Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Abstraction-Based Interprocedural Software Verification

Master's Thesis

Author
Márk Somorjai

Advisors
Mihály Dobos-Kovács
Levente Bajczi
Dr. András Vörös

December 15, 2024

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Background **4**

2.1 Control Flow Automata 4

2.2 Abstraction 6

2.3 Counterexample-Guided Abstraction Refinement 9

2.3.1 Abstractor 11

2.3.2 Refiner 12

2.4 Procedures 13

2.5 Contracts 14

2.5.1 ACSL 16

3 Related Work **18**

3.1 Inlining 18

3.2 Summaries 19

3.3 Contracts 19

3.3.1 Weakest Preconditions 19

3.3.2 Strongest Postconditions 20

4 Interprocedural Analysis **21**

4.1 Location Stack 21

4.2 Dataflow 25

4.2.1 Variable Instances 25

4.2.2 Parameter Assignments 26

5 Applying Abstraction to Stacks **27**

5.1 Stack Abstraction 27

5.1.1 Changes to Abstractor . . . 29
5.1.2 Changes to Refiner . . . 30
5.2 Case Study . . . 32
5.2.1 Explicit Abstraction . . . 33
5.2.2 Predicate Abstraction . . . 33
5.2.3 Comparison to CEGAR . . . 34
5.3 Evaluation . . . 35
5.3.1 Benchmark Setup . . . 35
5.3.2 Benchmark Results . . . 36
5.3.2.1 Basic Programs . . . 36
5.3.2.2 Recursive Programs . . . 37
5.3.2.3 Threats to Validity . . . 38

6 Contract Synthesis **39**
6.1 Transforming Abstract States to Contracts . . . 40
6.1.1 Predicate Domain . . . 42
6.1.2 Explicit Domain . . . 43
6.2 Case Study . . . 44
6.3 Evaluation . . . 46
6.3.1 Benchmark Setup . . . 46
6.3.2 Reachability Results . . . 47
6.3.3 Overflow Results . . . 48
6.3.4 Threats to Validity . . . 49

7 Conclusion **51**
7.1 Future Work . . . 52

Bibliography **54**

HALLGATÓI NYILATKOZAT

Alulírott *Somorjai Márk*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2024. december 15.

Somorjai Márk
hallgató

Kivonat

A szoftverek egyre jobban elterjednek életünkben. Használjuk üzenetek küldésére és fizetésre, de szoftver vezeti az autókat és üzemelteti a nukleáris erőműveket is. Ilyen rendszerekben egy hiba jelentős gazdasági következményekhez, környezeti károkhoz vagy akár életvesztéshez is vezethet, ezért az ezen rendszerekben való szoftverek helyességének bizonyítása kritikus. Ehhez formális módszereket lehet használni, melyek matematikailag precíz bizonyítékkal szolgálnak a helyességről, vagy cáfolattal a biztonsági tulajdonságok megsérüléséről.

Többféle megközelítése létezik a formális verifikációnak, mindegyiknek megvannak a saját kihívásai. A deduktív metódusok célja, hogy bizonyítsák a szoftver megfelelését a specifikációjának azáltal, hogy a programot és specifikációját matematikai bizonyítási célokká transzformálják, melyeket aztán tételbizonyítóknak adnak tovább. A megközelítés fő hátránya, hogy a specifikáció megfogalmazása kontraktusként jelentős mérnöki munkával jár, ami limitálja az eljárás skálázhatóságát. Egy másik megközelítés, a modellellenőrzés bejárja a program egy matematikai modelljének állapotterét, hogy eldöntse a program bizonyos biztonsági tulajdonságairól, hogy teljesülnek-e. A modellellenőrzés automatizálható, azonban gyakran kivitelezhetetlen a program állapotterének mérete miatt, mely exponenciálisan nő a program változóinak számával. Az állapottér méretét tovább növelik a procedúrák, melyek a program állapotát a hívási veremmel egészítik ki, amely végtelen mély lehet rekurzív programok esetén, így egy végtelen nagy állapotteret eredményezve.

A dolgozatomban absztrakció-alapú megközelítéseket javaslok az említett formális módszerek kihívásainak leküzdésére. Az absztrakciós technikák a program állapottéréből egy redukált absztrakt állapotteret állítanak elő az adatállapot felülbecslésével. Elsőként az absztrakció kiterjesztését mutatom be hívási vermekre egy absztrakció-alapú modellellenőrző algoritmusban, az ellenpélda-alapú absztrakciófinomításban (CEGAR). A hívási verem részeinek elabsztrahálásával a különböző hívási veremmel rendelkező, de hasonló állapotok tovább csoportosíthatók, így csökkentve az absztrakt állapottér méretét, ami javítja az interprocedurális verifikáció hatékonyságát. Ezután a modellellenőrzés és a deduktív verifikáció egy olyan kombinációját javaslom, ami automatizálttá és hatékonyabbá teszi a verifikációt. A modellellenőrző által kifejtett absztrakt állapottérből függvénykontraktusok szintetizálhatók, melyek felhasználásával deduktív módszerekkel bizonyíthatók a program különböző tulajdonságai. Ezáltal nem szükséges a kontraktusok előállításához a fejlesztők manuális munkája, így a verifikáció skálázhatóbb. Az utóbbi javaslat újdonsága, hogy a szintézis *környezetfüggő*, azaz a generált kontraktusok csak olyan feltételeket tartalmaznak amelyek ténylegesen előfordulhatnak a program függvényeiben, nem pedig a lehető legáltalánosabb formulát. A bemutatott ötleteket a Theta modellellenőrző keretrendszerben implementálom, a hatékonyságukat pedig szintetikus és ipari példákon értékelem ki.

A Kulturális és Innovációs Minisztérium ÚNKP-23-2-I-BME-83 kódszámú Új Nemzeti Kiválóság Programjának és a EKÖP-24-2-BME-220 kódszámú Egyetemi Kutatói Ösztöndíj Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

Abstract

Software is becoming more and more prevalent in our lives. We use it to send messages and execute payments, but nowadays software also drives cars and operates nuclear power plants. In such systems, failure can lead to significant financial loss, environmental damage or even the loss of life, therefore, ensuring the correctness of software in these system is critical. Formal methods can give a mathematically precise proof of correctness or a refutation to the safety properties of a system.

There are various approaches to formal verification, each with its own set of challenges. Deductive methods aim to prove that software conforms to its specification, by transforming the program and its specification to a collection of mathematical proof obligations, which are then passed on to theorem proving tools. The main disadvantage of the approach is that writing the specification as contracts requires significant engineering effort, limiting the scalability of deductive verification. Another method, model checking, explores the state space of a mathematical model of software, in order to decide whether or not certain safety properties of the program are satisfied. While model checking can be automatic, it can often be infeasible as the size of a program's state space grows exponentially with the number of variables in the program. The state-space is further blown up by procedures, which extend the state of the program with the call stack that can stretch infinitely deep in recursive programs, leading to an infinitely large state-space.

In this work, I propose abstraction-based approaches for combating the challenges of the aforementioned formal methods. Abstraction-based methods reduce the state-space into an abstract state-space by overapproximating the data state of the program. First, I present the extension of abstraction to the call stack in the abstraction-based model checking algorithm, Counterexample-Guided Abstraction Refinement (CEGAR). With parts of the call stack abstracted away, similarities between states with different stacks can be detected and the size of the abstract state-space can further be reduced, improving the efficiency of interprocedural verification. Then, I propose to combine model checking and deductive verification in a way that makes verification both automatic and more efficient. The abstract state-space explored by model checking is used to synthesize contracts for procedures, which are then utilized by deductive methods to verify properties of the program. This eliminates the necessity of the developers having to manually provide contracts, allowing verification to be more scalable. The novel contribution of the latter work is that the synthesis is *context-aware*, that is, the generated contracts only consider conditions that can actually occur in each procedure of the program, instead of generating the most general formulas. I implement the proposed techniques in the model checking framework Theta and evaluate their performance on synthetic and industrial examples.

Supported by the ÚNKP-23-2-I-BME-83 New National Excellence Program and the EKÖP-24-2-BME-220 University Excellence Scholarship Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Chapter 1

Introduction

In our technology-driven world, an increasing amount of tasks are automated using software. In addition to keeping us connected or providing entertainment, these days computers are also relied upon for flying planes or operating nuclear power plants. While the failure of software may not matter much when watching a movie, it can lead to catastrophic events if it happens mid-flight on a passenger aircraft. In such *safety-critical* systems, where failure can lead to significant financial loss, environmental damage or the loss of life, ensuring the correctness of software is of utmost importance. Conventional testing methods can prove the presence of errors by showing an example of the incorrect behavior. Conversely, the success of tests does not guarantee the absence of errors as it is infeasible for tests to cover all behaviors of the software, hence it cannot guarantee the correctness of software. In safety-critical systems, testing alone is insufficient, more precise methods have to be used, such as *formal verification*. Formal verification aims to either provide a mathematically precise proof for a software's correctness or a refutation to it.

There are different kinds of methods for formal verification, each with its own set of advantages and setbacks. *Model checking* [1] is an automated verification algorithm that explores the state-space of the program and checks whether or not an erroneous state can be reached. The greatest challenge of model checking is the size of the state-space: to represent all possible values of a 32-bit integer, 2^{32} states are needed. Furthermore, the number of states grows exponentially with the number of variables in the program, meaning that there are more possible states in a program with eight 32-bit integers than the number of atoms in the universe. This phenomenon is called the state-space explosion [20], which all model checking algorithms have to deal with if they aspire to be useful in practice. Abstraction-based model checking algorithms reduce the state-space into an abstract state-space by grouping states together based on their control locations and data states, that is, the values of their variables. The Gordian knot of such algorithms is finding the right level of abstraction, so that correctness of the program can be reasoned about. The *Counterexample-Guided Abstraction Refinement (CEGAR)* algorithm [19] takes an iterative approach to this: it overapproximates the state-space of the program with an *abstract state-space* and refines it repeatedly, until it reaches an adequate level of abstraction for giving a verdict about the program's correctness.

Another challenge of formal verification emerges from procedures in programs. Procedures are a widely used concept in all fields of software. They provide structure and allow the reuse of existing software. On the other hand, they disrupt the sequential flow of execution and generate new variable instances at each of their calls, which makes their *interprocedural verification* more challenging. Moreover, they extend the state of a program with a call stack, making it possible for the same control locations and data states to appear multiple

times in the state-space with different call stacks. Along with the fact that they allow recursion, procedures enable the state-space to grow infinitely large, because the call stack can stretch infinitely deep in recursive programs.

A different approach to formal verification is *deductive verification* [26]. It expresses the correctness of a program as a set of mathematical statements, then uses axioms and proof rules in automated or interactive theorem provers to verify them. Deductive verification is a powerful method as it can be used to prove both safety and behavioral properties of programs, which are specified by contracts [32]. Contracts also modularize the verification process, since conformity to the specification can be checked on a per-procedure basis. Writing these contracts, however, requires significant engineering effort and specialized knowledge in the field of formal methods, since the developer has to understand why the program should work correctly and needs to be able to convey that information at various abstraction levels throughout the software. Consequently, the scalability of deductive verification is limited, which restricts its applicability.

In this work, I propose approaches to interprocedural verification, using abstraction-based methods. An overview of the presented ideas and their relations can be seen in Figure 1.1.

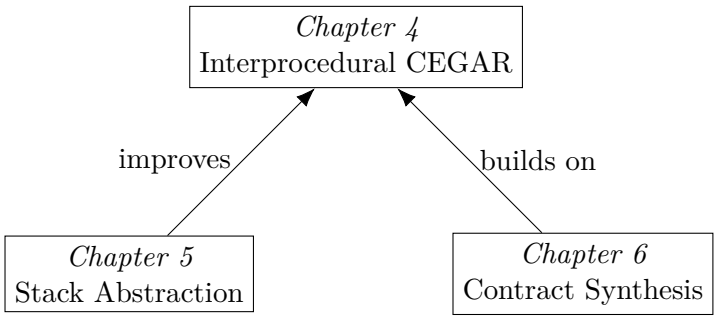


Figure 1.1: Overview of the presented work

First, I introduce a variant of the abstraction-based CEGAR algorithm to explore the reduced abstract state-space of the program in an interprocedural manner. This allows the verification of (transitively) recursive programs. More importantly, it enables the following two novel contributions of my work.

In Chapter 5, I propose an approach to improve the efficiency of abstraction-based interprocedural verification of programs. The idea is an improvement of the aforementioned interprocedural CEGAR algorithm. My novel approach reduces the size of the state-space of a procedural program by applying abstraction to call stacks, similarly to how abstraction is traditionally applied to the control locations and data states of the program. Not only does the proposition enhance the performance of verification on procedural programs, it also empowers the CEGAR algorithm to verify some infinitely recursive programs, which it is not able to do by default. The proposed approach is implemented in the open-source model checking framework Theta [28] and is evaluated on benchmarks from SV-COMP [5], the international competition for software verification.

In Chapter 6, I propose an automated approach for contract synthesis using abstraction-based model checking. The generated contracts then can be utilized by deductive verifiers in order to bring out the best of both approaches, making verification both automatic and more efficient. The idea builds upon the interprocedural CEGAR algorithm introduced in Chapter 4, using it to explore the abstract state-space of the program. I present a synthesis method for generating contracts from the explored abstract state-space, the products of which can then be passed onto deductive verifiers to prove various safety properties of

the program. Since both steps are automated, the manual labor of developers for writing contracts can be eliminated, making deductive verification more scalable. The novelty of this work is that the abstraction-based synthesis is *context-aware*, in the sense that the generated contracts only consider the conditions under which the procedure could have been called in the context of the program, as opposed to traditional weakest precondition synthesis techniques [23]. The proposed approach is also implemented in the open-source model checking framework Theta, making it capable of synthesizing contracts in the ANSI C Specification Language [3]. The implementation, combined with the open-source deductive verification platform Frama-C [30], is evaluated on benchmarks from SV-COMP [5].

The thesis is structured as follows: in Chapter 2, the preliminary concepts and definitions are introduced, which the rest of the work builds upon. In Chapter 3, various approaches to interprocedural verification are described. In Chapter 4, an interprocedural extension of the CEGAR algorithm is discussed. In Chapter 5, my first main contribution is presented: a way of applying abstraction to location stacks, as a means of improving the efficiency of interprocedural verification. My other main contribution is proposed in Chapter 6: a context-aware abstraction-based contract synthesis technique, enabling deductive verification. Implementations of both approaches are evaluated at the end of their respective chapters. Finally, in Chapter 7, my work is summarized and conclusions are drawn.

Chapter 2

Background[1]

To understand the presented work, some background knowledge is required about software verification. This chapter presents the necessary concepts and definitions, as well as their interpretation in the context of the presented work.

The goal of software verification is to mathematically prove certain properties of a program. One such property is the safety of a program, that is whether or not an erroneous location can be reached in the program. A program is *unsafe* if such a location can be reached from the initial location of the program using a finite number of transitions; otherwise, it is *safe*. To prove these properties *model checking* is often employed, during which the reachable states of the program are explored, and their erroneousness is decided. Due to the large state-space of programs, state reduction techniques are usually employed. A model checking algorithm using abstraction is described later in the section. But first, a formal representation of programs is introduced, which is often used in software verification.

2.1 Control Flow Automata

Software can take many shapes and forms, most notably, it can be represented as source code. While it is convenient for software development due to its readability, its usage in model checking can be complicated due to its complex syntax and semantics. For that purpose, a formal representation of the software is used, which allows for easier verification of basic properties, such as *error reachability*. A formal representation that is often used to model programs is the *Control Flow Automaton* [6].

A *Control Flow Automaton* represents a program as a directed graph, as described in the following.

Definition 1 (Control Flow Automata). A control flow automaton is a tuple $CFA = (V, L, l_0, E)$, where:

- V: A set of *variables*, where each $v \in V$ can have values from its domain D_v.
- L: A set of *locations*, where each *location* can be interpreted as a possible value of the program counter.
- $l_0 \in L$: The *initial location*, that is active at the start of the program.

[1] Some parts of this chapter are taken from my previous work [38].

- $E \subseteq L \times Ops \times L$: A set of transitions, where a transition is a directed edge going from one location in L to another, with a label $op \in Ops$, where Ops is a set of operations that can be executed as the program advances from one location to another. An $op \in Ops$ can be one of the following:
  - $v = expr$: An assignment of a variable, where the value of $v \in V$ becomes the evaluation of the right-hand side $expr$.
  - $havoc\, v$: A non-deterministic assignment of a variable, after which the value of $v \in V$ can be in anything from its domain D_v.
  - $[cond]$: A *guard* operation, where *cond* is an expression that evaluates to a boolean value. The transition can only be executed if the *cond* in the *guard* evaluates to *true*. ▪

In formal verification, it is also useful to distinguish *error locations*, which are locations where the program would behave in an undesirable way, as well as *final locations*, which have no *outgoing transitions*, that is, transitions that are directed away from them.

Definition 2 (Concrete State). A *concrete state* of a $CFA = (V, L, l_0, E)$ can be described as $s = (l, d_1, d_2, \ldots, d_n)$, where:

- $l \in L$ is the current location of the program,
- $d_1, d_2, \ldots, d_n \in D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$ is the *concrete data state*, where $d_1, d_2, \ldots, d_n$ are the values of all variables, that is, $\forall v_i \in V : v_i = d_i$.

The set of concrete states of the program is denoted by $\hat{S}_L = L \times \hat{S}$, where $\hat{S} = D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$ is the set of concrete data states. ▪

The state of the CFA in its initial location is its *initial state*. The uninitialized values of variables at the beginning of the program depend on the programming language. In a language where uninitialized variables have the value of whatever memory garbage is at their assigned location in the memory, the values of variables would be non-deterministic. Therefore, the CFA of programs written in such languages may have many initial states. Other languages (such as Java) assign a default value to uninitialized variables, resulting in a single initial state of the CFA.

All possible states of the CFA make up the *state-space* of the program. An *execution* of a program on the CFA can be represented as $(s_1, op_1, s_2, \ldots, op_{n-1}, s_n)$, an alternating sequence of locations and operations. The operations an execution's alternating sequence can then be interpreted as *transitions* in the state-space of the program.

Example 1. *Consider the following C program:*

```
int a;
int b = a % 10;
while (a > 100) {
    b = a % b;
    a = a − 1;
}
assert(b < 10);
```

It can be represented by the CFA in Figure 2.1.

Note how the uninitialized variable a is havoced on the edge going from the initial location.

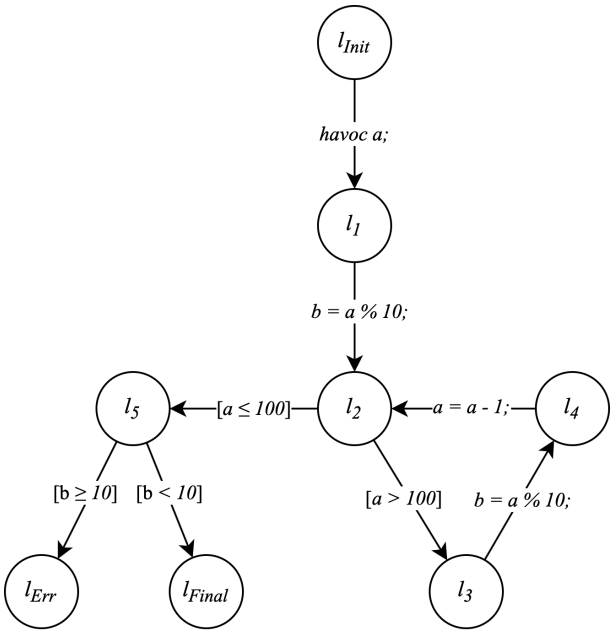


Figure 2.1: CFA of example C program.

2.2 Abstraction

The size of the state-space of a program presents the greatest challenge in software verification: just to represent all possible states with a single 32-bit integer variable 2^{32} states are needed, moreover, it grows exponentially with the number of variables present in the program. It goes without saying that checking the reachability of all states would be unfeasible, leaving the need for some kind of reduction technique on the state-space. One such technique is abstraction.

Definition 3 (Abstract Domain). An abstract domain is a tuple $D = (\mathcal{S}, \sqsubseteq, \mathit{concr})$, where:

- $\mathcal{S}$ is a lattice of *abstract data states*,
- $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ is a partial order conforming to the lattice,
- $\mathit{concr} : \mathcal{S} \to 2^{\hat{S}}$ is a function, mapping an abstract data state to its set of concrete data states.

The abstract domain is *overapproximating*, that is, the partial $\sqsubseteq$ and the *concr* function satisfy the following: $S_1 \sqsubseteq S_2 \Leftrightarrow \mathit{concr}(S_1) \subseteq \mathit{concr}(S_2)$, i.e., S_1 *overapproximates* S_2. ▪

The *precision* $\pi \in \Pi$ defines the level of abstraction. The *transfer function* $T : \mathcal{S} \times \mathit{Ops} \times \Pi \to 2^{\mathcal{S}}$ calculates the *successors* of an abstract data state with respect to the operation *Ops* and a target precision.

In software verification, two widely used abstract domains are *explicit-value abstraction* [7] and *predicate abstraction* [2].

Explicit-value abstraction defines an abstract data state $S \in \mathcal{S}$ by variable assignments, mapping variables to $\top, \bot$ or a value from their domain. $\top$ is the top element of the lattice and denotes an unknown value, while $\bot$, the bottom element of the lattice represents that

no assignment to the variable is possible. Referring to the value of a variable v stored in an abstract data state S as $S(v)$, this can be expressed as $S(v) \in D_v \cup \{\top, \bot\}$. The partial order for abstract data states $S_1, S_2 \in \mathcal{S}$ is defined as $S_1 \sqsubseteq S_2$, if for all variables v in S_2: $S_1(v) = S_2(v)$ or $S_1(v) = \bot$ or $S_2(v) = \top$. The function *concr* for an abstract data state S is $concr(S) = \hat{S}$ if $\forall v : S(v) = \top$, $concr(S) = \varnothing$ if $\exists v : S(v) = \bot$, otherwise it is $concr(S) = \{s \in \hat{S} \mid \forall v : v = S(v)\}$. A precision π is a subset of the variables that are being tracked.

In *predicate abstraction*, an abstract data state $S \in \mathcal{S}$ is the combination of first order logic predicates on the variables, e.g. $v_1 > 7 \wedge v_2 < 0$. The top and bottom elements are $\top = true$ $\bot = false$, respectively. The partial order corresponds to implication, that is, $\forall S_1, S_2 \in \mathcal{S} : S_1 \sqsubseteq S_2$ if $S_1 \Rightarrow S_2$. The function *concr* maps the abstract data state S to all concrete states, where the predicates of S evaluate to true. The predicates being tracked are given by the precision π.

Similarly to how an abstract data state represents a set of concrete data states of a CFA, the abstract states of a CFA can be defined by extending data states with locations.

Definition 4 (Abstract State). Given an abstract domain $D = (\mathcal{S}, \sqsubseteq, concr)$, an *abstract* state of a $CFA = (V, L, l_0, E)$ is a tuple $S_L = (l, S)$, where:

- $l \in L$ is the current location of the program,
- $S \in \mathcal{S}$ is an abstract data state, describing the data state of the program.

The set of the abstract states of a CFA $\mathcal{S}_L$ forms a lattice as well, where the partial order $\sqsubseteq_L$ for the abstract states $(l_1, S_1), (l_2, S_2) \in \mathcal{S}_L$ is defined as $(l_1, S_1) \sqsubseteq_L (l_2, S_2)$ if $l_1 = l_2$ and $S_1 \sqsubseteq S_2$.

The function $conr : \mathcal{S}_L \to 2^{\hat{S}_L}$ for an abstract state $(l, S) \in \mathcal{S}_L$ is defined as $conr((l, S)) = \{(l, s) \in \hat{S}_L \mid s \in concr(S)\}$. ▪

The transfer function $T_L : \mathcal{S}_L \times \Pi \to 2^{\mathcal{S}_L}$ in a $CFA = (V, L, l_0, E)$ for an abstract state $(l, S) \in \mathcal{S}_L$ is defined as $T_L((l, S), \pi) = \{(l', S') \in \mathcal{S}_L \mid (l, op, l') \in E, S' \in T(S, op, \pi)\}$, that is, the *successors* of an abstract state (l, S) are abstract states (l', S') for which there is an edge (l, op, l') in the CFA and the abstract data state S' is a successor of S with respect to the transfer function T and precision π.

The set of abstract states $\mathcal{S}_L$ of a CFA form the *abstract state-space* of a program, which can be represented by an *Abstract Reachability Graph* [9].

Definition 5 (Abstract Reachability Graph). An abstract reachability graph is a graph-like representation of the abstract state-space $\mathcal{S}_L$ of a $CFA = (V, L, l_0, E_{CFA})$. It is defined by the tuple $ARG = (N, E, C)$, where:

- $N \subseteq \mathcal{S}_L$ is the set of *nodes*, each corresponding to an abstract state.
- $E \subseteq N \times Ops \times N$ is the set of directed *edges* between nodes, labeled with operations of the CFA. An edge $((l_1, S_1), op, (l_2, S_2)) \in E$ is present if $(l_1, op, l_2) \in E_{CFA}$ and (l_2, S_2) is a successor of (l_1, S_1) with *op*.
- $C \in N \times N$ is the set of *covered-by edges*. A covered-by edge edge for the abstract states $S_1, S_2 \in N$ is present, i.e., $(S_1, S_2) \in C$, if $S_1 \sqsubseteq_L S_2$. In this case, S_1 is *covered by* S_2 or in other words, S_2 *covers* S_1. ▪

The edges between abstract states can be interpreted as transitions in the abstract state-space of the program. An *abstract path* is a directed path in the ARG, i.e., an alternating sequence $((l_1, S_1), op_1, (l_2, S_2), op_2, \ldots, op_{n-1}, (l_n, S_n))$ of abstract states an operations. An abstract path is *feasible*, if there is a concrete path $((l_1, s_1), op_1, (l_2, s_2), \ldots, op_{n-1}, (l_n, s_n)$ where $\forall i \in [1, n] : s_i \in concr(S_i)$. In practice, feasibility can be decided by converting the abstract states to logic formulae and querying an SMT solver (such as Z3 [21]) with the formula $S_1 \wedge op_1 \wedge S_2 \wedge \cdots \wedge op_{n-1} \wedge S_n$. Moreover, a satisfying assignment to the variables in this formula can be converted to a concrete path in the CFA.

A node $S_L = (l, S) \in N$ of an $ARG = (N, E, C)$ is called

- *expanded*, if $\forall S'_L \in T_L(S_L) : \exists (S_L, op, S'_L) \in E$ for some operation op, i.e., all of its successors according to the transfer function are in the ARG,
- *covered*, if $\exists (S_L, S'_L) \in C$ for some $S'_L \in N$, meaning S_L has an outgoing covered-by edge,
- *unsafe*, if $l = l_E$ is an error location of the CFA,
- *incomplete* otherwise.

An ARG is *unsafe* if there is an unsafe node in it and *complete* if none of its nodes are incomplete.

Example 2. *Consider the CFA in Example 1. An ARG of this CFA can be seen in Figure 2.2, using predicate abstraction with the precision* $\pi = \{a \geq 100, a > 100, b < 10\}$.

Starting from the abstract state of the initial location l_0 *of the CFA, the first edge does not provide any information about the data state of the program. The second transition using the edge* $(l_1, b = a\%10, l_2)$*, however, does provide information about the value of* b*, which is stored in the abstract state: it is less than 10. From here, there are two possible edges guarded by the value of* a*. These are both possible, since there's no information currently available about the value of* a*, therefore the ARG will branch in two directions.*

The one going towards l_5 *reaches the final location next, since the* $b < 10$ *information still holds. Since the aren't any outgoing edges from the final location, the corresponding node of the ARG is expanded without any outgoing edges.*

The other branch going towards l_3 *follows one iteration of the loop in the program. Let* $S_1 = (l_2, b < 10)$ *denote the branching node and* $S_2 = (l_2, a \geq 100 \wedge b < 10)$ *the other node with location* l_2*. The transfer function* T_L *describes an edge going from* S_1 *to the abstract state* $(l_3, a \geq 100 \wedge a > 100 \wedge b < 10)$*, because the guard on the edge* $(l_2, [a > 100], l_3)$ *of the CFA guarantees that the predicates* $(a > 100), (a \geq 100) \in \pi$ *are satisfied in all concrete states of the program in* l_3*. The predicates stay true along the next transition, since the CFA edge* $(l_3, b = a\%10, l_4)$ *has no effect on the value of* a*. However, during the next transition, the operation* $a = a - 1$ *can make the predicate* $a > 100$ *false. On the other hand,* $a \geq 100$ *still holds, as a result of* $a > 100$ *being true in the previous abstract state and the value of* a *only decreasing by 1. Thus, the transfer function* T_L *specifies an edge going to* S_2*.*

In S_2*, a covering relation can be found between the two abstract states in* l_2*. Their locations are the same and the abstract data state of the latter implies the abstract data state of the former:* $(a \geq 100 \wedge b < 10) \implies (b < 10)$*, which in the case of predicate abstraction means* $S_2 \sqsubseteq_L S_1$*. Therefore, there is a covered-by edge going from* S_2 *to* S_1*.*

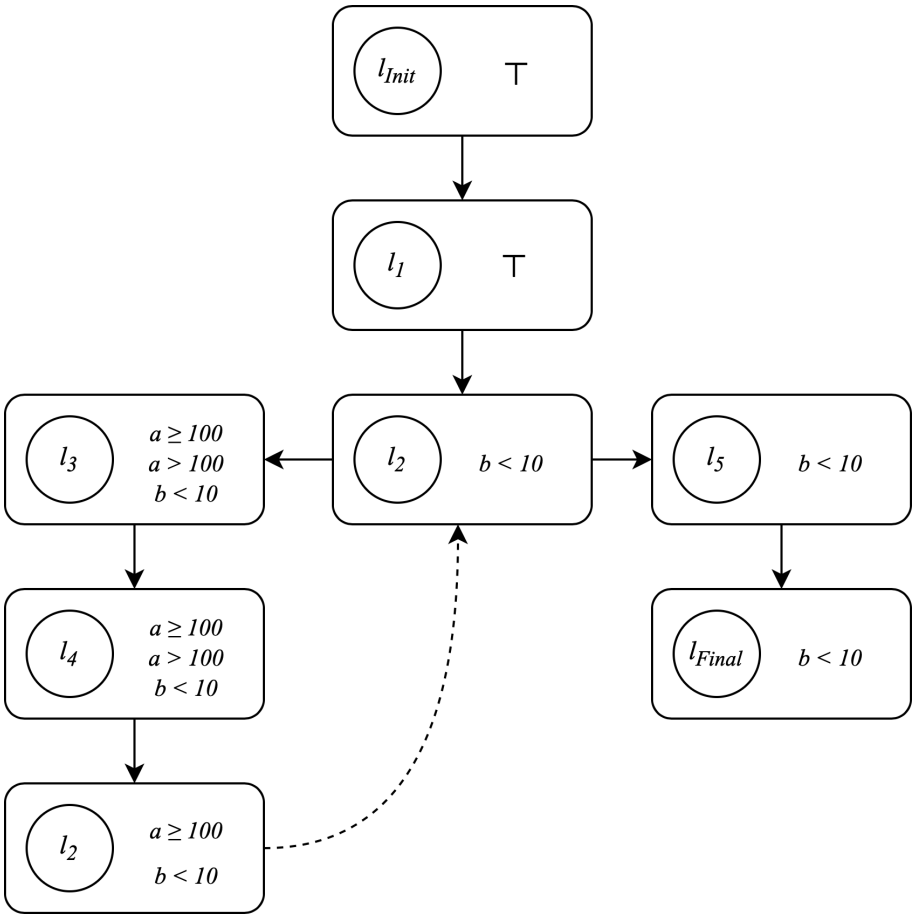


Figure 2.2: Fully expanded ARG of Example 1.

The node corresponding to S_2 is covered, all other nodes of the ARG are expanded and none of the nodes are unsafe because none are in the error location l_{Err}. Consequently, the ARG is complete and not unsafe, alias safe.

2.3 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [19] is an abstraction-based model checking algorithm. It takes the formal representation of a program (such as a CFA) with distinguished error locations and decides whether or not the program is safe, that is, if the error locations are reachable from the initial location of the program. It does this by either exploring all reachable abstract states of the program and deeming them non-erroneous or by providing a counterexample to the program's safety, which is a concrete execution of the program in which an error-state is reached.

The core of the algorithm is the CEGAR-loop on Figure 2.3, made up of two main parts: the *abstractor* and the *refiner*. The abstractor builds the ARG using the *covering* relation, a *transfer function* and a *precision* on abstract states, as introduced in Section 2.2. An abstract error-state is an overapproximation of the possible error-states, consequently, if no abstract error-state is reachable, then no concrete error-state is reachable, meaning the program is *safe*.

On the other hand, if an abstract error-state is reachable, the abstractor produces an *abstract counterexample*, that is, an abstract path starting at the initial abstract state and

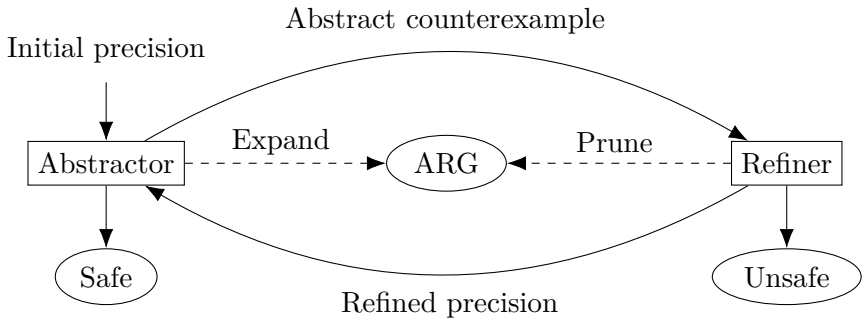


Figure 2.3: The CEGAR loop

ending in an abstract error-state. This is where the refiner comes in: it decides whether or not a concrete error state is reachable in the abstract error-state. If it can be reached, then the program is *unsafe*, and the path from the initial location of the CFA to a concrete error state is presented as a counterexample.

However, if a concrete error-state is not reachable, then the reachability of the abstract error-state is a result of the overapproximation of abstraction. Thus, the abstraction needs to be *refined* so that the abstract error-state does not contain the unreachable concrete error-state. This results in a refined precision, which is passed back to the abstractor after all unreachable abstract states are removed (*pruned*) from the abstract state-space.

The CEGAR loop is repeated until it either finds a concrete counterexample to the safety of the program or proves that no abstract error-state is reachable, that is, all nodes in the ARG are either expanded or covered. In the first case, the program is *unsafe*, while in the latter, it is *safe*. A pseudo-code of the CEGAR loop [27] is presented in Algorithm 2.1.

Algorithm 2.1: CEGAR loop

Input:
$CFA = (V, L, l_0, E)$: program
$D = (\mathcal{S}, \sqsubseteq, concr)$: abstract domain
π_0: initial precision
T: transfer function

Output: safe or unsafe

```
1  ARG ← ((l_0, ⊤), ∅, ∅)
2  π ← π_0
3  while true do
4  |  result, ARG ← Abstraction(ARG, D, π, T)
5  |  if result = safe then
6  |  |  return safe
7  |  else
8  |  |  result, π, ARG ← Refinement(ARG, π)
9  |  |  if result = unsafe then
10 |  |  |  return unsafe
11 |  |  end
12 |  end
13 end
```

The abstractor and its Abstraction procedure is presented in Section 2.3.1, while the refiner and the Refinement procedure are described in Section 2.3.2.

2.3.1 Abstractor

Using the concepts defined in Section 2.2, the abstractor's operation in the CEGAR loop is presented in Algorithm 2.2. The abstractor explores the abstract state-space by building the ARG, using the *covered-by* relation, the *transfer function* with some *precision*. If it builds the ARG to completion without encountering an erroneous abstract state, the program is deemed safe because abstract state-space is an *overapproximation* of the concrete state-space, meaning if a concrete error state was reachable in the concrete state-space, it would be reachable in the abstract state-space as well. On the other hand, if it reaches an erroneous state during exploration then it returns an *abstract counterexample* to the program's safety, that is, an abstract path going from the initial abstract state to an erroneous one in the ARG.

Algorithm 2.2: ABSTRACTION procedure

Input:
$ARG = (N, E, C)$: partially constructed abstract reachability graph
$D = (\mathcal{S}, \sqsubseteq, concr)$: abstract domain
π: current precision
T_L: transfer function
Output: (safe or unsafe, ARG)

```
1  waitlist ← {S ∈ N | S is incomplete}
2  while waitlist ≠ ∅ do
3  |  (l, S) ← pop waitlist
4  |  if l = l_E then
5  |  |  return (unsafe, ARG)
6  |  else if ∃(l', S') ∈ N : (l, S) ⊑_L (l', S') then
7  |  |  C ← C ∪ {((l, S), (l', S'))}
8  |  else
9  |  |  foreach (l', S') ∈ T_L((l, S), π) \ {⊥} do
10 |  |  |  waitlist ← waitlist ∪ {(l', S')}
11 |  |  |  N ← N ∪ {(l', S')}
12 |  |  |  E ← E ∪ {((l, S), op, (l', S'))}
13 |  |  end
14 |  end
15 end
16 return (safe, ARG)
```

The algorithm manages a *waitlist* of incomplete nodes that gets filled up as new abstract states are discovered. The `while` loop iterates through each abstract state (l, S) in the waitlist by popping the first element of it. Since the first element is taken in each iteration, the ordering of the waitlist can be used to define a search strategy in the abstract state-space (e.g. BFS). The abstractor does one of the following 3 operations on the popped abstract state:

- If the abstract state is in an erroneous location, then an abstract counterexample to the CFA's safety has been found. Therefore, the abstractor returns the verdict `unsafe` along with the partially built ARG.
- If the abstract state is not erroneous, but it is covered by another abstract state (l', S'), then a covered-by edge is created in the ARG. In this case, the node does not

need to be expanded, since S' overapproximates S, meaning that for every abstract path starting at (l, S) there is a corresponding one starting at $(l', S') = (l, S')$. Therefore, the expansion of (q, S) would be redundant, so it is not expanded.

- If the abstract state is neither erroneous or covered, then it is expanded, i.e., its successors are calculated using T_L and are added to the ARG.

If there are no more abstract states in the waitlist, it means that no erroneous abstract state was found - otherwise the procedure would have terminated already -, and that all nodes are either expanded or covered by another node. Therefore, the ARG is complete and safe by definition. Since the built ARG is an overapproximation of the concrete state space of the program, a concrete path to an erroneous location would have to appear in the ARG as an abstract counterexample; but none were found, thus, the program is safe.

2.3.2 Refiner

Using the concepts defined in Section 2.2, the refiner's operation in the CEGAR loop is presented in Algorithm 2.3. The task of the refiner is twofold: for one, it needs to decide whether an abstract counterexample to the program's safety is feasible or not. Additionally, if the abstract counterexample was found to be infeasible, it needs to refine the precision π so that the counterexample becomes unattainable in the abstract state space as well.

Algorithm 2.3: REFINEMENT procedure

Input:
$ARG = (N, E, C)$: partially constructed abstract reachability graph
π: current precision
Output: (spurious or unsafe, π', ARG)

$\sigma = ((l_1, S_1), op_1, \ldots, op_{n-1}, (l_n, S_n)) \leftarrow$ abstract path to unsafe node in ARG
if σ *is feasible* **then**
  return (unsafe, π, ARG)
else
  $(I_1, \ldots, I_n) \leftarrow$ interpolant for σ
  $(\pi_1, \ldots, \pi_n) \leftarrow (I_1, \ldots, I_n)$ converted to precisions
  $\pi' \leftarrow \pi \cup \bigcup_{1 \leq i \leq n} \pi_i$
  $i \leftarrow$ lowest i for which $I_i \notin \{true, false\}$
end
$N_i \leftarrow$ all nodes in the subtree rooted at (l_i, S_i)
$N \leftarrow N \setminus N_i$
$E \leftarrow \{(n_1, op, n_2) \in E \mid n_1, n_2 \notin N_i\}$
$C \leftarrow \{(n_1, n_2) \in C \mid n_1, n_2 \notin N_i\}$
return (spurious, π', ARG)

The algorithm starts off by finding an abstract counterexample in the ARG, that is, an abstract path $\sigma = ((l_1, S_1), op_1, (l_2, S_2), \ldots, op_{n-1}, (l_n, S_n))$ that starts off at the initial abstract state and ends in an erroneous one. This can be done by using any kind of search algorithm (e.g. BFS) from the initial node of the ARG.

Next, the feasibility of σ is decided, that is, whether there is a concrete path in the CFA $((l_1, s_1), op_1, (l_2, s_2), \ldots, op_{n-1}, (l_n, s_n)$ for which $\forall i \in [1, n] : s_i \in concr(S_i)$. A widely

used way of deciding this is converting the abstract states to logic formulae and querying an SMT solver with the formula $S_1 \wedge op_1 \wedge S_2 \wedge \cdots \wedge op_{n-1} \wedge S_n$:

- If the formula is satisfiable, then the counterexample is feasible and the verdict `unsafe` is returned.
- If the formula is unsatisfiable, then the counterexample is infeasible. In this case, the verdict is `spurious`, a refined precision π' is calculated and the ARG is pruned of its unreachable abstract states.

The new precision π' is calculated using an inductive sequence of assertions, which can be calculated via sequence interpolants [40] or Newton refinement [22]. For example, the elements of a sequence interpolant $(I_1, \ldots, I_n)$ correspond to the abstract states of the counterexample. The structure of the interpolant for some $k \in (1, n)$ is as follows: $I_i = true$ if $i < k$, $I_i = I_k$ if $i = k$ and $I_i = false$ if $i > k$. The sequence interpolant $(I_1, \ldots, I_n)$ is converted to precisions $(\pi_1, \ldots, \pi_n)$ according to the abstract domain (e.g. $\pi_i = I_i$ for predicate abstraction), then the new precision π' is created as the union of the created precisions.

Finally, the ARG pruned back using lazy abstraction [39], i.e., the descendants of the earliest abstract state (l_i, S_i) where the precision changed are removed. Since the sequence interpolants resemble the abstract states, the index i of the earliest abstract state can be found by looking for the first sequence interpolant that is not $true$ or $false$. With the index in hand, all descendants of (l_i, S_i) are removed from the ARG, along with all transitions and covered-by edges related to them.

The motivation behind pruning lazily is to keep as much of the unaffected part of the ARG as possible, so that the abstractor can reuse it in the next iteration of the CEGAR loop. Note, that during the pruning process, the remaining nodes may become incomplete due to the removal of transitions and covered-by edges.

2.4 Procedures

Procedures are a well-known concept in software that allow modularity, more structured software, as well as the reuse of already written software. However, their semantics and usage can differ between languages and different domains, hence the following definition is introduced.

Definition 6 (Procedure). A *procedure* is an encapsulated part of software represented by the tuple $P = (B, I, O)$, where:

- $B = (V, L, l_0, E)$: The encapsulated program *body*, represented as a CFA.
- $I \subseteq V$: A set of variables called *input parameters*, Unlike all other variables, they have a value assigned to them in the initial state of B.
- $O \subseteq V$: A set of variables called *output parameters*. Unlike all other variables, values assigned to them are preserved after the final state of B is reached. ▪

Some programming languages support *inout* parameters, where a variable is passed into the procedure by reference (or by a pointer), making all local modifications to a parameter

apply to the outer variable as well. These variables can be replaced by an input and an output parameter, therefore I chose not to distinguish them for the sake of simplicity.

Another commonly used feature in procedural programming languages are global variables that exist independently from procedures: they have a value at the start of the program and can be modified from any procedure. As with inout parameters, a global variable g can also be replaced with an input i_g and an output o_g parameter, where the i_g takes up the value of the global variable before the procedure is executed, while the o_g is assigned the value of i_g at the final location of the procedure. Since o_g keeps its value after the execution of the procedure, it can be used to update the value of g as if the procedure operated on it. Therefore, without loss of generality, only programs without global variables are considered in this work.

The utility of procedures comes with the introduction of *procedure calls*.

Definition 7 (Procedure call). A *procedure call* is an operation in programs which initiates the execution of the body of a procedure. It can be represented by the tuple $C = (P, A, R)$, where:

- $P = (B, I, O)$: The procedure being *called*, the CFA $B = (V, L, l_0, E)$ of which is to be executed.
- $A = \{a_1, a_2, \ldots, a_{|I|}\}$: *Arguments* are a set of expressions that are assigned to the input parameters I of the procedure, that is, $\forall i \in [1, |I|], v_i \in I, a_i \in A : v_i = a_i \in D_{v_i}$.
- $R = \{r_1, r_2, \ldots, r_{|O|}\}$: *Return variables* are a set of variables, to which the output parameters O of the procedure will be assigned to, that is $\forall i \in [1, |O|], v_i \in O, r_i \in R : r_i = v_i \in D_{r_i}$.

A *procedure call* consists of 3 steps:

1. The evaluations of the expressions in A are assigned to I, the input parameters of P.
2. Execution carries on from the initial location l_0 in B, the CFA representing the body of the procedure, until a final location is reached.
3. The output parameters O of P are assigned to the variables in R, after which execution continues from the location after the *procedure call*. ▪

It is important to note that calling a procedure essentially creates a new instance of it, meaning that if a procedure was called multiple times at the same time, the different executions of the body would not operate on the same set of variables.

As procedure calls are introduced to software verification, complications arise. One is the aforementioned handling of different variable instances, but problems emerge with abstract states and their covering relation as well. Handling the modified control flow with location stacks is described in Section 4.1, while solutions for the intricacies of data flow are discussed in Section 4.2.

2.5 Contracts

Procedures modularize software, providing interface of certain functionality to callers without exposure to implementation details. Even though the added level of abstraction is

certainly useful, it also loses information on what a procedure performs. To preserve information on functionality, contracts [32] can be used, which describe the assumptions on the inputs of a procedure and provide information on the outputs of it.

Definition 8 (Contract). A *contract* for a procedure $P = (B, I, O)$ with body $B = (V, L, l_0, E)$ specifies the conditions present before and after calling a procedure. It is represented by a pair $(Pre_P, Post_P)$, where:

- Pre_P is a set of first order logic predicates over the input parameters I, describing the preconditions of the procedure.
- $Post_P$ is a set of first order logic predicates over the output parameters O, describing the postconditions of the procedure.

If a procedure call $C = (P, A, R)$ is reached, then all predicates in Pre_P shall hold over the values of A substituted in for the input parameters I. When the procedure returns, all predicates in $Post_P$ shall hold over the values of output parameters O assigned to the return values R.

The procedure *conforms* to its contract if all of its terminating executions $(s_1, \ldots, s_F)$ that start in a state s_1 for which Pre_P holds end in a state s_F in which $Post_P$ is satisfied, that is:

- $(l_0, d_1, \ldots, d_n) = s_1 \in \hat{S}_L : s_1 \vdash Pre_P$
- $(l_F, d'_1, \ldots, d'_n) = s_F \in \hat{S}_L : s_F \vdash Post_P$ ▪

A contract imposes constraints and provides guarantees both inside the procedure and at call locations of it. The preconditions require the arguments at all call locations to fulfill the predicates in Pre_P, but provide guarantees inside the procedure about the input parameters. Correspondingly, the postconditions require the output parameters to satisfy the predicates in $Post_P$, while also ensuring that they hold over the return variables of the procedure. As a result, contracts allow verification to be carried out in an intraprocedural manner in two steps:

1. For every procedure, the conformity of the implementation CFA to its contract specification is to be verified.
2. The safety property in question can be verified, using the contracts of procedures at call locations instead of the bodies.

Contracts are useful for software verification in many ways. For one, they can formally specify the expected behavior of a system, so that the implementation then can be checked for conformity. Additionally, they can make the verification of other safety properties more efficient by enabling intraprocedural verification, with the preconditions further reducing the state-space of each procedure. Contracts also make deductive verification [26] possible, a methodology that expresses the correctness of programs as a set of mathematical statements and uses automated or interactive theorem provers to verify them. On the downside, creating contracts requires significant engineering effort and specialized knowledge, limiting its applicability.

2.5.1 ACSL

The *ANSI C Specification Language (ACSL)* [3] is a formal language for specifying the intended behavior of C programs. It can be used to annotate loops with invariants, locations with assertions and most notably for this work, procedures with contracts. The annotations are added to the code as special comments, so that the compilation process remains unaffected.

ACSL contains various language constructs, including high-level concepts – such as the aforementioned invariants and contracts – and low-level building blocks of logic formulae, including conjunction and comparison operators. It also contains C-specific elements, e.g. pointers or the ternary operator. Due to its vast scope, only relevant parts of the language are presented that are later used in this work.

The partial grammar for higher level constructs of ACSL is presented in Figure 2.4. Contract specifications are embedded in a block comment, containing various kinds of contract clauses. The preconditions and postconditions described in Contract 8 correspond to the requires- and ensures-clauses, the other clauses are omitted for simplicity.

function-contract	::=	**`/*@`** requires-clause* terminates-clause$^?$ decreases-clause$^?$ simple-clause* named-behavior* completeness-clause* **`*/`**
requires-clause	::=	clause-kind$^?$ **`requires`** pred **`;`**
ensures-clause	::=	clause-kind$^?$ **`ensures`** pred **`;`**
named-behavior	::=	**`behavior`** *id* **`:`** behavior-body
behavior-body	::=	assumes-clause* requires-clause* simple-clause*
completeness-clause	::=	**`complete behaviors`** (id (**`,`** id)*)$^?$ **`;`**
	|	**`disjoint behaviors`** (id (**`,`** id)*)$^?$ **`;`**

Figure 2.4: The partial ACSL grammar for contracts.

ACSL includes behaviors as a means for improving the legibility of contracts. They can be used to define separate contracts for different behaviors of the procedure. Behaviors can be defined as disjoint or complete, in which case a concrete state must only satisfy at most one or at least one of the behaviors' preconditions, respectively.

The partial grammar for the lower level elements of ACSL can be seen in Figure 2.5. It builds upon integer and boolean literals as well as variables to construct expressions with arithmetic and logic operators. Predicates and terms are distinguished, following the distinction between propositions and terms in first order logic.

literal	::=	`\true`	
	|	`\false`	boolean constants
	|	integer	integer constants
ident	::=	*id*	any valid C identifier
unary-op	::=	`+` | `-`	unary plus and minus
	|	`!`	boolean negation
bin-op	::=	`+` | `-` | `*` | `/` | `%`	
	|	`==` | `!=` | `<=` | `>=` | `>` | `<`	
	|	`&&` | `||` | `^^`	boolean operations
rel-op	::=	`==` | `!=` | `<=` | `>=` | `>` | `<`	
term	::=	literal	literal constants
	|	ident	variables, function names
	|	unary-op term	
	|	term bin-op term	
	|	`(` term `)`	parentheses
	|	term `?` term `:` term	ternary condition
pred	::=	`\true` | `\false`	
	|	term rel-op term	comparisons
	|	`(` pred `)`	parentheses
	|	pred `&&` pred	conjunction
	|	pred `||` pred	disjunction
	|	`!` pred	negation
	|	pred `^^` pred	exclusive or
	|	term `?` pred `:` pred	ternary condition
	|	pred `?` pred `:` pred	

Figure 2.5: The partial ACSL grammar for predicates.

Chapter 3

Related Work

Procedures make the analysis of programs more difficult by disrupting both the control and data flow of the program. This chapter covers various approaches to interprocedural analysis of procedural programs. First, inlining and its extension with summaries is described, in comparison with the work of Chapter 5. Then, contracts are discussed, with the relevant automated contract synthesis techniques compared to the approach presented in Chapter 6.

3.1 Inlining

Inlining handles the procedures of procedural programs by eliminating all procedures from them. The elimination is done by replacing every procedure call with the called procedure's body, along with the input and output parameter assignments.

An advantage of inlining is that it is straightforward way of handling procedures correctly. A downside of it is that it produces huge programs. The pitfall of inlining, however, is that it does not work on recursive, or even transitively recursive programs, i.e., programs where the procedure can reach a call of itself, potentially through other procedures. In such programs, each time a (transitively) recursive procedure call is replaced by the procedure's body, a new call of the procedure is created. Therefore, the number of procedure calls never reaches 0, so inlining does not terminate. *Bounded inlining* [17] handles this by setting a bound on the depth of inlining.

Bounded inlining inlines all procedure calls up to a certain depth k and cuts off every call that would go beyond this depth. This results in an underapproximation of the original program. Verification starts off with a bound $k = 0$, i.e., no procedure calls are inlined, they are just removed from the program. If a counterexample to the program's safety is found in the underapproximating program, then the counterexample exists in the original program as well, so the program is deemed unsafe. If no counterexample is found, then the bound is increased and another iteration is done. If all procedure calls have been inlined and no counterexample was found, then the program is safe.

Bounded inlining methods are used by *Bounded Model Checkers* [13]. They are usually combined with some other technique, such as *summaries*.

3.2 Summaries

A procedure summary [36] is an abstraction of a procedure. It captures the effect of computations that start at the entry and end at the exit points of a procedure. This can be done by an explicit tabulation of the relation between abstract initial and final states [34], or by defining a function from input abstract states to the output abstract states [41], for example. In the case of recursive procedures, transformers [41] or fixpoint algorithms [33] can be employed to to generate a summary, among others.

Procedure summaries cannot be used directly as a verification method. They are typically utilized by verification algorithms to replace procedure calls with their summaries, in order to reduce the number of states that need to be explored. CPARec [18] combines summaries with intraprocedural analyzers to verify recursive programs. Stratified inlining [31] is an extension of bounded inlining with summaries.

3.3 Contracts

A contract [32] – as introduced in Section 2.5 – in a software component describes requirements and guarantees that have to be satisfied when interacting with the component. For procedures, this corresponds to preconditions on the input parameters and postconditions on the output parameters of the procedure. Therefore, contract can be used similarly to summaries: a procedure call can be replaced with the procedure's contract during verification, in order to simplify verification. Contracts can be utilized by model checking tools [25], but their main usage comes in deductive verification tools, such as KeY [14] and Frama-C [30].

Contracts are a powerful concept. They serve as a specification language for the expected behavior of procedure and modularize the verification process. Finding correct and sufficient contracts, however, requires significant engineering effort and specialized knowledge, limiting its scalability. Consequently, there has been significant research done on the automation of generating contracts. In the following, two approaches to automated contract synthesis are discussed, which are related to the work of Chapter 6.

3.3.1 Weakest Preconditions

The main line of research on synthesizing contracts is the inference of preconditions of a procedure, which are sufficient to ensure that the procedure's assertions hold. The preconditions state the least amount of obligations that a caller has to fulfill in order for the procedure to run correctly.

Preconditions can be computed in many different ways. Traditional weakest precondition calculus [23] transforms postconditions to preconditions. They can also be computed using symbolic execution by exploring all paths that lead to violated assertions. As expected with symbolic execution, the latter approach would have issues with loops. More directly related to this work, there are algorithms for deriving preconditions using CEGAR [35], which handles loops better. The iterative algorithm can be used to refine an overapproximation of the weakest precondition, until only the sufficient preconditions remain.

The main difference between weakest precondition contracts and the contracts synthesized by the approach presented in Chapter 6 is what they represent. The weakest precondition is a *backward* approach that starts at the verification conditions and looks for the most

general conditions that are required for it to hold. This, however may be too broad for verification, and it can be unnecessary as the procedure can only be called under known conditions in the program. The approach presented in this work is a *forward* one, which takes all calling contexts into account. The resulting contract contains all possible preconditions under which the procedure could be called from in the context of the program. As opposed to the weakest precondition, this serves to simplify the deductive verification of the procedure.

3.3.2 Strongest Postconditions

Another related approach is strongest postcondition [24], sometimes also called a summary. Given a program and a precondition, the strongest postcondition captures all possible final states of an execution of the procedure starting with the precondition. The postcondition is the strongest, as it describes the possible postcondition to most detailed level. Therefore it is ideal for supporting the procedure-level modularity of deductive verification. On the other hand, the precision can come at a cost of overly verbose expressions. On top of that, it usually reflects more on the implementation of the program itself, rather the intention behind it.

Similarly to weakest preconditions, strongest postconditions can be calculated using symbolic execution [29], though it once again comes with loop unrolling issues. It can also generate huge formulas, as it is path based. Techniques have been developed to reduce the size of the formula [37] using passive single assignment and normalization.

The main difference of strongest postconditions compared to the approach presented in Chapter 6 is that for one it generates postconditions as opposed to preconditions. Moreover, it relies on preconditions to already be written, circumventing the goal of fully automating the contract synthesis process.

Chapter 4

Interprocedural Analysis

Procedures are widely used in all fields of software. They have many advantages – such as modularization and reutilization –, however, they make verification more difficult by disrupting the sequential flow of execution and generating new variables with each procedure call. While it is possible to inline procedures in some programs, to verify programs with (transitively) recursive procedures, a more delicate approach is required, as inlining does not terminate for these programs. In this chapter, I describe adjustments that can be used to support procedures and procedure calls in the abstraction-based model checking algorithm, CEGAR.

4.1 Location Stack

With the addition of procedures, the input of the model checking algorithm is no longer a single CFA, but several *Control Flow Automata (CFAs)*. The bridges connecting these CFAs are procedure calls: after a procedure call, execution carries on from the initial location of the called procedure's CFA. Calling is just one part of the task, though; the continued execution from the calling location, as the final location of the procedure's CFA is reached, also needs to be ensured. For this purpose, a *location stack* can be used, similar to the call stack that is employed in programs.

Definition 9 (Location Stack). A *location stack* q is a FILO data structure with *push* and *pop* operations, which stores all locations of a procedural program's CFAs from where procedure calls were made to reach the current location l_q. The current location l_q is always on the top of the stack, i.e., $l_q = top(q)$.

The set of all possible stacks of a program is denoted by Q. ▪

At the beginning, the location stack stores the location that represents the entry point of the program. Afterwards, the stack is modified in the following three situations, when an edge (l_i, op, l_j) of the CFA is reached:

- By default, the top location of the stack is replaced with the target of the transition by popping l_i from the stack and pushing l_j onto it.
- Additionally, if the operation op of the edge is a procedure call $C = (P, A, R)$, the initial location of the called procedure P's body is *pushed* onto the stack.

- If the target location l_j of the edge is the final location of a procedure, it is popped from the stack and execution carries on from the location underneath. If there are no locations left in the stack, execution stops as the program terminates.

With these rules, it is ensured that the desired properties of procedures are kept, as well as that the top location of the stack is always the current location.

The introduction of location stacks means that a concrete state of a CFA can no longer be described by a location and a concrete data state, since that does not store information about if and where execution should be continued from the final location. Instead, the location stack is what can accurately represent a concrete state of a CFA, because it also stores the procedure calls (and the CFAs) through which the current location was reached.

Definition 10 (Concrete State). A *concrete state* of a $CFA = (V, L, l_0, E)$ with procedure calls can be described as a tuple (q, s), where:

- $q \in Q$ is a location stack with the current location $top(q)$ on top of it,
- $s \in \hat{S}$ is a concrete data state of the program.

The set of concrete states of a procedural program is denoted by $\hat{S}_Q = Q \times \hat{S}$. ▪

The extension of concrete states with a location stack also impacts abstract states in a similar fashion. An abstract state of a CFA can no longer be described by a location and an abstract data state, since that would not be sufficient for the transfer function to calculate the successors of an abstract state in a final location. Therefore, an abstract state also needs to be extended with a location stack.

Definition 11 (Abstract State). An *abstract state* of a $CFA = (V, L, l_0, E)$ with procedure calls can be described as a tuple (q, S), where:

- q is a location stack with the current location $top(q)$ on top of it,
- $S \in \mathcal{S}$ is an abstract data state of the program.

The set of abstract states of a procedural program is denoted by $\mathcal{S}_Q = Q \times \mathcal{S}$. ▪

The function $concr : \mathcal{S}_Q \to 2^{\hat{S}_Q}$ for an abstract state $(q, S) \in \mathcal{S}_Q$ can then be defined as $concr((q, S)) = \{(q, s) \in \hat{S}_Q \mid s \in concr(S)\}$.

The transfer function

$$T_Q : \mathcal{S}_Q \times \Pi \to 2^{\mathcal{S}_Q}$$

in a procedural program with

$$\forall i \in [1, n] : CFA^i = (V^i, L^i, l_0^i, E^i)$$

for an abstract state

$$(q, S) \in \mathcal{S}_Q$$

is defined as

$$T_Q((q, S), \pi) = \{(q', S') \in \mathcal{S}_Q \mid \exists i \in [1, n] : (top(q), op, top(q')) \in E^i, S' \in T(S, op, \pi)\}.$$

That is, the *successors* of an abstract state (q, S) are abstract states (q', S') for which there is an edge $(top(q), op, top(q'))$ going between their stacks' top locations, and the abstract data state S' is a successor of S with respect to the transfer function T and precision π.

All that remains is the partial order $\sqsubseteq_Q$ between abstract states $(q_1, S_1), (q_2, S_2) \in \mathcal{S}_Q$ with stacks, in order to make $\mathcal{S}_Q$ a lattice. Carrying over the idea from $\sqsubseteq_L$, one could potentially define $\sqsubseteq_Q$ as $(q_1, S_1) \sqsubseteq_Q (q_2, S_2)$ if $S_1 \sqsubseteq S_2$ and their current locations are equal, that is, $top(q_1) = top(q_2)$. However, this definition alone would not result in correct analysis, as Example 3 shows.

Example 3. *Consider the following C program and its CFAs on Figure 4.1. The procedure* `reach_error()` *represents the program entering an unsafe state.*

```
void foo(int k) {
  return;
}

int main() {
  foo(2);
  foo(2);
  reach_error();
};
```

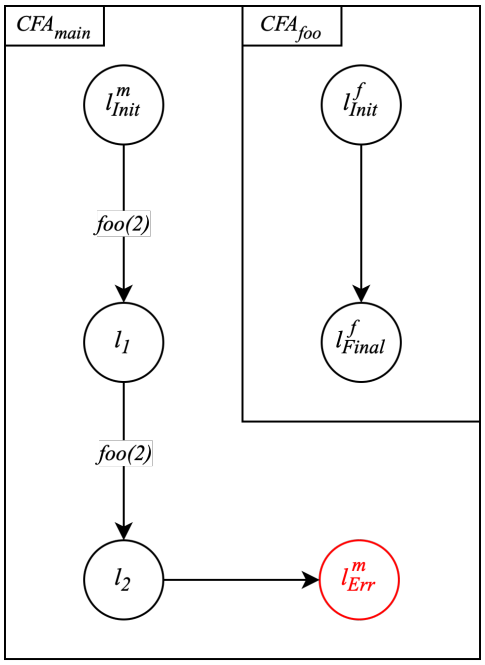


Figure 4.1: Example C program and its CFAs.

Examining the code, it can be seen that the the program will reach the erroneous state in its main function after two calls to `foo`*, as the procedure does nothing. Conversely, if the partial order $\sqsubseteq_Q$ would be defined in the aforementioned manner, the CEGAR algorithm would deem the program safe. The ARG built by the abstractor using the explicit domain can be seen on Figure 4.2.*

The first call to `foo` *is explored without a problem. During the second call, however, the abstract state in l^f_{Init} is in the same location as the one after the first call. Since both abstract states' data state is $k = 2$, the first one overapproximates the second, leading to a covered-by relation between them. The branch being the only one of the ARG, exploration stops at this point and the program is deemed (incorrectly) safe, as the fully built ARG has no nodes in erroneous states.*

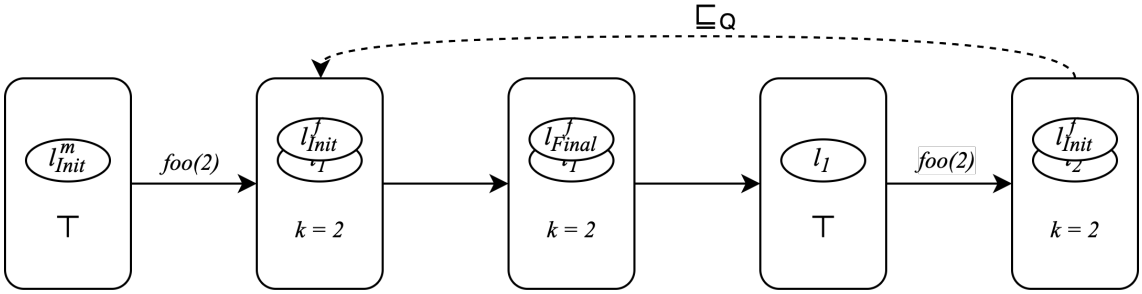


Figure 4.2: The ARG built with the wrong definition of $\sqsubseteq_Q$.

The problem with the definition of $(q_1, S_1) \sqsubseteq_Q (q_2, S_2)$ above is that the abstract state (q_2, S_2) is not necessarily an overapproximation of (q_1, S_1). Just because their current location is the same, their stacks underneath could be entirely different and map to disjoint sets of concrete states. The overapproximation was what allowed covered abstract states to not be expanded in Algorithm 2.2. Without overapproximation, it is no longer guaranteed that all paths of covered state are present from the covering state, which lead to an unsound analysis.

Another way of interpreting $\sqsubseteq_L$ on the abstract states is that not only should their current locations be equal, but everything other than their abstract data states should be identical. While for $\sqsubseteq_L$ the current location is the only information stored in the abstract states other than the data state, in the case of abstract states with stacks, the aforementioned condition would mean that their whole stack should match.

The partial order $\sqsubseteq_Q$ on abstract states with stacks is defined as $(q_1, S_1) \sqsubseteq_Q (q_2, S_2)$ if $S_1 \sqsubseteq S_2$ and $q_1 = q_2$, i.e, their stacks are identical. As opposed to the previous potential definition, this way of defining $\sqsubseteq_Q$ guarantees that (q_2, S_2) is an overapproximation of (q_1, S_1), because for $q = q_1 = q_2 : concr((q, S_1)) = \{(q, s_1) \in \hat{S}_Q \mid s_1 \in concr(S_1)\} \subseteq \{(q, s_2) \in \hat{S}_Q \mid s_2 \in concr(S_2)\} = concr((q, S_2))$ holds by definition of $S_1 \sqsubseteq S_2$. Thus, this definition of $\sqsubseteq_Q$ can be used for covering, keeping the analysis sound. Using this definition, the previous example program is verified correctly, as seen in Example 4 below.

Example 4. *Using the second, correct definition of the partial order $\sqsubseteq_Q$, the CEGAR algorithm would come to a different conclusion when verifying the program in Figure 4.1. The ARG built by the abstractor using the explicit domain can be seen on Figure 4.3.*

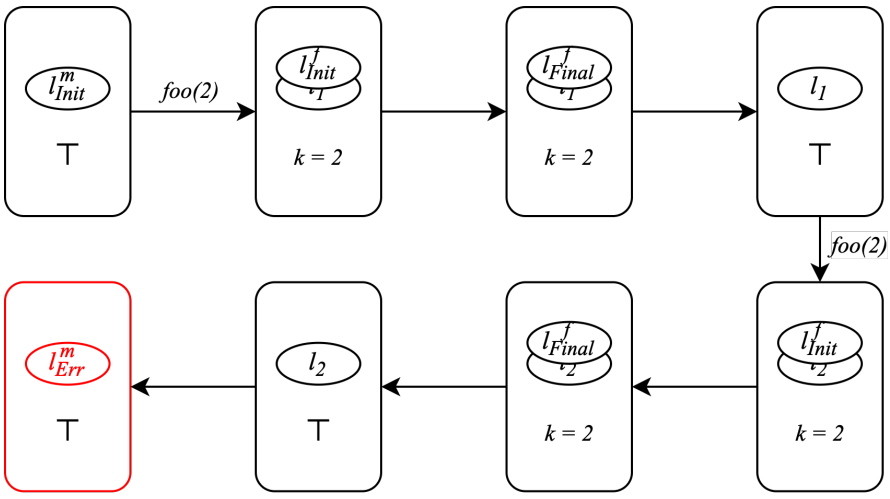


Figure 4.3: The ARG built with the correct definition of $\sqsubseteq_Q$.

When exploration reaches the second call to `foo`*, the abstract data states of the two abstract states in* l^f_{Init} *once again overapproximate one another. This time, however, this is no longer sufficient for covering, as even though they are in the same location, their location stacks are different: the first has* l_1 *at its bottom, while the other has* l_2*. Therefore, no covered-by edge is created and the abstract state-space is explored after the second call to* `foo` *returns, reaching the erroneous state highlighted in red. After the refiner checks that the found counterexample is feasible, the program would correctly be deemed safe.*

Even though the definition of $\sqsubseteq_Q$ leads to a sound analysis by preserving the overapproximation property, it loses out on some of the power of abstraction: not considering similarities between abstract data states because they have different stacks, leads to redundant calculations. In Chapter 5, a way of improving the efficiency of verification is introduced.

4.2 Dataflow

In *intraprocedural* verification, only a single instance can exist of variables and procedure parameters can be modeled as unknown inputs. When performing *interprocedural* analysis, tracking the flow of data is not as simple. For one, each call of a procedure creates new instances of its variables, which can take different values. On top of that, parameters act as a channel for communication between procedures, therefore, their values need to be conveyed both ways. The following sections describe how these problems can be addressed.

4.2.1 Variable Instances

A desired property of procedures is their template-like behavior, that is, new instances of their variables are created with every procedure call. One approach would be to copy the local variables uniquely with every procedure call.

However, the variables cannot be replaced on the CFA's transitions because there is only one CFA per procedure. Therefore an *instance mapping* is required, which associates a local variable with its uniquely copied version (*instance*). Note the use of *local variables*: global variables and output parameters do not need to be instantiated because, in the first case, there is just the single instance of them; in the second case their value can only be used in the next assignment anyway, so there is no point in managing separate versions of them. Using the mapping, local variables can be replaced by their mapped instances during verification when expressions are evaluated.

By default, *instance mappings* need to be created every time a procedure is called. However, due to the nature of CEGAR, previously instantiated versions of variables need to be accessible sometimes. This can happen, when the refiner creates a refined precision, and a new iteration of expansion starts. The refined precision contains information about previously created instances of variables that are used in comparison with the same variable versions' evaluations in the new iteration. One solution is to store the instance mappings associated with location stacks. This way, instances can be reused in procedures called from the same location stack, and the refined precision can be utilized.

To summarize, when a procedure is encountered during verification, the association of location stacks and instance mappings is checked. If an instance mapping exists for the location stack of the current state, then that mapping is used; if not, a new one is created with unique copies of the called procedure's local variables. This way, it is ensured that

variables on a CFA transition can be replaced with their correct instances at any point during verification with CEGAR.

4.2.2 Parameter Assignments

The last defined property of procedures that remains unaccounted for is parameters and their assignments. To address this, additional transitions can be created in the CFAs, with the assignments of parameters on them. Caution needs to be taken around which CFA to add these transitions to, and which version of variables to use.

Let $P_1 = (B_1, I_1, O_1)$ be the *outer* procedure, $P_2 = (B_2, I_2, O_2)$ be the *called* procedures and let $C = (P_2, A, R)$ be a procedure call on a transition between locations l_i and l_j in B_1.

The output parameters of P_2 will be used by variables in P_1, for this reason they need to be assigned in B_1 after the procedure call. This can be done by the following:

1. A new location l_k is created.
2. The transition with the procedure call is moved so that it goes from l_i to l_k.
3. A new transition is created from l_k to l_j, with the assignments output parameters $\forall i \in [1, |O_2|], r_i \in R, v_i \in O_2 : r_i = v_i$ as an operation.

Since output parameters do not have versions (because their value is only used right after the procedure call), no further effort is needed to have their correct versions present in the outer procedure.

The input expressions are used by variables in P_2, therefore it makes sense to assign them in B_2, before the initial location. Unlike output parameters, input parameters do have versions, therefore additional care needs to be taken with their assignments. For each procedure call $C = (P_2, A, R)$ of P_2, the following needs to be done $\forall A$ arguments:

1. Each variable of the CFA B_1 used in the input expressions $a_i \in A, \forall i \in [1, |A|]$ is replaced with a *prime version* of itself (e.g. $v \to v'$).
2. A new *initial parameter location* l_A is created.
3. A new transition is created from l_A to the initial location in P_2, with the assignments of input parameters $\forall i \in [1, |I_2|], v_i \in I_2, a_i \in A : v_i = a_i$ as an operation, using the modified input expressions.

A mapping of the procedure calls associated with their freshly created l_A can be used during verification, to push the correct l_A on top of the location stack whenever a procedure call is encountered. During such an encounter, the *marked versions* of variables mapped to the instances of their original counterparts in P_1 also need to be passed onto the *instance map* of P_2, to allow the assignment of the outer procedure's local variables.

Chapter 5

Applying Abstraction to Stacks

Procedures introduce procedure calls as a valid operation on CFA transitions, therefore, they need to be handled during verification. This calls for changes in how the model checking algorithm works and what information abstract states store. Chapter 4 describes the adjustments that can be used to support procedures and procedure calls in CEGAR. In this chapter, I first present an extension of abstraction to stacks that improves the efficiency of verification. Then, I show the modified algorithm on an example program. Finally, I evaluate the approach on a set of C programs.

5.1 Stack Abstraction

Using the whole stacks for the partial order $\sqsubseteq_Q$ between abstract states leads to redundant calculations in abstract states with the same top location and similar data states, making verification less efficient. Defining $\sqsubseteq_Q$ with only the current top locations of the stack would eliminate such calculations because such abstract states would be covered. However, that definition leads to unsound analysis. In the following, a combination of these two approaches is presented in CEGAR, in order to get a verification algorithm that is both *sound* and *performant*.

Let $\sqsubseteq_A$ denote the partial order between abstract states with stacks, which only requires their top locations to match as in the first attempted definition in Section 4.1: $(q_1, S_1) \sqsubseteq_A (q_2, S_2)$ if $top(q_1) = top(q_2)$ and $S_1 \sqsubseteq S_2$. The reason why this led to an unsound analysis was that (q_2, S_2) is not necessarily an overapproximation of (q_1, S_1). On the other hand, the redundant calculations of using $\sqsubseteq_Q$ come from repeating the same calculations in states that are to some extent, exactly the same.

The main idea behind combining these two approaches is to abstract away part of the stack, so that the resulting two abstract states overapproximate one another. Consider an ARG with abstract states $(q_1, S_1), (q_2, S_2) \in \mathcal{S}_Q$, where $(q_1, S_1) \sqsubseteq_A (q_2, S_2)$ but $(q_1, S_1) \not\sqsubseteq_Q (q_2, S_2)$. This can only occur, when $S_1 \sqsubseteq S_2$ and $top(q_1) = top(q_2)$, but $q_1 \neq q_2$. A part of such an ARG can be seen on Figure 5.1, where the dashed line labeled with $\sqsubseteq_A$ represents that the source node would cover the target node, if $\sqsubseteq_A$ was used for covering.

The key observation in this situation is that from the current location $l_1 = top(q_1) = top(q_2)$, the same states will be explored essentially from (q_2, S_2) as from (q_1, S_1), until it reaches the final location of l_1's CFA. The reason for this is the same as for regular covering: S_1 overapproximates S_2, therefore all paths from (l_1, S_2) will be present from (l_1, S_1). The only difference between the explored states is the bottom of the stack, below

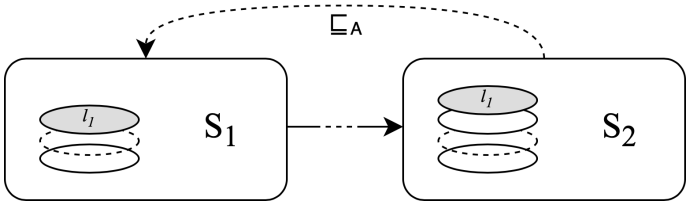


Figure 5.1: Part of an ARG with $\sqsubseteq_A$ between two abstract states.

l_1. However, this part of the stack has no effect on the control flow nor the data state until an element of it becomes the current location again, which happens exactly when the final location of said CFA is reached and the top location is popped.

Another perspective on this observation is that with the part of the stack below l_1 *abstracted away*, the abstract states overapproximate each other, because without the bottom of the stack, exploration would end at the final location l_F of l_1's CFA. In this sense, $\sqsubseteq_A$ is a good partial order for covering: there is no need to explore what happens from l_1 at (q_2, S_2) until l_F is reached, since all such paths are already present at (q_1, S_1). Nevertheless, the path after reaching l_F does have to be examined due to $q_1 \neq q_2$.

Both of the aforementioned demands can be met by popping the top location l_1 of q_2 and carrying on with exploration with the remainder of the stack q_2', instead of creating a covering edge. Popping the top location ensures that the *covered part* of (q_2, S_2), i.e., the paths going from l_1 to l_F are not traversed again. Continuing from the remainder of the stack instead of stopping with a covering edge guarantees that the *not overapproximated part* of (q_2, S_2), that is, the bottom part (q_2', S_2) is further explored. The exact way of this operation is described in Section 5.1.1, while the result of applying it to the ARG in Figure 5.1 can be seen on Figure 5.2.

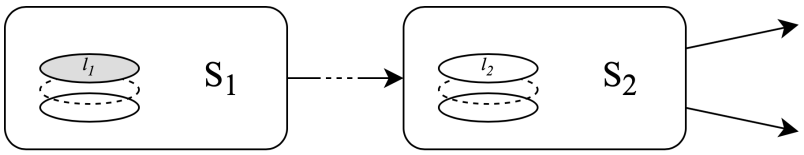


Figure 5.2: Part of the ARG in Figure 5.1 after popping.

One key feature of abstraction is that it is an overapproximation, i.e., for each path in the concrete state-space of the program, a corresponding abstract path can be found in the abstract state-space. To show that this property still holds with the introduced popping, assume without loss of generality that the input program has no global variables and no procedures with inout parameters (see Section 2.4 for details). The only threat to the overapproximation property then would be if the output parameters in the popped abstract data state did not match all of their possible concrete values. These parameters, however, remain uninitialized because the final location of the CFA was never reached. An uninitialized abstract state can correspond any value of the variable's domain, therefore, all concrete states are represented. Consequently, every abstract path that would start at (q_2, S_2) will have corresponding paths in the ARG: the part of the path going from l_1 to l_F is covered by one at (q_1, S_1), while the remainder of the path after l_2 is covered by a path starting at (q_2', S_2). Thus, the modified ARG will be an overapproximation of the original ARG, and transitively the concrete state-space of the program.

The fact that the ARG remains to be an overapproximation of the CFA's concrete state-space implies that if the program is unsafe, the verification algorithm will not give a

wrong answer, because an abstract path corresponding to the concrete counterexample to the program's safety will always be present in the abstract state-space as well. Conversely, if the program is safe, the ARG may still contain an abstract path leading to an erroneous state due the overapproximation. It is the task of the refiner to detect such an abstract counterexample's infeasibility and to provide a refined precision that prevents the infeasible step.

The proposed popping introduces a new kind of infeasibility. The top location may be popped at any abstract state of an ARG, not just where the state's current location is a final one, as in Figure 5.2. If an abstract state with such stack is popped, the ARG will have a transition that cannot happen in the concrete state space of the program: not due to the infeasibility of some assignment in the abstract data state, but because no operation in an inner CFA's non-final location can lead to the outer CFA. To avoid providing a counterexample with such impossible transitions, the feasibility check of the refiner needs to be updated to detect the infeasible pop in abstract counterexamples, and to provide a refined precision that can prevent it from happening in the next iteration. The latter is achieved by introducing a *stack precision* $\pi_Q \subseteq Q$, a set of location stacks in which no special popping should occur. The changes of the refiner are further discussed in Section 5.1.2.

The adjustments made to the refiner ensure the CEGAR loop will not terminate with an infeasible counterexample, because all infeasible abstract counterexamples are refined. Therefore, the verification algorithm will not give a wrong answer if the program is safe. Combined with the observation for unsafe programs above, the algorithm is *sound*.

In the following, the exact changes made to the abstractor and the refiner are described, then the modified algorithm is presented on an example program.

5.1.1 Changes to Abstractor

The abstractor of the CEGAR algorithm is responsible for exploring the abstract state-space of the program by building the ARG. It iteratively explores all non-covered abstract states until it either finds an erroneous one or all nodes are expanded. The abstractor's algorithm is described in Section 2.3.1.

The modifications proposed in Section 5.1 affect the abstractor in the following 3 ways:

1. The $\sqsubseteq_A$ relation also needs to be checked between abstract states.
2. If found, then the top location of the covered state's stack needs to be popped,
3. unless the stack is in π_Q, in which case it needs to be expanded.

The abstractor's modified pseudo-code is presented in Algorithm 5.1, with the changes highlighted in blue.

The input is slightly adjusted: the precision is changed to a tuple (π, π_Q) of a precision π and a stack precision π_Q. The former specifies the level of abstraction in the abstract domain, while the latter contains all location stacks which should not be popped, because the refiner found their popping in an abstract counterexample.

The body of the `while` loop decides what to do with the abstract state (q, S). Lines 6-15 contain the algorithmic modifications, where previously the covered-by edges were created. In line 6, an abstract state (q', S') is chosen, which overapproximates (q, S) with the partial order $\sqsubseteq_A$, corresponding to change 1. Lines 7-9 execute the previous behaviour,

Algorithm 5.1: Modified ABSTRACTION procedure

Input:
$ARG = (N, E, C)$: partially constructed abstract reachability graph
$D = (\mathcal{S}, \sqsubseteq, concr)$: abstract domain
(π, π_Q): current precision
T_Q: transfer function

Output: (safe or unsafe, ARG)

```
1  waitlist ← {S ∈ N | S is incomplete}
2  while waitlist ≠ ∅ do
3      (q, S) ← pop waitlist
4      if top(q) = l_E then
5          return (unsafe, ARG)
6      else if ∃(q', S') ∈ N : (q, S) ⊑_A (q', S') then
7          if (q, S) ⊑_Q (q', S') then
8              C ← C ∪ {((q, S), (q', S'))}
9              continue
10         else if q ∉ π_Q then
11             pop q
12             waitlist ← waitlist ∪ {(q, S)}
13             continue
14         end
15     end
16     foreach (q', S') ∈ T_Q((q, S), π) \ {⊥} do
17         waitlist ← waitlist ∪ {(q', S')}
18         N ← N ∪ {(q', S')}
19         E ← E ∪ {((q, S), op, (q', S'))}
20     end
21 end
22 return (safe, ARG)
```

meaning a covered-by edge is created if $(q, S) \sqsubseteq_Q (q', S')$, i.e., not only is their top location identical, but their whole stack. Under these conditions, the covered-by edge is justified by the definition of $\sqsubseteq_Q$. Lines 10-13 carry out change 2 by popping the top location of q and adding the modified abstract state back to the waitlist, so that it is only expanded in a later iteration if it is not erroneous and covered. The lack of else branches in lines 14-15 realize change 3 by continuing onto expansion if the stack to pop q was in the stack precision π_Q. This is imperative to avoid finding the same abstract counterexample in each iteration of the CEGAR loop, as it guarantees (in combination with the changes in Section 5.1.2) that an infeasible popping of an abstract state (q, S) with $q \in \pi_Q$ will not happen again.

5.1.2 Changes to Refiner

The refiner of the CEGAR algorithm is responsible for deciding the feasibility of an abstract counterexample to the program's safety. If the counterexample is infeasible, it also needs to refine the precision so that the counterexample becomes unattainable in the abstract state space as well. Refiners usually rely on SMT solvers for these tasks. The refiner's algorithm is described in Section 2.3.2.

The modifications proposed in Section 5.1 affect the refiner in the following 2 ways:

1. The abstract counterexample also needs to be checked for infeasible pops.
2. If such pop is found, then it needs to be prevented in the next iteration.

The refiner's modified pseudo-code is presented in Algorithm 5.2, with the changes highlighted in blue.

Algorithm 5.2: Modified REFINEMENT procedure

Input:
$ARG = (N, E, C)$: partially constructed abstract reachability graph
(π, π_Q): current precision
Output: (spurious or unsafe, (π, π_Q), ARG)

```
1  σ = ((q_1, S_1), op_1, ..., op_{n-1}, (q_n, S_n)) ← abstract path to unsafe node in ARG
2  if σ is feasible then
3  |  if ∃j ∈ [2, n] : |q_{j-1}| > |q_j|, top(q_j) was not final then
4  |  |  i ← lowest such j
5  |  |  π_Q ← π_Q ∪ {q_i}
6  |  else
7  |  |  return (unsafe, (π, π_Q), ARG)
8  |  end
9  else
10 |  (I_1, ..., I_n) ← interpolant for σ
11 |  (π_1, ..., π_n) ← (I_1, ..., I_n) converted to precisions
12 |  π ← π ∪ ⋃_{1≤i≤n} π_i
13 |  i ← lowest i for which I_i ∉ {true, false}
14 end
15 N_i ← all nodes in the subtree rooted at (q_i, S_i)
16 N ← N \ N_i
17 E ← {(n_1, op, n_2) ∈ E | n_1, n_2 ∉ N_i}
18 C ← {(n_1, n_2) ∈ C | n_1, n_2 ∉ N_i}
19 return (spurious, (π, π_Q), ARG)
```

The input and output precisions are changed to tuples (π, π_Q) of a precision π and a stack precision π_Q. The former specifies the level of abstraction in the abstract domain, while the latter contains all location stacks which should not be popped, because they appeared in an abstract counterexample.

The refiner gets an abstract counterexample from the ARG and decides whether it is feasible or not. Lines 3-8 on the *feasible branch* contain the algorithmic modifications, where previously the `unsafe` verdict was returned due to the σ being feasible. In line 3, the abstract counterexample is checked for having infeasible pops, corresponding to change 1. The condition $|q_{j-1}| > |q_j|$ describes that q_j has been popped, because it is smaller than the stack of the previous state. With this knowledge in mind, the interpretation of the condition $top(q_j)$ *was not final* is the location that was popped from q_j is final, which can be decided by storing the popped location in each ARG node, for example. Lines 4-5 enforce change 2 (alongside with the changes in Section 5.1.1), by adding the first such q_j in the abstract counterexample to the stack precision π_Q. The remaining lines 6-8 execute the previous behaviour of returning an `unsafe` verdict, when the counterexample is feasible with regards to popping as well.

The remainder of the algorithm is unchanged. It is worth noting, however, that lazy abstraction is applied for the change in stack precision as well. All descendants of (q_i, S_i) are removed from the graph, which is in line with what is expected from lazy abstraction: in the next iteration of the CEGAR loop, (q_i, S_i) will not be popped by the abstractor and the abstract state will have different successors.

5.2 Case Study

In this chapter, the modified verification algorithm is presented on an example C program. The built ARGs are shown for both explicit and predicate abstraction: predicate abstraction succeeds in verifying the program, while the explicit domain does not terminate. It is also demonstrated, that original CEGAR algorithm is not able to verify the example program.

Consider the following C program and its CFAs on Figure 5.3. The procedure call `reach_error()` represents the program entering some unwanted state.

```
int rec(int k) {
  if (k > 0) rec(k + 1);
  else reach_error();
}

int main() {
  rec(1);
};
```

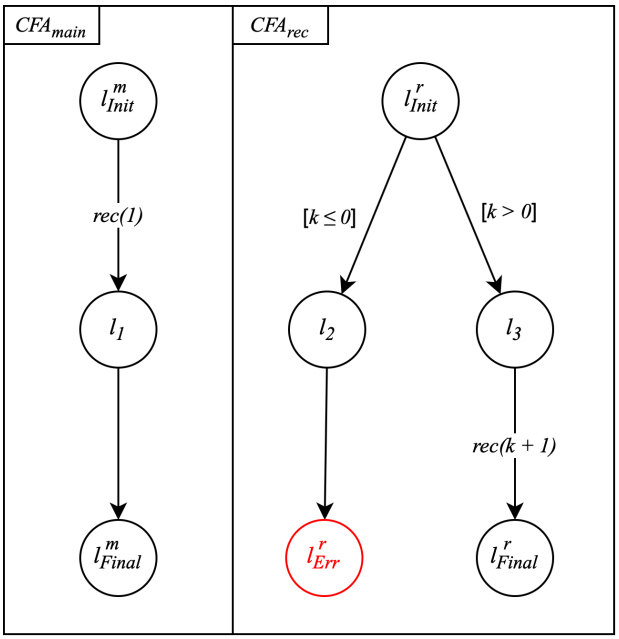


Figure 5.3: Example C program and its CFAs.

In the first iteration of the CEGAR loop the precision is empty, so no information about the value of k is available. In such case, it is assumed that k can have any value, therefore, the edge going out of l^r_{Init} guarded by $[k \leq 0]$ is available and the error location is reached. Thus, both explicit and predicate abstraction find the abstract counterexample on Figure 5.4. The erroneous abstract state is highlighted in red.

The abstract counterexample is passed onto the refiner, which checks its feasibility. One may notice that the transition guarded by $[k \leq 0]$ is not enabled in the concrete state space of the program, so the counterexample is deemed spurious and a refined precision is calculated. Given some refinement strategy, let us assume that the refiner creates the precision $\pi^e = \{k\}$ for explicit abstraction and the precision $\pi^p = \{k > 0\}$ for predicate abstraction. With this, the first iteration of the CEGAR loop is finished.

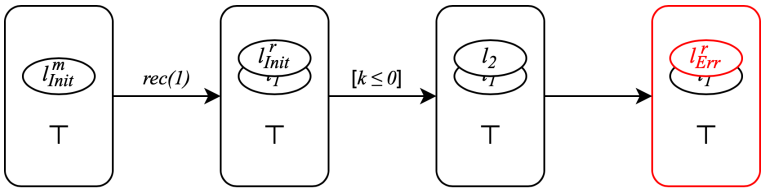


Figure 5.4: The abstract counterexample found in the first iteration.

From here, verification using explicit and predicate abstraction carries on differently. First, the second iteration with explicit abstraction is discussed.

5.2.1 Explicit Abstraction

The refined precision $\pi^e = \{k\}$ means that the value of k should be tracked while the abstract state-space is being explored. The first couple nodes of the built ARG are shown on Figure 5.5.

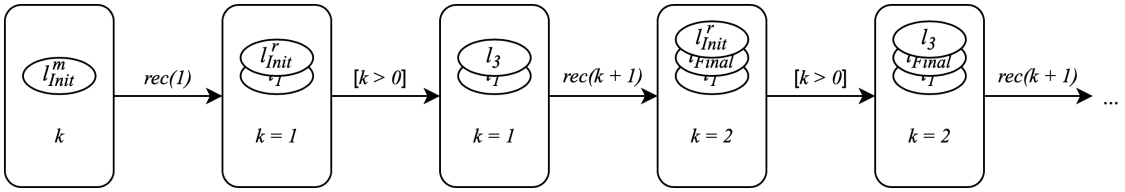


Figure 5.5: The ARG built using explicit domain.

With the first call of `rec(1)`, the tracked value of k becomes 1. Consequently, only the edge guarded by $[k > 0]$ is enabled in the second abstract state. The next call of `rec(k + 1)` changes the tracked value of k to 2. At this point, the top locations of the second and the current (fourth) node are both l^r_{Init}. However, the abstract data state $k = 1$ of the second node does not overapproximate the abstract data state $k = 2$ of the current node. Therefore, the partial order $\sqsubseteq_A$ does not hold between them, so no popping occurs.

Due to the tracked value $k = 2$, the only enabled transition in the fourth node is the edge guarded by $[k > 0]$. The fifth node has the same top location l_3 as the third one, however, their abstract data states $k = 1$ and $k = 2$ do not overapproximate one another once again, so $\sqsubseteq_A$ does not hold and no popping occurs. From here on, the value of k increases by 1 every time a new location is pushed onto the stack due to the `rec(k + 1)` procedure call. Thus, the same sequence of expansions is repeated infinitely, meaning the verification algorithm never terminates.

5.2.2 Predicate Abstraction

The refined precision $\pi^p = \{k > 0\}$ means that the truth of the predicate $k > 0$ should be tracked while the abstract state-space is being explored. The first couple of nodes of the built ARG are shown on Figure 5.6. The visualization of the ARG nodes is to be interpreted as all predicates that are displayed in a node hold in the abstract state.

With the first call of `rec(1)`, it is ensured that $k > 0$, so the predicate evaluates to true in the second node. Consequently, only the edge guarded by $[k > 0]$ is enabled in this abstract state. The next call of `rec(k + 1)` does not make the predicate false, because adding 1 to a positive number cannot make it ≤ 0. Thus, the predicate remains true in the

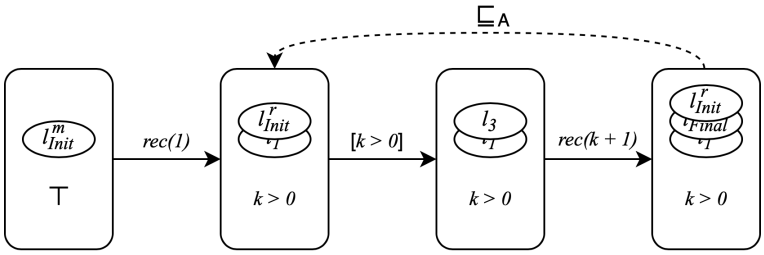


Figure 5.6: The ARG built using predicate abstraction, before popping.

fourth abstract state as well. At this point, the top locations of the second node (q_2, S_2) and the current (fourth) node (q_4, S_4) are both l^r_{Init}. The partial order $\sqsubseteq$ for predicate abstraction is implication: $\{k > 0\} \sqsubseteq \{k > 0\} \Leftrightarrow (k > 0) \implies (k > 0)$, which is true, meaning that $(q_2, S_2) \sqsubseteq_A (q_4, S_4)$, but $(q_2, S_2) \not\sqsubseteq_Q (q_4, S_4)$. According to Algorithm 5.1, this is exactly when the stack q_4 needs to be popped. The modified ARG can be seen on Figure 5.7, with the popped node highlighted in blue.

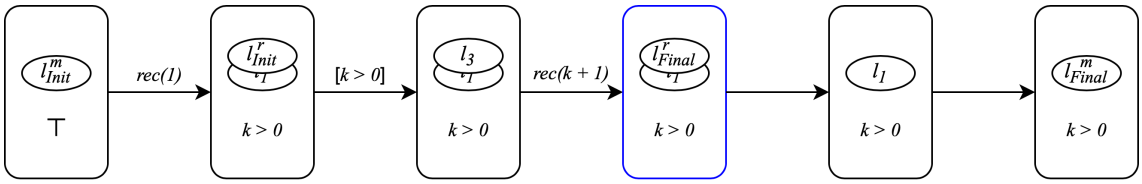


Figure 5.7: The final ARG built using predicate abstraction.

After popping, exploration continues from the popped state (q'_4, S_4), which is now in the final location l^r_{Final} of CFA_{rec}. On the next transition, the needs to be popped because a final location was reached, as described in Section 4.1. The only transition from this fifth node goes to the final location of the main procedure l^m_{Final} and with that, the ARG is fully expanded. There are no erroneous abstract states in it, therefore, the ARG is safe. It is also an overapproximation of the program's state-space, thus, the program is safe as well.

5.2.3 Comparison to CEGAR

In this section, the unmodified version of CEGAR is ran on the example program in Figure 5.3, using predicate abstraction.

In the first iteration of the CEGAR loop the precision is empty, so no information about the value of k is available. For the same reasons as with the modified version, the first iteration of the abstractor finds the abstract counterexample on Figure 5.4. Given some refinement strategy, let us assume that the refiner creates the same $\pi^p = \{k > 0\}$ precision as in the modified case. With that, the first iteration of the CEGAR loop is finished.

In the second iteration, the refined precision $\pi^p = \{k > 0\}$ means that the truth of the predicate $k > 0$ should be tracked while the abstract state-space is being explored. The first couple nodes of the built ARG are shown on Figure 5.8.

With the first call of `rec(1)`, it is ensured that $k > 0$, so the predicate evaluates to true in the second node. Consequently, only the edge guarded by $[k > 0]$ is enabled in this abstract state. The next call of `rec(k + 1)` does not make the predicate false, because adding 1 to a positive number cannot make it ≤ 0. Thus, the predicate remains true in the fourth abstract state as well. At this point, the top locations of the second node

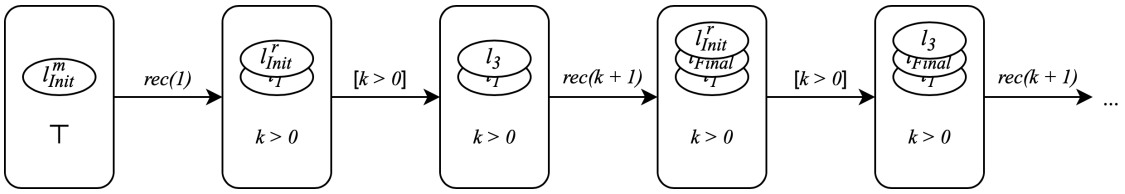


Figure 5.8: The ARG built using predicate abstraction, without popping.

(q_2, S_2) and the current (fourth) node (q_4, S_4) are both l^r_{Init} and their abstract data states imply one another. Even though their top locations is the same, for $\sqsubseteq_Q$ their whole stack would need to match. But $q_2 \neq q_4$, therefore, $(q_2, S_2) \not\sqsubseteq_Q (q_4, S_4)$ and no covered-by edge is created.

Due to the tracked predicate $k > 0$ being true in the fourth node, the only enabled transition in it is the edge guarded by $[k > 0]$. The fifth node once again has the same top location l_3 and abstract data state $\{k > 0\}$ as the third one, however, their whole stack are not identical, so $\sqsubseteq_Q$ does not hold and no covered-by edge is created between them. From here on, the height of the stack increases with each `rec(k + 1)` procedure call. Thus, the same sequence of expansions is repeated infinitely, meaning the verification algorithm never terminates.

5.3 Evaluation

In this chapter, an implementation of the modified CEGAR algorithm described in Section 5.1 is evaluated on a set of C programs. First, the benchmark environment is described, then the benchmark results are presented.

5.3.1 Benchmark Setup

The presented CEGAR modifications were implemented in Theta [28], an open-source formal model checking framework. Theta already had a highly configurable CEGAR engine with different abstraction domains, interpolation techniques and search strategies, among other options. The changes were implemented into the `xcfa` subproject of Theta in Java and Kotlin: new `Abstractor` and `Refiner` classes were created, the `Precision` class was extended, while the implementation of $\sqsubseteq_A$ was placed along other analysis utilities. The implementation is available on a fork of Theta[1] and has been merged into the main version of the tool.

The implementation was evaluated on 1219 C programs from the SV-COMP benchmark repository[2]. The verification tasks were chosen from 4 categories: 22 from *control*, 321 from *eca*, 779 from *loops* and 97 from *recursive*. The first 3 categories were chosen to measure the computational overhead of the new technique, since most of these tasks have 1-2 non-recursive procedures. The *recursive* category, on the other hand, has many recursive procedures and is a good indicator of the presented idea's efficiency.

As Theta is a highly configurable framework, 4 different configurations were tested:

- EXPL_BFS: explicit domain with sequential interpolation and BFS search strategy

[1] https://github.com/s0mark/theta/tree/interproc
[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

- EXPL_DFS: explicit domain with sequential interpolation and DFS search strategy
- PRED_BW: predicate domain with backward binary interpolation and BFS search strategy
- PRED_NWT: predicate domain with Newton-style interpolation [22] and BFS search strategy

All of the configurations above were run with lazy pruning strategy. On top of that, each configuration was benchmarked with (`POP`) and without (`NOPOP`) the presented CEGAR modification.

The execution of benchmarks was done using the BenchExec framework [10]. The tests were run on virtual machines equipped with 3 Intel Haswell/Skylake CPU cores and 16 GB of memory, in university cloud infrastructure[3]. The number of solved tasks and their execution times were measured with a 900 second timeout for each task, in order to allow for a wide variety of configurations to be tested within limited time constraints, in compliance with the benchmarking practice of SV-COMP.

5.3.2 Benchmark Results

The tested configurations only gave correct answers, meaning they either answered correctly or did not answer within the 15-minute timeout. Therefore, only the number of solved tasks of each configuration is presented on Figure 5.9. The blue and green bars correspond to the number of tasks solved by configurations with `NOPOP` and `POP`, respectively. For clarification, the exact numerical values are available in Table 5.1.

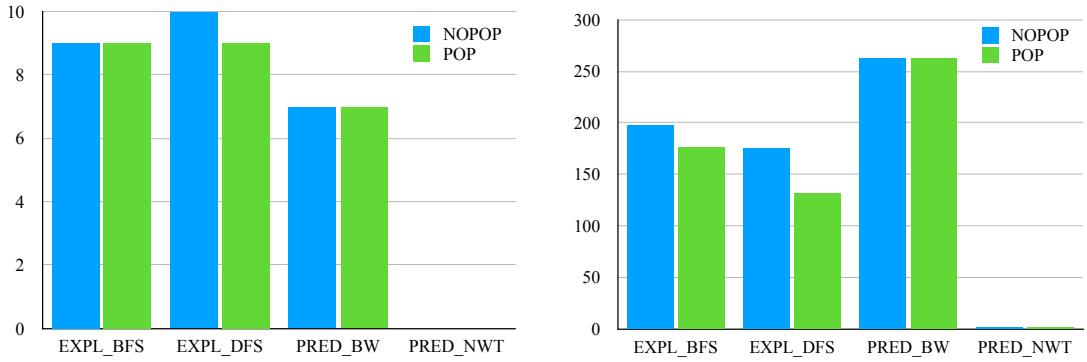
configuration	EXPL_BFS		EXPL_DFS		PRED_BW		PRED_NWT	
version	NOPOP	POP	NOPOP	POP	NOPOP	POP	NOPOP	POP
control	9	9	10	9	7	7	0	0
eca	197	177	175	131	262	263	2	2
loops	70	60	62	60	294	157	94	64
recursive	38	36	28	28	23	38	13	14

Table 5.1: Number of solved tasks by configuration.

5.3.2.1 Basic Programs

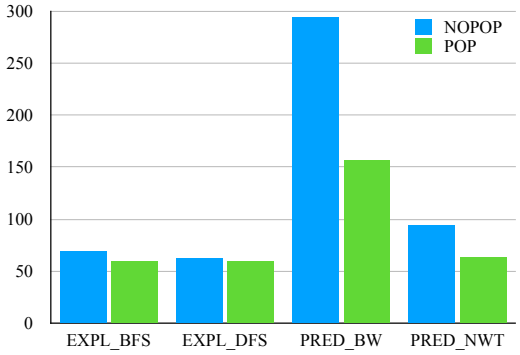
In categories *control*, *eca* and *loops*, `POP` mostly performed on par with or slightly worse than `NOPOP`. This is in line with expectations, since tasks in these categories have 1-2 non-recursive procedures. Consequently, there is no opportunity for popping to occur, but the extra checks with $\sqsubseteq_A$ are still performed, leading to a futile computational overhead. The largest difference in performance is in loop tasks, using predicate abstraction with backward binary interpolation, as seen on Figure 5.9c: the modified CEGAR version could only solve just over half as many tasks as the original version. One major contributing factor to such a drop in performance is that loop tasks only have a single procedure with a loop in it. Therefore, the size of the location stack is always one, in which case $\sqsubseteq_A$ and $\sqsubseteq_Q$ are equivalent. As a result, the same calculation is done on lines 6 & 7 in Algorithm 5.1, whereas in the original Algorithm 2.2 the computation of the partial order

[3] https://cloud.bme.hu

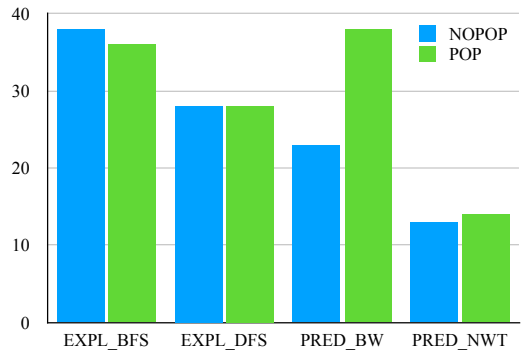


(a) Control

(b) ECA



(c) Loops



(d) Recursive

Figure 5.9: Number of tasks solved by configuration.

only happens once. The loops in the tasks' main procedures amplify the effect of the redundant calculation on performance, because the same location is visited repeatedly in the loop, leading to an increased number of computations of the partial order.

5.3.2.2 Recursive Programs

For *recursive* tasks, the two versions perform the same using explicit abstraction, as seen on Figure 5.9d. With predicate abstraction, however, the changes proposed in Section 5.1 lead to an improvement in efficiency: with backward binary interpolation, the number of solved tasks increased by over 65%. This promotes it to being the joint best configuration, matching the performance of the reigning champion EXPL_BFS which dominated the other configurations without popping. Moreover, over 20% of the programs that this configuration verified were tasks that no other `NOPOP` configuration could verify within the time constraints.

The lack of improvement using explicit abstraction can be attributed to the fundamental way recursion is used in programs: there usually is a variable k tracking the depth of the recursion. This variable gets added to the precision rather early during verification, because its value typically determines whether recursion continues or the procedure returns. Once k is in the precision of explicit abstraction, its value is tracked which blocks $\sqsubseteq_A$ from occurring between different abstract states with different sized stacks, because their abstract data states will differ in the value of k as a result of their recursion depth not being equal. Therefore, popping can not happen, hence the lack of improvement.

5.3.2.3 Threats to Validity

In this section, possible biases and threats to the validity of the benchmark results are discussed.

As mentioned in Section 5.3.1, the virtual machines used for benchmarking had either an Intel Haswell or Skylake processor. There is some 5-10% difference in the performance of these chips, which could influence the results if the configurations were assigned to different processors. The execution of benchmarks, however, was done in a distributed benchmarking environment which assigned each individual task and configuration to a random available worker. Consequentially, the difference between the chips only appears as mere noise in the results, and it can certainly not be responsible for the 60-100% efficiency losses and gains seen on Figure 5.9c and Figure 5.9d.

To get benchmark results in reasonable time, the aforementioned 15-minute limit was introduced. Given enough time, the number of solved tasks of configurations could have turned out differently. However, the verification of a program is not decidable in general, therefore, a limit always has to be put in place in practice. 15 minutes was chosen because it has been agreed upon by experts in the field of formal verification as the timeout used at SV-COMP [5], the international competition of software verification tools. Counting the solved tasks can be seen as a measure of practical performance.

Chapter 6

Contract Synthesis

Contracts are a powerful tool for software verification. They modularize the verification process and enable the use of deductive methods. Finding correct and sufficient contracts, however, requires significant engineering effort and specialized knowledge. To help alleviate that burden, contract generation would need to be automated. This chapter proposes a technique for synthesizing contracts from the output of the automated abstraction-based model checking algorithm CEGAR. An overview of the approach is presented in Figure 6.1, with the contributions of this work highlighted in bold. First, a brief description of the approach is given. Then, Section 6.1 introduces the transformations required for synthesizing contracts from an explored abstract state-space. Finally, the approach is evaluated on a case study in Section 6.2 and on a set of C programs in Section 6.3.

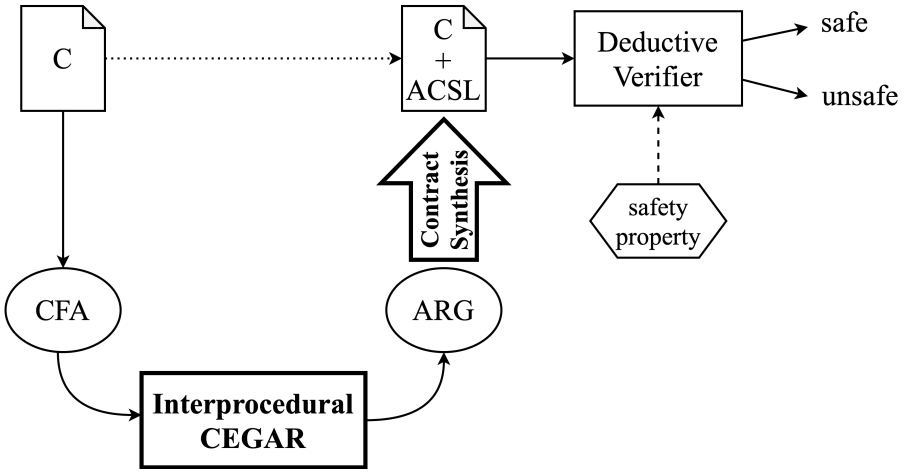


Figure 6.1: Overview of the presented approach.

In order to harness the power of deductive verification, the input program needs to be extended with contracts. Traditionally, the deductive verification workflow would require the manual labor of an engineer specialized in the field of formal methods to write the contracts, marked with the dotted line. The proposed approach automates this step by:

1. converting the input program to CFAs,
2. exploring the abstract state-space of the program using CEGAR extended with interprocedural analysis, then
3. synthesizing contracts from the generated ARG.

The synthesized contracts mainly differ from human-made contracts and the ones in Chapter 3 in that they represent a *forward* approach: they contain information about the conditions under which a procedure can be reached in the program, as opposed to a *backward* approach, which would specify the conditions that are required for some safety property to hold within the procedure. This allows the deductive verifier to build upon the reachability to verify *various* safety properties, whether it be error reachability or integer overflow. Thereby, the contracts serve as an intermediate representation for communication between the model checker algorithm and the deductive verifier.

The generated contracts convey information about the conditions a procedure can be reached under. To gather all possible abstract states, the CEGAR algorithm is run with no error detection – ignoring error locations –, so that exploration does not stop at error states and the whole ARG is expanded. Hence no refinement happens, only the abstractor runs with a certain safe verdict and a fully expanded ARG as its output. Without refinement, however, the precision has to be specified for the abstractor, as it would only be updated on refinement. The initial precision can also be generated in an automated way, using heuristics for extracting predicates and selecting variables from the program. These heuristics are described further in Section 6.1.1 for the predicate domain and Section 6.1.2 for the explicit domain. If the contracts generated with the chosen precision are too general for the deductive verifier to prove the desired safety property, one could run some iterations of CEGAR with said property in order to obtain more detailed contracts from a refined precision.

In order to attain information about the conditions each procedure is called from, the program is to be analyzed as a whole. While it is possible to inline procedures in some programs, to verify programs with (transitively) recursive procedures, a more delicate approach is required, as inlining does not terminate for these programs. The technical details of the introduction of procedures are described in Section 2.4, while Chapter 4 discusses the extension of abstract states with locations stacks. Then, Section 6.1 presents how contracts can be synthesized from a fully explored abstract state-space.

6.1 Transforming Abstract States to Contracts

The goal of this transformation is to synthesize contracts from the abstract state-space of the program, so that deductive verification can be employed for verifying safety properties, without the necessity of specialized engineers writing the contracts manually. The novelty of the CEGAR-based approach, enabled by the interprocedural analysis extensions presented in Chapter 4, is that it is *context-aware*, i.e., the synthesized contracts describe the context a procedure can be called from, as opposed to the requirements for the procedures correct behavior. This allows the generated contracts to be used by deductive verifiers for proving various safety properties, such as error reachability or integer overflow.

The core idea of the transformation is to collect all abstract states from the ARG in which the program was in the initial location of a procedure. For each procedure, the collected states can be joined together to obtain all states from which the procedure can be called from in the program. In other words, these states describe all the possible preconditions of the procedure within the context of the program, thus, they can be used as a contract to empower deductive verifiers.

The algorithm for synthesizing contracts is presented in Algorithm 6.1. It generates the abstract state-space represented by an ARG and transforms it into procedure contracts. The transformation is done from a CFA to contracts written in the ANSI C Specification

Language (ACSL), described in Section 2.5.1. The parsing of code to CFA as well as the enrichment of the code with the ACSL contracts is omitted from the presentation for the sake of simplicity.

Algorithm 6.1: Contract Synthesis

Input:
$Ps = \{P_i \mid P_i = (CFA_i, I_i, O_i)\}$: program's procedures
$D = (\mathcal{S}, \sqsubseteq, concr)$: abstract domain
π: initial precision
Output: $PCs = \{(Pre_{P_i}, Post_{P_i}) \mid \forall P_i \in Ps\}$: contracts of each procedure

1 $Ps' \leftarrow \{(CFA'_i, I_i, O_i) \mid \forall P_i = (CFA_i, I_i, O_i) \in Ps, CFA'_i = \text{noerrorlocs}(CFA_i)\}$
2 *result, ARG* $\leftarrow \text{CEGAR}(Ps', D, \pi, T_Q)$
3 $(N, \_, \_) \leftarrow ARG$
4 $PCs \leftarrow \varnothing$
5 **forall** $P_i = (CFA_i, I_i, O_i)$ **in** Ps **do**
6 $\quad (V, L, l_0, E) \leftarrow CFA_i$
7 $\quad R \leftarrow \{S \mid \forall (q, S) \in N : top(q) = l_0\}$
8 $\quad$ **if** $R = \varnothing$ **then**
9 $\quad\quad Pre_{P_i} \leftarrow$ ""
10 $\quad$ **else**
11 $\quad\quad Pre_{P_i} \leftarrow$ "`/*`"
12 $\quad\quad$ **forall** S **in** R **do**
13 $\quad\quad\quad S \triangleright S'$
14 $\quad\quad\quad Pre_{P_i} \leftarrow Pre_{P_i} \cdot$ "`@ behavior` i`: requires` S'`;`"
15 $\quad\quad$ **end**
16 $\quad\quad Pre_{P_i} \leftarrow Pre_{P_i} \cdot$ "`@ complete behaviors; */`"
17 $\quad$ **end**
18 $\quad PCs \leftarrow PCs \cup \{(Pre_{P_i}, \varnothing)\}$
19 **end**
20 **return** PCs

Building the ARG The input of the algorithm contains the program's procedures Ps, an abstraction D and an initial precision π of the provided domain. The procedures are cleansed from error locations, represented by the *noerrorlocs* call in line 1, so that the CEGAR algorithm – defined in Algorithm 2.1 – can run without error detection in line 2.

Since the input to the model checking algorithm does not contain error locations, the CEGAR loop will only execute the abstractor once, the return value of which is guaranteed to be safe along with a fully expanded ARG. Consequently, the refiner is not executed, meaning the precision does not change. Thus, the ARG only contains information that was specified to be tracked in the initial precision, hence it cannot be left empty as it is done when using CEGAR for verification. Heuristics for initial precision extraction from the input are described in Section 6.1.1 and Section 6.1.2 for the predicate and explicit domain, respectively.

Running CEGAR in this manner allows the generated contracts to be reused for various safety properties, e.g. error reachability or integer overflow. However, if the initial precision turns out to be too general for the deductive verifier to prove a safety property, this reusability can be traded for more detailed contracts by running the CEGAR loop for some iterations with some safety property.

Extracting Contracts To extract contracts from the fully expanded ARG for all procedures, the algorithm iterates through the procedures in lines 5-19. For each procedure P_i, in line 7 it collects the data state S of each ARG node that was in the initial location of the procedure, i.e., the top of the abstract state's location stack is equal to the initial location of P_i. By collecting all such data states, the set R will contain all possible states of the program in which the procedure could have been called, since the abstraction is overapproximating.

Lines 8-17 convert the collected set of abstract data states R into contracts. If no such abstract state was found, then the procedure cannot be called in the program, therefore, no contract is generated for it. On the other hand, if there are such states, then the nested loop on lines 12-15 converts each $S \in R$ abstract state into a separate behavior with its own precondition using the $\triangleright$ translation operator. These behaviors are nested in an ACSL-style comment and are appended by `complete behaviors` in line 16, conveying the fact that the abstract states of R are the only possible conditions under which the procedure can be called.

The translation operator $\triangleright$ converts an abstract data state into an ACSL contract predicate. It is defined partially with regards to both context and typing, in the style of sequent calculus [15]. For literals and variables in an abstract data state S, it is defined as:

$$\frac{}{S \vdash \top \triangleright \texttt{\textbackslash true}} \qquad \frac{}{S \vdash \bot \triangleright \texttt{\textbackslash false}}$$

$$\frac{}{S \vdash n : \text{integer} \triangleright n} \qquad \frac{}{S \vdash v : \text{var} \triangleright \text{name}(v)}$$

Semantically, this corresponds to the left hand operand of $\triangleright$ being converted to the right hand operand within the context of the abstract data state S, e.g. if $\top$ is found during the conversion of S, then it will be translated to `\true`. Integer literals are left in place, while variables are translated to their names in the program code, represented by the *name* function.

The remaining definition of $\triangleright$ is domain-dependent, therefore, the extensions to the definition above are described in Section 6.1.1 for the predicate domain and in Section 6.1.2 for the explicit domain.

6.1.1 Predicate Domain

In predicate abstraction, as introduced in Algorithm 2.2, an abstract data state is a combination of first order logic predicates over the variables. Thus, the translation of an abstract data state in this domain consists of mapping logic operations over boolean formulas ϕ and ψ, arithmetic operations over integers α and β as well as the if-then-else (ite) operation over ω – representing either integer or logic types – to their corresponding operations in ACSL:

$$\frac{S \vdash \alpha \triangleright \alpha'}{S \vdash +\alpha \triangleright \texttt{+}\alpha'} \qquad \frac{S \vdash \alpha \triangleright \alpha'}{S \vdash -\alpha \triangleright \texttt{-}\alpha'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha + \beta \triangleright \alpha' \texttt{ + } \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha - \beta \triangleright \alpha' \texttt{ - } \beta'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \cdot \beta \triangleright \alpha' \texttt{ * } \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \div \beta \triangleright \alpha' \texttt{ / } \beta'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha = \beta \triangleright \alpha' \texttt{ == } \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \neq \beta \triangleright \alpha' \texttt{ != } \beta'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha > \beta \triangleright \alpha' \texttt{ > } \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \geq \beta \triangleright \alpha' \texttt{ >= } \beta'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha < \beta \triangleright \alpha' \texttt{ < } \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \leq \beta \triangleright \alpha' \texttt{ <= } \beta'}$$

$$\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \bmod \beta \triangleright \alpha' \texttt{ \% } \beta'} \qquad \frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \oplus \psi \triangleright \phi' \texttt{ \^{}\^{} } \psi'}$$

$$\frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \wedge \psi \triangleright \phi' \texttt{ \&\& } \psi'} \qquad \frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \vee \psi \triangleright \phi' \texttt{ || } \psi'}$$

$$\frac{S \vdash \phi \triangleright \phi' \quad S \vdash \omega_1 \triangleright \omega_1' \quad S \vdash \omega_2 \triangleright \omega_2'}{S \vdash \mathrm{ite}(\phi, \omega_1, \omega_2) \triangleright \phi' \texttt{ ? } \omega_1' \texttt{ : } \omega_2'} \qquad \frac{S \vdash \phi \triangleright \phi'}{S \vdash \neg\phi \triangleright \texttt{!}\phi'}$$

Finding the right predicates to track during verification is one of the main challenges of automated verification. The following heuristics can be used for selecting the initial precision:

- `ALLASSUMES`: All predicates in guards or asserts.
- `PARAMASSUMES`: All predicates in guards or asserts on procedure parameters as well as on variables that appear in arguments of a procedure call.

The `ALLASSUMES` heuristic is more detailed but also more wasteful, while `PARAMASSUMES` makes a compromise on precision for efficiency gains by tracking less predicates.

6.1.2 Explicit Domain

In explicit-value abstraction, as introduced in Section 2.2, an abstract data state is a mapping of selected variables to literal values from their domain. Thereby, the translation of an abstract data state in this domain can be defined by the conversion of a mapping to en equality assumption:

$$\frac{S \vdash v \triangleright v' \quad S \vdash n \triangleright n'}{S \vdash S(v) = n \triangleright v == n}$$

The power of explicit-value abstraction lies in selecting the correct variables to track the values of. The following heuristics can be used for selecting the initial precision:

- `ALLVARS`: All variables of the program.
- `ALLASSUMES`: All variables that appear in a guard or assert in the CFAs of the program.
- `PARAMASSUMES`: All parameters of procedures, as well as variables that appear in arguments of a procedure call.

Of these heuristics, `ALLVARS` is the most detailed but also the most wasteful, while `ASSUMEVARS` and `PARAMVARS` make different compromises on precision to gain efficiency by abstracting some variables away.

6.2 Case Study

In this section, the contract synthesis process is presented on an example C program. The abstract state-space is explored in the predicate domain to demonstrate the power of abstraction on unknown values, where explicit value-tracking would be of no use.

Consider to program and its CFAs in Figure 6.2. It contains a naive implementation of the absolute value function for integers, which is used on an unknown positive value in the main function. The havoc edge on the variable n is omitted for simplicity, its value is considered to be unknown regardless.

```
int abs(int k) {
    if (k < 0) {
        return −k;
    } else {
        return k;
    }
}

int main() {
    int n;
    if (n > 0) {
        abs(n);
    }
};
```

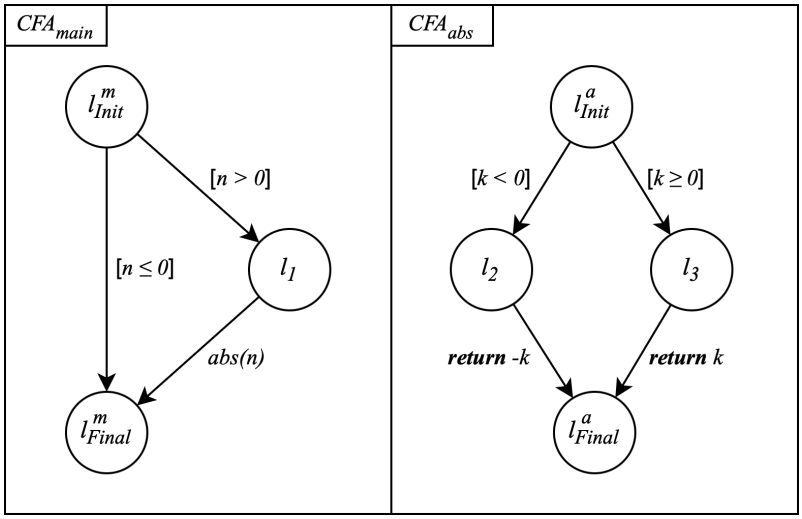


Figure 6.2: Example C program and its CFAs.

However, the implementation of `abs` is incorrect. If its input parameter k is equal to `INT_MIN` (i.e. –2147483648 for 32-bit integers), the negation in the first branch of the if will result in integer overflow, leading to undefined behavior. A deductive verifier that

is capable of detecting overflow would find this potential error as it goes through each procedure, but it would not be able to provide a verdict on the safety of the program, as it does not know whether or not there is a concrete execution of the program that will lead to an overflow.

Looking at the program through an interprocedural lens, one may notice that no overflow will actually occur in this program: even though the `abs` procedure alone could produce an overflow, it is only called with positive values in the context of the program, for which the implementation is correct. Let us see, how the presented contract synthesis method could provide the deductive verifier with the necessary information to reach this conclusion.

First, the interprocedural CEGAR algorithm is used to explore the abstract state-space of the program. The initial precision can be defined using either the `ALLASSUMES` or `PARAMASSUMES` heuristic, since both result in the same $\pi = \{n > 0, k < 0\}$ precision due to the fact that all guards in the program include parameters of a procedure or arguments of a procedure call. The explored abstract state-space is represented as an ARG in Figure 6.3. A predicate that is known to be false is shown after a semantic conversion, e.g. as $k \geq 0$ instead of $\neg(k < 0)$.

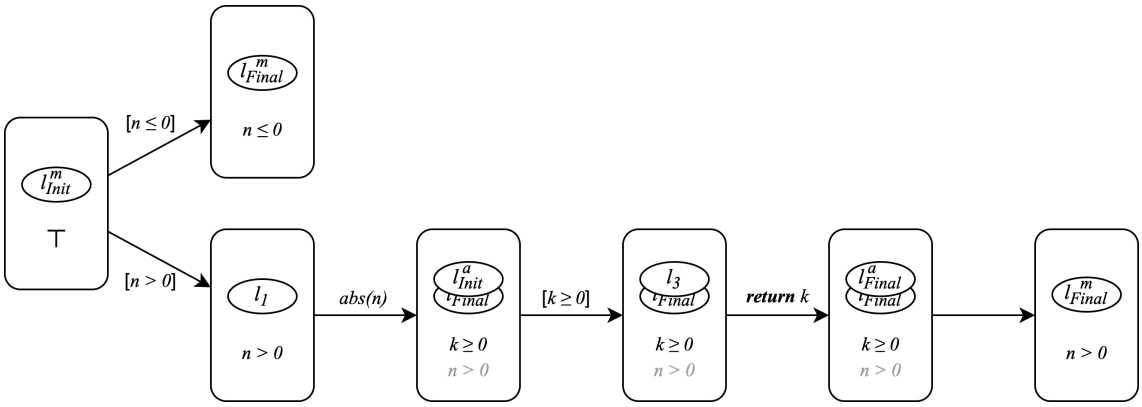


Figure 6.3: The fully expanded ARG of the example program.

The guard of the branching expression in the main procedure creates two paths in the abstract state-space: one going to l_1 where $n > 0$ is satisfied, and one going to the final location l^m_{Final} where the same predicate is known to be false. The former continues with a call to `abs`, where the other predicate in the precision $k < 0$ is guaranteed to be false, as the argument n of the procedure is known to be greater than 0. The abstract state in the initial location l^a_{Init} of the called procedure is therefore extended with the $k \geq 0$ predicate. Consequently, only the else branch of the procedure needs to be explored, which returns the value of k as is. With the final location l^a_{Final} of the procedure reached, exploration comes to an end at the final location l^m_{Final} of the program. Note, how the CEGAR algorithm does not need to know what property is to be verified as it only does an exploration of the abstract state-space.

With the ARG built up, the abstract states in the initial location of the `abs` procedure can be collected. There is only one such node in the ARG, where the abstract data state contains the predicates $k \geq 0$ and $n > 0$. Filtering for predicates with variables present in the procedure, the $k \geq 0$ predicate can be translated to $k \geq 0 \triangleright k$ `>=` 0, which is then wrapped in a behavior. The ACSL contract synthesized from the ARG can be seen in Figure 6.4.

Adding the synthesized contract with interprocedural cues to the program helps the deductive verifier prove safety. Going through each procedure, it would find that the context-

```
/* @ behavior abs_1: requires k >= 0;
   @ complete behaviors; */
int abs(int k) {
    if (k < 0) {
        return -k;
    } else {
        return k;
    }
}

int main() {
    int n;
    if (n > 0) {
        abs(n);
    }
};
```

Figure 6.4: Example C program with the synthesized ACSL contract.

aware preconditions described by the contract of `abs` rules out the branch with the possible integer overflow. Thus, the verifier would conclude that the program is indeed safe from integer overflows.

6.3 Evaluation

In this chapter, an implementation of the contract synthesis method described in Section 6.1 is evaluated along with the deductive verifier Frama-C [30] on a set of C programs. First, the benchmark environment is described, then the benchmark results are presented.

6.3.1 Benchmark Setup

The presented contract synthesis was implemented in Theta [28], an open-source formal model checking framework. Theta already had a highly configurable CEGAR engine with different abstraction domains, interpolation techniques and search strategies, among other options. The changes were implemented into the `xcfa` subproject of Theta in Java and Kotlin. A new `arg2acsl` package was created with for the contract synthesis. The implementation is available on a fork of Theta[1].

Theta was chained together with the SV-COMP version of Frama-C [8] to benchmark the potential benefits of this combined approach. The contracts were synthesized and placed in the program by the implementation in Theta, which was then fed to Frama-C for verification. Frama-C was used to verify error reachability by employing deductive verification using the Coq theorem prover [4], as well as to check integer overflow enabled by its non-deductive Evolved-Value Analysis (EVA) plugin [16]. In all cases, the configurations specified in the SV-COMP submission of the tool were used.

The toolchain was evaluated on 1213 error-reachability tasks and 1625 overflow tasks from the SV-COMP benchmark repository[2].

[1] https://github.com/s0mark/theta/tree/contract-synthesis
[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

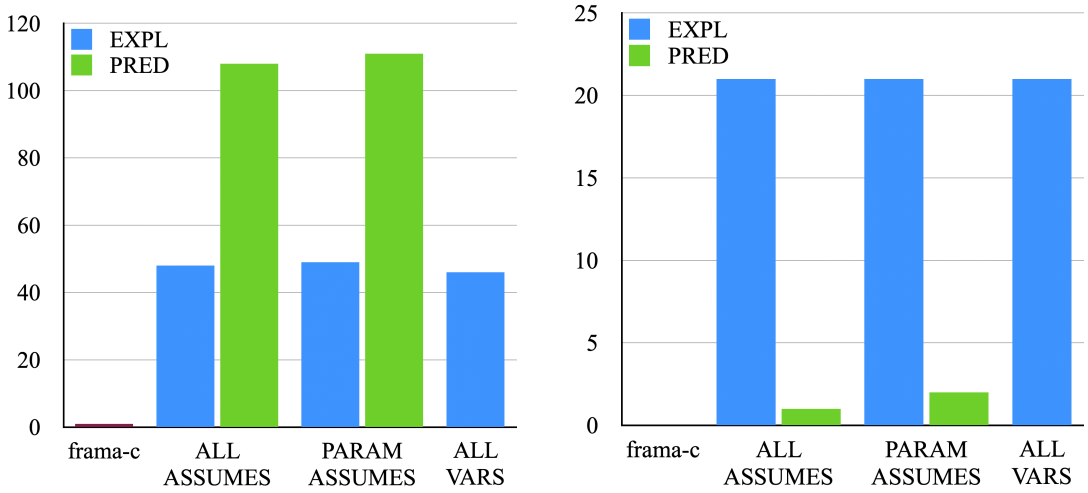
- For reachability, the verification tasks were chosen from 4 categories: 22 from *control*, 321 from *eca*, 779 from *loops* and 97 from *recursive*. The first 3 categories were chosen to measure the computational overhead of the new technique, since most of these tasks have 1-2 non-recursive procedures. The *recursive* category, on the other hand, has many recursive procedures and is a good indicator of the presented idea's efficiency.
- For overflow, the benchmarks of the regular overflow track of SV-COMP were used.

The contract synthesis of Theta was configured both with predicate and explicit abstraction. Each of them were tested with their corresponding initial precisions presented in Section 6.1.1 and Section 6.1.2.

The execution of benchmarks was done using the BenchExec framework [10]. The tests were run on virtual machines equipped with 3 Intel Xeon (Skylake) CPU cores and 15 GBs of memory, in the university cloud infrastructure[3]. The number of solved tasks and their execution times were measured with a 900 second timeout for each task, in order to allow for a wide variety of configurations to be tested within limited time constraints, in compliance with the benchmarking practice of SV-COMP.

6.3.2 Reachability Results

The tested configurations only gave correct answers, meaning they either answered correctly or did not answer within the 15-minute timeout. Therefore, only the number of solved tasks of each configuration is presented on Figure 6.5. The blue and green bars correspond to the number of tasks solved by configurations in the explicit and predicate domain, respectively. For clarification, the exact numerical values are available in Table 6.1.



(a) Loops **(b)** Recursive

Figure 6.5: Number of reachability tasks solved by configuration.

In categories *control* and *eca*, Frama-C was not able to verify a single task within the specified constraints, regardless of the added contract. For *eca* tasks, Frama-C always ran out of memory, probably due to the fact that these tasks are relatively large. For the *control* tasks, it either timed out or returned *unknown*, that is, it is neither able to

[3]https://cloud.bme.hu

category	Loops		Recursive	
configuration	EXPL	PRED	EXPL	PRED
frama-c	1		0	
ALLASSUMES	48	108	21	1
PARAMASSUMES	49	111	21	2
ALLVARS	46	-	21	-
theta	86	258	51	19

Table 6.1: Number of solved reachability tasks by configuration.

prove or refute the property in question. Thus, only the results for *loops* and *recursive* are presented in Figure 6.5 and Table 6.1.

For *loops* and *recursive* tasks, FRAMA-C was only able to verify a single task without contracts. The contracts synthesized by the presented approach enabled the deductive tool to verify a combined 156 C programs between all tracks and configurations, with the initial precision not making much of a difference. Thus, the approach does succeed in making deductive verification fully automated. On the downside, the combined approach was only able to solve about 50% of the tasks that THETA as a standalone verification tool was able to. However, there were a handful of tasks that THETA was not able to verify alone, but the combined toolchain successfully proved.

6.3.3 Overflow Results

For overflow tasks, FRAMA-C gives both correct and wrong results, with and without the synthesized contracts, thus, its evaluation is limited. The number of tasks solved correctly and incorrectly by each configuration can be seen in Table 6.2. The number of *new* tasks is also presented in the table and in Figure 6.6, that is, tasks which FRAMA-C could only verify – correctly or incorrectly – with the synthesized contracts. The green and red bars correspond to the number of correct and wrong verdicts, respectively.

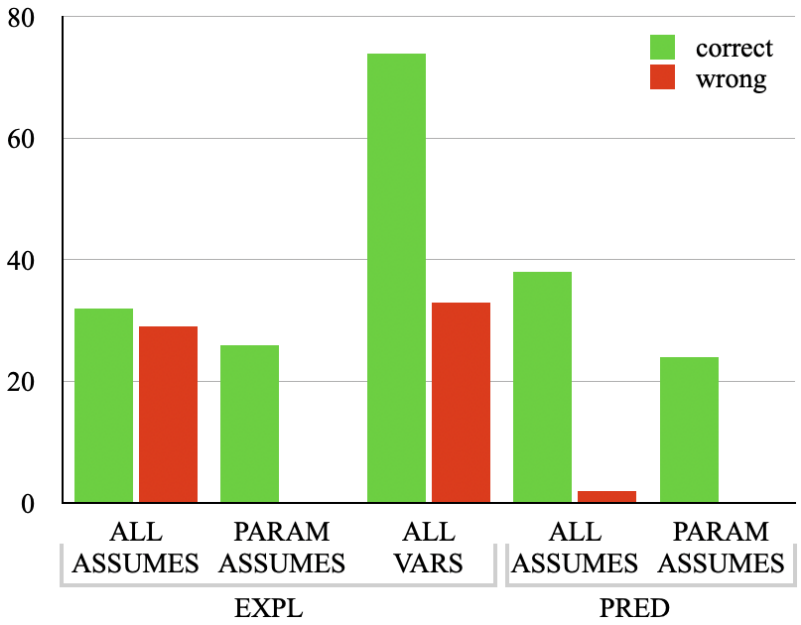


Figure 6.6: Newly solved overflow tasks by configuration and correctness

			EXPL			PRED	
			ALL	PARAM	ALL	ALL	PARAM
type	solution	frama-c	ASSUMES		VARS	ASSUMES	
solved	correct	665	251	291	266	300	319
solved	wrong	4	31	2	35	4	2
new	correct	-	32	26	74	38	24
new	wrong	-	29	0	33	2	0

Table 6.2: Number of solved overflow tasks by configuration and correctness.

The wrong solutions are discussed in Section 6.3.4. This section focuses on tasks that were influenced by this work, i.e., tasks Frama-C could only verify with contracts.

Once again, the combined approach was not more efficient than running a specialized tool alone, with the best performing configuration solving about 50% as many tasks as Frama-C on its own. However, the synthesized contracts did allow Frama-C to verify an additional 101 tasks, a 15% increase over what the tool could solve on its own. As opposed to the reachability benchmarks, the initial precision does make a significant difference in both the number of correctly and incorrectly solved tasks. Contracts synthesized using the `PARAMASSUMES` heuristic added no new wrong solutions, but they also increased the number of correct tasks by the least out of all configurations. The contracts generated in the explicit domain added more wrong solutions than ones coming from the predicate domain, which could be related to the fact that Frama-C uses its EVA plugin for proving overflow, as opposed to deductive methods.

Overall, the approach was able to help Frama-C verify more tasks. For all new tasks that the toolchain was able to verify, there was enough time to run Frama-C alone first, wait for the inconclusive result, then run the toolchain and get the correct solution all within the specified time limit. However, with the amount of incorrect solutions, it is tough to make conclusions.

6.3.4 Threats to Validity

In this section, possible biases and threats to the validity of the benchmark results are discussed.

To get benchmark results in reasonable time, the aforementioned 15-minute limit was introduced. Given enough time, the number of solved tasks of configurations could have turned out differently. However, the verification of a program is not decidable in general, therefore, a limit always has to be put in place in practice. 15 minutes was chosen because it has been agreed upon by experts in the field of formal verification as the timeout used at SV-COMP [5], the international competition of software verification tools. Counting the solved tasks can be seen as a measure of practical performance.

While all answers were correct in the reachability benchmarks, the toolchain provided a significant amount of wrong answers for overflow tasks. The wrong solutions could be caused by misconfiguration or non-compatible Frama-C versions, though the newest release of the tool was used with the configurations specified in its SV-COMP submissions, where it participated in the overflow track. They may just be caused by bugs in Frama-C or its EVA plugin. The newly added wrong tasks could be related to bugs Theta, as the tool does not participate in the overflow track of SV-COMP, thus, the support for overflows is not thoroughly tested.

As both Frama-C and Theta are highly configurable frameworks, there is a reasonable chance that the tools were not run in their best performing configurations in each scenario. There is always a trade-off in spending time on finding the best configurations of tools, as the amount of work put in usually has diminishing returns. The effort made in this work was running Theta in multiple configurations and using the configurations of Frama-C that the author chose to submit it with for SV-COMP, for each safety property.

Chapter 7

Conclusion

As technology is integrated into an increasing part of our lives, more and more tasks are automated using software. In addition to using it as a means of communication and entertainment, software is also used in safety-critical systems, such as cars and spaceships. In such systems, failure can lead to catastrophes, therefore, their correctness needs to be guaranteed. While conventional testing can only show the presence of incorrect behavior, formal verification can mathematically prove the absence of errors as well.

In one approach to formal verification, model checking is employed to explore the state-space of the program and look for erroneous states in it. The most challenging part of the method is the state-explosion problem, that is, the size of the state-space grows exponentially with the number of variables in the program. Procedures make interprocedural verification even more challenging by extending the state of a program with the call stack, which can lead to an infinitely large state-space. To counteract this, reduction techniques can be applied to the state-space of the program, such as abstraction. Traditional abstraction-based methods group states together by abstracting away some information from the data state of the program, e.g. the values of some variables. The abstraction-based model checking algorithm CEGAR was presented in Section 2.3.

In Chapter 4, I presented a variant of the CEGAR algorithm with interprocedural analysis. In addition to the adjustments to handling control- and dataflow, the partial order $\sqsubseteq_Q$ was introduced to keep the analysis sound. The extensions allowed the model checking algorithm to verify (transitively) recursive programs, while also laying the foundations for my two main contributions of this work.

Stack Abstraction In Chapter 5, I enhanced on the aforementioned interprocedural analysis with a novel approach, in order to improve the efficiency of interprocedural verification. The two partial orders in Section 5.1 were combined and integrated into CEGAR in a way that keeps the algorithm sound but makes it more performant. Changes necessary to the abstractor were discussed in Section 5.1.1, while the refiner's modifications were described in Section 5.1.2. The modified algorithm was presented in a case study in Section 5.2, where it was shown that the modified CEGAR algorithm can verify certain infinitely recursive programs, which it was not able to by default, without the modifications. The presented approach was implemented in the open-source model checking framework Theta. The implementation was evaluated in Section 5.3 on 1219 C programs in different configurations of the tool. All configurations gave only correct answers, meaning they either verified the program correctly or did not give an answer within the 15-minute timeout. For tasks without procedures, the computational overhead of the modifications

degraded performance to a varying degree: in most cases there was no or only a slight decrease in the number of solved tasks, in the worst case it was halved. For recursive tasks, the performance was improved by as much as 65% with predicate abstraction, promoting it to being the best configuration. Furthermore, over 20% of the programs that it verified were tasks that no other configuration could verify within the time constraints.

The other main contribution of this work was related to deductive verification, a different approach to formal verification. It can verify whether or not a program conforms to its specification, which can be specified by contracts. However, designing contracts that are useful for verification is a complex and tedious process, where engineers need to have a strong formal background.

Contract Synthesis In Chapter 6, I proposed a technique for synthesizing contracts, in order to enable automated deductive verification. The approach used the interprocedural model checking algorithm introduced in Chapter 4 to explore the abstract state-space of the program, from which contracts were synthesized that were then passed onto the deductive verifier. With each step being automated, the costly labor of manually writing contracts was eliminated. The novel contribution of the method was that the contract generation is *context-aware* in the sense that the generated contracts only consider conditions that can actually occur in each procedure of the program. In Section 6.1, heuristics were devised based on the structure of the procedures, in order to find the right level of abstraction. The proposed approach was also implemented in the open-source model checking framework Theta. The implementation was evaluated in Section 6.3, combined with the deductive verification platform Frama-C in a toolchain. The deductive method was only able to verify a single program's reachability by itself, which number increased to 156 when synthesized contracts were provided. Thus, the toolchain was able make the verification of overflow and error reachability automated. Additionally, there were tasks for both properties, which neither Theta or Frama-C was able to prove by itself, but the combined toolchain succeeded in verifying.

7.1 Future Work

For stack abstraction, the number of required CEGAR loop iterations could be reduced by putting more consideration into what is added to the stack precision π_Q. If a location l is found to have been impossibly popped in the abstract counterexample, the location stack of the popped abstract state gets added to π_Q. In the current version, this is followed by adding l's descendant locations to π_Q in separate CEGAR iterations. These iterations could be avoided by adding them at once when the impossible pop of l is detected, one just needs to figure out which of l's descendants need to be added.

A longer term goal is to extend popping with summaries. Currently, when a stack is popped in accordance with the presented idea, the return variables of the abstracted procedure call are left uninitialized. This could be improved upon by applying a summary of the procedure to the popped abstract state, resulting in a more precise abstraction.

As for contract synthesis, it would be worth experimenting with more iterations of CEGAR instead of just a single run of the abstractor. This would lose some of the generality of the synthesized contracts, but could result in improved efficiency for verifying certain safety properties.

On the longer term, verification witnesses [11] could be utilized instead of the ARG as the source of contract inference. This would widen the combination possibilities of Theta

with other tools. Moreover, it would allow the chaining of any tool generating witnesses to ones that utilize ACSL contracts, further enhancing collaboration between different verification methods [12].

In addition to preconditions, postconditions could also be synthesized from the ARG. The extraction of postconditions for each precondition would be a more involved process, as it would require walking through the ARG for each behavior. It would be a worthwhile investment, as the addition of postconditions could significantly help the deductive verifier.

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X.

[2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45319-2.

[3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specification language. In *ACSL: ANSI/ISO C Specification Language*, 2008. URL `https://api.semanticscholar.org/CorpusID:122818859`.

[4] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Springer Berlin, Heidelberg, 01 2004. ISBN 3540208542. DOI: `10.1007/978-3-662-07964-5`.

[5] Dirk Beyer. State of the art in software verification and witness validation: Sv-comp 2024. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–329, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57256-2. DOI: `10.1007/978-3-031-57256-2_15`.

[6] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.

[7] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1.

[8] Dirk Beyer and Martin Spiessl. The static analyzer frama-c in sv-comp (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 429–434, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99527-0. DOI: `10.1007/978-3-030-99527-0_26`.

[9] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, October 2007.

[10] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21

(1):1–29, Feb 2019. ISSN 1433-2787. DOI: 10.1007/s10009-017-0469-y. URL https://doi.org/10.1007/s10009-017-0469-y.

[11] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4), September 2022. ISSN 1049-331X. DOI: 10.1145/3477579.

[12] Dirk Beyer, Martin Spiessl, and Sven Umbricht. Cooperation between automatic and interactive software verifiers. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods*, pages 111–128, Cham, 2022. Springer International Publishing. ISBN 978-3-031-17108-6. DOI: 10.1007/978-3-031-17108-6_7.

[13] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49059-3. DOI: 10.1007/3-540-49059-0_14.

[14] Richard Bubel, Reiner Hähnle, and Maria Pelevina. Fully abstract operation contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 120–134, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45231-8. DOI: 10.1007/978-3-662-45231-8_9.

[15] Samuel R. Buss. Chapter i - an introduction to proof theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 1–78. Elsevier, 1998. DOI: https://doi.org/10.1016/S0049-237X(98)80016-5.

[16] David Bühler. *EVA, an Evolved Value Analysis for Frama-C : structuring an abstract interpreter through value and state abstractions*. PhD thesis, University of Rennes, 2017. URL http://www.theses.fr/2017REN1S016. Thèse de doctorat dirigée par Blazy, Sandrine et Yakobowski, Boris Informatique Rennes 1 2017.

[17] Prantik Chatterjee, Jaydeepsinh Meda, Akash Lal, and Subhajit Roy. Proof-guided underapproximation widening for bounded model checking. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 304–324, Cham, 2022. Springer International Publishing. ISBN 978-3-031-13185-1. DOI: 10.1007/978-3-031-13185-1_15.

[18] Yu-Fang Chen, Chiao Hsieh, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. Verifying recursive programs using intraprocedural analyzers. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis*, pages 118–133, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10936-7. DOI: 10.1007/978-3-319-10936-7_8.

[19] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, sep 2003. ISSN 0004-5411. DOI: 10.1145/876638.876643.

[20] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[21] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[22] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. newton in software model checking. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 487–497, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. DOI: 10.1145/3106237.3106307.

[23] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. DOI: 10.1145/360933.360975.

[24] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J., 1976. ISBN 013215871X; 9780132158718.

[25] Pontus Ernstedt. *Contract-Based Verification in TriCera*. PhD thesis, Uppsala University, 2022. URL https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-474539.

[26] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, Oct 2011. ISSN 1433-2787. DOI: 10.1007/s10009-011-0211-0.

[27] Ákos Hajdú. *Effective Domain-Specific Formal Verification Techniques*. Phd thesis, Budapest University of Technology and Economics, 2020. URL http://hdl.handle.net/10890/13523.

[28] Ákos Hajdu and Zoltán Micskei. Efficient strategies for cegar-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, Aug 2020. ISSN 1573-0670. DOI: 10.1007/s10817-019-09535-x. URL https://doi.org/10.1007/s10817-019-09535-x.

[29] Cliff Jones, A. Roscoe, and Kenneth Wood. *Reflections on the Work of C.A.R. Hoare*. Springer London, 01 2010. ISBN 1848829116, 9781848829114. DOI: 10.1007/978-1-84882-912-1.

[30] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. ISSN 1433-299X. DOI: 10.1007/s00165-014-0326-7.

[31] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 427–443, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31424-7. DOI: 10.1007/978-3-642-31424-7_32.

[32] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. DOI: 10.1109/2.161279.

[33] Andreas Podelski, Ina Schaefer, and Silke Wagner. Summaries for while programs with recursion. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 94–107, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0. DOI: 10.1007/978-3-540-31987-0_8.

[34] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916921. DOI: 10.1145/199448.199462.

[35] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 451–471, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.

[36] M Sharir and A Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, 1981.

[37] John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David R. Cok. Inferring concise specifications of apis. *ArXiv*, abs/1905.06847, 2019. URL https://api.semanticscholar.org/CorpusID:155100095.

[38] Márk Somorjai. Abstraction Based Techniques for Constrained Horn Clause Solving. Bachelor's thesis, Budapest University of Technology and Economics, 2023. URL https://diplomaterv.vik.bme.hu/en/Theses/Absztrakcio-alapu-technikak-CHC-problemak.

[39] Taku Terao. Lazy abstraction for higher-order program verification. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, PPDP '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364416. DOI: 10.1145/3236950.3236969.

[40] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *2009 Formal Methods in Computer-Aided Design*, pages 1–8, 2009. DOI: 10.1109/FMCAD.2009.5351148.

[41] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 221–234, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595936899. DOI: 10.1145/1328438.1328467.