Ludwig Maximilian University of Munich
Department of Informatics

CEGECoRe: Finding Precise Error Conditions Using CPAchecker

Felix Hajuj

Thesis submitted
within the Bachelor's Program
Computer Science & Computer-linguistic

Supervisor: Prof. Dr. Dirk Beyer
Mentor: Msc. Matthias Kettl

April 14, 2025

Acknowledgments

I would like to express my sincere gratitude to my mentor Matthias Kettl for his guidance, outstanding support and valuable feedback throughout week in and week out the course of this thesis. Special thanks goes to Prof. Dr. Dirk Beyer, firstly for the opportunity to be part of the SoSy chair and secondly for the chance to contribute to the chair by writing this thesis.

CEGECoRe: Finding Precise Error Conditions Using CPAchecker

ABSTRACT

Since we rely on software for almost every aspect of life, it is vital to be able to trust the software, especially in safety-sensitive systems where software failures can lead to critical consequences. Finding a precise description of error paths in software systems can prove to be quite helpful for developers, considering that most of the existing techniques, like testing and formal verification, provide incomplete error paths, leading to residual defects that are hard to trace back and fix when debugging, if the software error conditions are only partially known and described.

In this thesis, we introduce CEGECoRe - Counter Example Guided Error Condition Refinement - a CEGAR-based approach integrated into CPAchecker, that aims to identify and find a precise error condition of C programs. CEGECoRe extends traditional CEGAR to iteratively find and analyze concrete counterexamples, extract error traces from them via SMT solving techniques like quantifier elimination and instrumenting the program to exclude them from future iterations. Thus refining an evolving error condition until convergence, that is until no further counterexamples are found and the program can be proven safe under the exclusion of the final error condition. The evaluation on 1188 SV-COMP benchmark tasks compares CEGECoRe against DescribErr. While DescribErr achieves a broader overall coverage, in terms of task count, CEGECoRe demonstrates complementary strengths by covering 64 tasks that DescribErr did not. The results highlight the potential of counterexample-driven error condition synthesis.

Contents

1 Introduction

Overview

Modern software systems are increasingly becoming more critical to everyday life, especially in safety-sensitive domains, where even minor errors can lead to catastrophic failures. Despite the increasing need for robust and reliable software, it is often the case that such measures are not adapted by the industry. This is due to the fact that software system are growing in size and complexity in a such rapid rate, that trying to identify all errors within a system is becoming a very tedious and practically infeasible task for developers to do manually. Traditional error-analysis methods (e.g. testing) often fall short by identifying only a subset of the error paths. This incomplete identification can result in accumulation of undetected errors in large systems, which trickles down the development life-cycle and can cause subsequent residual defects that are very hard to catch and repair later on; such weaknesses in software system increases the overall maintenance time and costs [1].

In this context, automatically finding *precise error conditions*, which is a condition that accounts for *all* error-inducing inputs in a software program, becomes a challenging task. We aim to tackle this task by introducing a CEGAR [2] based approach integrated into CPAchecker [3], a novel and state of the art framework and tool for formal software verification and program analysis, based on the concept of Configurable Program Analysis (CPA). CPAchecker provides a great platform that supports various analysis techniques in the software verification domain such as SMT solving techniques, abstraction, model-checking, counterexample-guided refinement and much more.

In our methodology, we define an error condition (EC) as a sequence of statements that precisely describe error behaviors of a program, more formally, it is a finite logical formula that represents exactly and only all error-inducing inputs in a program.

To illustrate our approach, consider Figure 1.2. In this simple C program, an integer variable `x` is nondeterministically assigned, and two error conditions are specified in two locations in the program. For this example, the expected precise error condition is:

$$x < 0 \vee (x \bmod 2 = 0) \tag{1.1}$$

This condition precisely captures all inputs that would trigger an error in this program,

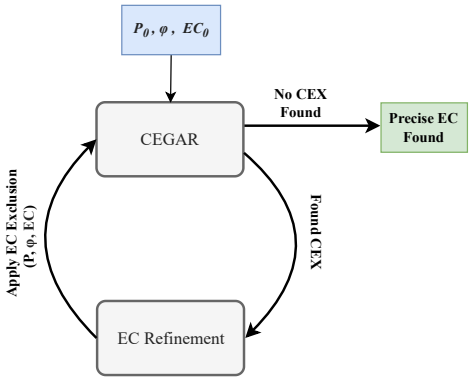


Figure 1.1: CEGECoRe Workflow

```c
int x = __VERIFIER_nondet_int();
// EC -> x is negative
if (x < 0) {
// Error State 1
    reach_error();
    return 1;
}
// EC -> x is even
if (x % 2 == 0) {
// Error State 2
    reach_error();
    return 1;
}
```

Figure 1.2: Simple C Program Example

i.e., when the *reach_error* function is reached (either when `x` is negative or when `x` is even).

CEGECoRe

We call our approach CEGECoRe (Counterexample Guided Error Condition Refinement). In CEGECoRe we strive to automatically synthesize a precise error condition for a given C program P_0 and a specification φ, where φ represents the unreachability property (i.e., no execution should reach *reach_error*). However the approach is not limited to synthesize conditions for unreachability errors, but can be extended to other verification properties used in SV-COMP, by replacing and adjusting property φ to other software verification properties like no-overflow and valid-memsafety [4].

In Figure 1.1, we demonstrate the workflow and the main components of CEGECoRE. Starting with the inputs to the algorithm, we assume that P_0 is an unsafe program, meaning that there exists a counterexample CEX for P_0, where CEX is an execution path that violates the input property φ. The initial error condition, denoted as EC_0, is simply *true*, implying that no error-inducing inputs are excluded at the start.

Our approach then proceeds by entering a CEGAR-like iterative refinement loop. In each iteration, the following steps are performed:

1. **CEGAR Loop:** The analysis starts by entering the traditional CEGAR loop, which aims to eliminate spurious counterexamples until a concrete counterexample CEX is found or the program is proven safe. This loop consists of the following steps:

a) **Predicate Abstraction and Model-Checking:** The first step is to over-approximate the behavior of the program P_0 by computing an abstraction of its concrete state space through predicate analysis [5]. The analysis starts with a finite set of predicates π (precision), which is derived from concrete executions of the program at the current location and is initially very coarse. The abstract states are represented as formulas over these predicates in the precision π for each program location. Over-approximation means that the abstraction may include some spurious behaviors (paths that are infeasible in the actual concrete program) but guarantees that no *real* error paths are missed. Using CPAchecker we perform a reachability analysis on the abstracted program.

b) **Feasibility Check:** Once an abstract error state (counterexample CEX) is reached in the reachability analysis, the CEX path is reconstructed in the concrete program P_0 and it is checked for its feasibility (whether it is a spurious or a real counterexample) using SMT solving methods [6].

c) **Precision Refinement:** If the SMT solver deem the found CEX to be real, then it is returned. However if the CEX is spurious, we want to exclude it from the next CEGAR iteration. This exclusion is done by refining the precision π, which involves adding new constraints (predicates) to it. These predicates can be derived from the counterexample CEX via techniques like Craig's interpolation [7]. After the precision refinement, the process is repeated with the new precision until the the property φ is proven safe for program P_0 or a concrete counter example is found.

2. **EC Refinement:** Since we assume that an error state exists in P_0, we will eventually find a *real* counterexample CEX from the CEGAR loop. Upon finding CEX, the approach excludes it from the next CEGECoRe iteration, essentially complementing CEGAR and eliminating concrete counterexamples. This exclusion is done by extracting information (predicates) from CEX via SMT solving techniques and adding it to the previous error condition, thus refining it and making it more precise. We introduce three different refinement strategies:

   a) **Generate Model Refiner:** In this strategy, an SMT solver is used to compute a *model* for the counterexample CEX. For instance, in our simple example 1.2, the SMT solver might find the model $x = -1$.

   b) **AllSAT Refiner:** Rather than stopping at a single model, the AllSAT refiner attempts to enumerate *all* satisfying models for the counterexample CEX (e.g., set of models $x = -1, x = -2, x = -3, \dots$). AllSAT restricts the solution search space and tries to recognize patterns from satisfying models candidates and generalize the condition offering more efficiency to the trivial model generation that might not terminate for a formula with infinite satisfying assignments.

   c) **Quantifier Elimination Refiner:** This strategy leverages *quantifier elimination* techniques in SMT solving and manipulates and transfers a formula

with quantifiers into an equivalent formula with those quantifiers removed, which is then easier to analyze. In our example, let the condition $(x < 0)$ be reformulated with an existential quantifier over integers,

$$\exists k \in \mathbb{Z}(x = k \wedge k < 0)$$

in the theory of linear integer arithmetic (LIA), the quantifier is eliminated trivially because k is redundant, this produces:

$$QE(\exists k \in \mathbb{Z}(x = k \wedge k < 0) \equiv (x < 0)$$

This describes then all negative integers and the error condition $(x < 0)$ can derived through this elimination of the quantifiers.

Each of these refinement strategies find a solution in the form of a predicate that satisfies the concrete counterexample CEX. The found predicate is then added to EC_i and excluded from the next iteration. The analysis is then re-run with the same property φ, the updated error condition, denoted as EC_i and the (remaining) program P_i, which is the original program P_0 excluding the error inducing inputs covered by the current error condition EC_i, ensuring that in the next iteration a new concrete counterexample is found.

3. **Termination:** The refinement loop repeats until no further concrete counterexamples are found (CEGAR termination), indicating that the program, under the accumulated error condition EC, is safe and EC is precise, or until a preset time limit is reached.

The entire CEGECoRe approach is implemented in Java and integrated into CPAchecker. CPAchecker offers a a great environment and starting point for our work, its modular architecture allows us to build in our technique in an non-complicated manner. Preliminary experiments on SV-Benchmarks indicate that our method is competitive with existing approaches such as DescribErr [8]. In a larger context, CEGECoRe can offer improvements in error detection and debugging efficiency.
The remainder of this thesis is organized as follows:

- **Chapter 2 (Related Work):** Provides an overview of existing techniques in error analysis that are similar to our work and how our proposed approach fit within the broader context of software verification and debugging.
- **Chapter 3 (Background):** Introduces the theoretical foundations, including CPA, SMT solving, and the principles of CEGAR, necessary to understand our approach.
- **Chapter 4 (CEGECoRe):** Details the CEGECoRe approach, including the three different refinement strategies.
- **Chapter 5 (Implementation):** Describes the implementation of the CEGECoRe approach in CPAchecker.

- **Chapter 6 (Evaluation):** Presents the experimental evaluation of the approach on SV-Benchmarks and compares its effectiveness and efficiency against DescribErr.
- **Chapter 7 (Conclusions and Future Work):** Summarizes the contributions of the thesis, discusses its limitations, and outlines directions for future research.

2 Related Work

There has been a lot of research over the past years that have addressed the challenge of identifying error paths. An invariant by definition is a logical property or condition that holds throughout a specific phase or sequence of operations within a program or algorithm. When a program's execution reaches a point where an invariant is not maintained, it may indicate a bug within the code. This is why, synthesizing invariants that capture error behaviors in a program, can be of great importance when it comes to debugging and proving the correctness of a program. Broadly speaking, the techniques used for finding program invariants can be split into two main categories: dynamic and static.

Dynamic invariant detection is a technique, which involves instrumenting a program to track and monitor the values of variables during the execution time. By running the program on a diverse set of test cases and then monitoring the behavior of the values, it it possible to observe some reoccurring patterns and then infer the invariants from those patterns [9]. The *DAIKON* system [10], for example, is a great tool that applies the dynamic invariant approach, where the tool executes the program and observes the values of its variables and then suggests a likely set of candidate invariants (or properties) that holds at a certain point in the program. Various approaches (e.g., IDiscovery [11], PIE [12], ICE [13], NumInv [14], SymInfer [15]) have made improvements on *DAIKON*, by building on top of it, usually starting with a dynamic inference of candidate invariants and then statically verifying that they hold for all inputs. IDiscovery technique for instance, integrates a feedback loop of new test case generated by symbolic execution of the instrumented code to further refine the generated set of candidate invariants. Although dynamic invariants detection approaches have been successful and are generally efficient and can handle expressive invariants, they are highly dependent on the quality of the test cases, and they frequently generate spurious invariants that may not apply to all possible inputs.

Static methods on the other hand, deduce invariants *without* executing the program itself, but rather by analyzing the source code or an abstracted model of the source code. Several approaches have applied static invariants synthesis with algorithms like abstract interpretation [16], symbolic analysis [17] and constraint or template-based synthesis [18].

Another static approach that is becoming more common in recent times, is the use of counterexamples to guide the refinement of abstractions, which is also the key idea behind our approach CEGECoRe. CEGAR [2] is a verification technique that starts with a coarse abstraction of the system and incrementally refines it based on counterexamples. When an error is detected in the abstract model, the corresponding counterexample is checked for feasibility in the concrete system. If it turns out to be spurious, the abstraction is refined—typically by adding new predicates—to eliminate the spurious behavior. This iterative process continues until either a real counterexample is found or the system is proven to be error-free.

This iterative approach has been integrated into several systems and verification tools. These methods however are mostly focused on refining the state-space abstraction rather than refining and finding a precise error conditions. Our approach utilizes CEGAR to find concrete counterexamples and then iteratively excluding them from the program, complementing CEGAR, which excludes and eliminates spurious counterexamples. The refinement loop in our approach is implemented within CPAchecker and leverages SMT solving techniques like quantifier elimination and AllSAT algorithms in order to synthesize the error condition, which is our motivation for CEGECoRe.

3 Background

To understand the contributions of this thesis, we first define the fundamental concepts and tools that our approach build on.

3.1 Control Flow Automaton (CFA)

A software program can be modeled in many different ways. Source code for example is human friendly and is a suitable format for developers to read and write, whereas compiled binary code is machine friendly and more suitable for a computer to execute. A *control flow automaton* (CFA) is a directed graph representation of a program. CFA is a tuple of $P = (L, l_0, G)$ where

- $L = \{l_0, \ldots, l_n\}$ is finite set of program locations, modeling the program counter.
- $l_0 \in L$ representing the program entry.
- Set $G \subseteq L \times Op \times L$ represents all the control-flow edges, modeling all possible transitions between two locations in L, where Op represents a program's operation, which is generally an assumption (e.g., $[x > 0]$) or an assignment (e.g., $x = 0;$) statement.

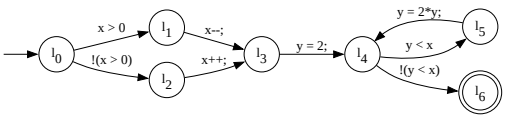
Figure 3.1b represents the constructed CFA of an example piece of code in figure 3.1a.

3.2 Program Property

To verify a program's correctness, it is essential to define what "correct" means by specifying the properties to be proven. Let $\varphi \in \Phi$, where Φ denotes the set of specification formulas describing the intended behavior of the program. These properties fall into two categories: *implicit* and *explicit*. Implicit properties capture general defects that are universally undesirable (such as out-of-bounds indexing, arithmetic overflows, null pointer dereferences, deadlocks, and resource leaks), while explicit properties are tailored to a program's specific functional requirements and are defined using assertions that ensure certain conditions hold at designated locations in the program.

```
1 if (x > 0) {
2     x--;
3 }
4 else {x++;}
5 y = 2;
6 while (y < x) {
7     y = y * 2;
8 }
```

(a) Piece of C code.



(b) Corresponding CFA representation of the code.

Figure 3.1: Comparison between the C code and its Control Flow Automaton (CFA) representation.

In a Control Flow Automaton (CFA), assertions are modeled as conditional branches leading to an *error location* $l_e \in L$ if the condition fails. Multiple assertions can share this error location l_e to indicate failures. We can then define a formal verification, as the task of determining the reachability of an error location l_e within the CFA. And we consider a program to be safe if the error location is unreachable; otherwise, it's considered unsafe [19].

The properties addressed in our approach fall under the category of *ACTL* (All Computation Tree Logic) properties. ACTL is a subset of Computation Tree Logic (CTL) [20] that includes only universal path quantifiers, such as AG (for "always globally") and AF (for "always eventually"). These operators express that a certain condition holds on all possible execution paths of the program. For example, a property like AG $\neg$`error` asserts that the error state is never reached on any execution path. This aligns with our focus on safety properties, where the goal is to ensure that "something bad never happens" across all possible executions.

3.3 Software Verification

Software verification is the process of ensuring that a software system meets specified requirements and it serves the purpose it was designed for. This process includes various techniques, including testing, static analysis, and formal methods, that help in verifying the *correctness* of the software [21].

3.3.1 Model Checking.

In software verification model checking [22, 23] is a static formal verification technique, that involves creating a finite abstract state space model of a software system and automatically and systematically verifying whether a certain specification or property is violated in the given system. Model checkers are usually associated with high precision, i.e. a correct verification result, but on the expense of time, because model checking explores the whole state space and keeps track of each distinct execution path (no merging

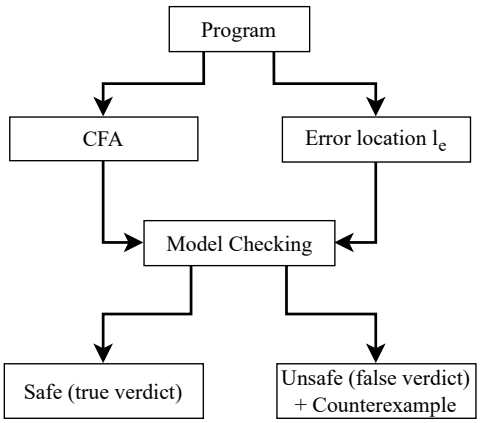


Figure 3.2: Model Checking

of different paths). This path-sensitivity allows for more precise verification, especially when checking for property violation.
If there exists *no* path that starts from the initial state(s) of the CFA and leads to a state with error location $l_e \in L$ (violation of the specified property), the verification verdict is *true*, implying that the program is safe. However if there exists such a path and $l_e \in L$ can be *reached*, the verification verdict is *false*, implying the program is unsafe for the specified property and this path is returned as proof, also referred to as counterexample 3.2.

3.3.2 Program Analysis.

Program analysis [23] in software verification is another formal static verification technique. It starts similar to model checking with an over-approximat-ion of the program's behavior without executing it . But rather than exhaustively exploring every distinct execution path as in model checking, program analyzers use methods like abstract interpretation [16] and data-flow analysis to propagate and combine (merge) abstract state information in the CFA edges. This merge is done at control-flow join points using a join operator. This merging of states and summarizing of the behavior of multiple paths into one increases the efficiency and the scalability of the verification process, especially on larger programs, where model-checking becomes practically infeasible due to the state space explosion. However this over-approximation costs the losing of precision.

3.4 Configurable Program Analysis (CPA)

3.4.1 CPA Definition

Configurable Program Analysis (CPA) [24] is a unifying framework that bridges both concepts of model checking and program analysis (mentioned in 3.3).

CPA defines an abstract domain to represent program states, a transfer relation to propagate these states along a CFA, and operators (such as merge and stop) to manage state combination and termination. By tuning these operators, CPA can be configured to merge information for efficiency or maintain distinct execution paths for higher precision. This flexibility allows the analysis to balance scalability with accuracy, adapting to the specific verification needs of a software system. In essence, CPA serves as the theoretical foundation that enables the integration and comparison of different verification approaches within one coherent framework. Formally denoted a CPA $\mathbb{C}$ is a four tuple $(D, \rightsquigarrow, merge, stop)$ and operates on a CFA (L, l_0, G). Each component in the CPA is explained below:

Lattice Before defining the components of the CPA, we provide the definition of lattices. Let E be a set equipped with a partial order $\sqsubseteq$. This partial order is defined by the following properties:

- **Reflexivity:** $\forall e \in E : e \sqsubseteq e$.
- **Transitivity:** $\forall e, e', e'' \in E : \; e \sqsubseteq e' \wedge e' \sqsubseteq e'' \Rightarrow e \sqsubseteq e''$.
- **Antisymmetry:** $\forall e, e' \in E : \; e \sqsubseteq e' \wedge e' \sqsubseteq e \Rightarrow e = e'$.

If every subset of E has a least upper bound, then the structure $(E, \sqsubseteq, \sqcup, \top)$ is called a semi-lattice. For any two elements e, $e' \in E$, the join operator $\sqcup$ yields their least upper bound, and the top element $\top$ is an upper bound for every element in E (formally, one may define $\top = \bigcup E$). This lattice structure is essential in abstract interpretation as it guarantees that the abstract domain has a well-defined notion of combining information and that the analysis will eventually converge.

Abstract Domain. The abstract domain is defined as

$$D = (C, \mathcal{E}, [\![\cdot]\!]),$$

where:

- C is the set of concrete states.
- $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ forms a *semi-lattice* (a special case of a lattice):
  - E is the set of abstract states.
  - The relation $\sqsubseteq \subseteq E \times E$ is a preorder operator (i.e., it is reflexive and transitive) which orders abstract states by their precision (with more precise states being "lower" in the order).

  - The join operator $\sqcup : E \times E \to E$ computes the least upper bound of two abstract states with respect to $\sqsubseteq$. This operator is **commutative**, **associative**, and **idempotent**—properties that ensure that combining abstract states is both well-defined and robust.
  - $\top \in E$ is the least precise (or most general) abstract state representing the entire state space, and $\bot \in E$ is the most precise (or often infeasible) state.
- The *concretization function* $[\![\cdot]\!] : E \to 2^C$ maps an abstract state to the set of concrete states it represents.

Transfer Relation. The transfer relation $\rightsquigarrow$: $E \times G \times E$ describes how abstract states evolve as control flows through the program. For an abstract state $e \in E$ and a CFA edge $g \in G$ with $(g = (l, op, l'))$, the relation computes all the corresponding abstract successor states e' that overapproximate the effect of executing the operation op on edge e. We denote this as: $e \overset{op}{\rightsquigarrow} e'$.

Merge Operator. The merge operator, $merge : E \times E \to E$, is used to combine two abstract states when execution paths converge. In the context of the underlying lattice $(E, \sqsubseteq, \sqcup, \top)$, the merge operator is implemented as an upper-bound operator – typically the join operator $\sqcup$. In other words, when merging e and e' (with $e, e' \in E$), we require that both states are subsumed by their merge, i.e.,

$$e \sqsubseteq \ merge(e, e') \quad \text{and} \quad e' \sqsubseteq \ merge(e, e').$$

This ensures that the combined abstract state is at least as general as each individual state, thereby preserving soundness while reducing the overall number of states.

Stop Operator. The stop operator, $stop : E \times 2^E \to \{true, false\}$, determines whether a newly derived abstract state should be added to the waitlist. One implementation is to check this new state is already covered by an existing state in the reached set. A Formally, for a new abstract state $e \in E$ and a set of reached states $R \subseteq E$, if there exists an $e' \in R$ such that

$$e \sqsubseteq e',$$

then $stop(e, R)$ returns `true`. This use of the lattice order $\sqsubseteq$ explicitly connects the lattice structure to CPA, and prevents redundant exploration by halting further refinement of abstract states that do not add new information.

3.4.2 Abstract State, Abstract Path and Counterexample

Abstract State. An abstract state is an element $e \in E$ in the abstract domain D that encapsulates the current program location along with a Boolean formula (or similar representation) summarizing the relevant properties of program variables in that location. The concretization function $[\![e]\!]$ maps e to the set of concrete states represented.

Abstract Path. An abstract path is a sequence of abstract transitions induced by a corresponding sequence of CFA edges. Formally, given a sequence of CFA edges

$$(l_0, op_1, l_1),\ (l_1, op_2, l_2),\ \ldots,\ (l_{n-1}, op_n, l_n),$$

the corresponding abstract path is

$$e_0 \xrightarrow{op_1} e_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} e_n,$$

where e_0 is the initial abstract state and each e_i is computed using the transfer relation $\rightsquigarrow$.

Counterexample. A counterexample is defined as an abstract path that leads to an error state—i.e., an abstract state $e_e \in E$ at an error location l_e such that the corresponding concrete states in $[\![e_e]\!]$ violate the specified property. Since counterexamples are defined over sequences of CFA edges, mapping back to a concrete execution is straightforward by following the transfer relation along these edges.

3.4.3 Reachability Analysis

Reachability analysis is the process of computing the set of states that can be reached from a given initial state by repeatedly applying the transfer relation. Formally, for a CPA $\mathbb{C} = (D, \rightsquigarrow, merge, stop)$ and the abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$, with $e_0 \in E$ being the initial abstract state corresponding to the initial concrete state at the initial location l_0 of a Control-Flow Automaton (CFA) (L, l_0, G). The *reachable set* $\mathcal{R}$ is the smallest set satisfying:

$$e_0 \in \mathcal{R} \quad \text{and} \quad \forall\, e \in \mathcal{R}, \forall\, g \in G \text{ with } e \overset{op}{\rightsquigarrow} e',\ e' \in \mathcal{R}.$$

This computation is performed iteratively using the transfer relation, merge, and stop operators until a fixed point is reached.

3.4.4 Abstract Reachability Graph (ARG)

An *Abstract Reachability Graph (ARG)* is a helpful graphical representation of the abstract space. In an ARG, each node represents an abstract state, while each directed edge corresponds to an abstract transition $e \overset{op}{\rightsquigarrow} e'$ defined by a CFA edge. An ARG is used to identify execution paths that leads to error states (i.e. counterexamples). If ARG contains a path ending in an abstract state at an error location in the program, there is a corresponding execution path in the concrete program (which can be feasible and infeasible).

3.4.5 CPA Algorithm

The CPA algorithm [24] performs reachability analysis on a control-flow automaton (CFA) by iteratively computing and merging abstract states. Initially, both the waitlist

Algorithm 1: CPA Algorithm adapted from [24]

Input : $(D, \rightsquigarrow, merge, stop)$: a CPA
Input : $e_0 \in E$: initial abstract state (where E is the lattice of D)
Input : cfa: CFA (needed for the $\rightsquigarrow$)
Output: R: reached set (all reachable abstract states)
Result : all reachable abstract states

```
1  waitlist = {e_0};
2  reached = {e_0};
3  while waitlist ≠ ∅ do
4      choose e ∈ waitlist;
5      waitlist = waitlist \ {e};
6      for each e' such that e ⇝ e' do
7          for each e'' ∈ reached do
8              e_new = merge(e', e'');
9              if e_new ≠ e'' then
10                 waitlist = (waitlist ∪ {e_new}) \ {e''};
11                 reached = (reached ∪ {e_new}) \ {e''};
12             end
13         end
14         if ¬stop(e', reached) then
15             waitlist = waitlist ∪ {e'};
16             reached = reached ∪ {e'};
17         end
18     end
19 end
20 return reached;
```

and the reached set contain the initial abstract state e_0. In each iteration, an abstract state is removed from the waitlist and its successors are computed using the transfer relation, which captures the effect of program operations. Each computed successor is then merged with every state in the reached set using the merge operator; if the merge operator produces a new state, the corresponding element in the reached set is updated accordingly.

The stop operator then determines whether this new state is already covered by an element in the reached set (with respect to the lattice order $\sqsubseteq$). It is crucial that the stop operator is configured properly, because one could for example set $stop(e, R)$ to always return `true`—causing the algorithm to terminate immediately—this would result in only a trivial abstraction and not compute the necessary over-approximation of the reachable state space. The algorithm terminates when the waitlist is empty, yielding an over-approximation of all reachable abstract states, as illustrated in Algorithm 1.

3.5 Predicate Analysis

Predicate analysis is a program analysis that applies predicate abstraction [5], which is a technique that simplifies program verification by mapping the (possibly infinite) set of concrete states into a finite abstract domain defined by a fixed set of logical predicates over the variables that appears in the program. The basic idea of predicate abstraction is that instead of inspecting individual values of variables (e.g. $x = 1, y = 2$), values are grouped together into equivalence classes. This means that for example any value of x that is smaller than the value of y satisfies $(x < y)$ and can be considered as equivalent under one abstract state represented by the predicate $(x < y)$. Therefore in predicate analysis instead of tracking concrete values of x and y, one might only track if $(x < y)$ holds or not, reducing the state space from infinite value pair of x and y to two abstract states $(x < y)$ or $\neg(x < y)$.

In predicate analysis, the precision π is a finite set of predicates that determines the granularity of the over-approximation of abstract states. Techniques such as counterexample guided abstraction refinement (CEGAR) and interpolation can dynamically refine this precision during analysis and lazy abstraction refinement allows various precisions at different program locations, but for simplicity a fixed set of predicates is assumed throughout the analysis [23].

Predicate Abstraction Example. Consider the simple C program shown in 3.3.

In this example, the initial value of `x` is nondeterministic. A predicate abstraction might select the predicate set

$$\pi = \{\, p_1 : x > 0,\ p_2 : x = 0 \,\},$$

and define the abstraction function as

$$\alpha_\pi(c) = (b_1, b_2), \quad \text{with} \quad b_1 = \begin{cases} 1 & \text{if } c \models (x > 0), \\ 0 & \text{otherwise,} \end{cases} \qquad b_2 = \begin{cases} 1 & \text{if } c \models (x = 0), \\ 0 & \text{otherwise.} \end{cases}$$

For instance, if the nondeterministic input yields $x = 1$, then

$$\alpha_\pi(1) = (1, 0),$$

indicating that $x > 0$ holds. In the true branch of the `if` statement, the operation $x = x - 1$ produces $x = 0$, and hence

$$\alpha_\pi(0) = (0, 1).$$

This abstract state corresponds to an error since the assertion `assert(x != 0)` is violated. Predicate analysis in CPAchecker would detect this error path and, through refinement if necessary, determine that the property does not hold.

```
1
2 extern int __VERIFIER_nondet_int();
3
4 int main() {
5     int x = __VERIFIER_nondet_int();
6     if (x > 0) {
7         x = x - 1;
8     } else {
9         x = x + 1;
10    }
11    assert(x != 0);
12    return 0;
13 }
```

Figure 3.3: Simple Example Program for Predicate Abstraction

3.5.1 Predicate Analysis CPA

In our approach we focus on predicate based abstraction and in order to realize this, we define predicate analysis as its own CPA, taken from [23]. Predicate CPA is represented as a four-tuple:

$$\mathbb{C}_{\text{pred}} = (D_{\text{pred}}, \rightsquigarrow_{\text{pred}}, merge_{\text{pred}}, stop_{\text{pred}}) \text{ operated on a } CFA = (L, l_0, G),$$

with precision π defined as a finite set of predicates over program variables X, with $false \in \pi$. Given a set $r \subseteq \pi$, we denote φ_r as a conjunction of all predicates in r, $\varphi_r = \bigwedge_{p \in r} p$, with $\varphi_{\{\}} = true$.

Abstract Domain. The abstract domain $D_{\text{pred}} = (C, \mathcal{E}_{\text{pred}}, [\![\cdot]\!]_{\text{pred}})$ is based on the idea of representing regions by conjunctions of predicates. Where the components of the domain are defined as follows:

- C represents the concrete states.
- The semi lattice $\mathcal{E}_{\text{pred}} = \left(2^{\pi}, \sqsubseteq, \sqcup, \top\right)$, where the partial order $\sqsubseteq$ is defined by $r \sqsubseteq r'$ if $r \supseteq r'$. In this ordering, a set with more predicates represents a stronger (i.e., more constrained) abstraction since the conjunction of predicates in r implies the conjunction in any $r' \subseteq r$. The least upper bound of two abstract states is given by their intersection, i.e., $r \sqcup r' = r \cap r'$. Top element $\top = \emptyset$, meaning least constraint on an abstract state, so that all concrete states are considered.
- The concretization function is defined as

  $$[\![r]\!]_{pred} = \{c \in C \mid c \models \varphi_r\},$$

  maps abstract states to concrete states, where, as mentioned earlier, φ_r is a conjunction of all predicates in r, with $\varphi_{\{\}} = true$.

Transfer Relation. The transfer relation $\rightsquigarrow_{\text{pred}}$ relates an abstract state $r \subseteq \pi$ and a control-flow edge $g \in G$ to a successor state r'. There is a transfer

$$r \overset{g}{\rightsquigarrow}_{\text{pred}} r'$$

if the strongest post-condition $\text{post}(\varphi_r, g)$ is satisfiable and if r' is the largest subset of π such that, for each predicate $p \in r'$, the abstraction satisfies

$$\varphi_r \implies \text{pre}(p, g),$$

where $\text{pre}(p, g)$ denotes the weakest precondition for the predicate p with respect to the edge g. The operators post and pre are defined so that

$$[\![\text{post}(\varphi, g)]\!] = \{c' \in C \mid \exists c \in C : c \xrightarrow{g} c' \text{ and } c \models \varphi\},$$

$$[\![\text{pre}(\varphi, g)]\!] = \{c \in C \mid \exists c' \in C : c \xrightarrow{g} c' \text{ and } c' \models \varphi\}.$$

In practice, the Cartesian abstraction allows these checks to be implemented with separate entailment queries for each predicate in π.

Merge Operator. The merge operator $merge_{\text{pred}}$ is defined as $merge_{\text{sep}}$, meaning that abstract states are *not* merged when the control flow converges. Each abstract state is maintained separately to preserve the precision necessary for accurate analysis.

Stop Operator. Similarly, the stop operator $stop_{\text{pred}}$ is defined as $stop_{\text{sep}}$. This operator checks for termination by considering each abstract state individually, ensuring that the analysis stops when every concrete state is covered by some abstract state without merging distinct paths.

3.6 Craig Interpolation

Craig interpolation [7] is used to extract new predicates for refinement when a spurious counterexample is detected. Suppose we have two formulas φ_1 and φ_2 (derived from different segments of an execution trace) such that their conjunction is unsatisfiable:

$$\varphi_1 \wedge \varphi_2 \quad \text{is unsatisfiable.}$$

A Craig interpolant I is a formula that satisfies:

1. $\varphi_1 \models I$ (every model of φ_1 satisfies I),
2. $I \wedge \varphi_2$ is unsatisfiable, and
3. I only contains symbols common to both φ_1 and φ_2.

In our context, φ_1 might represent the accumulated conditions along a path leading to a potential error, while φ_2 represents the remaining part that leads to an error state. The interpolant I captures the missing information—i.e., the necessary condition that invalidates the error path. By adding the predicates from I to the precision, the abstraction is refined to eliminate the spurious counterexample.

3.7 SMT Solving

Satisfiability Modulo Theories (SMT) [25, 6] solving works by determining whether a boolean formula is satisfiable (i.e. has a solution (*model*)) within a context of specific first order theories e.g., linear arithmetic, bit-vectors, arrays). Given a boolean formula φ in a theory T with variables $x_1, \ldots, x_n$, an SMT solver determines whether there exists an interpretation I (a model) that assigns values to variables such that

$$I \models_T \varphi,$$

where $\models_T$ denotes satisfaction under theory T. If φ is satisfiable, the solver may also perform additional tasks such as model generation, enumerating all satisfying assignments (AllSAT), or eliminating quantifiers. If φ is unsatisfiable however, the solver can perform additional checks (e.g., interpolation, unsat-core, etc).
In software verification, SMT solvers such as MathSAT [26] and Z3 [27] integrate these techniques to reason about different complex verification tasks.

In our context, SMT solvers are used to find models for path formulas representing concrete counterexamples. Essentially finding input value assignments to program variables that induce an error in the program.

3.7.1 Model Generation

When an SMT solver determines that formula φ is satisfiable, it can generate a model M, which is one possible solution to φ (assignment of values to the free variables in the formula). Formally, a *model* M for a a formula φ in theory T is a function $M : Var(\varphi) \to D_T$, where D_T is the domain of T (e.g., integers for linear arithmetic) and $Var(\varphi)$ denotes the free variables in φ. M satisfies φ (we write $M \models_T \varphi$) if substituting each variable x_i with $M(x_i)$ makes φ true under T.

Example: For a given formula φ in linear integer arithmetic set to:

$$(x > 0) \wedge (x < 2),$$

an SMT solver might return a model M with $M = \{x \to 1\}$.

3.7.2 AllSAT

AllSAT extends model generation by enumerating all possible satisfying assignments (models) for a given formula. Formally, given a formula φ, the AllSAT procedure returns a set:

$$\{M_1, M_2, \ldots, M_k\} \quad \text{where} \quad \forall i : M_i \models_T \varphi \quad \text{and} \quad \forall i \neq j : M_i \neq M_j\,,$$

Example: For the formula:

$$\varphi \equiv (x = 0) \vee (x = 1),$$

AllSAT returns:

$$\{M_1 = \{x \to 0\}, M_2 = \{x \to 1\}\}.$$

For formulas over finite domains or with a finite number of satisfying assignments, All-SAT can, in principle, enumerate all models.

3.7.3 Quantifier Elimination

Quantifier elimination transforms a formula with quantifiers into an equivalent quantifier-free formula within the same logic theory (i.e. removing quantifiers while preserving satisfiability). Quantifier elimination become very important when formulas contain quantifiers, because the solving process becomes significantly more complex and by eliminating quantifiers the SMT solver can work with simpler quantifer-free formulas. For a formula

$$\varphi(x) \equiv \exists y\, \psi(x, y),$$

a quantifier elimination procedure produces a quantifier-free formula $\phi(x)$ such that for all $M : Var(\phi) \to D_T$,

$$M \models_T \phi \iff \exists M' : Var(\varphi) \to D_T \text{ s.t. } M' \models_T \varphi \text{ and } M'(x) = M(x).$$

In order to achieve this elimination of quantifiers, different SMT solvers employ different strategies and algorithms for different theories, each with advantages and disadvantages regarding computation complexity (e.g., virtual substitution [28, 29], cylindrical algebraic decomposition (CAD) [30]).

Example: Consider the formula

$$\varphi(x) \equiv \exists y\, (y^2 = x \wedge y \geq 0).$$

Step by step elimination:

1. Solve $y^2 = x : y = \sqrt{x}$ or $y = -\sqrt{x}$.
2. With $y \geq 0$, only $y = \sqrt{x}$ is valid.
3. $\sqrt{x}$ is a real number only if $x \geq 0$.
4. Thus, the quantifier-free equivalent is $x \geq 0$.

Quantifier elimination yields the quantifier-free formula $x \geq 0$.

3.8 Counterexample-Guided Abstraction Refinement

3.8.1 CEGAR Definition

Counterexample-Guided Abstraction Refinement (CEGAR) [2] is an iterative approach used in software verification to progressively improve an abstract model until the property φ is either proven to hold or a concrete counterexample is identified, as illustrated in Figure 3.4.
In our context, we use CEGAR with a predicate CPA for the abstraction and model checking. Given a predicate CPA $\mathbb{C}^{\pi_0}_{pred}$ as defined in 3.4.1, where π_0 is the initial coarse precision (a fixed, small set of predicates), program P and a property φ, the CEGAR loop proceeds as follows:

1. **Abstraction and Model-Checking:** Analyze P using $\mathbb{C}^{\pi_0}_{pred}$. This produces an abstract model of P, denoted as $\mathcal{M}$. The analysis then represents $\mathcal{M}$ as an abstract reachability graph (ARG) and performs a reachability analysis on it. If the analysis discovers an error path (counterexample) CEX in $\mathcal{M}$, then this path is extracted as a candidate witness for a violation of φ.

   $$CEX = e_0 \xrightarrow{op_1} e_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} e_{error},$$

   where s_0 is an initial abstract state and s_{error} is an abstract error state.

2. **Feasibility Check:** The crucial step is to determine if the abstract counterexample CEX corresponds to a real execution in the concrete model $\mathcal{M}$.

   Reconstruct a concrete path from CEX using the $\mathbb{C}^{\pi_0}_{pred}$ domain's concretization function and check its feasibility (via an SMT solver, 3.7). There are two cases from this model check:

   - If the counterexample CEX_c is feasible (i.e., there is a genuine *concrete* corresponding execution path), then φ is violated, the algorithm terminates and this path of CEX_c is reported.
   - If the counterexample CEX_s is spurious, formally if any state s_i for $1 < i < n$ along $CEX_s = e_0 \xrightarrow{op_1} e_1 \xrightarrow{op_2} \cdots \xrightarrow{op_n} e_n$ is empty (i.e., it does not correspond to any concrete execution path in P), then the abstraction is too coarse and the precision π should be refined. Spurious counterexamples arise because the abstraction process may have introduced behaviors not present in the original system due to the loss of precision.

3. **Precision Refinement:** When a spurious counterexample is detected, a *refinement operator* is invoked. This refinement involves distinguishing between concrete states that were previously grouped into the same abstract state. Depending on the abstraction analysis used, there are different algorithms that can implement this refinement operator (e.g. partitioning abstract states) [2]. In our analysis we use predicate abstraction, one way to implement this refinement process is by adding new predicates to the precision leading to a finer grained abstraction that

eliminates such spurious behaviors. This is because the analysis of CEX_s can reveal some conditions or expressions that were not tracked by the current precision but are essential to differentiate between concrete states along the path. Such predicates capture these conditions and are found using techniques like computing weakest preconditions or using Craig's interpolation 3.6.

Formally we can denote:

$$refine : T \times P \to P$$

where T is the set of error traces representing the abstract path extracted by the analysis. Given the spurious trace $\tau \in T$ and the current precision π_i, the refinement operator computes an updated precision π_{i+1}, where:

$$\pi_{i+1} = \pi_i \cup \Delta\pi,$$

where $\Delta\pi$ represents the additional predicates that rule out the spurious error. **Note** that CPAchecker utilizes lazy abstraction [31], that allows different precisions π_l for each program location l and the refinement process adds a new predicate $p_i \in \Delta\pi$ to only those locations l_i where the interpolant I_i includes p_i. This way the abstraction is kept as coarse as possible where the precision is unnecessary.

4. **Termination:** A refined abstraction is then recomputed by re-running the predicate CPA with the new precision $\mathbb{C}_{pred}^{\pi_{i+1}}$. This iterative process of of abstract model checking, counterexample validation, and refinement continues until convergence, i.e., until either a feasible counterexample is found (indicating a concrete error) or the property φ is verified safe with no counterexample discovered in the program P.

3.8.2 Correctness of CEGAR

This iterative process guarantee that for an abstract model $\mathcal{M}$ of a given program P under precision π, if CEGAR terminates, i.e., returns a counterexample CEX, then CEX is a feasible path in P violating a given safety property φ (proof in [2]).

3.8.3 Example CEGAR Iteration

Consider the simple C program in Figure 3.5, we follow each step of a single CEGAR iteration on this program:

1. **Abstraction and Model-Checking:** The analysis begins by abstracting P using an initial (coarse) set of predicates, say $\pi_0 = \{\ x > 0\}$. An abstraction function α is applied and maps concrete states to abstract states (elements of the abstract domain D) by tracking which predicates hold. For example, if $x = 1$ then $\alpha(1) = \{x > 0\}$. The abstract model is constructed via a CPA-based reachability analysis that uses the transfer relation to propagate these abstract states. In our example, the branch `if(x > 0)` is considered, and in the true branch the assignment $x = x - 1$ is applied, potentially leading to an abstract state where x could be 0 (i.e., the condition $x > 0$ no longer holds).

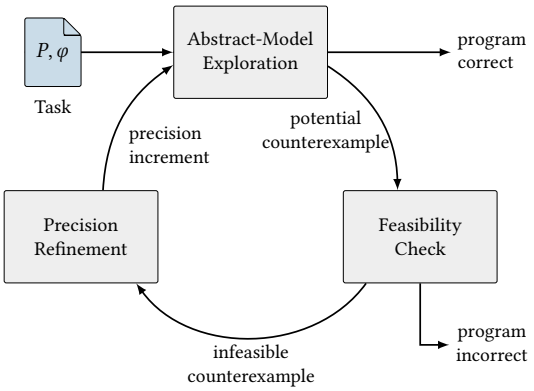


Figure 3.4: CEGAR Workflow adapted from [32].

2. **Counterexample Extraction:** An error is signaled in the abstract model when an abstract state at an error location (where the assertion `assert(x != 0)` fails) is reached. A counterexample CEX_0 is then extracted as a sequence of CFA edges:

$$(l_0, op_1, l_1),\ (l_1, op_2, l_2),$$

   where l_0 is the initial location, op_1 corresponds to the nondeterministic assignment to x, and op_2 corresponds to taking the true branch followed by $x = x - 1$. This path leads to an abstract state representing $x = 0$.

3. **Feasibility Check:** The extracted counterexample CEX_0 is then validated against the concrete semantics using an SMT solver. The solver checks if there exists a concrete execution that follows the CFA edges in CEX_0 and ultimately produces a state where $x = 0$ (violating `assert(x != 0)`). If no such concrete execution exists, the counterexample is deemed spurious.

4. **Precision Refinement:** When the counterexample is spurious, the precision is refined using Craig's interpolation to capture the missing distinctions. For instance, the refinement might add a predicate such as $x \leq 0$ or provide a more precise separation between the cases $x > 0$ and $x = 0$. The abstract model is then recomputed with the updated predicate set π_1, which eliminates the spurious counterexample. The CEGAR loop is repeated with the refined precision until either a concrete counterexample is found or the program is verified to be safe.

```
1  extern int __VERIFIER_nondet_int();
2
3  int main() {
4      int x = __VERIFIER_nondet_int();
5      if (x > 0) {
6          x = x - 1;
7      }
8      // Error: if x becomes 0, the assertion fails.
9      assert(x != 0);
10     return 0;
11 }
```

Figure 3.5: Simple Example Program for CEGAR.

4 Counterexample-Guided Error Condition Refinement (CEGECoRe)

Based on the provided material and presented theory in the background chapter, we want to now propose a new approach for synthesizing error conditions within CPAchecker. We introduce Counterexample-Guided Error Condition Refinement (CEGECoRe).

4.1 Overview

Traditional CEGAR terminates upon proving that a specification is not violated in a program or upon finding the first concrete counterexample (CEX), providing partial insights into error conditions. This suffices for proving unsafety of a program, but it fails to provide a complete characterization of all error-inducing inputs. CEGECoRe addresses this gap by extending the CEGAR loop and iteratively finding concrete counterexamples and refining an error condition (EC) that precisely captures all violating inputs. In this chapter we will discuss mainly the *EC Refinement* component of CEGECoRe, which is essentially the main component extending and complementing traditional CEGAR, which we have thoroughly discussed in section 3.8. Then we discuss some theoretical insights about the approach, its limitations and potential adaptation of the framework.

4.2 CEGECoRe Framework

CEGECoRe's workflow, Figure 4.1, can be formalized as an iterative process that increasingly build a precise error condition. We start with inputs similar to CEGAR, an input program P_0, a specified property φ (e.g., unreachability) and we introduce an evolving error condition at iteration i and denote it with EC_i. EC_i is a predicate that represents the set of inputs to P_0, that lead to an error after i refinement iterations. Formally, we can view EC_i as a formula over the program's input variables. For the CEGAR loop we use a predicate CPA $\mathbb{C}^{\pi_0}_{pred}$, as defined in 3.8, with π_0 as the initial coarse precision representing the initial abstraction model of P_0.

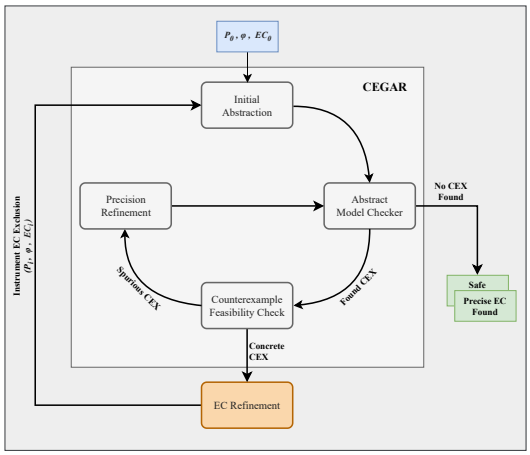


Figure 4.1: CEGECoRe Workflow (Expanded)

- **Initialization:** We start with the most general (coarsest) assumption about error inducing inputs. Initially, no specific inputs have been identified as causing errors, so the error condition is set to $EC_0 = true$. This true predicate implies that at the start we do not exclude any input, because every input is potentially error-inducing since we have no information yet (i.e., we start with an over-approximation that includes all inputs until we start the refinement cycle). Note that in this initial iteration we place no restrictions on the program's inputs or behaviors (P_0 is analyzed in its original form).

- **CEGAR loop:** We enter the traditional CEGAR loop, 3.8, which iteratively refines $\mathcal{M}_i$ until either P_i is safe under φ, in which case the algorithm terminates and the found EC_i is returned (note that if this happens in the first iteration of *CEGECoRe* (i.e., $P_i = P_0$) then $EC_i = EC_0 = true$, implying that no input to P_0 violates property φ), or a feasible concrete counterexample CEX_i is found in P_i.

- **Error Condition Refinement:** The main step in CEGECoRe is to refine the error condition EC_i using the concrete counterexample information. Intuitively, since CEX_i is a real failing execution, it provides evidence of a particular error-inducing condition on the inputs. CEGECoRe extracts a logical predicate ψ_i that generalizes the input conditions of this counterexample via a chosen refinement strategies (discussed in the next section), and then incorporates ψ_i into the error condition which then will be excluded in the next iteration. Formally, let ψ_i be a predicate over the input variables such that ψ_i is satisfied by the input assignment(s) in CEX_i. We refine the error condition by adding this predicate as a disjunct to the previous EC_{i-1}. We update:

$$EC_i = EC_{i-1} \vee \psi_i$$

Initially EC_0 was true (over-approximating all possibilities). After the first coun-

terexample, we set $EC_1 = \mathit{true} \vee \psi_1$, which is logically equivalent to ψ_1. Thus EC_1 effectively becomes ψ_1, capturing the condition under which the first counterexample CEX_1 occurred. If a second counterexample is found, we extract ψ_2 from it, and update the error condition again, $EC_2 = \psi_1 \vee \psi_2$. In general, $EC_k = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_k$ will represent the accumulated disjunction of all discovered error predicates up to iteration k. This means that any input satisfying EC_k is either one of the previously found error-inducing cases or falls into an error scenario equivalent to one of them.

It is crucial for the refinement step to also restrict the program from finding the same predicate ψ_i again in the next iteration. In order to achieve this exclusion of ψ_i, we constrain the program's input any input satisfying ψ_i is not considered again. In practice, this can be done by instrumenting the program or the analysis with an assumption that $\neg EC_i = \neg(\psi_1 \vee \psi_2 \vee \cdots \vee \psi_i)$ holds (i.e. disallow the already found error conditions for subsequent runs), or by modifying the model checker to treat paths satisfying predicates in EC_i as already explored. Thus, we obtain a "new" program $P_{i+1} = P_i$ with inputs restricted to those not satisfying ψ_i. This ensures that the next CEGAR iteration will search for new concrete counterexamples outside the already known error condition region.

The refinement of EC is guided by SMT-based predicate extraction from the counterexample. CEGECoRe can employ one of three strategies to derive ψ_i from the counterexample CEX_i. We formalize these strategies in the next section. After refining EC and excluding ψ_i from the program, CEGECoRe repeats the loop. A new CEGAR search on the restricted program P_{i+1} is performed to find another concrete counterexample outside of the already explored error paths . Each new concrete counterexample then yields another predicate to add to EC. We continue iterating this counterexample-guided manner.

- **Termination:** CEGECoRe terminates when after some iterations n the CEGAR loop reports that the (remaining) program P_n (representing the original program with all inputs satisfying EC_n excluded) is safe, meaning no further counterexample (feasible error path) exists under the current input restrictions ($\neg EC_n$). This implies that all error-inducing inputs have been accounted for by EC_n, since the only way P_n can be safe is if every input that would cause an error has been excluded.

  Upon termination we conclude that EC_n is a precise error condition for the program, i.e., for any initial state or input x_0 that *satisfies* EC_n, the program will reach an error (φ is violated), and for any x_0 *not satisfying* EC_n, the program is safe (φ is not violated). In the best case scenario, assuming that enough time and solver power are provided to find all concrete counterexamples, CEGECoRe's final output is an EC_n that exactly describes the program's erroneous behavior.

The CEGECoRe algorithm is summarized as pseudo code in Algorithm 2.

Algorithm 2: CEGECoRe Algorithm

Input : p: program
φ: safety property (e.g., *unreach_call*)
Output: ec: final refined error condition

```
1  ec = true;
2  while true do
3      verdict, cex = CEGAR(PredicateCPA, p, φ);
4      if verdict == false then
5          condition = extract_condition(cex);
6          ec = ec ∨ (condition);
7          p = restrict_program(p, condition)
8      else if verdict == true then
9          return ¬ec;
10     else
11         return Failed;
```

4.3 Refinement Strategies for Error Conditions

The main challenge in CEGECoRe is computing a suitable predicate ψ_i from each found concrete counterexample. Various strategies can offer trade-offs between generalization and complexity of the extracted predicate. In CEGECoRe we explore three refinement strategies for synthesizing predicates from counterexamples:

- **Generate Model Refiner:** extract a single concrete model (assignment) from the counterexample.
- **AllSAT Refiner:** enumerate multiple satisfying assignments for the counterexample's path condition to derive a more general predicate.
- **Quantifier Elimination Refiner:** eliminate existential quantifiers from the path formula to obtain a general condition characterizing the error.

Each strategy starts from the path formula or counterexample formula that led to the error and yields a predicate that will be used to refine EC. We now describe each strategy formally and provide a small illustrative example.

4.3.1 Generate Model Refiner

In this strategy, we use an SMT solver to compute **one satisfying assignment (model)** for the counterexample's path condition (recall that a counterexample CEX corresponds to a path through the program with certain branch conditions). From this path, we can derive a logical formula ϕ that must hold for the path to be feasible, we also refer to this path as *formula path*. For instance, if along the counterexample path the program

made a branch decision *if* $(x > 0)$ and later *if* $(y == x + 1)$, etc., then the formula path ϕ might look like $((x > 0) \wedge (y = x + 1) \wedge \cdots)$ up to the point of reaching the error. Now because CEX is a real execution (delivered by CEGAR), ϕ is satisfiable. We define a **model** M as a mapping from variables to values such that $M \models \phi$ (path formula ϕ holds holds under that assignment of values).

Using the SMT solver, the Generate Model Refiner obtains one such model M_i for the path formula ϕ of the counterexample CEX_i. Formally, let $Var(\phi)$ be the set of free variables (e.g., input variables) in the path formula. The solver returns a model $M_i : Var(\phi) \to D$ (where D is the domain, e.g., integers) such that $M_i \models \phi$. This model provides a specific valuation for the inputs that triggers the error path.

We then **derive the predicate** ψ_i from this model by capturing those concrete values as equalities (or as a conjunction of literals). Essentially, ψ_i is the conjunction of $x = v$ for each input variable x assigned value v by M_i. This means that, ψ_i represents exactly that single combination of inputs.

For example, lets suppose CEGAR has delivered a counterexample with formula path $\phi : (x > 0) \wedge (x < 5)$. The SMT solver might return the satisfying model $M : x \mapsto 1$ (assigning value 1 to x). The Generate Model strategy would then take M and form the **predicate** $\psi : (x = 1)$. This predicate ψ is true for the program input of the variable $x = 1$, which indeed is an input that led to the error in this counterexample. We add ψ to the error condition EC to record that input $x = 1$ causes an error and in the next iteration, the program analysis will exclude $x = 1$ from consideration (restricting the program so that $\neg(x = 1)$ holds).

Discussion: The generate model strategy is the most specific form of refinement. It tries to enumerate all possible models by focusing on one concrete input scenario at a time. It has the advantage that it is very cheap computationally (just one model generation) and straightforward (any SMT solver can provide a model for a satisfiable formula). However, the extracted predicate ψ_i we add is very **narrow**, meaning it is too specific, it may represent just one point in the input space. If there are many inputs that cause similar errors, the generate model refiner will require many iterations to find all of them. In the worst case of infinitely many error-inducing inputs (e.g., an entire range of integers), this strategy alone would not terminate or would enumerate satisfiable models endlessly. For example, if the error condition in reality is $x < 0$ (any negative x causes an error), the generate model refiner might first add $x = -1$, then $x = -2$, then $x = -3$, and so on, never generalizing to the condition $x < 0$. Thus, while this strategy is quite simple and sound (every predicate it adds truly corresponds to a real error input by construction), it might produce a very large or even infinite refinement sequence for programs with wide range of error conditions.

4.3.2 AllSAT Refiner

Instead of searching for a single model that satisfies the counterexample formula path, the AllSAT refiner strategy tries to cover all satisfying assignments for the formula path. The term AllSAT refers to the procedure that attempts to enumerate all solutions (satisfying assignments) for a satisfiability problem within some bounds. Formally, given

a formula ϕ (formula path of counterexample), an AllSAT procedure will return a set of models $M_1, M_2, \ldots, M_k$ such that for each M_i, $M_i \models \phi$, and for any possible assignment M that satisfies ϕ, M is equivalent to one of the returned M_i (i.e., the set is covering all satisfying assignments up to some equivalence), 3.7. For formulas over finite domains, AllSAT can enumerate all models exhaustively; for infinite domains or large solution spaces, typically AllSAT will either return a representative subset or use a symbolic representation to implicitly represent infinite groups of solutions.

In CEGECoRe, the AllSAT refiner uses an SMT solver (with designated AllSAT algorithms) to cover multiple satisfying assignments for the counterexample path formula, rather than just one. The idea is to capture a broader region of the error condition in one refinement step. Once a set of models $M_1, \ldots, M_k$ is obtained, the solver then **generalizes** these concrete assignments into a single predicate ψ_i that is satisfied by all of them. There are various generalization techniques and algorithms and each SMT-solver (e.g., Z3 [27], MATHSAT [26]) implements it differently. For instance, given models $x = -2, 0, 2$, one trivial generalization is the disjunction that exactly enumerates them: $(x = -2) \vee (x = 0) \vee (x = 2)$. This disjunction covers those specific cases but is obviously incomplete (it misses $x = 4$, $x = -4$, etc.). Another straightforward example, consider a case where the only safe input is a specific value and every other value causes error (e.g., a path formula with branch $if(x! = 1)$ leading to an error). If the the AllSAT yielded models $\{x = -1, x = 0, x = 2\}$ for ϕ, the solver might recognize that all those values share the property $x \neq 1$ and return $\psi : x \neq 1$. This ψ would be true for $x = -2, 0, 2$ (and for all other values except 1), representing all error-inducing inputs in this scenario. Therefore, ψ generalizes the finite set of models into a potentially infinite condition (e.g., $x \neq 1$ covers an infinite set of integers). This shows how AllSAT can lead to predicate covering broader set of assignments. In general, the AllSAT refiner aims to converge much faster on the final error condition by adding a stronger predicate than a single point. Fewer CEGECoRe iterations are needed if each added ψ_i covers a broad set of error-inducing inputs.

Discussion: The benefit of AllSAT refinement is that it can dramatically reduce the number of iterations by covering many input cases at once. It leverages the solver to explore the solution space of the path formula more thoroughly. However, it comes with a cost. Enumerating many models can be expensive, especially if the solution space is large. AllSAT procedures may struggle or even not terminate on formulas with infinitely many solutions unless heuristics or bounds are used. Moreover, in a lot of cases generalizing from a set of models to a more precise predicate is non-trivial. This means the procedure could end up simply listing many models without finding a nice pattern, which still requires multiple disjuncts (essentially, it becomes similar to the Generate Model Refiner, but enumerating all models in one iteration). Despite these challenges, when successful, AllSAT-based refinement can quickly zero down on conditions like "$x < 0$" or "$x \neq 1$" that cover infinitely many concrete cases, therefore greatly accelerating the convergence of EC.

4.3.3 Quantifier Elimination Refiner

The third strategy uses logical quantifier elimination (QE) to derive the predicate ψ_i. Quantifier elimination is a technique from logic and SMT solving that, given a formula with existential or universal quantifiers, produces an equivalent formula with those quantifiers removed.

In CEGECoRe, we typically consider the existential scenario: the counterexample indicates the existence of certain variable values that lead to error. For example, reaching an error might imply that "there exists some intermediate state or some input values such that conditions X, Y, Z hold". If we express the condition for the error path as an existential formula, quantifier elimination can sometimes produce a clearer description of the inputs, that is easier to analyze. Formally, suppose the path formula can be written in the form:

$$\varphi(x_1, \ldots, x_m) \equiv \exists y_1, \ldots, y_n \Psi(x_1, \ldots, y_m, y_1, \ldots, y_n)$$

where $x_1, \ldots, x_m$ are the input (free) variables and $y_1, \ldots, y_n$ are existentially quantified variables introduced, for example, as witnesses for some conditions (they could represent **nondeterministic choices** or intermediate values on the path). The solver can apply quantifier elimination to produce an equivalent quantifier-free formula $\tilde{\Psi}(x_1, \ldots, x_m)$ that depends only on the x variables. This quantifier-free formula $\tilde{\Psi}$ is logically equivalent to the existence of some y making Ψ true, i.e., $\tilde{\Psi}(x)$ holds if and only if there exists a y such that $\Psi(x, y)$ was true. In essence, $\tilde{\Psi}(x)$ characterizes exactly those x for which the path to error is feasible.

A classic example of quantifier elimination is over arithmetic constraints. Suppose during the counterexample path, a condition appears $\exists y; (y^2 = x) \wedge (y \geq 0)$ to witness a square root computation (this the same as saying "there exists a y such that $y = \sqrt{x}$"). The presence of $\exists y$ makes the formula more complex to reason about directly in terms of x. Applying quantifier elimination could yield an equivalent formula without y. In this example, the result would be $x \geq 0$, since the condition "there exists a real $y \geq 0$ with $y^2 = x$" is true exactly when x is non-negative. Thus, $\tilde{\Psi}(x)$ is $x \geq 0$. In the error condition context, if a counterexample formula path required the existence of some value to satisfy a constraint, QE gives us a direct constraint on the input.

Discussion: Quantifier elimination can be seen as a more symbolic form of generalization compared to AllSAT. Instead of enumerating models, it manipulates the formula to abstract away details. When it succeeds, it often produces a very clean predicate (like $x < 0$, or $x + y > 100$, etc.) which can immediately capture an entire infinite set of inputs. This can make CEGECoRe converge in very few iterations. For example, if the first counterexample path already contains enough information, quantifier elimination might yield the full error condition in one iteration (though this is optimistic). The downside however is that quantifier elimination is not always available or efficient for all theories. It may fail or produce extremely large formulas for complex conditions (e.g., non-linear arithmetic, bit-level operations, or data structures), and not all SMT solvers support quantifier elimination for all theories. Therefore, this strategy although can be very powerful and efficient (in terms of extracting predicates) but its applicability depends on the nature of the path formula.

These strategies are not mutually exclusive, they can be chosen based on the situation. One could try a cheap model generation first, and if progress is slow, switch to an AllSAT or QE approach for stronger predicates. In CEGECoRe design, all three strategies are approaches to implement the $extract_{condition}(cex)$ functionality in 2, each with its pros and cons.

4.4 Theoretical Analysis

Now that we have described the itterative approach of CEGECoRe, we discuss some theoretical properties of the approach, namely convergence and correctness of the error condition.

4.4.1 Convergence of the Error Condition

By convergence, we mean that the error condition EC_i reaches a fixed point as i grows: eventually $EC_{n+1} = EC_n$ for some n, and the algorithm terminates. In practice, convergence is detected when an iteration of the CEGAR loop yields no new counterexample, indicating no further error scenarios to discover. At that point, EC_n no longer changes (no new ψ to add). In the worst case, CEGECoRe could enumerate many distinct counterexamples without repetition. If that set is infinite and no generalization is made, the process might not converge quickly (or at all). However, if CEGECoRe is able to find a general enough predicate before resources are exhausted, it will converge.

When CEGECoRe does terminate at iteration n, we have a final error condition EC_n. This EC_n is a disjunction of all $\psi_1 \dots \psi_n$ found. No further counterexample exists outside EC_n, because if an input outside EC_n could produce an error, the CEGAR loop would have found a concrete counterexample for it before terminating. Thus, we can conclude that EC_n cannot be improved and it is considered the fixed-point error condition.

Note that convergence in CEGECoRe is in concept analog to how CEGAR's abstraction refinement converges. In CEGAR, either the abstraction becomes precise enough to prove safety or a concrete counterexample is found; in CEGECoRe, either the error condition becomes comprehensive enough to cover all error-inducing inputs (allowing the remainder of the state space to be proven safe) or the process continues to find new counterexamples. In both cases, if the procedure terminates normally, we reach a fixed point that represents a precise solution to the problem.

4.4.2 Soundness and Correctness of CEGECoRe

CEGECoRe inherits the **soundness** of the CEGAR loop for counterexample discovery. That is, any concrete counterexample it finds is a real error in the program (because of the feasibility check in each iteration). When the algorithm terminates with a final error condition EC_n, this EC_n is guaranteed to represent only error-inducing inputs and to include all such inputs. We can argue about this in two cases:

- EC_n **covers only error-inducing inputs (no false positives):** Each predicate ψ_i added to EC came directly from a feasible counterexample. That means for each ψ_i, there was an actual execution of the program where ψ_i held on the input and the program reached an error. If the final EC_n is the disjunction $\psi_1 \vee \psi_2 \vee \cdots \vee \psi_n$, then for any input that satisfies EC_n, it satisfies at least one ψ_i, which means that input will follow the corresponding error path and reach an error state. This ensures the **soundness of the error condition** (it never over-approximates beyond actual errors).

- EC_n **covers all error-inducing inputs (no false negatives):** If the algorithm terminated, it means the last CEGAR check found no counterexample outside the current EC_n. Similarly, the program P_n with inputs restricted to $\neg EC_n$ was verified safe. This implies that any input not satisfying EC_n cannot lead to an error (otherwise that input would be part of a concrete counterexample outside EC_n). This means that $\neg EC_n$ describes the region of the input space that is entirely safe.

  Because of the two previous arguments, we can say the final EC_n is a complete and correct error condition for the program. CEGECoRe provides in a sense, a certificate of "unsafety" that is stronger than a single counterexample, but rather it provides a formula that characterizes all **real** counterexamples. This result can be very useful for understanding the program's behavior or for documentation, since it tells us under what condition the program fails.

  Note that if the original program was actually safe from the start (no real counterexample exists), CEGECoRe would detect that through the initial CEGAR loop and terminate immediately, usually returning $EC_0 = \mathit{true}$ (there are no error-inducing inputs). It is more interesting to consider an unsafe program, where CEGECoRe's output is meaningful. In both cases, the procedure's outcome is sound: either "no error inputs" (safe program) or a precise description of error inputs.

4.5 Limitations and Considerations

Despite the conceptual strength that CEGECoRe offer, the approach has several practical limitations.

- **Dependency on SMT Solvers:** CEGECoRe heavily depends on SMT solvers not just to check feasibility of paths, but also to perform complex tasks like All-SAT and quantifier elimination. The effectiveness of each refinement strategy is therefore restricted by what the solver being used can do and how powerful it is.

- **Performance and Scalability:** Model checking the program for new counterexamples iteratively can be time-consuming, especially if there are many distinct error inputs. Each CEGAR is potentially expensive, and doing it repeatedly adds overhead. In the case of only excludes a very small portion of the state space the

total runtime can blow up (For example when using the trivial generate-model strategy on a program with many failing cases).

- **Interpretation of the Error Condition:** Although the goal is to find a “precise” error condition, the final formula EC_n could be very large or complex. It might be a long disjunction of many predicates, which could be hard to interpret and make sense of. Nevertheless we can look at EC_n, even when complex, as formal artifact that can be further analyzed and simplified in post processing.
- **Termination:** Although in theory, if we have powerful enough procedures, CEGECoRe should eventually find all counterexamples and deliver a precise error condition, in practice the algorithm might not terminate, because resource limits (time and memory) might cause the process to stop early.
- **Choosing a Refinement Strategy:** The "correct" choice of a refinement strategy in each CEGECoRe iteration depdends on the problem at hand. Each of the three might perform better on some problems and worse on others. A potential improvement could be adding heuristics to distinguish problem types and choosing a fitting refinement strategy, which we know performs better for that certain type of problem.

5 Implementation

This chapter presents the implementation details of the CEGECoRe algorithm within CPAchecker. Building upon the theoretical foundation discussed in previous chapters. Here we concentrate on the design of the software components that enable counterexample-guided error-condition refinement. The implementation was done as an extension to CPAchecker's existing architecture, emphasizing its modularity and reusability. In particular, we introduce several key Java classes and interfaces, such as FindErrorCondition, CompositeRefiner and Refiner, that together orchestrate the CEGECoRe refinement loop. We also discuss how multiple refinement strategies (Generate Model, AllSAT, and Quantifier Elimination) are realized, and how they interact through a composite design.

Figure 5.1 provides a high-level UML diagram of these components and their interactions.

5.1 Overview of the Main Components and Class Architecture

FindErrorCondition. The implementation introduces a new CPAchecker algorithm called `FindErrorCondition` (in package `core.algorithm.preciseErrorCondition)`. This class is the entry point for CEGECoRe, coordinating the overall refinement loop. The `FindErrorCondition` class implements CPAchecker's `Algorithm` interface so that it can plug into the CPAchecker verification workflow like any other algorithm. Internally, it wraps a standard verification algorithm (usually CPAchecker's predicate analysis) to perform safety checking on the program while refining the error condition. Key fields of `FindErrorCondition` include: a reference to the underlying `Algorithm` and `ConfigurableProgramAnalysis` (CPA) being used to find counterexamples, a `LogManager` for logging, and a `FormulaContext` for managing formula-related objects (solver and formula manager). It also maintains configuration options such as `maxIterations` (to limit the number of refinement iterations, if desired), refiners (an array specifying which refinement strategies to use), `parallelRefinement` (a boolean flag to run strategies in parallel or sequentially), `refinerTimeout` (time limit for each refinement step), and `withFormatter` (whether to pretty-print the final error condition formula for readability). These options are annotated with CPAchecker's `@Option` mechanism and are in-

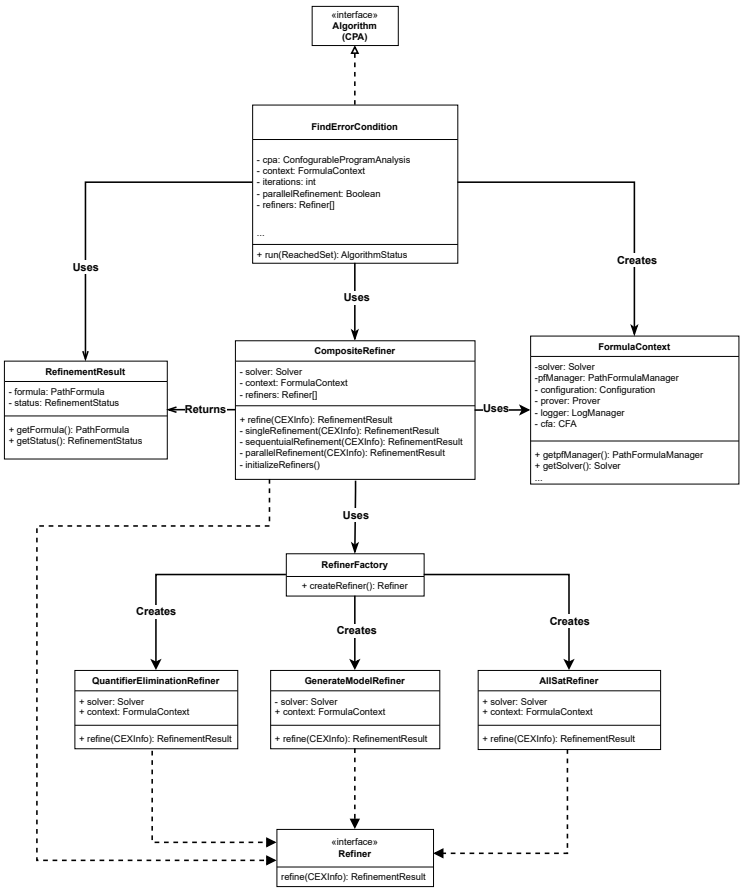


Figure 5.1: UML class diagram of key components in CEGECoRe's implementation and their interactions

jected from the configuration, allowing easy tuning via configuration files. For example, by default the `refiners` option might include two strategies (e.g. Quantifier Elimination and AllSAT) and `parallelRefinement=true`, but a user can choose a single strategy or sequential mode by changing these options.

FormulaContext. The `FormulaContext` is another important class in the implementation. It serves as a container for all formula-related utilities needed during refinement. Specifically, `FormulaContext` holds a reference to the underlying SMT solver and the path formula manager. In CPAchecker, the predicate analysis uses a `PathFormulaManager` to convert program paths into logical formulas (path formulas) and a `Solver` to decide satisfiability. The `FormulaContext` encapsulates a `Solver` instance (along with its `FormulaManager`) and a `PathFormulaManager`, as well as the CFA (control-flow automaton) of the program and some configuration/logging objects. By bundling these into one context, we can easily pass `FormulaContext` to each refiner, ensuring they all

use a consistent view of the solver and formula creation utilities. Note that CEGECoRe may use multiple solver instances, the main CPAchecker analysis might use one SMT solver (e.g., MathSAT5 for interpolation), while a specific refiner (Quantifier Elimination) might require another solver (Z3). The `FormulaContext` class provides a method `createContextFromThis(String solverName)` to spawn a secondary context with a different SMT solver. For example, the `QuantifierEliminationRefiner` uses this to obtain a Z3-based context for performing quantifier elimination (since MathSAT5 does not support that operation). All refiners share the original `FormulaContext` (and any secondary contexts created from it), instead of each creating their own solver, which ensures consistency (the same logical formulas can be translated and reused) and avoids unnecessary overhead.

CompositeRefiner. The `CompositeRefiner` class is central to our design. It implements `Refiner` and internally manages a set of actual refinement strategy instances. During initialization, `CompositeRefiner` uses the provided array of `RefinementStrategy` (an `enum` listing available strategies, e.g., `GENERATE_MODEL, ALLSAT, QUANTIFIER_ELIMINATION`) to create the corresponding refiner objects via a factory (RefinerFactory.createRefiner(...)). For each strategy, an appropriate `Refiner` implementation is constructed and stored in a map (keyed by the strategy enum). For example, if configured, it will instantiate a `GenerateModelRefiner`, an `AllSatRefiner`, and/or a `QuantifierEliminationRefiner`. All of these share the same `FormulaContext` (passed in the constructor) so they operate on the same formula space.

During the refinement loop, when `FindErrorCondition` calls compositeRefiner.refine(counterexample), the `CompositeRefiner` decides which mode to use based on how many strategy instances it has and the `parallelRefinement` flag. We implemented three modes of operation in `CompositeRefiner`

1. The *single* mode is the simplest: if only one refinement strategy is configured, the composite directly delegates to it. This mode has minimal overhead and is ideal when evaluating or using a single strategy in isolation.

2. In *sequential* mode, multiple strategies are available, but only one is executed at a time. Each strategy is applied to the counterexample in sequence, with timeout per strategy. If the first fails, the next is tried, and so on, until one succeeds or all fail. This mode increases robustness without incurring the CPU overhead of parallelism and is especially useful when some strategies perform better on certain kinds of counterexamples. The trade-off is longer runtime, but determinism and low resource usage make this a practical default in constrained environments.

3. The *parallel* mode runs all configured strategies simultaneously and uses the result from the first one that finishes successfully. A fixed thread pool launches each refinement task, and an `ExecutorCompletionService` monitors their progress. If one strategy succeeds, the others are cancelled to conserve resources. This mode is the most aggressive in resource consumption but often yields the fastest wall-clock time.

5.2 Refinement Strategies and Solver Integration

we implemented three distinct refinement strategies as separate classes, each conforming to the `Refiner` interface: GenerateModelRefiner, AllSatRefiner, and QuantifierEliminationRefiner. Each strategy uses the SMT solver in a different way to extract predicates (i.e., candidate formulas) from the counterexample, reflecting the approaches described theoretically in Section 4.3. Below we describe each strategy's purpose and inner workings, and how they leverage SMT solver capabilities. All strategies operate on the feasible counterexample path provided (we assume the counterexample is concrete or has been confirmed by CPAchecker's feasibility check, so a model for the path exists in principle).

Three refinement strategies are implemented:

- **GenerateModelRefiner:** This is the simplest strategy, focusing on one counterexample input at a time. The `GenerateModelRefiner` uses the solver's model generation capability to obtain a concrete assignment for the program's input variables that leads to the counterexample. In implementation, we take the path formula for the counterexample's error path (a Boolean formula encoding all assumptions along that path) and assert it to the solver. Because the counterexample is an actual bug witness, this formula is satisfiable. We then query the solver for a model (truth assignment) of the formula. The solver (e.g., MathSAT5 or Z3) provides values for each variable in the formula; we are particularly interested in the input (nondeterministic) variables. The refiner collects the model assignments for any input-related variables – for example, if the program has a nondet input x and in the counterexample $x = 5$, the model will include xß5. These assignments are then combined into a single formula m (essentially a conjunction of equalities capturing this input). In our implementation, we identify inputs by naming conventions (e.g., variables containing "_nondet" in their names are treated as nondet inputs, as is common in SV-COMP) and conjoin their model values: e.g., $nondet1 = 5 \wedge nondet2 = 0 \wedge \ldots$ for all such inputs. This formula m represents one particular error-inducing input. The error condition refinement is then simply to exclude this input in the future. We do so by negating m and conjoining it to the growing error-condition exclusion formula: $EC := EC \wedge \neg m$. Intuitively, we tell the verifier "forbid this specific input combination, and then try again." The `GenerateModelRefiner` sets its `RefinementResult` status to `SUCCESS` whenever it successfully obtains such a model and updates the exclusion formula. On the next iteration, the program with the new assumption $\neg m$ will force the analysis to find a different counterexample (if any exist) that violates the property. Over multiple iterations, this will enumerate distinct error-causing inputs one by one.

- **AllSAT Refiner:** The `AllSatRefiner` class implements a refinement strategy that generalizes over sets of counterexample inputs by enumerating satisfying assignments using the SMT solver's AllSAT functionality. Upon receiving a counterexample, it constructs the corresponding path formula and extracts a set of Boolean atoms (e.g., (x > 0)), which represent conditions that distinguish different potential input behaviors. These atoms are passed to the solver, which

iteratively enumerates all satisfying combinations of them that make the path feasible. Each such combination is treated as a separate model, represented as a conjunction of literals, and collected using a custom `AllSAT callback`. The disjunction of all these model formulas captures the full input space leading to the error along that path. This combined formula is then negated and added to the growing error condition, excluding a broad set of inputs in one refinement step. This approach is efficient when the number of distinct input conditions is small, but can be expensive or infeasible if the space is large or continuous.

- **Quantifier Elimination Refiner:** we rely here on the SMT solver's ability to perform quantifier elimination. In `QuantifierEliminationRefiner` it is a bit more complex internally, because it uses two solver contexts. The main CPAchecker analysis is using MathSAT5 (which does not support quantifier elimination on arbitrary formulas), so we create a secondary Solver context with a solver that does (e.g., Z3). We then translate the counterexample path formula into the language of that solver. This translation adds overhead and complexity to the strategy, especially when the path formula we are dealing with is quite large. The path formula is essentially $\exists(v) : P(x, v)$ where x are input variables and v are other existentially quantified variables (program variables along the path, which can be considered existentially quantified since the path's feasibility guarantees their existence). We then task the solver to eliminate those existential variables, yielding a quantifier-free formula $\Phi(x)$ over only the inputs. In practice, Z3 provides APIs to eliminate quantifiers for certain theories (e.g. linear integer arithmetic) or uses heuristics (qe tactics) to produce an equivalent formula without the quantified vars. We apply a variable partition, we instruct the elimination procedure to treat all variables that are not inputs (not containing "_nondet" in their name) as quantified, and preserve the input ones. The result is a formula $QE(x)$ that implies the path. After obtaining this result from Z3, we translate it back into the primary solver (so that it can be used with the same `FormulaManager` as the rest of the analysis). This translation back adds another layer of complexity and sometimes fail because the quantifier elimination has modified the path formula and thus the syntax of the quantified formula in the solver Z3 language is also changed and the translation fails because parts of the syntax are not recognized by the original solver (i.e., MATHSAT5). The obtained $QE(x)$ is expected to be TRUE for exactly those input assignments that satisfy the counterexample's path constraints. We then exclude it by conjoining $\neg QE(x)$ to the error condition. If Z3 times out, or if it throws an exception (e.g., if the theory is too complicated for elimination), the `QuantifierEliminationRefiner` will fail for that iteration. We chose Z3 as the quantifier-elimination solver because of its robust support for quantifiers and is supported by CPAchecker.

Each strategy is encapsulated in its own class and interacts with the `FormulaContext`. The choice of Z3 for quantifier elimination is deliberate: Z3 supports quantifier elimination in the theory of linear arithmetic and integrates smoothly with CPAchecker's

formula infrastructure, whereas alternatives like MathSAT5 lack reliable quantifier elimination capabilities.

5.3 Configuration Options

CEGECoRe exposes several parameters through CPAchecker's configuration system:

- `findErrorCondition.maxIterations`: Limits the number of refinement steps.
- `findErrorCondition.refiners`: Specifies the set of refinement strategies to use (GENERATE_MODEL, QUANTIFIER_ELIMINATION, ALLSAT).
- `findErrorCondition.parallel`: Enables parallel execution mode if multiple refiners are specified.
- `findErrorCondition.timeout`: Sets a timeout (in seconds) per refiner.
- `findErrorCondition.qSolver`: Selects the SMT solver for quantifier elimination.
- `findErrorCondition.withFormatter`: Enables pretty-printing of the final error condition.

These options affect the performance of the algorithm. For instance, enabling parallel refinement improves coverage but incurs higher CPU usage. Choosing a timeout that is too low may prevent powerful strategies like QE from succeeding, while increasing it can lead to higher solve times. Additionally, CEGECoRe keeps the global CPAchecker configurations such as the overall wall-clock timeout, memory limits, and the underlying CPA configurations.

In 5.2 is an example of CEGECoRe run command.

```
bin/cpachecker --heap 10000M --timelimit '1800 s' \
  --option cpa.predicate.memoryAllocationsAlwaysSucceed=true \
  --predicateAnalysis \
  --option analysis.algorithm.findErrorCondition=true \
  --option findErrorCondition.refiners=QUANTIFIER_ELIMINATION \
  --option findErrorCondition.withFormatter=true \
  --option cpa.predicate.memoryAllocationsAlwaysSucceed=true \
  --spec test/programs/benchmarks/properties/unreach-call.prp \
  --64 test/examples-findErrorCondition/program.c
```

Figure 5.2: Example Run command for CEGECoRe

6 Evaluation

In the following chapter, we present the evaluation of the proposed CEGECoRe approach. The evaluation is centered on the following main research questions (when referring in this chapter to a task being "solved", we mean that the approach managed to find a precise error condition):

- **RQ1 (Effectiveness & Uniqueness):** How many SV-COMP tasks can be solved (that is, for how many tasks a precise error condition can be found) by the CEGECoRe approach compared to the sister approach DescribErr? And can CEGECoRe solve tasks that DescribErr can not?
- **RQ2 (Efficiency):** How does the resource consumption (in terms of CPU and wall time) of the CEGECoRe approach compare to DescribErr?
- **RQ3 (Refinement-Strategy):** How do the different refinement strategies implemented in the CEGECoRe approach in CPAchecker behave when trying to identify precise error conditions?

CPAchecker [33] is an open-source verification tool and framework that implements the CPA concept (described in 3.4) in Java to analyze C programs. It transforms the input source code into a CFA and applies configurable program analyses to systematically explore the abstract state space for property violations. Depending on its configuration, CPAchecker can operate in a mode that efficiently merges states for scalability for example for very large and complex programs or in a highly precise mode that preserves distinct execution paths, similar to model checking.

6.1 Experimental Environment

Benchmark Set

The experiments are conducted on the widely used SV-Benchmarks suite, which provides of a comprehensive set of C programs for evaluating software verification tools and methods. In this collection of C programs, we examine the *ReachSafety* category, which comprises of the following subcategories:

- ReachSafety-BitVectors
- ReachSafety-Combinations
- ReachSafety-ControlFlow
- ReachSafety-ECA
- ReachSafety-Floats
- ReachSafety-Hardware
- ReachSafety-Heap
- ReachSafety-Loops
- ReachSafety-ProductLines
- ReachSafety-Sequentialized
- ReachSafety-XCSP
- SoftwareSystems-AWS-C-Common-ReachSafety
- SoftwareSystems-BusyBox-ReachSafety
- SoftwareSystems-coreutils-ReachSafety
- SoftwareSystems-DeviceDriversLinux64-ReachSafety
- SoftwareSystems-uthash-ReachSafety

Here, a program error is indicated by the *reach_error* function, serving as an equivalent to *assert(0)*, such an error is flagged, when this function is reached by a program execution. Each of these tasks has an assigned label, called *expected_verdict*, which specifies whether the task is expected to produce a *true* or *false* result by the verifier.

Since we are interested in finding error conditions of programs, we examine only the programs with *false* expected verdict; we find 3131 such tasks. From this set of programs, we exclude 1820 tasks, because their CPAchecker's predicate analysis delivers a different result from the assigned expected verdict. This happens due to errors or timeouts in CPAchecker. In order to have fair comparison between the two approaches we want to evaluate the same set of tasks, we perform an intersection on the two sets and the result is 1188 total tasks. These tasks cover various categories (for example, device-driver benchmarks, memory safety tasks, and crafted examples like AWS C-common harnesses and RERS problems) in SV-Benchmarks, providing a broad assessment of CEGECoRe's ability to find precise error conditions.

Run Configuration

All experiments were conducted utilizing the BenchExec benchmarking framework [34], which facilitated the reliable execution and concurrent monitoring of our tests across multiple cloud-based machines. BenchExec simplifies the adjustment of run configurations and resource allocations through a definition of an XML file, and provides an understanding of the results via interactive tables and plots. For the configuration of our experiments we have specified that the analysis in CPAchecker[3] is invoked with a wall-time limit of 900 seconds and a hard time limit of 1000 seconds, along with a memory limit of 7 GB per task. The configuration mandates the use of 2 CPU cores and targets machines with an Intel Core i7-6700 @ 3.40 GHz and running on operating system Ubuntu 22.04.

It is **important** to note that the DescribErr results, with which we draw our comparisons, employed a significantly extended wall-time timeout of 12,000 seconds per run, as well as the utilization of eight processing units and a 31 GB memory limitation. Due to the time constraints of this thesis, we have been unable to execute the CEGECoRe approach under identical conditions. This is why, we have opted to exclude certain runs (tasks) from the DescribErr results that exceeded a duration of 900 seconds in order to ensure an equivalent comparison between the two approaches. Nonetheless, we express a keen interest in exploring and examining the outcomes of the CEGECoRe approach under similar configuration settings in future research.

6.2 Results Overview

We first present an overview of the outcomes across all tasks, highlighting how many tasks were solved by each approach, as well as how often they timed out or crashed. A task is considered solved for CEGECoRe if the tool managed to refine an error condition successfully (outputting "true" at the end, meaning no further counterexamples exist under the derived condition), and for DescribErr if it outputs "false (precise error condition)" (meaning it produced a precise error condition). Table 6.1 summarizes the performance of each refinement strategy of CEGECoRe and each variant of DescribErr on the 1188 tasks. We report the number of tasks solved, its percentage out of the overall task set and the number of tasks that resulted in timeouts or errors (unexpected failures).

Looking at the table 6.1, several high-level observations can be made. DescribErr solved a substantially larger fraction of tasks than CEGECoRe. The best DescribErr configuration (Template-based) managed to produce precise error conditions for 327 tasks (27.5% of the total), whereas the best single CEGECoRe strategy (Generate Model) solved 107 tasks (9.0%). Even when considering all strategies, CEGECoRe's overall coverage is much lower, if we combine CEGECoRe's results (i.e., count a task as solved if any of the strategies solved it), the total unique tasks solved by CEGECoRe is 150 (12.6%). In contrast, DescribErr's collectively solved 399 unique tasks (about 33.5%). This indicates that CEGECoRe is currently less effective (in terms of tasks solved) than the DescribErr approach (we will analyze potential reasons under RQ1).

Approach (Strategy)	Solved Tasks	Timeouts	Errors/Crashes
CEGECoRe – Generate Model	107 (9.0%)	682	399
CEGECoRe – AllSAT	28 (2.4%)	845	315
CEGECoRe - Quantifier Elimination	70 (5.9%)	618	500
DescribErr - Counterexample	292 (24.6%)	700	196
DescribErr - Interval	299 (25.2%)	703	186
DescribErr - Template	327 (27.5%)	429	432

Table 6.1: Overall results for CEGECoRe (with All-SAT, Model-generation, and Quantifier-Elimination refiners) vs. DescribErr. Timeouts indicate the number of tasks that exceeded the set CPU limit. Other Failures include runs that crashed or ran out of memory before completion.

In terms of timeouts, both approaches have a significant number, showcasing the difficulty of the problem and the strict time limit. CEGECoRe's strategies timed out on the a large portoion of tasks. For instance, AllSAT refinement hit the timeout on 845 tasks – this is unsurprising, as enumerating many models can be very slow. The Quantifier Elimination strategy timed out on 618 tasks, and Generate Model on 682 tasks. DescribErr's counterexample and interval strategies timed out on roughly 700 tasks each, similar to CEGECoRe's numbers, but the template strategy has a notably lower timeout count (429). The error cases include cases like crashes (recorded as segmentation faults), Java exceptions (for example, "ERROR (interpolation failed)" in some Generate Model runs, or an "EXCEPTION" in Quantifier Elimination if Z3's API threw an error), and assertion violations. For CEGECoRe-Quantifier Elimination, in particular, 5 tasks ended with out-of-memory (OOM) errors and a few with unhandled exceptions, indicating the heavy memory usage of quantifier elimination on some inputs. CEGECoRe-AllSAT and CEGECoRe-Generate Model both saw a significant number of segmentation faults (233 and 328 respectively), likely stemming from low-level issues in the SMT solver or the JavaSMT layer when handling numerous queries.

In summary, DescribErr solves more tasks overall, while CEGECoRe in its current state suffers from many timeouts. However, CEGECoRe's different strategies have managed to solve (i.e., find a precise error condition) some tasks that DescribErr did not which is an improvement and a potential to leverage both approaches to get a more comprehensive precise error condition. Another important detail to recall is that the set time limit for CEGECoRe was only 900s, which for strategies like quantifier elimination on a complex very long task might not be sufficient.

6.3 RQ1: Effectiveness & Uniqueness, CEGECoRe vs. DescribErr

The evaluation shows that DescribErr outperforms CEGECoRe in terms of solved tasks, Figure 6.1. DescribErr (in its best configuration) solved 327 tasks, whereas CEGECoRe (best single strategy solved 107, and even combining all strategies yields 150 in com-

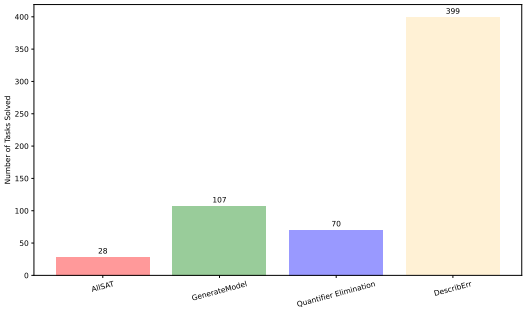


Figure 6.1: Tasks solved by all strategies

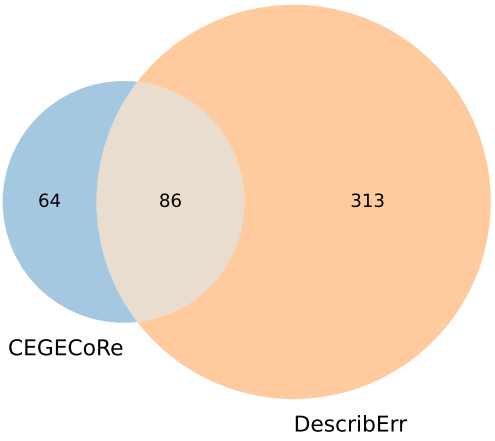


Figure 6.2: Unique tasks solved by each approach

parison to the 399 of DescribErr). This answers RQ1 in favor of DescribErr: it can find precise error conditions for a significantly larger portion of tasks. The difference is quite pronounced – roughly, DescribErr covered about one-third of the tasks, while CEGECoRe covered about one-eighth.

There are a few potential reasons for this discrepancy. CEGECoRe's approach, by design, requires multiple iterations of counterexample generation and refinement. If a single iteration is computationally expensive or the number of counterexamples is large, it may not complete within the 15-minute time limit. DescribErr, in contrast, attempts to infer the error condition directly, for example template-based analysis uses predefined logical formula structures (e.g., linear inequalities or boolean predicates) that can efficiently capture complex input conditions when the true error condition matches the structure of a template. Particularly in structured domains like RERS or ECA, where CEGECoRe struggles.

In many tasks, CEGECoRe generated one or a few counterexamples but failed to reach a fixed-point before timing out. Meanwhile, DescribErr's template analysis could infer the complete condition in a single, although sometimes costly, inference step. Moreover, DescribErr exhibits greater robustness—it handles failures gracefully and times out cleanly, whereas CEGECoRe experienced more solver crashes, reducing its effective coverage.

However, CEGECoRe showed unique strengths on 64 tasks that DescribErr could not solve, as shown in Figure 6.2. These tasks often involved low-level harnesses or negated preconditions, such as in certain `aws-c-common` or device-driver benchmarks, where CEGECoRe's counterexample-guided refinement allowed it to succeed. Although less effective overall, CEGECoRe demonstrates clear complementary capabilities by solving some tasks that static inference methods like DescribErr could not.

6.4 RQ2: Efficiency, CEGECoRe vs. DescribErr

To evaluate efficiency, we compared CEGECoRe and DescribErr based on CPU time, memory usage, and how effectively each tool used its time limit (900s for CEGECoRe and 7200s for DescribErr) 6.3. On average, CEGECoRe was much faster for the tasks it solved: AllSAT completed in 5.95 seconds, GenerateModel in 51.97 seconds, and Quantifier Elimination (QE) in 62.25 seconds. DescribErr's strategies, in contrast, consumed significantly more time per solved task, averaging 179.72 seconds (Counterexample), 143.78 seconds (Interval), and a significant 2703.46 seconds for the Template strategy.

Similarly the memory usage results follows this trend. CEGECoRe stayed within modest bounds (averaging 260 MB), while DescribErr consumed substantially more memory, particularly in the Template strategy (1.7 GB). However, these resource differences correlate with coverage, DescribErr solves more tasks overall (potentially more complex ones), justifying its higher consumption. When both tools solve the same task, CEGECoRe can be marginally faster ,though not by a large or consistent margin.

The results show that for RQ2: CEGECoRe is more time and memory-efficient on tasks it can solve, but these tend to be trivial cases. DescribErr, especially with its template approach, is slower and more resource-intensive per solved task but delivers results for a broader and harder range of programs. Therefore, from a perspective of prioritizing outcome over runtime, DescribErr currently offers better efficiency in practice. However, CEGECoRe shows promise particularly in some specific cases. There are potential improvements and further optimizations that might yield better results.

6.5 RQ3: Refinement-Strategy Comparison

Figure 6.4 shows that the three refinement strategies in CEGECoRe (Generate Model, AllSAT, and Quantifier Elimination) own complementary strengths. Generate Model was the most effective, solving 107 tasks. Its basic and simple idea made it a reliable, especially for easy to moderate sized error input spaces. Therefore enumerating them and excluding one counterexample at a time worked in these cases. Despite its low overhead, it did run into timeouts on tasks with very large or infinite error spaces, where it could not refine quickly enough. Still, it had fewer crashes than the more complex strategies, making it a good candidate as a baseline strategy for CEGECoRe.

Quantifier Elimination (QE) solved 70 tasks, fewer than Generate Model, but it performed especially well on benchmarks where the error condition could be symbolically captured. In such cases, QE often completed the refinement in a single step, leveraging Z3's built-in quantifier elimination. However, it also showed a high failure rate, with over 600 timeouts and numerous solver errors, Table 6.1, typically on tasks involving complex formulas, non-linear arithmetic, or large variable scopes. Despite this, QE contributed 36 uniquely solved tasks and overlapped with Generate Model in 26 tasks, highlighting its ability to solve problems that Generate Model could not. The four tasks solved by all three strategies were mostly trivial cases.

AllSAT had the weakest performance, solving only 28 tasks, with just 3 solved uniquely. Its poor results were likely due to model enumeration overhead, when faced with many

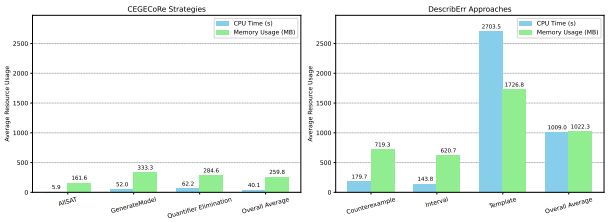


Figure 6.3: Resource usage in solved tasks.

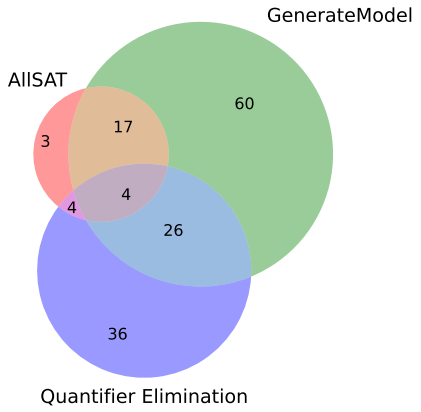


Figure 6.4: CEGECoRe refinement strategies Venn Diagram - Overlap.

satisfying assignments, the AllSAT solver could not terminate or crashed.

The Venn diagram in Figure 6.4, shows that no single strategy could cover all solvable tasks. Out of 150 tasks solved by CEGECoRe in total, 60 were solved only by Generate Model, and 36 only by QE showcasing the difference in types of problem that each of them can solve. A smart composite refinement approach here is if we ideally run QE and AllSAT in parallel and falls back to Generate Model. This could achieve full coverage under time constraints. The current evaluation highlights the value of combining strategies rather than relying on any one alone for every CEGECoRe iteration. It also suggests future directions (i.e., improving AllSAT's efficiency, optimizing QE's stability, and integrating adaptive strategy selection based on task characteristics.3)

6.6 Threats to Validity

Benchmark Selection Bias (External Validity): We performed our experiments on SV-COMP benchmarks, which are well-known and varied, but our results might not generalize to all possible programs (e.g., industrial code bases or on other properties like memory safety, not just reachability). We also use CPAchecker, which is an award winning verification tool and framework, namely we rely on CPAchecker predicate analysis its implementation of the CEGAR loop for our implementation. However taking the concept of CEGECoRe over to other verification tools, might yield different results.

Configuration and Implementation Factors (Internal Validity): The CEGECoRe implementation is a prototype in CPAchecker. The many crashes indicate there is room for improvement. It's possible that with bug fixes CEGECoRe would solve additional tasks (some tasks may have actually been solvable but crashed midway). Small changes like solver version or different CPU could potentially also impact which tasks time out.

We assumed that when the result of the analysis was "true" (i.e., the program deemed safe by CPAchecker's predicate analysis), it indeed found a precise error condition. We did not manually validate each reported condition for precision. It is possible (though unlikely with a tool like CPAchecker) that a reported error condition is not 100% precise. We trust the tools' internal checks – e.g., CEGECoRe would only output "true" after a final verification that no further counterexample is found. However, any unsoundness of the tool would violate our assumptions. Given the soundness of the CPAchecker framework, which have been used in competitions and is highly regarded in the field, we are confident in the soundness of our results.

Resource Limits: We have chosen 900s and 7GB as run configuration for CEGECoRe. CEGECoRe might have solved a few more tasks if given more time (some runs had remaining counterexamples but ran out of iterations/time). Therefore, the exact numbers may vary with different assignment of time and memory limits per run. In theory, given infinite time CEGECoRe would eventually find all error conditions, since it is complete.

7 Conclusion

7.1 Summary

In this thesis, we introduced CEGECoRe, a novel counterexample-guided refinement approach for synthesizing precise error conditions in software verification, and integrated it into the CPAchecker framework. Unlike traditional CEGAR, CEGECoRe continues refining after discovering the first counterexample, aiming to characterize all failing inputs through iterative abstraction. We implemented three SMT-based refinement strategies (Generate Model, AllSAT, and Quantifier Elimination) within a modular *CompositeRefiner*, enabling flexible sequential or parallel configuration of the refinement step. The technical integration required careful management of formula contexts and solver interactions to ensure both correctness and extensibility. This allowed CEGECoRe to produce concrete, verifiable error condition formulas on real-world C programs.

Our evaluation on 1188 SV-COMP benchmarks shows that while the sister approach DescribErr achieves higher overall coverage, CEGECoRe contributes in a different aspect, it managed to solve 64 tasks that DescribErr did not. Each CEGECoRe strategy showed different strengths. Generate Model was the most reliable, QE was effective on arithmetic-heavy inputs, and AllSAT had limited but sound performance. Although CEGECoRe faced scalability challenges, especially with timeouts and solver failures, it demonstrated that iteratively finding concrete counterexamples, refining the error condition and excluding them from future runs, can successfully synthesize error conditions. This highlights CEGECoRe's complementary value and supports its integration as part of a diversified verification pipeline.

7.2 Future Work

There are several promising directions to extend CEGECoRe. We believe that a key opportunity is extending the refinement loop duration, repeating experiments with a 2-hour time limit (as used by DescribErr) could uncover CEGECoRe's ability to solve harder tasks it currently times out on.

One potential improvement is adaptive strategy selection. As we have seen each strategy works well on different types of problems. A potential version of CEGECoRe could

include a mechanism to dynamically choose or prioritize strategies based on feedback. For example, if a counterexample's path formula is purely linear arithmetic, favor quantifier elimination. If it's largely Boolean with many combinations, perhaps try AllSAT or a BDD-based refiner. If it's very complex or unknown, start with model enumeration to gather more data. Initial ideas were discussed, for example using simple heuristics on the formula structures. Implementing this could improve performance and potentially reduce unnecessary timeouts.

Another direction that is interesting to explore, is to develop additional or improved refinement strategies. For instance, a smarter AllSAT that can integrate heuristics on how to better generalize the condition after finding a few satisfying models. Another example could be to focus on developing a smarter refinement strategy, such as counterexample clustering (i.e., using a technique (potentially with CEGAR as well) to find multiple concrete counterexamples first and then try to learn and infer a general condition that covers them).

7.3 Data-Availability-Statement

All data generated or analyzed during this study are available as supplementary material. In particular, the detailed results outputted by CPAchecker. In the .CSV files once find the outcomes and resource usage per task as well as the configuration and run command. The plotted graphs discussed in this chapter have been also provided (merged in one pdf). This allows independent verification of our results and further exploration. The CEGECoRe implementation has been integrated into CPAchecker's open-source which can be found under the FindErrorCondition branch.

Bibliography

[1] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8): 707–740, 2016.
doi: 10.1109/TSE.2016.2521368.

[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45047-4.
doi: 10.1007/10722167_15.

[3] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 184–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.
doi: 10.1007/978-3-642-22110-1_16.

[4] Dirk Beyer. Software verification with validation of results. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54580-5.
doi: 10.1007/978-3-662-54580-5_20.

[5] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Computer Aided Verification*, pages 72–83, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69195-2.
doi: 10.1007/3-540-63166-6_10.

[6] 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.

doi: 10.1007/3-540-45319-9_19.

[7] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
doi: 10.2307/2963593.

[8] Dirk Beyer, Lars Grunske, Matthias Kettl, Marian Lingsch-Rosenfeld, Moeketsi Raselimo, and Stefan Winter. Synthesis of precise error conditions. 2024.

[9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 213–224, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581130740.
doi: 10.1145/302405.302467.

[10] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007. ISSN 0167-6423.
doi: https://doi.org/10.1016/j.scico.2007.01.015.

[11] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 362–372, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452.
doi: 10.1145/2610384.2610389.

[12] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 42–56, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612.
doi: 10.1145/2908080.2908099.

[13] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 499–512, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492.
doi: 10.1145/2837614.2837664.

[14] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 605–615, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058.

doi: 10.1145/3106237.3106281.

[15] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. Syminfer: inferring program invariants using symbolic states. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, page 804–814. IEEE Press, 2017. ISBN 9781538626849. URL https://dl.acm.org/doi/10.5555/3155562.3155662.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450373500. doi: 10.1145/512950.512973.

[17] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 363–374, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542517.

[18] Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma. Linear invariant generation using non-linear constraint solving. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 420–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45069-6. doi: 10.1007/978-3-540-45069-6_39.

[19] Dirk Beyer. Software verification with validation of results. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, page 331–349, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662545799. doi: 10.1007/978-3-662-54580-5_20.

[20] Wojciech Penczek, Bożena Woźna, and Andrzej Zbrzezny. Bounded model checking for the universal fragment of ctl. *Fundam. Inf.*, 51(1–2):135–156, January 2002. ISSN 0169-2968. doi: 10.5555/2371116.2371125.

[21] Martin Brain and Elizabeth Polgreen. A pyramid of (formal) software verification. In Andre Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods*, pages 393–419, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-71177-0. doi: 10.1007/978-3-031-71177-0_24.

[22] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262032708.

doi: 10.5555/332656.

[23] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. *Combining Model Checking and Data-Flow Analysis*, pages 493–540. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8.
doi: 10.1007/978-3-319-10575-8_16.

[24] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

[25] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. ISSN 0001-0782.
doi: 10.1145/1995376.1995394.

[26] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36742-7.
doi: 10.1007/978-3-642-36742-7_7.

[27] 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.

[28] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1):3–27, 1988. ISSN 0747-7171.
doi: https://doi.org/10.1016/S0747-7171(88)80003-8.

[29] V. Weispfenning. Quantifier Elimination for Real Algebra — the Quadratic Case and Beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8 (2):85–101, January 1997. ISSN 1432-0622.
doi: 10.1007/s002000050055.

[30] George E. Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991. ISSN 0747-7171.
doi: 10.1016/S0747-7171(08)80152-6.

[31] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, January 2002. ISSN 0362-1340.
doi: 10.1145/565816.503279.

[32] Dirk Beyer, Jan Haltermann, Thomas Lemberger, and Heike Wehrheim. Decomposing software verification into off-the-shelf components: an application to cegar. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 536–548, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211.
doi: 10.1145/3510003.3510064. URL
https://doi.org/10.1145/3510003.3510064.

[33] D Baier, D Beyer, P.C. Chien, M.C. Jakobs, M. Jankola, M. Kettl, N.Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, H. Wachowitz, and P. Wendler. Software verification with cpachecker 3.0: Tutorial and user guide. In *Formal Methods*, pages 543–570. Springer Nature Switzerland, 2024. ISBN 978-3-031-71177-0.
doi: 10.1007/978-3-031-71177-0_30.

[34] D Beyer, S Löwe, and P Wendler. Reliable benchmarking: requirements and solutions. *Int J Softw Tools Technol Transfer*, 01 2019.
doi: 10.1007/s10009-017-0469-y.

List of Figures

List of Algorithms

List of Tables