields the proof state:

```
proof (prove)
goal (1 subgoal):
 1. greater_or_equal (myadd n y) n
    greater_or_equal
     (myadd (MyNat.Suc n) y)
     (MyNat.Suc n)
```

The difficult here is that in our addition we only defined the computation through the second argument. Now Isabelle is not sure how to use "y". The simplest way to solve this is to construct a new lemma, that addition is commutative.

```
lemma add_commutative[simp]:  "myadd x y = myadd y x"
  sorry
```

Note that we used `[simp]` after the definition of the lemma, this allows Isabelle to use this lemma by default while applying *simp*. Our previous proof is now updated with this new lemma:

```
goal:
No subgoals!
```

This means that the new lemma was enough to construct the proof. Of course, this proof is only valid as long as the `myadd` procedure is commutative. For a real example of how Isabelle does this proof, you can follow the definition of naturals (hold `Ctrl` and click on the type). Isabelle definition is quite similar to the one we proposed.

A.2 GOTO

With the preliminaries in place, we now turn to the formalization of the GOTO language in Isabelle. The theory file begins by importing the `Main` library, which already includes fundamental datatypes such as natural numbers, Booleans, and lists. We then extend this basis by defining the additional structures required for our semantics. These include symbolic

values, arrays represented as functions from natural numbers to elements, and environments mapping symbols to arrays. Example definitions illustrate how to construct lists, arrays, and environments.

```
theory Goto
  imports Main
begin
```

Considering the preliminaries: *Naturals*, *Booleans*, and *Lists* are already available. We then, define the missing elements:

```
value "0::nat" (* Shows 0 *)
value "False::bool" (* Shows False *)

(* Lists *)
definition my_list :: "nat list" where
  "my_list = 1 # 2 # 3 # []"

(* Symbol *)
datatype Symbol = Symbol nat

(* Array *)
type_synonym 'a array = "nat   'a"

definition my_array :: "nat   nat" where
  "my_array x = (if x = 1 then 0 else 1)"

value "my_array 1" (* Shows 0 *)

(* Environment *)
type_synonym ev = "Symbol   nat array"
```

A.2.1 Trace Expressions

Trace expressions represent the symbolic terms manipulated during execution. We define a datatype `TraceExpression` with constructors for Boolean constants, logical operators,

comparisons, integer constants, arithmetic, arrays, and conditionals. Each of these expressions can be evaluated relative to an environment using the function `EvalExpr`. Its definition specifies how to interpret each form of expression, returning arrays as values. For example, Boolean constants are mapped to arrays containing 0 or 1, arithmetic expressions are evaluated modulo a given width, and conditionals (`ITE`) branch on the value of a guard expression.

```
datatype TraceExpression = ConstantBool bool
  | And TraceExpression TraceExpression
  | Not TraceExpression
  | GreaterThan TraceExpression TraceExpression
  | Constant nat nat
  | Add TraceExpression TraceExpression nat
  | ConstantArray "nat array"
  | Select TraceExpression TraceExpression
  | With TraceExpression TraceExpression TraceExpression
  | ITE TraceExpression TraceExpression TraceExpression
  | SymbolValue TraceExpression
```

We can now define the `EvalExpr` procedure:

```
fun EvalExpr :: "TraceExpression  ev  nat array" where
  "EvalExpr (SymbolValue te) env = env (Symbol ((EvalExpr
   te env) 0))"
| "EvalExpr (Constant n width) _ = ToArray (n mod width)"
| "EvalExpr (Add e0 e1 width) env = ToArray ((
    ((EvalExpr e0 env) 0) +
    ((EvalExpr e1 env) 0)) mod width)"
| "EvalExpr (ConstantBool b) env = ToArray (if b then 1
   else 0)"
| "EvalExpr (Not te) env = ToArray
    (if ((EvalExpr te env) 0) = 0 then 1 else 0)"
| "EvalExpr (And e0 e1 ) env = ToArray ( if
    ((EvalExpr e0 env) 0  0)
    ((EvalExpr e1 env) 0  0) then 1 else 0)"
| "EvalExpr (GreaterThan e0 e1) env = ToArray
    (if ((EvalExpr e0 env) 0) > ((EvalExpr e1 env) 0) then
```

```
   1 else 0)"
| "EvalExpr (ConstantArray a) env = a"
| "EvalExpr (With a i u) env =
  (EvalExpr a env)((EvalExpr i env 0) := (EvalExpr u env 0)
   )"
| "EvalExpr (Select a i) env = ToArray ((EvalExpr a env) ((
   EvalExpr i env) 0))"
| "EvalExpr (ITE cond et ef) env =
    (if (EvalExpr cond env 0)  0  then (EvalExpr et env)
   else (EvalExpr ef env) )"
```

A.2.2 Trace Statements

Building on expressions, we next introduce trace statements, which include assignments, assumptions, and assertions. The evaluation of a trace is defined by the function `EvalTrace`, which processes a list of statements in sequence relative to an environment. Assignments update the environment by binding a symbol to the value of an expression, assumptions restrict the execution to cases where the guard evaluates to true, and assertions check properties, returning success or failure accordingly. To illustrate the semantics, we reconstruct the example from Chapter 3, defining a simple trace and evaluating it under two different environments. This demonstrates how assertions may succeed or fail depending on the symbolic values provided.

```
datatype TraceStmts = TAssign Symbol TraceExpression
  | TAssume TraceExpression
  | TAssert TraceExpression
```

Finally for the trace, we can define the `EvalTrace`:

```
fun EvalTrace :: "TraceStmts list  ev  bool" where
  "EvalTrace [] _ = False"
| "EvalTrace ((TAssign s te) # xs) env =
  EvalTrace xs (env(s := ToArray (EvalExpr te env 0)))"
| "EvalTrace ((TAssume te) # xs) env =
  (if (EvalExpr te env 0) = 0 then False else EvalTrace xs
   env)"
```

```
| "EvalTrace ((TAssert te) # xs) env =
  (if (EvalExpr te env 0) = 0 then True else EvalTrace xs
   env)"
```

We can now recreate the same Example from Chapter 3.

```
definition my_trace :: "traceStmt list" where
  "my_trace = (TAssign (Symbol 1)
                (Add
                  (SymbolValue (Constant 0 32))
                  (Constant 1 32) 32))
              # (TAssert (GreaterThan (Constant 9 32) (
   SymbolValue (Constant 1 32))) ) # []"
```

With both environments.

```
definition environment1 :: ev where
  "environment1 = (s . (x . 5))"

definition environment2 :: ev where
  "environment2 = (environment1 (Symbol 0 := x.10))"
```

We can use the environments to evaluate the trace

```
value "EvalTrace my_trace environment1" (* False *)
value "EvalTrace my_trace environment2" (* True *)
```

A.2.3 GOTO Statements

We then extend the formalization from traces to GOTO statements, which provide a higher-level representation including conditionals (`IfThenGoto`) alongside assignments, assumptions, and assertions. Loops are introduced as pairs of program locations, though their implementation is simplified here for clarity. Renumbering mechanisms are also formalized, allowing us to track variable updates symbolically using conditional expressions. This enables us to reason about different execution paths and construct equivalent symbolic representations.

```
datatype GotoStmt = Assign Symbol TraceExpression
```

```
  | Assume TraceExpression
  | Assert TraceExpression
  | IfThenGoto traceExpression nat
```

Loops

```
datatype Loop = L nat nat

fun getLoops :: "gotoStmt list  Loop list" where
  "getLoops [] = []"
| "getLoops ((IfThenGoto _ _) # xs) = []"
| "getLoops xs = []"
```

Renumbering

```
datatype renumbering = R "symbol  (traceExpression × symbol
   ) list"
definition defaultRenumber :: renumbering where
"defaultRenumber = R (s. [])"

fun updateSymbol :: "renumbering  symbol  symbol
   traceExpression  renumbering" where
"updateSymbol (R r) old new guard = R (r(old := (guard, new
   ) # (r old)))"

fun to_ite :: "(traceExpression × symbol) list
   traceExpression" where
"to_ite [] = ConstantBool False"
| "to_ite ((guard,Symbol s)#[]) = SymbolValue (Constant s
   32)"
| "to_ite ((guard,Symbol s)#xs) = ITE
  guard (SymbolValue (Constant s 32)) (to_ite xs)"


```

```
15 fun renumber :: "traceExpression  renumbering
     traceExpression" where
16 "renumber (And t1 t2) r = And (renumber t1 r) (renumber t2
     r)"
17 | "renumber (Not te) r = Not (renumber te r)"
18 | "renumber (GreaterThan t1 t2) r = GreaterThan (renumber
     t1 r) (renumber t2 r)"
19 | "renumber (Add t1 t2 w) r = Add (renumber t1 r) (renumber
      t2 r) w"
20 | "renumber (Select t1 t2) r = Select (renumber t1 r) (
     renumber t2 r)"
21 | "renumber (With t1 t2 t3) r = With (renumber t1 r) (
     renumber t2 r)  (renumber t3 r)"
22 | "renumber (ITE t1 t2 t3) r = ITE (renumber t1 r) (
     renumber t2 r)  (renumber t3 r)"
23 | "renumber (SymbolValue te) (R foo) = (case te of
24   Constant v w  to_ite ((foo (Symbol v)))
25   | _  SymbolValue te)"
26 | "renumber te r = te"
27
28 fun createRenumbering :: "gotoStmt list  renumbering
     renumbering" where
29 "createRenumbering [] r = r"
30 | "createRenumbering ((Assign s _)#xs) (R r) =
31     createRenumbering xs (R (r(s := [(ConstantBool True, s)
     ])))"
32 | "createRenumbering (_#xs) r = createRenumbering xs r"
```

A.2.4 Bounded Symbolic Execution

The bounded symbolic execution semantics are encoded using a record type `symexState`, which stores the program counter, unwinding bound, last instruction index, symbol counter, renumbering information, guard conditions, and accumulated trace. Auxiliary functions compute the maximum symbol index in expressions and programs, and construct initial states.

State

```
record symexState =
  pc :: nat
  k  :: nat
  last :: nat
  symbols :: nat
  r :: renumbering
  guard :: traceExpression
  symexTrace ::  "traceStmt list"
```

Auxiliary functions

```
fun getMaxSymbolTE :: "traceExpression  nat" where
"getMaxSymbolTE (SymbolValue (Constant n _)) = n"
| "getMaxSymbolTE (SymbolValue t0) = getMaxSymbolTE t0"
| "getMaxSymbolTE (Not t0) = getMaxSymbolTE t0"
| "getMaxSymbolTE (And t0 t1) = max (getMaxSymbolTE t0) (
   getMaxSymbolTE t1)"
| "getMaxSymbolTE (Add t0 t1 _) = max (getMaxSymbolTE t0) (
   getMaxSymbolTE t1)"
| "getMaxSymbolTE (GreaterThan t0 t1) = max (getMaxSymbolTE
    t0) (getMaxSymbolTE t1)"
| "getMaxSymbolTE (Select t0 t1) = max (getMaxSymbolTE t0)
   (getMaxSymbolTE t1)"
| "getMaxSymbolTE (With t0 t1 t2) = max (max (
   getMaxSymbolTE t0) (getMaxSymbolTE t1)) (getMaxSymbolTE
   t2)"
| "getMaxSymbolTE (ITE t0 t1 t2) = max (max (getMaxSymbolTE
    t0) (getMaxSymbolTE t1)) (getMaxSymbolTE t2)"
| "getMaxSymbolTE te = 0"

fun getMaxSymbol :: "gotoStmt list  nat" where
"getMaxSymbol [] = 0"
| "getMaxSymbol ((Assign (Symbol n) e)#xs) = max (max n (
   getMaxSymbolTE e)) (getMaxSymbol xs)"
```

```
| "getMaxSymbol ((Assume e)#xs) = max (getMaxSymbolTE e) (
   getMaxSymbol xs)"
| "getMaxSymbol ((Assert e)#xs) = max (getMaxSymbolTE e) (
   getMaxSymbol xs)"
| "getMaxSymbol ((IfThenGoto e _)#xs) = max (getMaxSymbolTE
    e) (getMaxSymbol xs)"

fun createDefState :: "gotoStmt list  nat  symexState"
   where
  "createDefState program unwind =
     pc = 0,
      k=unwind,
      last = (length program),
      symbols = Suc (getMaxSymbol program),
      r = (R (x . [(ConstantBool False, x)])),
      guard = (ConstantBool True),
      symexTrace = [] "
```

Symbolic iteration

```
function symexKIt :: "gotoStmt list  symexState
   symexState" where
"symexKIt program state = (if (Suc (pc state) (last state)
    (k state)  0) then
  case program ! (pc state) of
   Assume t
    symexKIt program ( state pc := ((pc state)+1),
            symexTrace :=  (TAssume (renumber t (r state)))
    # symexTrace state,
            guard := (Implies (guard state) t) )
   | Assert t
    symexKIt program (state  pc := ((pc state)+1),
            symexTrace :=  (TAssert (renumber t (r state)))
    # symexTrace state,
            guard := (Implies (guard state) t) )
```

```
12    | Assign s t   symexKIt program (state
13              pc := ((pc state)+1),
14              symbols := (symbols state) + 1,
15              r := updateSymbol (r state) s (Symbol (symbols
    state)) (guard state),
16              symexTrace :=  (TAssign (Symbol (symbols state)
    ) (renumber t (r state))) # symexTrace state  )
17     | IfThenGoto te n   (if (Suc (pc state)   n) (Suc n   (
    last state)) then
18          (let elseClause = symexKIt program (state
19               pc := ((pc state)+1),
20              last := n,
21              guard := (Implies (guard state) (Not te)) ) in
22           symexKIt program ( statepc := n, symbols := symbols
     elseClause, r := r elseClause, symexTrace := symexTrace
     elseClause ))
23        else
24          (let thenState = symexKIt program (state
25                pc := n,
26                 k := k state - 1,
27               guard := Implies (guard state) te  )
28          in
29          symexKIt program  thenStatepc :=  ((pc state)+1), k
    := k state  )
30 )
31
32  else state)"
33  by pat_completeness auto
34
35 termination symexKIt
36   sorry
```

A.2.5 Examples

Finally, we provide examples to illustrate the semantics in practice. These include a simple program without loops, a program with looping behavior, and a program with nondeterministic branching. Each example shows how the symbolic execution framework captures the program's behavior in Isabelle, linking back to the semantics introduced in Chapter 3.

Non-loop program

The following program:

```
definition program1 :: "gotoStmt list" where
  "program1 = [
    Assign (Symbol 0) (Constant 1 32),
    Assign (Symbol 1) (Constant 0 32),
    Assert (SymbolValue (Constant 2 32))] "
```

We, can check the result of the symbolic execution with bound 2:

```
value "symexK program1 2"
"[TAssign (Symbol 3) (Constant 1 32),
  TAssign (Symbol 4) (Constant 0 32),
  TAssert (SymbolValue (Constant 2 32))]"
  :: "traceStmt list"
```

We can then try with an environment to show that it fails:

```
definition environment3 :: ev where
  "environment3 = (environment1 (Symbol 2 :=  x.0))"

definition environment4 :: ev where
  "environment3 = (environment1 (Symbol 2 :=  x.1))"

value "EvalTrace (symexK program1 2) environment4" // False
```

Non-loop conditional program

For programs with conditional jumps values

```
definition listing_3_5 :: "gotoStmt list" where
  "listing_3_5 = [IfThenGoto (SymbolValue (Constant 1 32))
   3,
    Assign (Symbol 0) (Constant 2 32),
    IfThenGoto (ConstantBool True) 4,
    Assign (Symbol 2) (Constant 0 32),
    Assert (SymbolValue (Constant 0 32))]"
```

In this program, the assertion only fails if $s_1 \neq 0 \wedge s_0 = 0$. We generate the following unwind:

```
"[TAssign (Symbol 3) (Constant 2 32),
  TAssign (Symbol 4) (Constant 0 32),
  TAssert
   (ITE (Not
          (And
            (Not (Not (ConstantBool True)))
            (Not (Not (SymbolValue (Constant 1 32))))))
     (SymbolValue (Constant 3 32)) (SymbolValue (Constant 0
    32)))]"
```

Then we can test this trace with the eval method:

```
definition environment4 :: ev where
  "environment4 = (environment1 (Symbol 1 := x.1, Symbol 0
   := x.0))"

definition environment5 :: ev where
  "environment5 = (environment1 (Symbol 1 := x.1, Symbol 0
   := x.1))"

definition environment6 :: ev where
  "environment6 = (environment1 (Symbol 1 := x.0, Symbol 0
   := x.0))"

value "EvalTrace (symexK listing_3_5 2) environment4"
// True
```

```
value "EvalTrace (symexK listing_3_5 2) environment5"
// False
value "EvalTrace (symexK listing_3_5 2) environment6"
// False
```

Loop Program

For loops, we can try a simple program:

```
definition listing_loop :: "gotoStmt list" where
  "listing_loop = [
    Assign (Symbol 0) (Constant 2 32),
    Assign (Symbol 1) (Constant 3 32),
    Assign (Symbol 2) (Constant 4 32),
    IfThenGoto (SymbolValue (Constant 3 32)) 0,
    Assert (SymbolValue (Constant 0 32))]"
```

With symbolic execution it generates the following trace:

```
"[TAssign (Symbol 4) (Constant 2 32),
  TAssign (Symbol 5) (Constant 3 32),
  TAssign (Symbol 6) (Constant 4 32),
  TAssign (Symbol 7) (Constant 2 32),
  TAssign (Symbol 8) (Constant 3 32),
  TAssign (Symbol 9) (Constant 4 32),
  TAssert
   (ITE (traceExpression.Not
          (And (traceExpression.Not (traceExpression.Not (
   ConstantBool True)))
            (traceExpression.Not (SymbolValue (Constant 3
   32)))))
     (SymbolValue (Constant 7 32))
     (ITE (ConstantBool True) (SymbolValue (Constant 4 32))
    (SymbolValue (Constant 0 32))))]"
  :: "traceStmt list"
```