MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Abstraction-Based Interprocedural Software Verification

Scientific Students' Association Report

Author:

Márk Somorjai

Advisors:

Mihály Dobos-Kovács
Levente Bajczi
Dr. András Vörös

2023

Contents

Kivonat

A technológiavezérelt világunkban egyre több feladat van automatizálva szoftver által. Szoftvert használunk üzenetek küldéséhez és fizetéshez, de manapság rábízzuk nukleáris erőművek működtetését és autók vezetését is. Az ilyen biztonságkritikus rendszerekben a szoftverbe vetett bizalom nem elegendő, mert egy fellépő hiba hatalmas gazdasági veszteséggel, környezeti károkkal vagy akár életvesztéssel is járhat. A biztonságkritikus rendszerek szoftverének helyességét tehát biztosítani kell. Bár a hagyományos tesztelési technikák tudnak példát mutatni a rendszer helytelen viselkedésére, a hibák hiányát nem tudják garantálni. A formális verifikáció ezzel szemben matematikai bizonyítást vagy cáfolatot tud adni a rendszer biztonsági tulajdonságairól.

A formális verifikációnak megvannak a saját kihívásai. Ahhoz, hogy egy program helyes legyen, az elérhető állapotai, azaz az a változók lehetséges értékei az elérhető vezérlési helyeken nem sérthetik meg a biztonsági kritériumot. A program állapoterének mérete azonban exponenciálisan nő a programváltozók számának függvényében, ami gyakorlatban lehetetlenné teszi az összes állapot ellenőrzését. Az állapotteret tovább növelik a szoftver minden területén megjelenő procedurák. Ezek struktúrálják és újrahasznosíthatóvá teszik a szoftvert, azonban megnehezítik a szoftver interprocedurális analízisét a végrehajtás folyamának megzavarásával és új változópéldányok generálásával a meghívási helyeiknél. Ezen felül a program állapotát a hívási veremmel egészítik ki, amely végtelen mély lehet rekurzív programok esetén, így egy végtelen nagy állapotteret eredményezve.

A hatalmas állapotterek problémáját tipikusan redukciós eljárásokkal kezelik formális szoftververifikációban. Az absztrakciós technikák az állapottér elemeit valamilyen információ elhagyásával csoportosítják absztrakt állapotokba, így egy redukált absztrakt állapotteret állítva elő. Hagyományosan az absztrakciót a változók értékeire alkalmazzák. A dolgozatomban az absztrakció kiterjesztését mutatom be hívási vermekre egy absztrakció-alapú modellellenőrző algoritmusban, az ellenpélda-alapú absztrakciófinomításban (CEGAR). A hívási verem részeinek elabsztrahálásával különböző hívási veremmel rendelkező, hasonló állapotok tovább csoportosíthatók, így csökkentve az absztrakt állapottér mértetét. Ez javítja a szoftverek interprocedurális verifikációjának hatékonyságát, különösképp rekurzív progamok esetén, ahol a hívási verem nagyobb mértékben járul hozzá az állapottér növekedéséhez. A bemutatott ötlet egy prototípusát a Theta modellellenőrző keretrendszerben implementálom, hatékonyságát pedig egy esettanulmányon értékelem ki.

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

Abstract

In our technology-driven world, an increasing amount of tasks are automated using software. We use it to deliver messages and execute payments, but nowadays we also trust it to operate nuclear power plants and drive cars. In such safety-critical systems, trust alone is not sufficient, because failure can lead to significant financial loss, environmental damage or even the loss of life. Thus, the correctness of software in safety-critical systems needs to be ensured. While conventional testing can provide examples of the incorrect behavior of a system, it does not guarantee the absence of errors. Formal verification, on the other hand, can give a mathematical proof of correctness or a refutation to the safety properties of a system.

Formal verification of software comes with its own set of challenges. For a program to be correct, its reachable states, that is, the possible values of variables at reachable locations must not violate the safety property in question. The size of a program's state-space, however, grows exponentially with the number of variables in the program, making checking every state practically infeasible. The state-space is further blown up by procedures, a widespread concept in all fields of software. Procedures provide structure and allow the reuse of existing software, but they make their interprocedural analysis more difficult by disrupting the sequential flow of execution and generating new variable instances at each of their calls. On top of that, they extend the state of the program with the call stack, which can stretch infinitely deep in recursive programs, and as a result, lead to an infinitely large state-space.

The problem of large state-spaces in formal software verification is typically handled using some kind of reduction technique. Abstraction-based methods reduce the state-space into an abstract state-space, grouping states together by removing some information from a state of the program. Traditionally, abstraction is only applied to the values of variables. In this work, I propose 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. This improves the efficiency of interprocedural software verification, especially in the case of recursive programs, where the call stack has a greater effect on the size of the state-space. I implement a prototype of the proposed idea in the model checking framework THETA, and evaluate its performance on a case study.

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

Chapter 1

Introduction

As our lives are becoming more and more intertwined with technology, the range of tasks taken over by software keeps broadening. Not only does it keep us connected or provide entertainment, these days software is also used for flying planes or operating nuclear power plants. While the failure of software may not matter much when playing a video game, it can lead to catastrophic events if it happens mid-flight on a passenger aircraft. Systems, in which the cost of failure is significant financial loss, environmental damage or the loss of life are called *safety-critical.* It goes without saying, that ensuring the correctness of software in such systems is imperative. Conventional testing methods are capable of proving the presence of errors by showing an example of the incorrect behavior. On the contrary, the success of tests does not guarantee the absence of errors, since it is infeasible for them to cover all behaviors of the software. Therefore, in safety-critical systems, testing alone is not sufficient and other techniques need to be employed. One such technique is *formal verification.*

The goal of formal verification is to either provide a mathematically precise proof for a software's correctness, or generate a refutation to it. In order to do so, it employs *model checking*, which explores the state-space of the program and checks whether an erroneous state can be reached or not. The greatest challenge of formal verification stems from the size of the state-space: to represent all possible states 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, leading to 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 [12], 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.

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.

In this work, I propose an approach to improve the efficiency of abstraction-based interprocedural verification of programs. 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. The idea is presented as a modification of the *Counterexample-Guided Abstraction Refinement (CEGAR)* algorithm. Not only does the proposition improve 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. A prototype of the proposed approach was implemented in the open-source model checking framework Theta [17] and was evaluated on benchmarks from SV-COMP [2], the international competition for software verification.

The report 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, my main contribution is presented: a way of applying abstraction to location stacks, as a means of improving the efficiency of interprocedural verification. In Chapter 5, a prototype implementation of the presented idea is evaluated. Finally, in Chapter 6, 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* [3].

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 [23].

- $E \subseteq L \times Ops \times L$: 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.

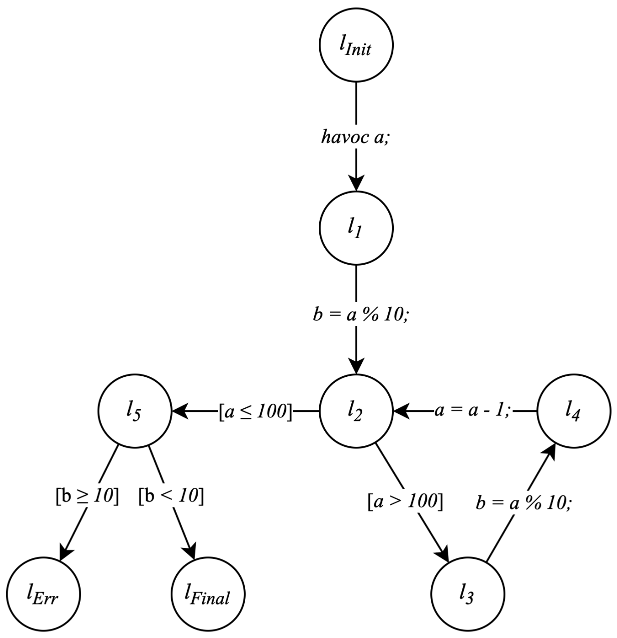


Figure 2.1: CFA of example C program.

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 $D = (\mathcal{S}, \sqsubseteq, concr)$, 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,
- $concr : \mathcal{S} \rightarrow 2^{\hat{S}}$ 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: $S_1 \sqsubseteq S_2 \Leftrightarrow concr(S_1) \subseteq concr(S_2)$, i.e., S_1 *overapproximates* S_2. ▪

The *precision* $\pi \in \Pi$ defines the level of abstraction. The *transfer function* $T : \mathcal{S} \times Ops \times \Pi \rightarrow 2^{\mathcal{S}}$ 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* [4] and *predicate abstraction* [1].

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* [5].

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 $((l_1, S_1), op_1, (l_2, S_2), op_2, \dots, op_{n-1}, (l_n, S_n))$ of abstract states an operations. An abstract path is *feasible*, if there is a concrete path $((l_1, s_1), op_1, (l_2, s_2), \dots, op_{n-1}, (l_n, s_n)$ 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 [13]) with the formula $S_1 \wedge op_1 \wedge S_2 \wedge \dots \wedge op_{n-1} \wedge S_n$. 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.

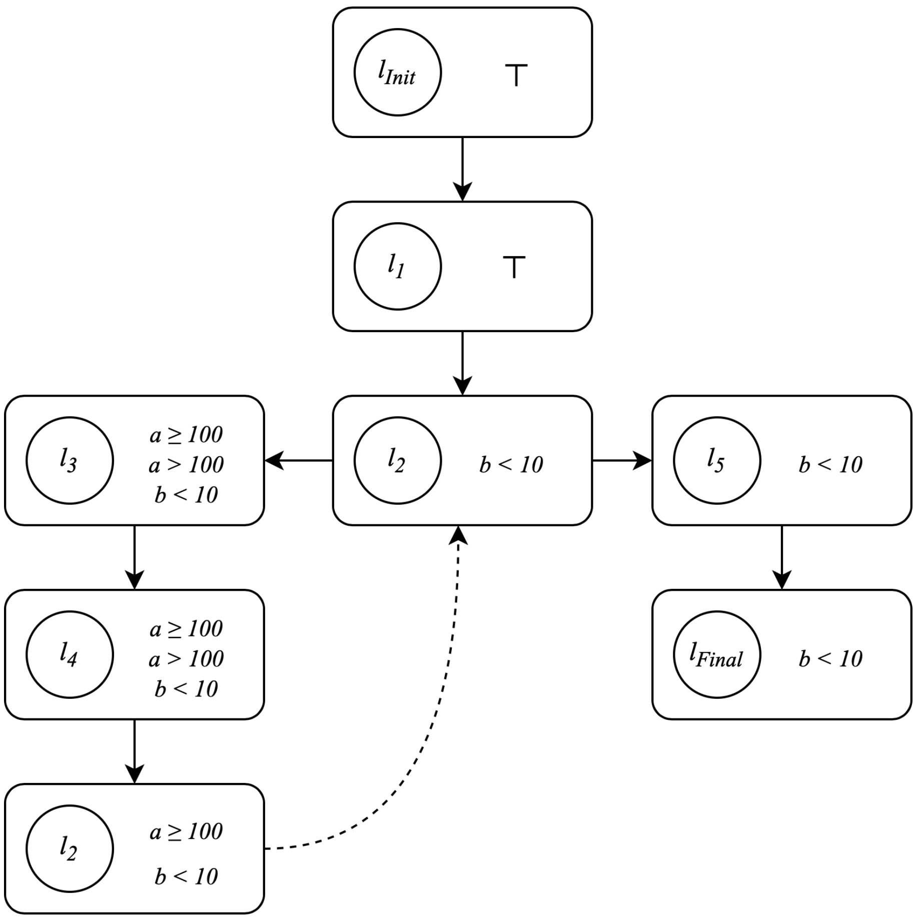

l_{Init} $\top$
l_1 $\top$
l_3 $a \geq 100$ $a > 100$ $b < 10$
l_2 $b < 10$
l_5 $b < 10$
l_4 $a \geq 100$ $a > 100$ $b < 10$
l_{Final} $b < 10$
l_2 $a \geq 100$ $b < 10$

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) [11] 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

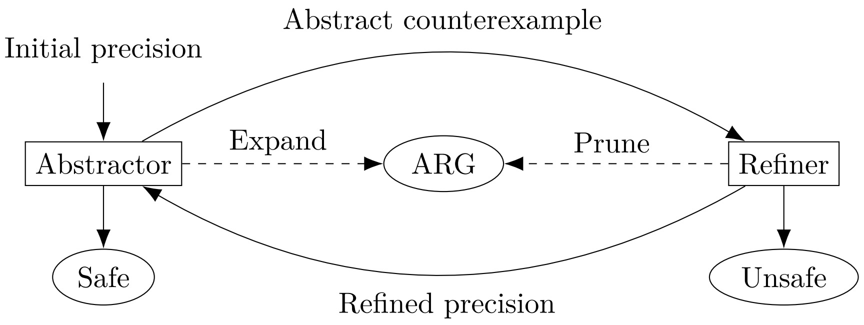


Figure 2.3: The CEGAR loop

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 [16] 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
    $\pi_0$: initial precision
    $T$: transfer function
Output: safe or unsafe
1  $ARG \leftarrow ((l_0, \top), \varnothing, \varnothing)$
2  $\pi \leftarrow \pi_0$
3  while $true$ do
4      $result, ARG \leftarrow$ ABSTRACTION$(ARG, D, \pi, T)$
5      if $result$ = safe then
6          return safe
7      else
8          $result, \pi, ARG \leftarrow$ REFINEMENT$(ARG, \pi)$
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
    $\pi$: current precision
    $T_L$: transfer function
Output: (safe or unsafe, $ARG$)
1  $waitlist \leftarrow \{S \in N \mid S$ is incomplete$\}$
2  while $waitlist \neq \varnothing$ do
3      $(l, S) \leftarrow$ pop $waitlist$
4      if $l = l_E$ then
5          return (unsafe, $ARG$)
6      else if $\exists (l', S') \in N : (l, S) \sqsubseteq_L (l', S')$ then
7          $C \leftarrow C \cup \{((l, S), (l', S'))\}$
8      else
9          foreach $(l', S') \in T_L((l, S), \pi) \setminus \{\bot\}$ do
10             $waitlist \leftarrow waitlist \cup \{(l', S')\}$
11             $N \leftarrow N \cup \{(l', S')\}$
12             $E \leftarrow E \cup \{((l, S), op, (l', S'))\}$
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
    $\pi$: current precision
Output: (spurious or unsafe, $\pi'$, $ARG$)
1  $\sigma = ((l_1, S_1), op_1, \ldots, op_{n-1}, (l_n, S_n)) \leftarrow$ abstract path to unsafe node in $ARG$
2  if $\sigma$ is feasible then
3      return (unsafe, $\pi$, $ARG$)
4  else
5      $(I_1, \ldots, I_n) \leftarrow$ interpolant for $\sigma$
6      $(\pi_1, \ldots, \pi_n) \leftarrow (I_1, \ldots, I_n)$ converted to precisions
7      $\pi' \leftarrow \pi \cup \bigcup_{1 \leq i \leq n} \pi_i$
8      $i \leftarrow$ lowest $i$ for which $I_i \notin \{true, false\}$
9  end
10 $N_i \leftarrow$ all nodes in the subtree rooted at $(l_i, S_i)$
11 $N \leftarrow N \setminus N_i$
12 $E \leftarrow \{(n_1, op, n_2) \in E \mid n_1, n_2 \notin N_i\}$
13 $C \leftarrow \{(n_1, n_2) \in C \mid n_1, n_2 \notin N_i\}$
14 return (spurious, $\pi'$, $ARG$)
```

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 $S_1 \wedge op_1 \wedge S_2 \wedge \dots \wedge op_{n-1} \wedge S_n$:

- 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 [25] or Newton refinement [14]. For example, the elements of a sequence interpolant $(I_1, \dots, I_n)$ 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 $(I_1, \dots, I_n)$ is converted to precisions $(\pi_1, \dots, \pi_n)$ 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 [24], 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.
- $A = \{a_1, a_2, \dots, a_{|I|}\}$: *Arguments* are a set of expressions that are assigned to the input parameters I of the procedure, that is, $\forall i \in [1, |I|], v_i \in I, a_i \in A : v_i = a_i \in D_{v_i}$.
- $R = \{r_1, r_2, \dots, r_{|O|}\}$: *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, as it is closely connected to the main topic of this work. In the following, solutions for the intricacies of data flow are introduced.

2.4.1 Variable Instances

A desired property of procedures is their template-like behaviour, 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.

2.4.2 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. $v \rightarrow v'$).

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 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 is discussed, then its extension with summaries is described. At last, contracts are introduced.

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 a 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* [9] 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* [7]. They are usually combined with some other technique, such as *summaries.*

3.2 Summaries

A procedure summary [22] 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 [21], or by defining a function from input abstract states to the output abstract states [26], for example. In the case of recursive procedures, transformers [27] or fixpoint algorithms [20] 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 [10] combines summaries with intraprocedural analyzers to verify recursive programs. Stratified inlining [18] is an extension of bounded inlining with summaries.

3.3 Contracts

A contract [19] 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 reduce the size of the state-space.

Contracts of procedures have to be specified by the developer. On one hand, this allows them to capture fundamental information about the procedure without the need for summary calculation, since the developer may be able to provide insight about the program using their expertise. On the other hand, contracts require the developer to have a strong background in formal verification. Some verification tools employing contract-based verification are KeY [8] and TriCera [15].

Chapter 4

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. First, I describe the adjustments that can be used to support procedures and procedure calls in CEGAR. Then, I present an extension of abstraction to stacks that improves the efficiency of verification. Finally, I show the modified algorithm on an example program.

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 8 (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 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 9 (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 10 (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, $top(q_1) = top(q_2)$.

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.

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 the following, a way of improving the efficiency of verification is introduced.

4.2 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 4.1, 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.

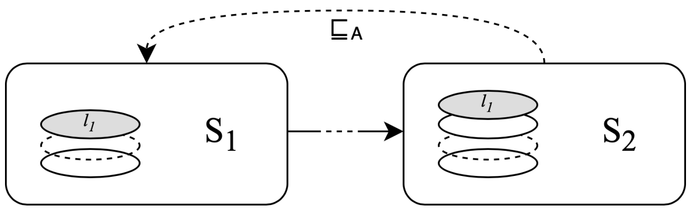


Figure 4.1: Part of an ARG with $\sqsubseteq_A$ between two abstract states.

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 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 4.2.1, while the result of applying it to the ARG in Figure 4.1 can be seen on Figure 4.2.

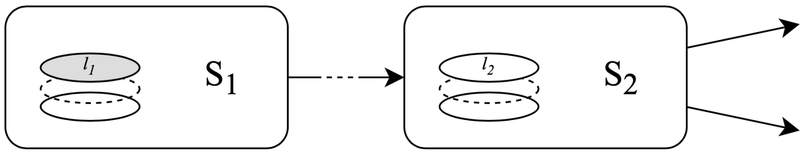


Figure 4.2: Part of the ARG in Figure 4.1 after popping.

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 4.2. 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 4.2.2.

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.

4.2.1 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 4.2 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 4.1, with the changes highlighted in blue.

Algorithm 4.1: Modified ABSTRACTION procedure

```
    Input:
        $ARG = (N, E, C)$: partially constructed abstract reachability graph
        $D = (\mathcal{S}, \sqsubseteq, concr)$: abstract domain
        $(\pi, \pi_Q)$: current precision
        $T_Q$: transfer function
    Output: (safe or unsafe, $ARG$)
 1  $waitlist \leftarrow \{S \in N \mid S$ is incomplete$\}$
 2  while $waitlist \neq \varnothing$ do
 3      $(q, S) \leftarrow$ pop $waitlist$
 4      if $top(q) = l_E$ then
 5          return (unsafe, $ARG$)
 6      else if $\exists (q', S') \in N : (q, S) \sqsubseteq_A (q', S')$ then
 7          if $(q, S) \sqsubseteq_Q (q', S')$ then
 8              $C \leftarrow C \cup \{((q, S), (q', S'))\}$
 9              continue
10          else if $q \notin \pi_Q$ then
11              pop $q$
12              $waitlist \leftarrow waitlist \cup \{(q, S)\}$
13              continue
14          end
15      end
16      foreach $(q', S') \in T_Q((q, S), \pi) \setminus \{\bot\}$ do
17          $waitlist \leftarrow waitlist \cup \{(q', S')\}$
18          $N \leftarrow N \cup \{(q', S')\}$
19          $E \leftarrow E \cup \{((q, S), op, (q', S'))\}$
20      end
21  end
22  return (safe, $ARG$)
```

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, 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 4.2.2) that an infeasible popping of an abstract state (q, S) with $q \in \pi_Q$ will not happen again.

4.2.2 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 4.2 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 4.2, with the changes highlighted in blue.

Algorithm 4.2: Modified REFINEMENT procedure

```
Input:
    $ARG = (N, E, C)$: partially constructed abstract reachability graph
    $(\pi, \pi_Q)$: current precision
Output: (spurious or unsafe, $(\pi, \pi_Q)$, $ARG$)
$\sigma = ((q_1, S_1), op_1, \ldots, op_{n-1}, (q_n, S_n)) \leftarrow$ abstract path to unsafe node in $ARG$
if $\sigma$ is feasible then
    if $\exists j \in [2, n] : |q_{j-1}| > |q_j|$, $top(q_j)$ was not final then
        $i \leftarrow$ lowest such $j$
        $\pi_Q \leftarrow \pi_Q \cup \{q_i\}$
    else
        return (unsafe, $(\pi, \pi_Q)$, $ARG$)
    end
else
    $(I_1, \ldots, I_n) \leftarrow$ interpolant for $\sigma$
    $(\pi_1, \ldots, \pi_n) \leftarrow (I_1, \ldots, I_n)$ 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 $(q_i, S_i)$
$N \leftarrow N \setminus N_i$
$E \leftarrow \{(n_1, op, n_2) \in E \mid n_1, n_2 \notin N_i\}$
$C \leftarrow \{(n_1, n_2) \in C \mid n_1, n_2 \notin N_i\}$
return (spurious, $(\pi, \pi_Q)$, $ARG$)
```

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 4.2.1), 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.

4.3 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 4.3. 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);
};
```

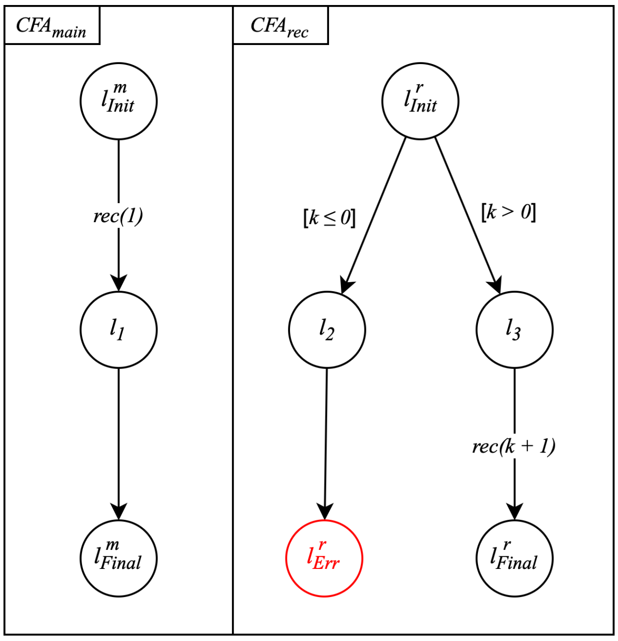


Figure 4.3: Example C program and its CFAs.

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 4.4. The erroneous abstract state is highlighted in red.

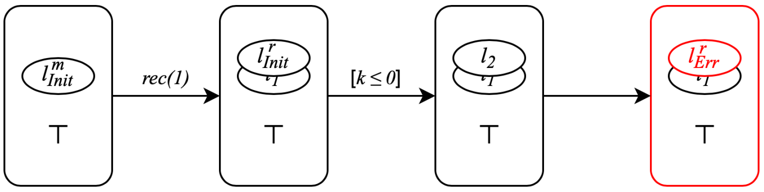


Figure 4.4: The abstract counterexample found in the first iteration.

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.

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

4.3.1 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 4.5.

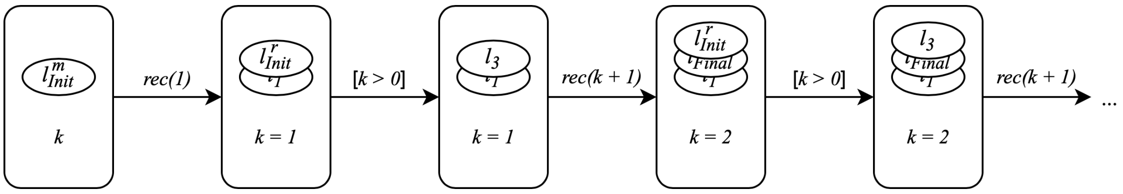


Figure 4.5: The ARG built using explicit domain.

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.

4.3.2 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 4.6. The visualization of the ARG nodes is to be interpreted as all predicates that are displayed in a node hold in the abstract state.

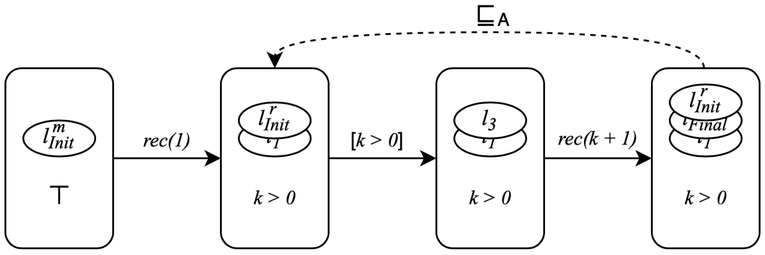


Figure 4.6: The ARG built using predicate abstraction, before popping.

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 (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 4.1, this is exactly when the stack q_4 needs to be popped. The modified ARG can be seen on Figure 4.7, with the popped node highlighted in blue.

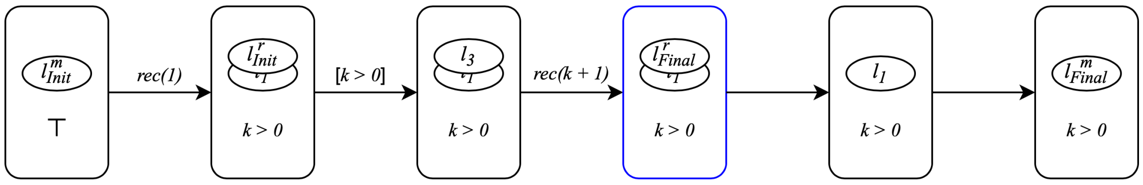


Figure 4.7: The final ARG built using predicate abstraction.

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.

4.3.3 Comparison to CEGAR

In this section, the unmodified version of CEGAR is ran on the example program in Figure 4.3, 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 4.4. 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 4.8.

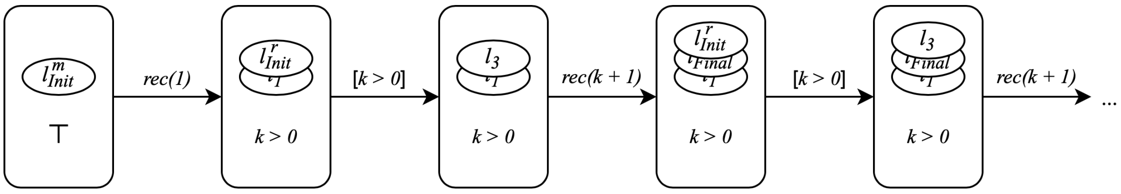


Figure 4.8: The ARG built using predicate abstraction, without popping.

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 (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.

Chapter 5

Evaluation

In this chapter, a prototype implementation of the modified CEGAR algorithm described in Chapter 4 is evaluated on a set of C programs. First, the benchmark environment is described, then the benchmark results are presented.

5.1 Benchmark Setup

A prototype of the presented CEGAR modifications was implemented in THETA [17], 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 as the prototype is finalized, a pull request will be made as well.

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
- 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 [14] and BFS search strategy

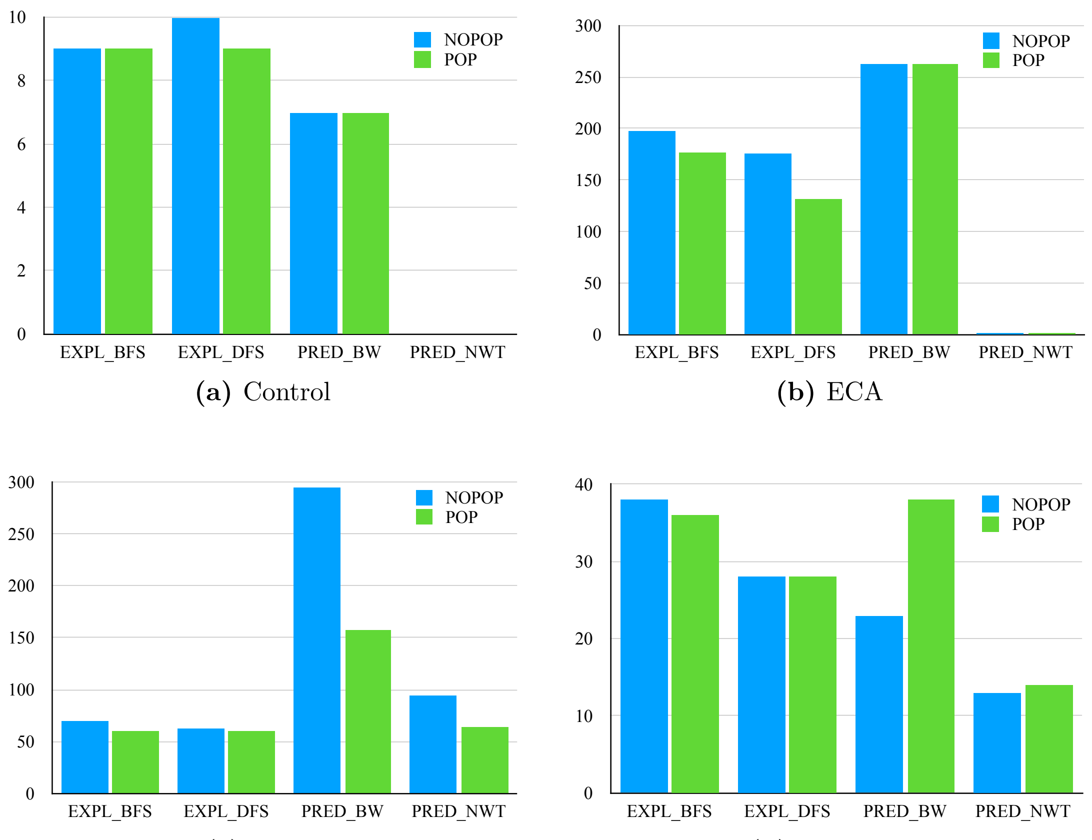
[1] https://github.com/s0mark/theta/tree/interproc
[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

All of the configurations above were ran 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 [6]. The tests were run on virutal 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.

5.2 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 5.1. 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.



(a) Control **(b)** ECA

(c) Loops **(d)** Recursive

Figure 5.1: Number of tasks solved by configuration.

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

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.

5.2.1 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 5.1c: 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 4.1, whereas in the original Algorithm 2.2 the computation of the partial order 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.

5.2.2 Recursive Programs

For *recursive* tasks, the two versions perform the same using explicit abstraction, as seen on Figure 5.1d. With predicate abstraction, however, the changes proposed in Section 4.2 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.

5.2.3 Threats to Validity

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

As mentioned in Section 5.1, 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 5.1c and Figure 5.1d.

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 [2], the international competition of software verification tools. Counting the solved tasks can be seen as a measure of practical performance.

Chapter 6

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 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 formal verification is the state-explosion problem, that is, the size of the state-space grows exponentially with the number of variables in the program. To counteract this, reduction techniques are applied to the state-space of the program, such as abstraction. Abstraction groups states together by abstracting away some information from the state of the program, e.g. the values of some variables. The abstraction-based model checking algorithm CEGAR was presented in Section 2.3.

Procedures are a widespread concept in all fields of software. They allow the reuse of existing software, but they also make interprocedural verification more challenging by creating new variable instances with each of their calls. On top of that, they extend the state of a program with the call stack, which can lead to an infinitely large state-space in the case of recursive programs, where the stack stretches infinitely deep.

In Chapter 4, I presented a novel approach to improve the efficiency of interprocedural verification. The main idea was to extend abstraction to the location stacks of states in order to further reduce the size of the abstract state-space. This was achieved by introducing 2 partial orders in Section 4.2 for covering: $\sqsubseteq_Q$ was a sound, but wasteful version and $\sqsubseteq_A$ was an underapproximating, but performant one. The two partial orders 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 presented in Section 4.2.1, while the refiners modifications were described in Section 4.2.2. The modified algorithm was presented in a case study in Section 4.3, 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.

A prototype of the presented approach was implemented in the open-source model checking framework Theta. The implementation was evaluated in Chapter 5 on 1219 C programs in different configurations of Theta. 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.

6.1 Future Work

In the short-term, the prototype implementation could be finalized and integrated into Theta as an optional configuration. This would allow a portfolio to only use the modified algorithm on tasks that it brings an improvement to, e.g. recursive programs. The goal is to submit the feature as part of the experimental version of Theta for next year's SV-COMP '24.

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.

Bibliography

[1] 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.

[2] Dirk Beyer. Competition on software verification and witness validation: Sv-comp 2023. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8. DOI: `10.1007/978-3-031-30820-8_29`.

[3] 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.

[4] 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.

[5] 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.

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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`.

[10] 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`.

[11] 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`.

[12] 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`.

[13] 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`.

[14] 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`.

[15] 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`.

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

[17] 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`.

[18] 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`.

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

[20] 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`.

[21] 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`.

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

[23] 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`.

[24] 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`.

[25] 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`.

[26] 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`.

[27] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. *SIGPLAN Not.*, 43(1):221–234, jan 2008. ISSN 0362-1340. DOI: `10.1145/1328897.1328467`.