

TDK-dolgozat

Somorjai Márk
2024.



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

Automatic Contract Synthesis for Deductive Verification

Scientific Students' Association Report

Author:

Márk Somorjai
Grade 6

Advisors:

Mihály Dobos-Kovács,
PhD student

Levente Bajczi,
PhD student

Dr. András Vörös,
associate professor

2024.

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background	3
2.1 Control Flow Automata	3
2.2 Abstraction	5
2.3 Counterexample-Guided Abstraction Refinement	8
2.3.1 Abstractor	10
2.3.2 Refiner	11
2.4 Procedures	12
2.4.1 Variable Instances	13
2.4.2 Parameter Assignments	14
2.5 Contracts	15
2.5.1 ACSL	16
3 Related Work	18
3.1 Weakest Preconditions	18
3.2 Strongest Postconditions	19
4 Contract Synthesis	20
4.1 Interprocedural Analysis	21
4.2 Transforming Abstract States to Contracts	25
4.2.1 Explicit Domain	27
4.2.2 Predicate Domain	28
4.3 Case Study	29
5 Evaluation	31
5.1 Benchmark Setup	31

5.2	Reachability Results	32
5.3	Overflow Results	33
5.4	Threats to Validity	34
6	Conclusion	35
6.1	Future Work	35
	Bibliography	37

Kivonat

A szoftver egyre jobban elterjed életünkben. Használjuk üzenetek küldésére és fizetésre, de szoftver vezeti az autókat és üzemelteti a nukleáris erőműveket is. Ilyen rendszerekben egy hiba jelentős gazdasági következményekhez, környezeti károkhoz vagy akár életvesztéshez is vezethet, ezért az ezen rendszerekben való szoftverek helyességének bizonyítása kritikus. Ebből adódóan a hibák hagyományos tesztelési technikákkal való keresése nem elégséges, hiszen a tesztelés csak a helytelen működésre tud példát adni, a hibák teljes hiányát nem tudja bizonyítani. A helyesség bizonyításához formális módszereket lehet használni, melyek matematikailag precíz bizonyítékkal szolgálnak a helyességről, vagy cáfolattal a biztonsági tulajdonságok megsérüléséről.

Többféle megközelítése létezik a formális verifikációnak, mindegyiknek megvannak a saját kihívásai. A deduktív metódusok célja, hogy bizonyítsák a szoftver megfelelését a specifikációjának azáltal, hogy a programot és specifikációját matematikai bizonyítási célokká transzformálják, melyeket aztán tételbizonyítóknak adnak tovább. A megközelítés fő hátránya, hogy a fejlesztőn múlik, hogy megértse a program helyességének okát, majd ezt az információt kontraktusokká alakítsa a szoftver különböző absztrakciós szintjein. Ez jelentős mérnöki munkával jár, ami limitálja az eljárás skálázhatóságát. Egy másik megközelítés, a modellellenőrzés bejárja a program egy matematikai modelljének állapotterét, hogy eldöntse a program bizonyos tulajdonságairól – mint a hibák elérhetősége –, hogy teljesülnek-e. A modellellenőrzés automatizálható, azonban gyakran kivitelezhetetlen a program állapotterének mérete miatt, mely exponenciálisan nő a program változóinak számával. Ezen probléma kezelésére tipikusan redukciós technikákat alkalmaznak az állapottérre, mint például az absztrakció.

A dolgozatomban a modellellenőrzés és a deduktív verifikáció egy olyan kombinációját mutatom be, amely mindkettőnek a jó tulajdonságait őrzi meg, ezáltal automatizálttá és hatékonyabbá téve a verifikációt. Elsőként az absztrakció-alapú modellellenőrzés bejárja a program redukált állapotterét. Ezután a kifejtett állapottérből függvénykontraktusok szintetizálhatók, melyek felhasználásával deduktív módszerekkel bizonyíthatók a program különböző tulajdonságai. Ezáltal nem szükséges a kontraktusok előállításához a fejlesztők manuális munkája, így a verifikáció skálázhatóbb. A munkám újdonsága, hogy a szintézis *környezetfüggő*, azaz a generált kontraktusok csak olyan feltételeket tartalmaznak amelyek ténylegesen előfordulhatnak a program függvényeiben, nem pedig a lehető legáltalánosabb formulát. A bemutatott ötlet egy prototípusát a FRAMA-C deduktív verifikációs platformra építve implementálom a THETA modellellenőrző keretrendszerben, a hatékonyságát pedig szintetikus és ipari példákon értékelem ki.

Abstract

Software is becoming more and more prevalent in our lives. We use it to send messages and execute payments, but nowadays software also drives cars and operates nuclear power plants. In such systems, failure can lead to significant financial loss, environmental damage or even the loss of life, therefore, ensuring the correctness of software in these system is critical. Consequentially, using conventional testing to find bugs is not sufficient, as testing can only provide examples of incorrect behavior and is not able to prove the complete absence of errors. For proving correctness, formal methods can be used, which can give a mathematically precise proof of correctness or a refutation to the safety properties of a system.

There are various approaches to formal verification, each with its set of own challenges. Deductive methods aim to prove that the a software conforms to its specification, by transforming the program and its specification to a collection of mathematical proof obligations, which are then passed on to theorem proving tools. The main disadvantage of the approach is that it depends on the developer of the system to understand why the software should work correctly, then convey that information through the use of contracts at varying levels of abstraction throughout the software. This requires significant engineering effort, limiting the scalability of deductive verification. Another approach, model checking, explores the state space of a mathematical model of software in order to decide whether or not certain properties – e.g. error reachability – of the program are satisfied. While model checking can be automatic, it can often be infeasible as the size of a program’s state space grows exponentially with the number of variables in the program. This issue is typically handled by applying reduction techniques to the state space, such as abstraction.

In this work, I propose to combine model checking and deductive verification in a way that brings out the best of the two approaches, making verification both automatic and more efficient. Abstraction-based model checking is used first to explore the reduced abstract state space of the program. The explored state space is then used to synthesize contracts for procedures in the software, which are then utilized by deductive methods to verify properties of the program. This eliminates the necessity of the developers having to manually provide contracts, allowing verification to be more scalable. The novel contribution of this work is that the synthesis is *context-aware*, that is, the generated contracts only consider conditions that can actually occur in each procedure of the program, instead of generating the most general formulas. I implement a prototype of the proposed approach using the model checking framework THETA and the deductive verification platform FRAMA-C, and evaluate its performance on synthetic and industrial examples.

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

Chapter 1

Introduction

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

There are different kinds of methods for formal verification, each with its own set of advantages and setbacks. *Model checking* [1] is an automated verification algorithm that explores the state-space of the program and checks whether or not an erroneous state can be reached. The greatest challenge of model checking is the size of the state-space: to represent all possible values of a 32-bit integer, 2^{32} states are needed. Furthermore, the number of states grows exponentially with the number of variables in the program, meaning that there are more possible states in a program with eight 32-bit integers than the number of atoms in the universe. This phenomenon is called the state-space explosion [15], 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.

Deductive verification [20] takes a different approach to formal verification. It expresses the correctness of a program as a set of mathematical statements, then uses axioms and proof rules in automated or interactive theorem provers to verify them. Deductive verification is a powerful method as it can be used to prove both safety and behavioral properties of programs, which are specified by contracts [25]. Contracts also modularize the verification process, since conformity to the specification can be checked on a per-procedure basis. Writing these contracts, however, requires significant engineering effort and specialized knowledge in the field of formal methods, since the developer has to understand why the program should work correctly and needs to be able to convey that information at various abstraction levels throughout the software. Consequently, the scalability of deductive verification is limited, which restricts its applicability.

In this work, I propose an automated approach for contract synthesis using abstraction-based model checking. The generated contracts then can be utilized by deductive verifiers in order to bring out the best of both approaches, making verification both automatic and more efficient. I present a variant of the abstraction-based *Counterexample-Guided Abstraction Refinement (CEGAR)* algorithm to explore the reduced abstract state-space of the program in an interprocedural manner. Next, I propose a synthesis method for generating contracts from the explored abstract state-space, the products of which can then be passed onto deductive verifiers to prove various safety properties of the program. Since both steps are automated, the manual labor of developers for writing contracts can be eliminated, making deductive verification more scalable. The novelty of this work is that the abstraction-based synthesis is *context-aware*, in the sense that the generated contracts only consider the conditions under which the procedure could have been called in the context of the program, as opposed to traditional weakest precondition synthesis techniques [18]. A prototype of the proposed approach is implemented in the open-source model checking framework THETA [22], making it capable of synthesizing contracts in the ANSI C Specification Language [3]. The implementation, combined with the open-source deductive verification platform FRAMA-C [24], is evaluated on benchmarks from SV-COMP [5], 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, related approaches to contract synthesis are described. In Chapter 4, my main contribution is presented: a context-aware abstraction-based contract synthesis technique, as a means of making verification more efficient by enabling deductive 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

Background¹

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

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

2.1 Control Flow Automata

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

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

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

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

¹Some parts of this chapter are taken from my previous work [28].

- $E \subseteq L \times Ops \times L$: 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 $s = (l, d_1, d_2, \dots, d_n)$, where:

- $l \in L$ is the current location of the program,
- $d_1, d_2, \dots, d_n \in D_{v_1} \times D_{v_2} \times \dots \times D_{v_n}$ is the *concrete data state*, where $d_1, d_2, \dots, d_n$ 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 $\hat{S} = D_{v_1} \times D_{v_2} \times \dots \times D_{v_n}$ 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 $(s_1, op_1, s_2, \dots, op_{n-1}, s_n)$, 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.

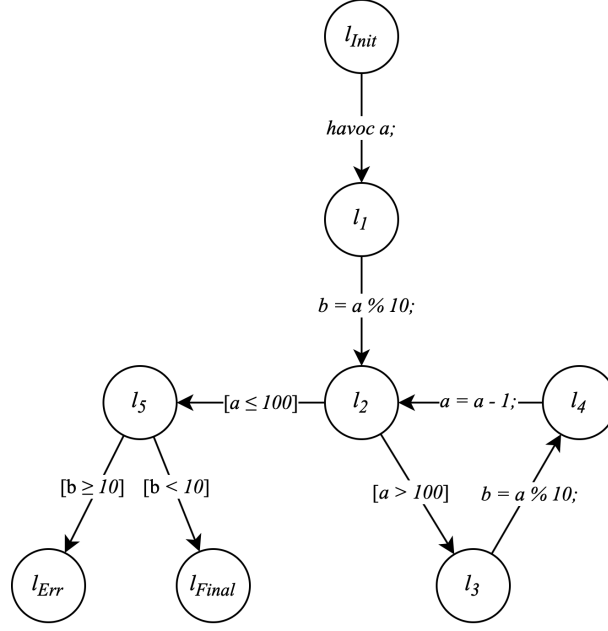


Figure 2.1: CFA of example C program.

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

The *precision* $\pi \in \Pi$ defines the level of abstraction. The *transfer function* $T : \mathcal{S} \times \text{Ops} \times \Pi \rightarrow 2^{\mathcal{S}}$ 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* [7] and *predicate abstraction* [2].

Explicit-value abstraction defines an abstract data state $S \in \mathcal{S}$ by variable assignments, mapping variables to $\top$, $\perp$ or a value from their domain. $\top$ is the top element of the lattice and denotes an unknown value, while $\perp$, 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 $S(v) \in D_v \cup \{\top, \perp\}$. 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 $S_1(v) = \perp$ or $S_2(v) = \top$. The function *concr* for an abstract data state S is $\text{concr}(S) = \hat{S}$ if $\forall v : S(v) = \top$, $\text{concr}(S) = \emptyset$ if $\exists v : S(v) = \perp$, otherwise it is $\text{concr}(S) = \{s \in \hat{S} \mid \forall v : v = S(v)\}$. 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 $\top = \text{true}$ and $\perp = \text{false}$, 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 $D = (\mathcal{S}, \sqsubseteq, \text{concr})$, 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 $\text{conr} : \mathcal{S}_L \rightarrow 2^{\hat{\mathcal{S}}_L}$ for an abstract state $(l, S) \in \mathcal{S}_L$ is defined as $\text{conr}((l, S)) = \{(l, s) \in \hat{\mathcal{S}}_L \mid s \in \text{concr}(S)\}$. ▪

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

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

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

- $N \subseteq \mathcal{S}_L$ is the set of *nodes*, each corresponding to an abstract state.
- $E \subseteq N \times Ops \times N$ is the set of directed *edges* between nodes, labeled with operations of the CFA. An edge $((l_1, S_1), \text{op}, (l_2, S_2)) \in E$ is present if $(l_1, \text{op}, l_2) \in E_{CFA}$ and (l_2, S_2) is a successor of (l_1, S_1) with op .
- $C \subseteq N \times N$ 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 and 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 $\forall i \in [1, n] : s_i \in \text{concr}(S_i)$. In practice, feasibility can be decided by converting the abstract states to logic formulae and querying an SMT solver (such as Z3 [16]) 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 there aren't any outgoing edges from the final location, the corresponding node of the ARG is *expanded* without any outgoing edges.

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

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

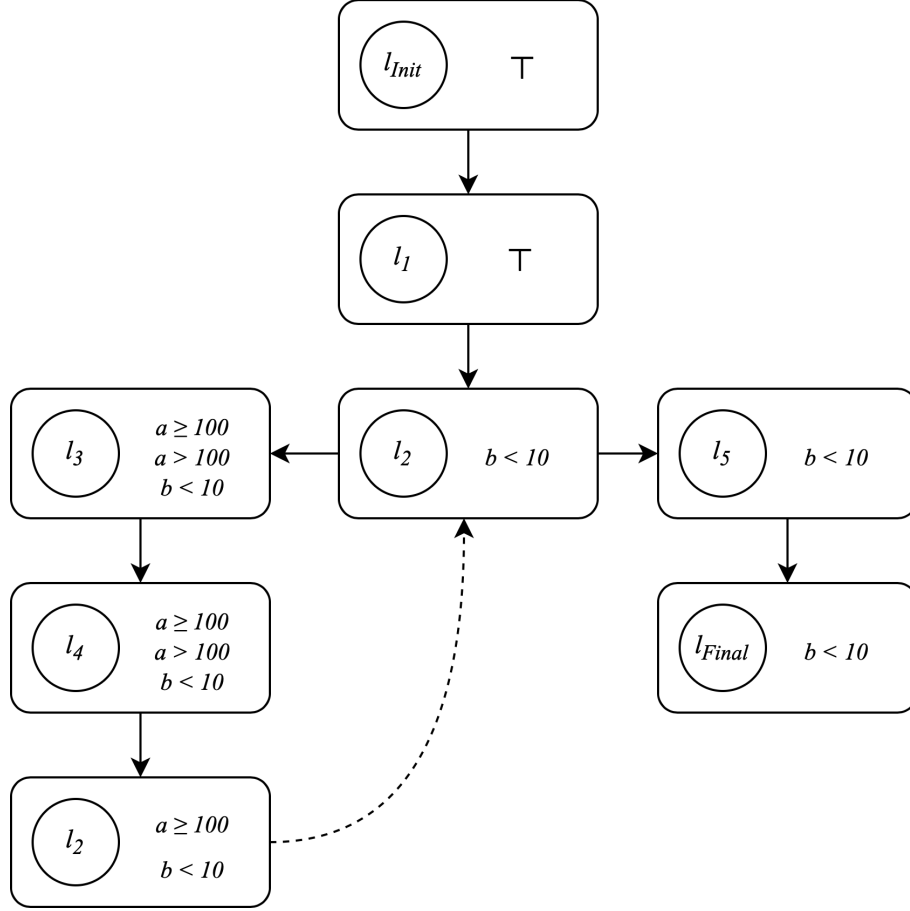


Figure 2.2: Fully expanded ARG of Example 1.

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

2.3 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [14] 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 in 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

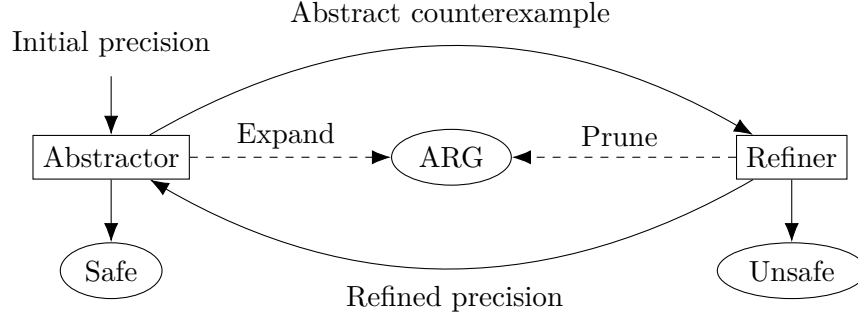


Figure 2.3: The CEGAR loop

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 [21] is presented in Algorithm 2.1.

Algorithm 2.1: CEGAR loop

Input:

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

Output: safe or unsafe

```

1   $ARG \leftarrow ((l_0, \top), \emptyset, \emptyset)$ 
2   $\pi \leftarrow \pi_0$ 
3  while true do
4       $result, ARG \leftarrow \text{ABSTRACTION}(ARG, D, \pi, T)$ 
5      if  $result = \text{safe}$  then
6          return safe
7      else
8           $result, \pi, ARG \leftarrow \text{REFINEMENT}(ARG, \pi)$ 
9          if  $result = \text{unsafe}$  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
 $D = (\mathcal{S}, \sqsubseteq, \text{concr})$: abstract domain
 π : current precision
 T_L : transfer function

Output: (safe or unsafe, ARG)

```

1 waitlist  $\leftarrow \{S \in N \mid S \text{ is incomplete}\}$ 
2 while waitlist  $\neq \emptyset$  do
3    $(l, S) \leftarrow \text{pop } \textit{waitlist}$ 
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  $(l', S') \in T_L((l, S), \pi) \setminus \{\perp\}$  do
10      waitlist  $\leftarrow \textit{waitlist} \cup \{(l', S')\}$ 
11       $N \leftarrow N \cup \{(l', S')\}$ 
12       $E \leftarrow E \cup \{(l, S), \text{op}, (l', S')\}$ 
13    end
14  end
15 end
16 return (safe,  $ARG$ )

```

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

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

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

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

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

2.3.2 Refiner

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

Algorithm 2.3: REFINEMENT procedure

Input:

$ARG = (N, E, C)$: partially constructed abstract reachability graph

π : current precision

Output: (spurious or unsafe, π' , ARG)

```

1  $\sigma = ((l_1, S_1), op_1, \dots, op_{n-1}, (l_n, S_n)) \leftarrow$  abstract path to unsafe node in  $ARG$ 
2 if  $\sigma$  is feasible then
3   | return (unsafe,  $\pi$ ,  $ARG$ )
4 else
5   |  $(I_1, \dots, I_n) \leftarrow$  interpolant for  $\sigma$ 
6   |  $(\pi_1, \dots, \pi_n) \leftarrow (I_1, \dots, I_n)$  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 $\sigma = ((l_1, S_1), op_1, (l_2, S_2), \dots, op_{n-1}, (l_n, S_n))$ 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 $((l_1, s_1), op_1, (l_2, s_2), \dots, op_{n-1}, (l_n, s_n))$ for which $\forall i \in [1, n] : s_i \in \text{concr}(S_i)$. 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 [30] or Newton refinement [17]. 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: $I_i = \text{true}$ if $i < k$, $I_i = I_k$ if $i = k$ and $I_i = \text{false}$ 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 [29], 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 for all 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.

2.5 Contracts

Procedures modularize software, providing interface of certain functionality to callers without exposure to implementation details. Even though the added level of abstraction is certainly useful, it also loses information on what a procedure performs. To preserve information on functionality, contracts [25] can be used, which describe the assumptions on the inputs of a procedure and provide information on the outputs of it.

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

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

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

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

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

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

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

Contracts are useful for software verification in many ways. For one, they can formally specify the expected behavior of a system, so that the implementation then can be checked for conformity. Additionally, they can make the verification of other safety properties more efficient by enabling intraprocedural verification, with the preconditions further reducing the state-space of each procedure. Contracts also make deductive verification [20] possible, a methodology that expresses the correctness of programs as a set of mathematical statements and uses automated or interactive theorem provers to verify them. On the downside, creating contracts requires significant engineering effort and specialized knowledge, limiting its applicability.

2.5.1 ACSL

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

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

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

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

Figure 2.4: The partial ACSL grammar for contracts.

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

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

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

Figure 2.5: The partial ACSL grammar for predicates.

Chapter 3

Related Work

Contracts are a powerful concept in software verification. They serve as a specification language for the expected behavior of procedure and modularize the verification process. Finding correct and sufficient contracts, however, requires significant engineering effort and specialized knowledge, limiting its scalability. Consequently, there has been significant research done on the automation of generating contracts. This chapter discusses approaches to automated contract synthesis that is related to this work.

3.1 Weakest Preconditions

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

Preconditions can be computed in many different ways. Traditional weakest precondition calculus [18] transforms postconditions to preconditions. They can also be computed using symbolic execution by exploring all paths that lead to violated assertions. As expected with symbolic execution, the latter approach would have issues with loops. More directly related to this work, there are algorithms for deriving preconditions using CEGAR [26], which handles loops better. The iterative algorithm can be used to refine an overapproximation of the weakest precondition, until only the sufficient preconditions remain.

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

3.2 Strongest Postconditions

Another related approach is strongest postcondition [19], sometimes also called as summary. Given a program and a precondition, the strongest postcondition captures all possible final states of an execution of the procedure starting with the precondition. The postcondition is the strongest, as it describes the possible postcondition to most detailed level. Therefore it is ideal for supporting the procedure-level modularity of deductive verification. On the other hand, the precision can come at a cost of overly verbose expressions. On top of that, it usually reflects more on the implementation of the program itself, rather the intention behind it.

Similarly to weakest preconditions, strongest postconditions can be calculated using symbolic execution [23], though it once again comes with loop unrolling issues. It can also generate huge formulas, as it is path based. Techniques have been developed to reduce the size of the formula [27] using passive single assignment and normalization.

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

Chapter 4

Contract Synthesis

Contracts are a powerful tool for software verification. They modularize the verification process and enable the use of deductive methods. Finding correct and sufficient contracts, however, requires significant engineering effort and specialized knowledge. To help alleviate that burden, contract generation would need to be automated. This chapter proposes a technique for synthesizing contracts from the output of the automated abstraction-based model checking algorithm CEGAR. An overview of the approach is presented in Figure 4.1, with the contributions of this work highlighted in bold. First, a brief description of the approach is given. Then, in Section 4.1, the extensions required for making CEGAR sound in an interprocedural context are addressed. Finally, Section 4.2 introduces the transformations required for synthesizing contracts from an explored abstract state-space.

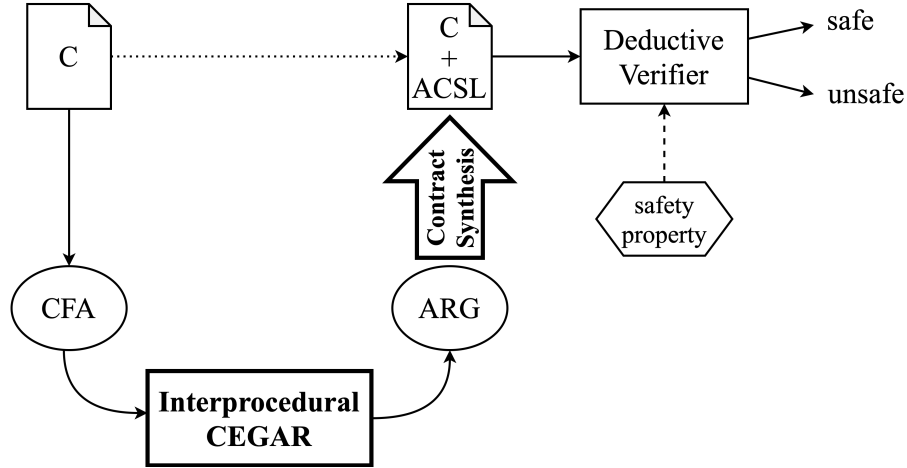


Figure 4.1: Overview of the presented approach.

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

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

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

The generated contracts convey information about the conditions a procedure can be reached under. To gather all possible abstract states, the CEGAR algorithm is run with no error detection – ignoring error locations –, so that exploration does not stop at error states and the whole ARG is expanded. Hence no refinement happens, only the abstractor runs with a certain safe verdict and a fully expanded ARG as its output. Without refinement, however, the precision has to be specified for the abstractor, as it would only be updated on refinement. The initial precision can also be generated in an automated way, using heuristics for extracting predicates and selecting variables from the program. These heuristics are described further in Section 4.2.1 for the explicit domain and Section 4.2.2 for the predicate domain. If the contracts generated with the chosen precision are too general for the deductive verifier to prove the desired safety property, one could run some iterations of CEGAR with said property in order to obtain more detailed contracts from a refined precision.

In order to attain information about the conditions each procedure is called from, the program is to be analyzed as a whole. While it is possible to inline procedures in some programs, to verify programs with (transitively) recursive procedures, a more delicate approach is required, as inlining does not terminate for these programs. The technical details of the introduction of procedures are described in Section 2.4, while Section 4.1 discusses the changes that are relevant to this work, namely the extension of abstract states with locations stacks. Then, Section 4.2 presents how contracts can be synthesized from a fully explored abstract state-space.

4.1 Interprocedural Analysis

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

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

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

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

- By default, the top location of the stack is replaced with the target of the transition by popping l_i from the stack and pushing l_j onto it.
- Additionally, if the operation op of the edge is a procedure call $C = (P, A, R)$, the initial location of the called procedure P 's body is *pushed* onto the stack.
- If the target location l_j of the edge is the final location of a procedure, it is popped from the stack and execution carries on from the location underneath. If there are no locations left in the stack, execution stops as the program terminates.

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

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

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

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

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

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

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

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

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

The function $concr : \mathcal{S}_Q \rightarrow 2^{\hat{S}_Q}$ 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

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

in a procedural program with

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

for an abstract state

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

is defined as

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

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

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

Example 3. Consider the following C program and its CFAs on Figure 4.2. The procedure `reach_error()` represents the program entering an unsafe state.

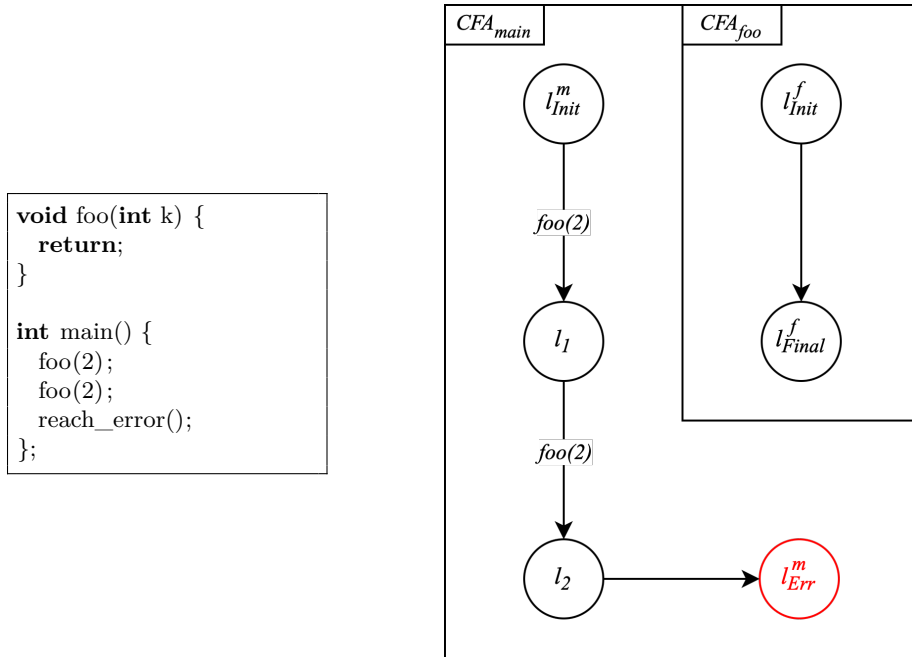


Figure 4.2: Example C program and its CFAs.

Examining the code, it can be seen that the the program will reach the erroneous state in its main function after two calls to `foo`, as the procedure does nothing. Conversely, if the partial order $\sqsubseteq_Q$ would be defined in the aforementioned manner, the CEGAR algorithm would deem the program safe. The ARG built by the abstractor using the explicit domain can be seen on Figure 4.3.

The first call to `foo` is explored without a problem. During the second call, however, the abstract state in l_{Init}^f is in the same location as the one after the first call. Since both abstract states' data state is $k = 2$, the first one overapproximates the second, leading to a

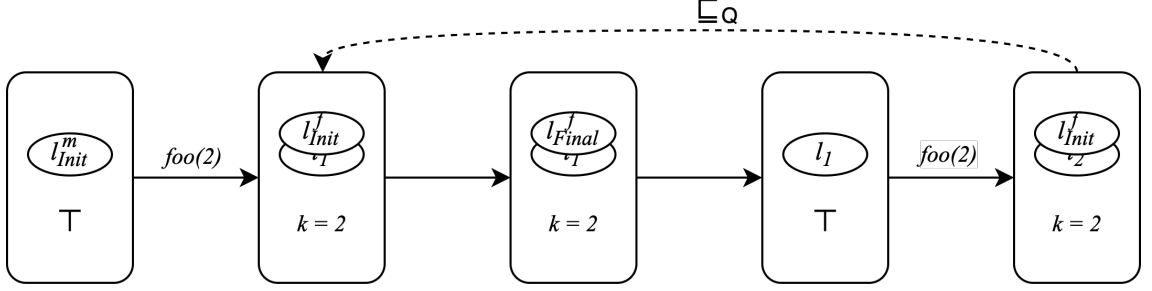


Figure 4.3: The ARG built with the wrong definition of $\sqsubseteq_Q$.

covered-by relation between them. The branch being the only one of the ARG, exploration stops at this point and the program is deemed (incorrectly) safe, as the fully built ARG has no nodes in erroneous states.

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 $q = q_1 = q_2$: $\text{concr}((q, S_1)) = \{(q, s_1) \in \hat{S}_Q \mid s_1 \in \text{concr}(S_1)\} \subseteq \{(q, s_2) \in \hat{S}_Q \mid s_2 \in \text{concr}(S_2)\} = \text{concr}((q, S_2))$ holds by definition of $S_1 \sqsubseteq S_2$. Thus, this definition of $\sqsubseteq_Q$ can be used for covering, keeping the analysis sound. Using this definition, the previous example program is verified correctly, as seen in Example 4 below.

Example 4. Using the second, correct definition of the partial order $\sqsubseteq_Q$, the CEGAR algorithm would come to a different conclusion when verifying the program in Figure 4.2. The ARG built by the abstractor using the explicit domain can be seen on Figure 4.4.

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

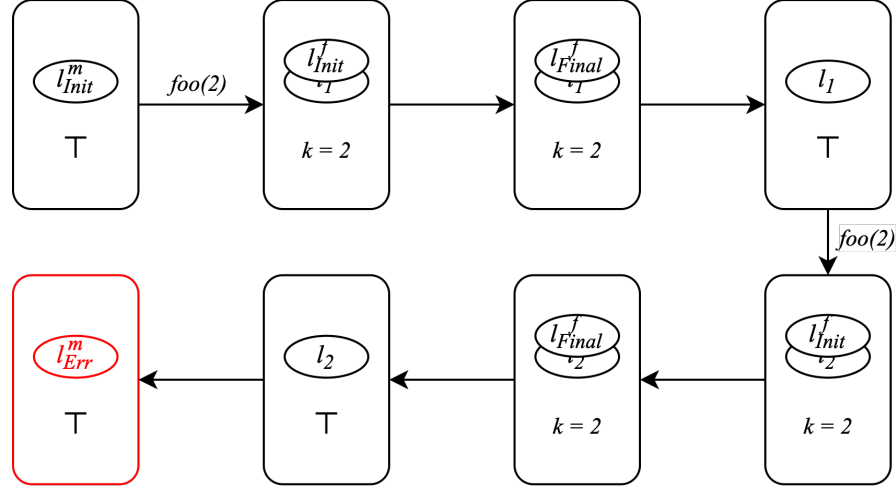


Figure 4.4: The ARG built with the correct definition of $\sqsubseteq_Q$.

4.2 Transforming Abstract States to Contracts

The goal of this transformation is to synthesize contracts from the abstract state-space of the program, so that deductive verification can be employed for verifying safety properties, without the necessity of specialized engineers writing the contracts manually. The novelty of the CEGAR-based approach, enabled by the interprocedural analysis extensions presented in Section 4.1, is that it is *context-aware*, i.e., the synthesized contracts describe the context a procedure can be called from, as opposed to the requirements for the procedures correct behavior. This allows the generated contracts to be used by deductive verifiers for proving various safety properties, such as error reachability or integer overflow.

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

The algorithm for synthesizing contracts is presented in Algorithm 4.1. It generates the abstract state-space represented by an ARG and transforms it into procedure contracts. The transformation is done from a CFA to contracts written in the ANSI C Specification Language (ACSL), described in Section 2.5.1. The parsing of code to CFA as well as the enrichment of the code with the ACSL contracts is omitted from the presentation for the sake of simplicity.

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

Since the input to the model checking algorithm does not contain error locations, the CEGAR loop will only execute the abstractor once, the return value of which is guaranteed to be safe along with a fully expanded ARG. Consequently, the refiner is not executed, meaning the precision does not change. Thus, the ARG only contains information that was specified to be tracked in the initial precision, hence it cannot be left empty as it is

Algorithm 4.1: CONTRACT SYNTHESIS

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

```

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

done when using CEGAR for verification. Heuristics for initial precision extraction from the input are described in Section 4.2.1 and Section 4.2.2 for the explicit and predicate domain, respectively.

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

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

Lines 8-17 convert the collected set of abstract data states R into contracts. If no such abstract state was found, then the procedure cannot be called in the program, therefore, no contract is generated for it. On the other hand, if there are such states, then the nested loop on lines 12-15 converts each $S \in R$ abstract state into a separate behavior with its own precondition using the $\triangleright$ translation operator. These behaviors are nested in

an ACSL-style comment and are appended by **complete behaviors** in line 16, conveying the fact that the abstract states of R are the only possible conditions under which the procedure can be called.

The translation operator $\triangleright$ converts an abstract data state into an ACSL contract predicate. It is defined partially with regards to both context and typing, in the style of sequent calculus [12]. For literals and variables in an abstract data state S , it is defined as:

$$\begin{array}{c} \overline{S \vdash \top \triangleright \backslash \mathbf{true}} \\ \overline{S \vdash \perp \triangleright \backslash \mathbf{false}} \\ \overline{S \vdash n : \text{integer} \triangleright n} \\ \overline{S \vdash v : \text{var} \triangleright \text{name}(v)} \end{array}$$

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

The remaining definition of $\triangleright$ is domain-dependent, therefore, the extensions to the definition above are described in Section 4.2.1 for the explicit domain and in Section 4.2.2 for the predicate domain.

4.2.1 Explicit Domain

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

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

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

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

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

4.2.2 Predicate Domain

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

$$\begin{array}{c}
\frac{S \vdash \alpha \triangleright \alpha'}{S \vdash +\alpha \triangleright +\alpha'} \qquad \frac{S \vdash \alpha \triangleright \alpha'}{S \vdash -\alpha \triangleright -\alpha'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha + \beta \triangleright \alpha' + \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha - \beta \triangleright \alpha' - \beta'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \cdot \beta \triangleright \alpha' * \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \div \beta \triangleright \alpha' / \beta'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha = \beta \triangleright \alpha' == \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \neq \beta \triangleright \alpha' != \beta'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha > \beta \triangleright \alpha' > \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \geq \beta \triangleright \alpha' \geq \beta'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha < \beta \triangleright \alpha' < \beta'} \qquad \frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \leq \beta \triangleright \alpha' \leq \beta'} \\
\\
\frac{S \vdash \alpha \triangleright \alpha' \quad S \vdash \beta \triangleright \beta'}{S \vdash \alpha \bmod \beta \triangleright \alpha' \% \beta'} \qquad \frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \oplus \psi \triangleright \phi' \wedge \psi'} \\
\\
\frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \wedge \psi \triangleright \phi' \&\& \psi'} \qquad \frac{S \vdash \phi \triangleright \phi' \quad S \vdash \psi \triangleright \psi'}{S \vdash \phi \vee \psi \triangleright \phi' || \psi'} \\
\\
\frac{S \vdash \phi \triangleright \phi' \quad S \vdash \omega_1 \triangleright \omega'_1 \quad S \vdash \omega_2 \triangleright \omega'_2}{S \vdash \text{ite}(\phi, \omega_1, \omega_2) \triangleright \phi' ? \omega'_1 : \omega'_2} \qquad \frac{S \vdash \phi \triangleright \phi'}{S \vdash \neg \phi \triangleright !\phi'}
\end{array}$$

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

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

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

4.3 Case Study

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

Consider to program and its CFAs in Figure 4.5. It contains a naive implementation of the absolute value function for integers, which is used on an unknown positive value in the main function. The havoc edge on the variable n is omitted for simplicity, its value is considered to be unknown regardless.

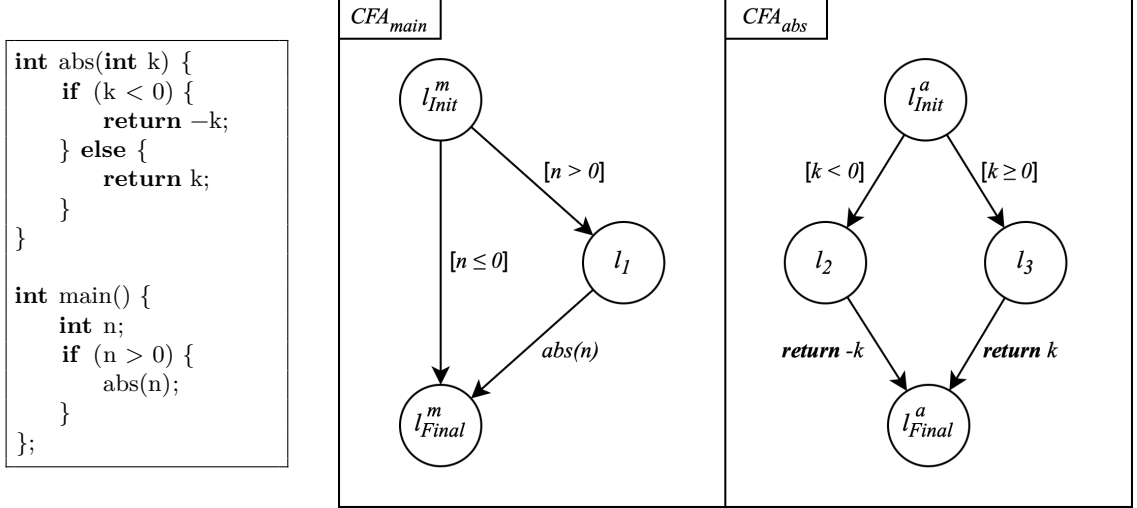


Figure 4.5: Example C program and its CFAs.

However, the implementation of `abs` is incorrect. If its input parameter k is equal to `INT_MIN` (i.e. -2147483648 for 32-bit integers), the negation in the first branch of the `if` will result in integer overflow, leading to undefined behavior. A deductive verifier that is capable of detecting overflow would find this potential error as it goes through each procedure, but it would not be able to provide a verdict on the safety of the program, as it does not know whether or not there is a concrete execution of the program that will lead to an overflow.

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

First, the interprocedural CEGAR algorithm is used to explore the abstract state-space of the program. The initial precision can be defined using either the `ALLASSUMES` or `PARAMASSUMES` heuristic, since both result in the same $\pi = \{n > 0, k < 0\}$ precision due to the fact that all guards in the program include parameters of a procedure or arguments of a procedure call. The explored abstract state-space is represented as an ARG in Figure 4.6. A predicate that is known to be false is shown after a semantic conversion, e.g. as $k \geq 0$ instead of $\neg(k < 0)$.

The guard of the branching expression in the main procedure creates two paths in the abstract state-space: one going to l_1 where $n > 0$ is satisfied, and one going to the final location l_{Final}^m where the same predicate is known to be false. The former continues with

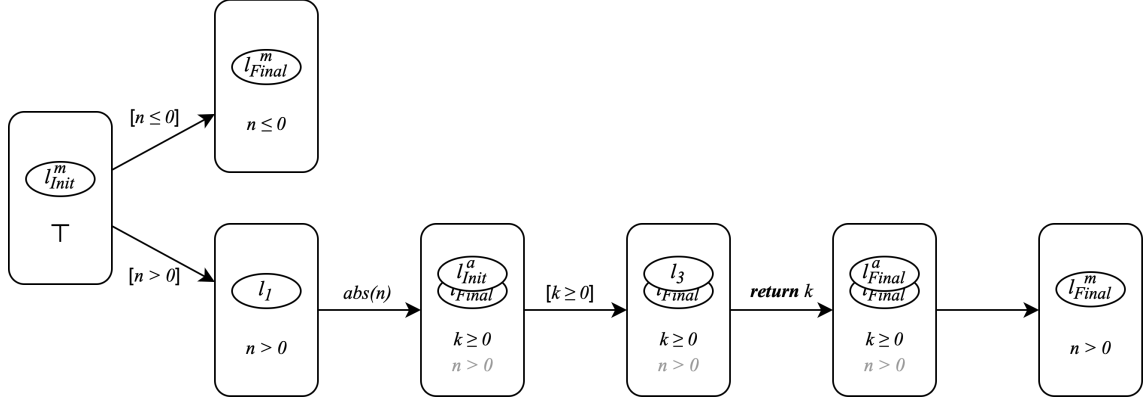


Figure 4.6: The fully expanded ARG of the example program.

a call to `abs`, where the other predicate in the precision $k < 0$ is guaranteed to be false, as the argument n of the procedure is known to be greater than 0. The abstract state in the initial location l_{Init}^a of the called procedure is therefore extended with the $k \geq 0$ predicate. Consequently, only the else branch of the procedure needs to be explored, which returns the value of k as is. With the final location l_{Final}^a of the procedure reached, exploration comes to an end at the final location l_{Final}^m of the program. Note, how the CEGAR algorithm does not need to know what property is to be verified as it only does an exploration of the abstract state-space.

With the ARG built up, the abstract states in the initial location of the `abs` procedure can be collected. There is only one such node in the ARG, where the abstract data state contains the predicates $k \geq 0$ and $n > 0$. Filtering for predicates with variables present in the procedure, the $k \geq 0$ predicate can be translated to $k \geq 0 \triangleright k \geq 0$, which is then wrapped in a behavior. The ACSL contract synthesized from the ARG can be seen in Figure 4.7.

```

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

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

```

Figure 4.7: Example C program with the synthesized ACSL contract.

Adding the synthesized contract with interprocedural cues to the program helps the deductive verifier prove safety. Going through each procedure, it would find that the context-aware preconditions described by the contract of `abs` rules out the branch with the possible integer overflow. Thus, the verifier would conclude that the program is indeed safe from integer overflows.

Chapter 5

Evaluation

In this chapter, a prototype implementation of the contract synthesis method described in Chapter 4 is evaluated along with the deductive verifier FRAMA-C [24] 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 and the contract synthesis was implemented in THETA [22], an open-source formal model checking framework. THETA already had a highly configurable CEGAR engine with different abstraction domains, interpolation techniques and search strategies, among other options. The changes were implemented into the xcfa subproject of Theta in Java and Kotlin. A new `arg2acsl` package was created with for the contract synthesis. The implementation is available on a fork of THETA¹ and as the prototype is finalized, a pull request will be made as well.

THETA was chained together with the SV-COMP version of FRAMA-C [8] to benchmark the potential benefits of this combined approach. The contracts synthesized and placed in the program by the prototype implementation, which was the fed to FRAMA-C for verification. FRAMA-C was used to verify error reachability by employing deductive verification using the Coq theorem prover [4], as well as to check integer overflow enabled by its non-deductive Evolved-Value Analysis (EVA) plugin [13]. In all cases, the configurations specified in the SV-COMP submission of the tool were used.

The toolchain was evaluated on 1213 error-reachability tasks and 1625 overflow tasks from the SV-COMP benchmark repository².

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

¹<https://github.com/s0mark/theta/tree/contract-synthesis>

²<https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks>

The contract synthesis of THETA was configured both with predicate and explicit abstraction. Each of them were tested with their corresponding initial precisions presented in Section 4.2.1 and Section 4.2.2.

The execution of benchmarks was done using the BenchExec framework [10]. The tests were run on virtual machines equipped with 3 Intel Xeon (Skylake) CPU cores and 15 GBs of memory, in the university cloud infrastructure³. 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 Reachability Results

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

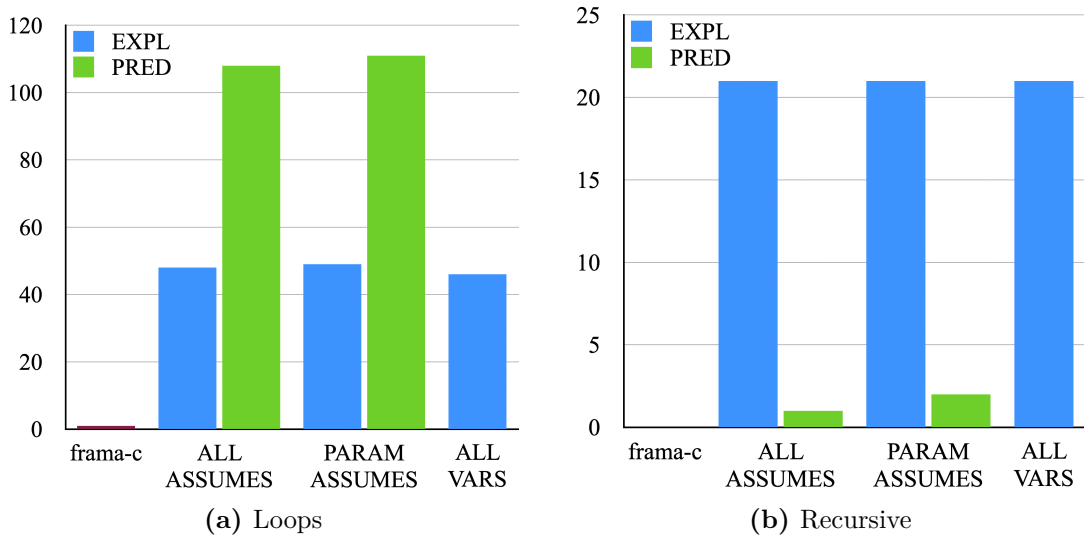


Figure 5.1: Number of reachability tasks solved by configuration.

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

Table 5.1: Number of solved reachability tasks by configuration.

In categories *control* and *eca*, FRAMA-C was not able to verify a single task within the specified constraints, regardless of the added contract. For *eca* tasks, FRAMA-C always

³<https://cloud.bme.hu>

ran out of memory, probably due to the fact that these tasks are relatively large. For the *control* tasks, it either timed out or returned *unknown*, that is, it is neither able to prove or refute the property in question. Thus, only the results for *loops* and *recursive* are presented in Figure 5.1 and Table 5.1.

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

5.3 Overflow Results

For overflow tasks, FRAMA-C gives both correct and wrong results, with and without the synthesized contracts, thus, its evaluation is limited. The number of tasks solved correctly and incorrectly by each configuration can be seen in Table 5.2. The number of *new* tasks is also presented in the table and in Figure 5.2, that is, tasks which FRAMA-C could only verify – correctly or incorrectly – with the synthesized contracts. The green and red bars correspond to the number of correct and wrong verdicts, respectively.

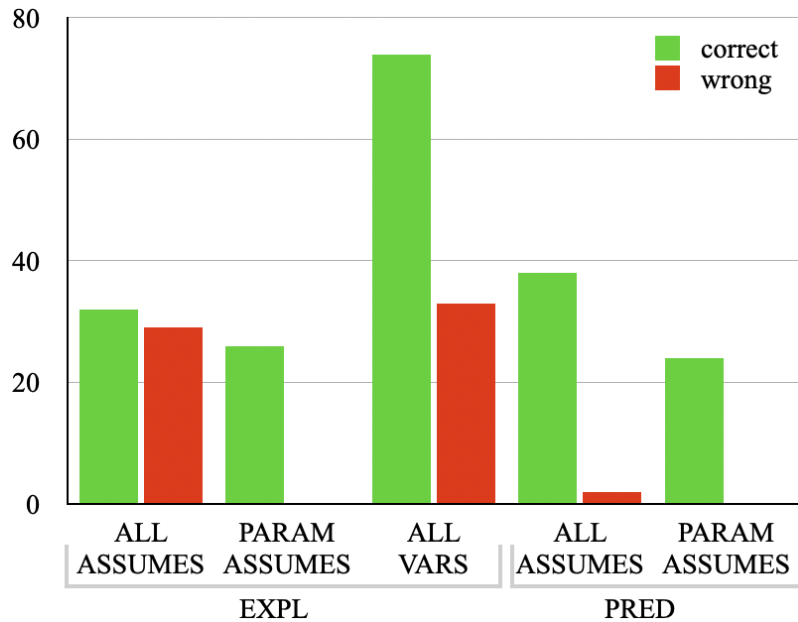


Figure 5.2: Newly solved overflow tasks by configuration and correctness

The wrong solutions are discussed in Section 5.4. This section focuses on tasks that were influenced by this work, i.e., tasks FRAMA-C could only verify with contracts.

Once again, the combined approach was not more efficient than running a specialized tool alone, with the best performing configuration solving about 50% as many tasks as FRAMA-C on its own. However, the synthesized contracts did allow FRAMA-C to verify an additional 101 tasks, a 15% increase over what the tool could solve on its own. As opposed to the reachability benchmarks, the initial precision does make a significant difference in both the number of correctly and incorrectly solved tasks. Contracts synthesized using

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

Table 5.2: Number of solved overflow tasks by configuration and correctness.

the PARAMASSUMES heuristic added no new wrong solutions, but they also increased the number of correct tasks by the least out of all configurations. The contracts generated in the explicit domain added more wrong solutions than ones coming from the predicate domain, which could be related to the fact that FRAMA-C uses its EVA plugin for proving overflow, as opposed to deductive methods.

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

5.4 Threats to Validity

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

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

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

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

Chapter 6

Conclusion

Software systems are responsible for critical services, where functional correctness is important to be ensured. Design-by-contract programming is gaining and increasing importance in the development of critical systems as it combines formal specification, verification with software design and supports the continuous, modular verification of critical software. However, designing contracts that are useful for the verification is a complex and tedious work, where engineers are needed to have a strong formal background. The goal of this work was to provide an efficient contract synthesis algorithm - relying on existing software verification methods - to automatize the manual work done by experts and support efficient, deductive, contract-based software verification.

I introduced a new algorithm, based on the Counterexample-Guided Abstraction Refinement approach, to explore the possible state spaces of the procedures, and extract useful contracts to represent the logical behavior of the procedures. The algorithm provides interprocedural analysis, so it can handle complex sequences of procedure calls, and exploits various abstraction and refinement techniques to focus the exploration and represent only the necessary information. A novel contribution in my work is that the contract generation is *context-aware* in the sense that the generated contracts only consider conditions that can actually occur in each procedure of the program. I devised heuristics to find the good abstraction based on the structure of the procedures.

I implemented a prototype of the proposed approach using the model checking framework THETA and the deductive verification platform FRAMA-C, and evaluated its performance on synthetic and industrial examples.

The presented work advances design-by-contract programming with efficient verification support to improve the quality of future critical software.

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 the THETA to be combined with other verification tools that support ACSL contracts.

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

On the longer term, verification witnesses [11] could be utilized instead of the ARG as the source of contract inference. This would further widen the combination possibilities of THETA with other tools. Moreover, it would allow the chaining of any tool generating witnesses to ones that utilize ACSL contracts.

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

Bibliography

- [1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 026202649X.
- [2] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45319-2.
- [3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acs1: Ansi/iso c specification language. In *ACSL: ANSI/ISO C Specification Language*, 2008. URL <https://api.semanticscholar.org/CorpusID:122818859>.
- [4] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq’Art: The Calculus of inductive constructions*. Springer Berlin, Heidelberg, 01 2004. ISBN 3540208542. DOI: 10.1007/978-3-662-07964-5.
- [5] Dirk Beyer. State of the art in software verification and witness validation: Sv-comp 2024. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 299–329, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57256-2. DOI: 10.1007/978-3-031-57256-2_15.
- [6] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.
- [7] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37057-1.
- [8] Dirk Beyer and Martin Spiessl. The static analyzer frama-c in sv-comp (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 429–434, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99527-0. DOI: 10.1007/978-3-030-99527-0_26.
- [9] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, October 2007.
- [10] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21

- (1):1–29, Feb 2019. ISSN 1433-2787. DOI: 10.1007/s10009-017-0469-y. URL <https://doi.org/10.1007/s10009-017-0469-y>.
- [11] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4), September 2022. ISSN 1049-331X. DOI: 10.1145/3477579.
 - [12] Samuel R. Buss. Chapter i - an introduction to proof theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 1–78. Elsevier, 1998. DOI: [https://doi.org/10.1016/S0049-237X\(98\)80016-5](https://doi.org/10.1016/S0049-237X(98)80016-5).
 - [13] David Bühler. *EVA, an Evolved Value Analysis for Frama-C : structuring an abstract interpreter through value and state abstractions*. PhD thesis, University of Rennes, 2017. URL <http://www.theses.fr/2017REN1S016>. Thèse de doctorat dirigée par Blazy, Sandrine et Yakobowski, Boris Informatique Rennes 1 2017.
 - [14] 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.
 - [15] 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.
 - [16] 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.
 - [17] 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.
 - [18] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. DOI: 10.1145/360933.360975.
 - [19] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J., 1976. ISBN 013215871X; 9780132158718.
 - [20] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, Oct 2011. ISSN 1433-2787. DOI: 10.1007/s10009-011-0211-0.
 - [21] Ákos Hajdú. *Effective Domain-Specific Formal Verification Techniques*. Phd thesis, Budapest University of Technology and Economics, 2020. URL <http://hdl.handle.net/10890/13523>.
 - [22] 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>.

- [23] Cliff Jones, A. Roscoe, and Kenneth Wood. *Reflections on the Work of C.A.R. Hoare*. Springer London, 01 2010. ISBN 1848829116, 9781848829114. DOI: 10.1007/978-1-84882-912-1.
- [24] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. ISSN 1433-299X. DOI: 10.1007/s00165-014-0326-7.
- [25] B. Meyer. Applying ‘design by contract’. *Computer*, 25(10):40–51, 1992. DOI: 10.1109/2.161279.
- [26] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 451–471, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.
- [27] John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David R. Cok. Inferring concise specifications of apis. *ArXiv*, abs/1905.06847, 2019. URL <https://api.semanticscholar.org/CorpusID:155100095>.
- [28] 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>.
- [29] 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.
- [30] 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.