Ludwig-Maximilians-Universität München

Institut für Informatik

Implementation of Value Analysis over Symbolic Memory Graphs in CPAchecker

Master's Thesis in Computer Science

30.9.2022

Daniel Baier

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Thomas Bunk

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, wurden auch als solche gekennzeichnet. Weiterhin versichere ich, dass diese Masterarbeit, weder in gleicher noch in ähnlicher Form, bei einer anderen Prüfungsbehörde von mir vorgelegt worden ist.

$Unterschrift:$

$Ort, Datum:$

Abstract

Concrete-value analysis is a software verification technique that uses concretely known values in programs to analyze them. It is independent of an external solver and has been successfully extended by the techniques CEGAR and interpolation to improve upon its results. Despite the simplicity of the value analysis, it is up to this day performing well in the ongoing competitions on software verification. However, the analysis lacks the ability to track memory and memory operations accurately and is therefore not able to analyze memory safety properties of a program or analyse programs with a heap memory in general. Symbolic memory graphs (SMGs) are a successful approach to model and abstract memory in order to analyze memory-safety related issues in programs.

This thesis extends the domain of concrete-value analysis with SMGs and refines the resulting software analysis using a CEGAR approach. Besides tracking the complete memory of a program closely enough to analyze memory-safety, concrete values in memory structures are also tracked by SMGs. CEGAR is used to track only necessary program variables, avoiding the problem of state-space-explosion in many cases. Furthermore, this thesis focuses on abstraction with linked lists using the technique of SMGs. The abstraction of linked lists not only reduces the resources needed by the analysis of lists, but also enables the analysis to verify programs with large lists. CEGAR is again used to improve upon this by tracking only values necessary to prove or refute a violation. Also, the type of the list used to abstract lists is refined, again based on the needs of the analysis. The resulting analysis is capable of tracking all the memory of a program and analyse programs based on it.

The analysis is implemented in the CPACHECKER framework, and is subsequently tested on a large set of C programs taken from the benchmark repository of the Competition on Software Verification (SV-COMP). The evaluation highlights that there are problems with respect to the combination of SMGs and the value analysis, but demonstrates altogether an overall solid performance.

Zusammenfassung

Concrete-Value Analysis ist eine Technik zur Verifikation von Software, die konkret bekannte Werte zur Analyse von Programmen nutzt. Die Technik ist unabhängig von externen Solvern und die Kombination der Analyse mit den bekannten Verfahren CEGAR und Interpolation verbesserte die Leistung. Trotz der Einfachheit der Value Analyse, erzielt sie nach wie vor gute Ergebnisse in der International Competition on Software Verification. Der Analyse fehlt jedoch die Möglichkeit Speicher und Speicheroperationen genau abzubilden, was dazu führt, dass dies auch nicht analysiert werden kann. Symbolic Memory Graphs (SMGs) sind eine bereits erfolgreich eingesetzte Technik, Speicher und Speichernutzung zu modellieren und dabei Speicherprobleme zu finden.

Diese Arbeit erweitert die Domäne der Value Analyse durch SMGs und verfeinert die Softwareanalyse durch einen CEGAR Ansatz. Neben der Nutzung von SMGs um Speicher genau genug abzubilden, mit dem Ziel diesen zu analysieren, werden auch bekannte, konkrete Werte darin gespeichert. Der CEGAR Ansatz wird genutzt, um ausschließlich Programm-Variablen abzubilden, die nötigt sind für die Analyse. Dies reduziert zum Beispiel Fälle, in denen die Anzahl der möglichen Zustände explodiert. Weiterhin liegt der Fokus dieser Arbeit auf der Abstraktion von verketteten Listen durch SMGs. Diese Abstraktion reduziert nicht nur die zur Analyse benötigten Ressourcen, sondern macht große Listen überhaupt erst analysierbar. Dies wird durch einen auf CEGAR basierenden Ansatz unterstützt, der ausschließlich die konkreten Werte idendifiziert, die für die Analyse des Programms nötig sind. Basierend auf demselben CEGAR Ansatz wird auch die Form der abstrahierten Listen verbessert.

Die im CPACHECKER Framework implementierte Analyse, wird anschließend mithilfe einer großen Teilmenge der Benchmarks der internationalen Competition on Software Verification (SV-COMP) für C-Programme getestet. Die Ergebnisse der Tests zeigen Probleme mit der Kombination von SMGs und Value Analyse auf, aber auch insgesamt solide Ergebnisse.

Contents

List of Abbreviations

CEGAR	Counterexample-Guided Abstraction Refinement
CFA	Control Flow Automaton
CPA	Configurable Program Analysis
DLL	Doubly Linked List
DLS	Doubly Linked List Segment
JVM	Java Virtual Machine
SLL	Singly Linked List
SLS	Singly Linked List Segment
SMG	Symbolic Memory Graph
SMT	Satisfiability Modulo Theories
SV	Software Verification
SV-COMP	International Competition on Software Verification

List of Figures

List of Algorithms

List of Tables

1 Introduction

Finding bugs in programs is still one of the greatest issues facing software development. There have been many documented cases of bugs that caused major problems. For example, the start of the first Ariane 5 rocket ended in an explosive disaster caused by an unexpected integer overflow. Since testing on its own can find bugs, but never proove their nonexistence, verifying programs using formal verification techniques got more popular in recent years. One of those techniques is model checking, which tries to verify that a given specification holds for a program using mathematical models. Those specifications can, for example, use assertions to specify a property that has to hold at a specific program location. Another possibility is that a property has to hold throughout the program execution, such as memory safety related issues like no invalid pointer dereferences being possible. Model checking algorithms[9, 30, 4] often times use external SAT or SMT solvers to reason about the feasibility of the built model of a program with respect to the given specification. While those algorithms tend to perform well on a wide variety of tasks[4, 7], they tend to track information symbolically and as a result the solvers can be overpowered by the computational complexity of the model. This can be improved by using techniques like *counterexample-guided abstraction refinement* (CEGAR)[26]. However, for verification tasks where bit-precise reasoning is necessary, for example the aforementioned memory safety overflow issue, abstraction techniques are often times too coarse. Tracking only concrete information can help improving this issue for tracked information and when used exclusively can even make the use of external solvers unnecessary. This form of analysis is called *explicit-value analysis*[16], or simply referred to as *value analysis*. A basic version of this analysis for the C programming language could e.g. be tracking only stack-based memory assignments, meaning only global and local variables. The retained concrete information can then easily be modeled for bit-precise reasoning, without much overhead. Another benefit of this technique is that CEGAR can be applied as well. The advantages of a value analysis however come with the disadvantage of imprecision for non-deterministic values, such as external program input or program variables that are not initialized. One more advantage, however, is that the domain of the value analysis can be extended from just tracking concrete assignments of stack variables, to domains that are more expressive.

Figure 1 shows an example written in the C programming language, which a value analysis based on stack memory alone cannot show that it is unsafe. The program first requests heap memory, and afterwards writes consecutive values into it, incrementing the memory address each time after a write. The subsequent assertion fails, as the accessed memory exceeds the boundaries of the requested memory. Depending on the options and flags used to compile the program, the execution may crash or succeed, thus opening the possibility of external exploits. An obvious route to extend the value domain would be the extension of the modeled memory from stack based memory to a full memory model that includes the heap. The modeling of memory for software verification is however difficult. Consider linked lists, which

```
#include <assert.h>

int main() {
  int *array = malloc(sizeof(int) * 10);

  for (int i = 0; i < 10; i++) {
    *array = i;
     array++;
  }

  assert(*array == 9);
}
```

Figure 1: Example program 1, showing usage of a heap array with pointer arithmetic leading to an invalid dereference.

can be cyclic, and arrays, whose length can be dependent on variables with values that are only known on runtime. One approach for this is based on graphs, which has the following of advantages: First, the ability to abstract memory by merging similar (sub-)graphs, reducing their size from a potentially infinite graph to a manageable size, while still retaining a lot of information. Then, the (sub-)graphs properties can be analyzed by a so called *shape analysis*[34, 25, 31] to find specification violations in regard to memory safety. Third, memory graph based analyses can be improved similarly to other algorithms by using CEGAR[11] to track only necessary information. Lastly, based on the type of memory being tracked, the shape of the graphs can be refined to fit the needs of the analysis[14].

One proven possibility for such memory graphs would be the so-called *symbolic memory graphs*[27] (SMGs). Symbolic memory graphs have been used to verify memory safety fast, successful, and without background solver[28, 5, 32, 4, 7].

The goal of this thesis is the incorporation of SMGs as domain into a concrete-value analysis. The memory graphs used to track heap memory are abstracted and the resulting shapes are refined[14]. Due to the scope of by this thesis, heap abstraction and refinement will be demonstrated using linked lists only. The analysis is then subsequently improved by the incorporation of CEGAR. First, CEGAR is used to track only program variables needed for the current verification[16, 17, 18]. Then, the same approach is used for heap memory, in that the analysis tracks only mandatory information[11]. This is implemented in the CPACHECKERframework, written in Java, and focuses on analyzing programs written in C. The aim of the analysis is to track concrete program variables, including pointers and heap memory, and then use this information to reason about potential specification violations. The reasoning needed is provided by the value analysis, which provides the necessary information about concrete values and determines possible paths throughout the program. Meanwhile the shape analysis provides memory safety information while exploring the state-space of the program. The memory safety properties that can be analyzed are, invalid memory access, invalid freeing of memory and memory leaks. The SMG domain therefore aims for the resulting analysis to be more precise in heap related tasks, especially regarding linked lists.

The structure of this thesis is as follows. It first gives an overview about related projects, with the aim of analyzing memory safety in chapter 2. Chapter 3 then describes the necessary background knowledge for the used concepts. Following that, in chapter 4, the new concrete-value analysis with SMGs and shape analysis, as well as improvements such as CEGAR are discussed. Then the implementation in the CPAchecker framework itself is described in chapter 5. This includes design choices and general layout as well as implementation and also design problems. Following this, the implementation is evaluated in chapter 6. This chapter aims at showing general functionality by verifying tasks from the *International Competition on Software Verification* (SV-COMP)[4].

The results are then compared to the state-of-the-art value analysis in CPAchecker, as well as the state-of-the-art software verification tool PredatorHP. Finally, the analysis is discussed in regards to its results and future work directions in chapter 7.

2 Related Work

2.1 PredatorHP

PredatorHP[1] [28, 32] is a software verifier for the C programming language, specialized in memory related analysis. While it was originally based on separation logic, it is now purely based on SMGs and shape analysis, with a specialty in low level list abstraction. It can check properties related to pointer arithmetic and use, memory reinterpretation, address alignment, over/underflow errors and more. PredatorHP does not use any external SAT or SMT solvers and is purely based on an internal symbolic execution procedure. SV-COMP participations have shown that the tool is comparably fast, and scores highly in the MemorySafety category. The SMGs used by the concrete-value analysis are based on the SMGs used by PredatorHP. The downside of PredatorHP is that it only verifies memory safety or heap manipulation related specifications, compared to the goal of this work which aims to combine memory safety, heap manipulation and general reachability.

2.2 Forester

Another software verification tool specialized in memory analysis is Forester [2][29]. It too uses shape analysis, but on the basis of tree automata, which combine separation logics separation and scaling abilities with abstract regular tree model checking. As a result, Forester can analyze heap and especially list structures of different flavors for memory related issues. The used automata are realized without any external solvers, similar to PredatorHP or concrete-value analysis. Forester also competed in past SV-COMPs with moderate success[6]. While Forester approach is distinct from the chosen SMGs in this work, they both are partially based on separation logic.

2.3 SLAyer and SpaceInvader

SLAyer [3] and SpaceInvader [2] both use separation logic to perform shape analysis on heap memory. SLAyer used the Z3 SMT solver to aide its analysis, while SpaceInvader does not use any external prover or solver. Both can analyze pointer related issues such as dangling pointers, basic heap and have a limited list support. However, they both suffer from the same limits in that they do not allow outside pointers to specific list elements besides the first, or pointers from inside a list to the rest of the heap. Furthermore it is hard to compare them in a more recent context, as none of the tools did compete in the SV-COMP, nor are they actively maintained. The separation logic used by both is related to the SMGs used in PredatorHP and this thesis, as SMGs were developed partially out of separation logic[27].

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`, 1.9.2022
[2] `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`, 1.9.2022

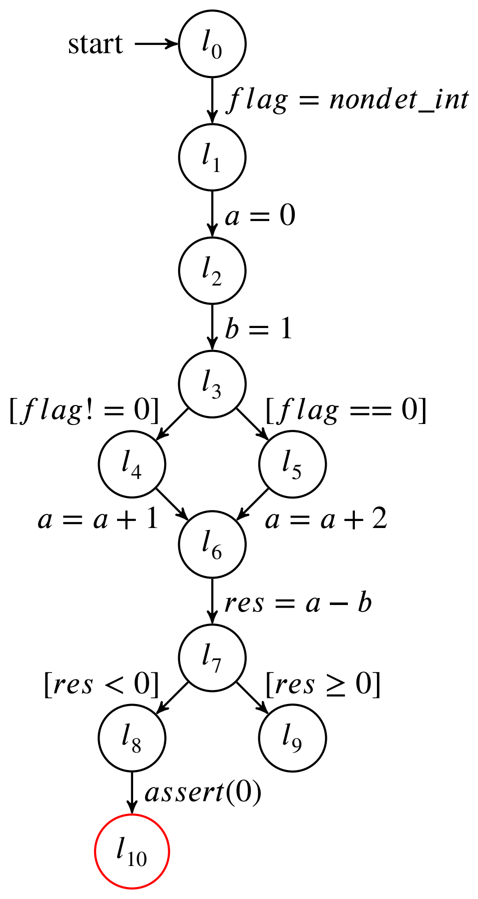
2.4 Symbiotic

SYMBIOTIC[3] [23, 24] is an actively developed, open source software analysis framework, that has been a top contender in multiple SV-COMP categories for several years. Upon those the MemorySafety category, as well as heap related subcategories. The open source project uses pointer analysis, instrumentation, static program slicing and symbolic execution for memory related tasks. SYMBIOTICuses a portfolio of external programs, for example the symbolic executors KLEE[21] or Slowbeast[4], but also complete verifiers, for example CPACHECKER or PREDATORHP. The approach of SYMBIOTIC 9 , to verify memory related issues, is to use loop folding combined with backwards symbolic execution supported by Slowbeast.

[3]`http://staticafi.github.io/symbiotic/`, 1.9.2022
[4]`https://gitlab.com/mchalupa/slowbeast`, 1.9.2022

```
#include <assert.h>

int main() {
  int flag = nondet_int();
  int a = 0;
  int b = 1;

  if (flag) {
    a = a + 1;
  } else {
    a = a + 2;
  }

  int res = a - b;
  if (res < 0) {
    assert(0);
  }
}
```

(a) Program 2



(b) CFA for program 2

Figure 2: Example program 2 and its CFA.

3 Background

To understand the following thesis fully, multiple concepts have to be introduced. Firstly, the idea of *control flow automatons* (CFA) is needed to understand the subsequent *configurable program analysis* (CPA) and its components. Then the CPACHECKER framework is described, in which the new analysis of this paper is implemented. Further, the current state-of-the-art value analysis, including CEGAR, value-domain interpolation and refinement selection, is presented. Finally, symbolic memory graphs (SMGs), including list abstraction are discussed in conjunction with shape refinement. It is to note that these background information are presented in a manner that allows the reader to understand the later used concepts and are not meant to give a full overview.

3.1 Control Flow Automaton (CFA)

A control flow automaton (CFA) is a triple *(L, l_0, G)* that represents a directed, potentially circular graph. The triple consists of L, the the set of program locations, $l_0 \in L$, the start location and G, representing all possible transitions between the locations in L. A transition consists of edges $g = (l, op, l')\in G$, with operations $op \in Ops$ and $l \in L$ and $l' \in L$. Operations can be thought of as the execution of a program, e.g., assumptions like **res < 0**, declarations like **int a = 0**, or statements like **a = a + 1**. Therefore edges $g \in G$ represent transitions from one location $l \in L$ to another $l' \in L$. Figure 2 shows an example program with its CFA. It can be seen how each location, represented as a node, is transitioned into another via edges. Also, the CFA branches for conditional statements into every possible path, as it does not check the feasibility of paths. Examples would be for example assumptions or loop-heads. By inlining a function calls CFA at the position of the call, we can create a single CFA for any program. This resulting CFA, modeling an entire program, can then be traversed by an analysis like a value analysis using the following configurable program analysis concept.

3.2 Configurable Program Analysis (CPA)

A configurable program analysis (CPA)[12] is a way of formalizing software analysis algorithms and is made up of 4 components. The abstract domain D, a transfer relation $\rightsquigarrow$, a merge operator $merge$ and a stop operator $stop$. Consisting of a four-tuple, a CPA C is formally written as $(D, \rightsquigarrow, merge, stop)$ and runs on a CFA. This CPA definition can be extended by an dynamic precision adjustment to the CPA+[13] algorithm. CPA+ $(D, \Pi, \rightsquigarrow, merge, stop)$, an extension of the CPA algorithm, adds a set of precisions Π, and a precision adjustment operator to its definition. In the following work, we assume all CPAs to be a CPA+, and just assume that if there is no precision or precision adjustment operator mentioned, that the precision is empty and that the precision adjustment operator does not change the precision. If the precision is not mentioned in definitions, it can be assumed that it is not necessary to describe that component of the analysis.

3.2.1 Lattice and Semi-Lattice

As the definition of the abstract domain depends on the definition of a *lattice* and *semi-lattice*, those need to be defined first. A lattice is based on a partial order $\sqsubseteq$ over a set E such that $\sqsubseteq \subseteq E \times E$. The partial order $\sqsubseteq$ is reflexiv, transitive and antisymmetric. The smallest element $e \in E$ satisfying $e' \sqsubseteq e$ for all $e' \in M$ is defined as the least upper bound of a set M with $M \subseteq E$. In contrast, the biggest element $e \in E$ satisfying $e \sqsubseteq e'$ for all $e' \in M$ is defined as the greatst lower bound for a set $M \subseteq E$. A semi-lattice $(E, \sqsubseteq, \sqcup, \top)$ exists, if a least upper bound for every possible subset of E can be found in E. In a semi-lattice, $\sqcup$ returns the least upper bound of the two elements $e \in E$ and $e' \in E$. The bottom element $\bot$ is the greatest lower bound of a lattice, while the least upper bound of the set E defines the top element $\top$.

3.2.2 Abstract Domain

Now the abstract domain can be defined over a set of concrete states C as $D = (C, \mathcal{E}, [\![\cdot]\!])$, with the just defined semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ and a concretization function $[\![\cdot]\!] : E \mapsto 2^{C}$. The concretization function associates an abstract state $e \in E$ to a set of concrete states

$C_e \in C$. Note that the set of concrete states C_e may be infinite. Multiple concrete states may be over-approximated via the chosen abstraction of the analysis to be represented in a single abstract state.

3.2.3 Precision

A precision π is used to determine the coarseness of an analysis and can be used to, for example, specify the tracked program variables. The precisions of the abstract domain are specified by the set of precisions Π. For different abstract states $e \in E$, the CPA tracks different precisions $\pi \in \Pi$. An abstract state with its precisions is denoted as $(e,\pi) \in E \times \Pi$.

3.2.4 Transfer Relation

The transfer relation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ returns possible successors $e' \in E$ for each given abstract state $e \in E$. These successors however depend on the precision $\pi \in \Pi$. Each transfer from abstract state $e \in E$ to a successor element e' is labeled with a CFA edge $g \in G$. For a tuple (e, g, e´, π) $\in \rightsquigarrow$ and e $\rightsquigarrow$ (e´, π), we write e $\overset{g}{\rightsquigarrow}$ (e´, π)).

3.2.5 Merge Operator

Two abstract states can be combined into a single, new abstract state using the operator merge $merge : E \times E \mapsto E$. The results of $merge\ (e,e')$ can vary from returning e again, to $\top$. It is to note that the merge operator does weaken the second state and is therefore not commutative. The two merge operators important for this thesis are $merge_{sep}(e,e') = e$, which does not merge at all, and $merge_{join}\ (e,e') = e \sqcup e'$, with $\sqcup$ being the join operator of the lattice.

3.2.6 Stop Operator

The stop operator $stop : E \times 2^E \mapsto \mathbb{B}$ checks whether an abstract state $e \in E$, given as the first parameter, is covered by the set of abstract states R given as the second parameter. If the abstract state is covered, it returns true to indicate that the analysis can stop for this specific abstract state. In the case that the stop operator does not return true, the analysis continues with the next abstract state. A abstract state is considered covered if, for example, the set of abstract states subsume the current state. The most important stop operator for this thesis is the stop sep operator $stop_{sep}\ (e,R) = (\exists e' \in R : e \sqsubseteq e')$.

3.2.7 Precision Adjustment Operator

Using the precision adjustment operator $prec : E{\times}\Pi{\times}2^{E\times\Pi}$, a new abstract state and precision is returned for the entered abstract state, its precision and set of abstract states with precisions. The precision adjustment operator is applied after the transfer relation $\rightsquigarrow$ and may change the precision or widen the abstract state.

Algorithm 1: CPA algorithm[13].

```
Input: A CPA ($D$, $\Pi$, $\rightsquigarrow$, $merge$, $stop$, $prec$), an initial abstract state $e_0 \in E$ with
       precision $\pi_0 \in \Pi$, where $E$ denotes the set of elements of the semi-lattice of $D$
Output: A set of reachable abstract states
Variables: A set $reached$ of elements of $E \times \Pi$, a set $waitlist$ of elements of $E \times \Pi$
1  $waitlist := \{(e_0, \pi_0)\}$;
2  $reached := \{(e_0, \pi_0)\}$;
3  while $waitlist \neq \emptyset$ do
4      pop $(e,\pi)$ from waitlist;
5      // Adjust the precision
6      $(\hat{e}, \hat{\pi}) = prec(e,\pi,reached)$;
7      for each $e'$ with $\hat{e} \rightsquigarrow (e', \hat{\pi})$ do
8          for each $(e'', \pi'') \in reached$ do
9              // Combine with existing abstract state
10             $e_{new}$ := merge$(e', e'', \hat{\pi})$;
11             if $e_{new} \neq e''$ then
12                 $waitlist := (waitlist \cup \{(e_{new}, \hat{\pi})\} \setminus \{(e'', \pi'')\})$;
13                 $reached := (reached \cup \{(e_{new}, \hat{\pi})\} \setminus \{(e'', \pi'')\})$;
14         if $\neg stop(e', \{e \mid (e,\cdot) \in reached\}, \hat{\pi})$ then
15             $waitlist := waitlist \cup \{(e', \hat{\pi})\}$;
16             $reached := reached \cup \{(e', \hat{\pi})\}$;
17 return $\{e \mid (e,\cdot) \in reached\}$;
```

3.3 Composite CPA

A composite CPA[12] can simply be seen as the combination of multiple different CPAs into a single CPA. Instead of redefining the CPA operators ($\rightsquigarrow$, *merge* and *stop*) for each CPA combination, they are extended by two composite operators, the strengthening operator $\downarrow$ and the compare relation $\preceq$. The strengthening operator $\downarrow$ is used to compute a single successor state for two given abstract successor states. This can result in a stronger successor state, compared to a simple product of the two abstract states. In contrast, the compare relation $\preceq$ is used to compare elements of different lattices. Using these, any combination of CPAs is possible, for example a concrete-value analysis tracking only global and local program variables could be combined with shape analysis to track information about the heap.

3.4 CPA Algorithm

The CPA algorithm[12, 13], given as algorithm 1, starts from an initial state e_0 and computes a set of abstract states reachable from e_0. Operations are based on a set *reached* and a set *waitlist*, each containing abstract states and their precision. A state is then taken from the waitlist and given to the transfer relation $\rightsquigarrow$, which computes its successors. The resulting abstract states are then given to the precision adjustment operator *prec*. If one of the successor states are found to be a state that is considered as a target by the used specification, the analysis

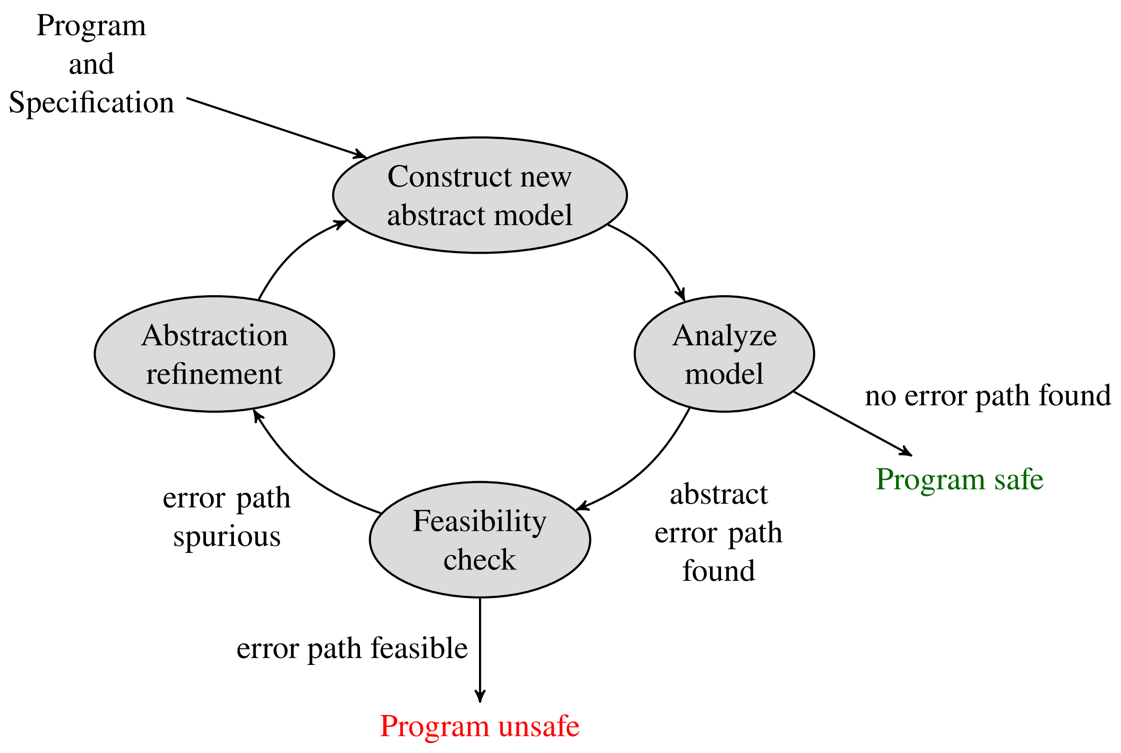


Figure 3: The abstract procedural CEGAR algorithm. As depicted by Bunk[20], slightly modified.

returns all found abstract states without continuing. Otherwise, the algorithm applies the merge operator on the newly found abstract states and each member of the reached set. If a new state is produces by the merge operator, it is put into both sets, replacing the old abstract state from *reached* just used to merge. Lastly the stop operator decides whether or not a found abstract successor is already covered, or should be put into the waitlist to be analyzed further. This process is then repeated until the waitlist is empty, finally returning the set of reached states.

3.5 Counterexample-Guided Abstraction Refinement (CEGAR)

Counterexample-guided abstraction refinement (CEGAR) is a well known technique to improve the performance of software analysis algorithms. Generally the algorithm can be summed up using figure 3. First a new abstract model is constructed for the analyzed program and specification. Then the model is analyzed and if no error could be found the program is regarded as safe. Otherwise, the abstract error path found is checked for feasibility, returning unsafe for feasible error paths. If the abstract error path is spurious however, the abstraction is refined before the process is started again from the beginning.
In the context of CPAs this is achieved by modifying the precision of the analysis continuously until a program can be proven safe or unsafe.
Algorithm 2 shows a CEGAR algorithm based on the CPA algorithm. In this CEGAR algorithm used by CPAchecker and this thesis, an initially empty precision is used to find any

Algorithm 2: CEGAR algorithm modified to be usable with CPAs, cf. [16]

```
Input: A CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, merge, stop, prec)$, an initial abstract state $e_0 \in E$ with a
       precision $\pi_0 \in \Pi$
Output: the verification result true, or false (with counterexample)
Variables: a set $reached \subseteq E \times \Pi$, a set $waitlist \subseteq E \times \Pi$, an error path
       $\sigma = \langle (op_0, l_0), ..., (op_n, l_n) \rangle$
reached := $\{(e_0, \pi_0)\}$
waitlist := $\{(e_0, \pi_0)\}$
$\pi := \pi_0$
while true do
    (reached,waitlist) := CPA($\mathbb{D}$,reached,waitlist)
    if $waitlist = \emptyset$ then
        // no error path found: verdict is true
        return true
    $\sigma := ExtractErrorPath(reached)$
    if $IsFeasible(\sigma)$ then
        // error path is feasible: verdict false, report bug
        return false
    else
        // error path is infeasible: restart with refined precision
        $\pi := \pi \cup Refine(\sigma)$
        reached := $\{(e_0, \pi)\}$
        waitlist := $\{(e_0, \pi)\}$
```

specification violation present in the analyzed program. If the analysis concludes that the just run CPA can not find any violations, the program is safe. If however an abstract violation path is found, a feasibility check is used to check if the path is reachable using the current precision. This feasibility check tries to verify that a concrete error path exists and prove the program unsafe. If however the error path is infeasible, the used precision is refined and the process is repeated. A example for this, in regards to the following value analysis, can be that the analysis tracks only variables used in the analysis that are also tracked by the precision. Such an analysis would tracks no variables in the beginning. Each time the program can not be proven safe or unsafe, the precision is extended by *Refine* to include more program variables. Doing this, the analysis tracks only variables that are needed to prove the final verdict, but ignores unneeded variables, potentially reducing the work the actual analysis has to do.

3.6 Verification Witnesses

Verification witnesses[8] attest the result of a software verification. Assuming that an analysis found a result and wants to communicate it, beyond just the verdict, witnesses represent a standardized certificate for either the correctness of the analyzed program with respect to the specification, or a violation of the specification. A violation witness has to document the cause of the specification violation through for example a counterexample. This counterexample can as an example consist of the concrete variable and also heap assignments leading to the specification violation at key program locations. In contrast, correctness witnesses have to provide evidence that a specification violation can not occur during program execution. Witnesses can be checked for correctness themselves using a witness validator. But they can also easily be used to improve the results of software analysis algorithms. It is for example possible to use a counterexample trace as a starting precision of a different analysis to the one that provided the witness.

3.7 CPAchecker

CPACHECKER[15] is a framework for configurable software verification and a proven[4, 7] formal verification tool. Developed at the Software and Computational Systems Lab (SoSy-Lab) at the LMU Munich, CPACHECKER continues to rank among the top software verifiers in the international competition on software verification[4]. CPACHECKER provides each analysis with an CFA generated from a given program to be analyzed. Then, a CPA takes that CFA to analyze it, returning the verdict in the end. Since CPACHECKER was built around the concept of configurable problem analysis and also CEGAR from its inception, it is possible to add new CPAs, with or without CEGAR, with relative ease. Additionally, it possesses multiple helpful features, such as support for composite CPAs, violation and correctness witnesses. The analysis described in this work is integrated into CPACHECKER as a new CPA, and subsequently tested using BENCHEXEC[19], a reliable benchmarking tool that posses compatibility for CPACHECKER.

3.8 Value Analysis with CEGAR and Interpolation

The analysis presented in this work is based upon concrete-value analysis, mainly because of it was found to be fast compared to other algorithms and does not need a background solver. Also, it can solve a large amount of tasks, compared to its simplicity, even if it can not compete with advanced analyses such as symbolic execution. The following sections describe value analysis[16], as well as CEGAR and interpolation[16, 17, 18] for the value domain as it is used in CPACHECKER.

3.8.1 Concrete-Value Analysis

As already mentioned in the introduction, a concrete-value analysis uses concrete value information to reason about the correctness of a program. To achieve this, the analysis essentially tries to find a way to violate the specification, which is in most cases a path through the program to a violating program line. Only if none can be found a program is considered save. Each program variable x in the set of program variables X is derived from the possible operations $op \in Ops$ of the program. A program variable x can be assigned a value, e.g. from the set $\mathbb{Z}$ for integer values. The abstract states of a value analysis with value domain are represented as abstract variable assignments υ, with the *value domain* defined as the domain used by the value analysis tracking only concrete program variables. Since the analysis might not be able to determine a concrete value, for example due to an external function call or a unknown function argument, not all variable assignments can be represented as abstract variable assignments. For this reason, abstract variable assignments are modeled as partial functions. The definition range for a partial function f is defined as $\text{def}(f) = \{ x \mid \exists y : f(x) = y\}$, with $y \in Y$ being a new definition range similar to that of X. The strongest postcondition $f' = \widehat{SP}_{op}(f)$ is defined as the application of an operation $op \in Ops$ on an abstract variable assignment f, with the result being a new abstract variable assignment f'.

As a result, the analysis tracks known concrete stack memory variables, including arrays and structs. Since the value analysis does not use a background solver to ease its computations, only equalities and non-equalities are evaluated using the known concrete values. Using only equalities, most branching program statements, for example if statements or loops, can still be evaluated. If values can not be determined, for example through previous unknown values or nondeterministic values through outside function input, a unknown value is used. These unknowns can then be assumed in conditional statements, as they could equal a value. Using this assumption, all possible resulting branches are explored further. If the conditional statement is an equality assumption, unknown values can be assigned a concrete value. An example would be the statement *if (x == 2)* for an unknown value *x*. One assumption is that the unknown value is equal to the checked value, in which case the unknown values value is changed to the value of the concrete value, in this case 2. In the other case the unknown values is unchanged, but the conditional statements result is reversed.

Taking a look at program 2 as an example, we see that there are three values: *flag*, *a* and *b*. The values for *a* and *b* are known, while *flag* is unknown, as it is determined by user input. The if statement in line 7 can be broken down into 2 assumptions for the unknown value *flag*. One assumption is that *flag* equals 0, and the other is that it does not, which means that it equals 1, as the variable *flag* is only used with boolean values throughout the program. From this point onwards two states exist, one for *flag* assumed to be the concrete value 1, the other for *flag* being 0. As a result, both paths following the if statement are evaluated, one for each state. When the CFA meets again in l_5, the two states have diverged, as in one *flag* equals 1 and *a* equals 2 and in the other *flag* equals 0 while *a* equals 3. As a result, both branches have different concrete values for the variable *res* after l_6 and also evaluate the following if statement differently. The branch that originally assumed *flag* to be 0, and as a result calculated *res* to be -1, violates the assert statement, proving program 1 unsafe.
Using the CFA locations l ∈ L and used operations op ∈ Ops, we can define a path $\varphi = \langle (op_0, l_0), ..., (op_n, l_n) \rangle$ from the beginning of the program to the property violation. This violation path can then be used together with the concrete variables at each location to build a violation witness.

```
#include <assert.h>

int main(int x) {

  if (x < 0) {
    if (x == 0) {
      assert(0);
    }
  }
}
```

Figure 4: Example program 3, showcasing a safe program that the value analysis determines as unsafe.

A extended version of tracking unknown values is the usage of symbolic values that track operations performed on them. Using this extended unknown value tracking, (non-)equality information of symbolic values can be checked. This comes at a higher cost, as the values themselves have to be saved and computed. Also, the domain used by the value analysis can easily be extended with more advanced techniques. The value analysis implemented in CPAchecker for example uses a composite CPA with pointer tracking capabilities to extend its domain.

Another valuable insight is, that the value analysis tends to over-approximate values through the assumption behavior and therefore tends to not prove programs incorrect that are correct. An example of this behavior is provided in figure 4. The analysis would assume $x < 0$ to be feasible, but can not assume a concrete value. Then $x == 0$ would also be assumed to be feasible, leading to an invalid violation. This problem however can be negated easily, as a violation witness can be produced and checked by a more sophisticated analysis, ruling it out.

3.8.2 Value Analysis with CEGAR and Interpolation

Basic value analysis tends to suffer from state-space explosion for large tasks, or even smaller tasks with difficult to find specification violations but a lot of branching. An example of this is provided in figure 5. While only the variable *flag* is relevant to the possible assertion violation, the just defined analysis would repeat the while loop endlessly. The reasons is that the if statement inside the loop depends on two unknown values in *result* and *x*. The value of *result* is assumed to be equal to 0 in one state and not equal to *0* in another each loop. While the state with the variable *result* equaling *0* leaves the loop, the other state unrolls another loop iteration. Similarly, the assumption for *x* results in the same problem. This leads to endless loop unrolls, as the assumption statement inside the loop can never be evaluated into concrete values for the next iteration of the loop. And since the assumptions that can be made to leave the loop can't prove a violation, the analysis would continue until it is either stopped or it runs out of resources.

The most basic method to avoid this issue is counterexample guided abstraction refinement

```
 1 #include <assert.h>
 2
 3 int main(int x) {
 4   int flag = 0, ticks = 0;
 5   int result;
 6
 7   while(1) {
 8     ticks = ticks + 1;
 9     result = nondet();
10
11     if (result == 0 || ticks > x) {
12       break;
13     }
14   }
15
16   if (flag > 0) {
17     assert(0);
18   }
19 }
```

Figure 5: Example program 4[16], showcasing state space explosion for plain value analysis.

(CEGAR). CEGAR uses a precision, in this case a value precision π that is a set of program variables that are tracked. The precision adjustment operator *prec* makes sure that no program variable is assigned a value if the variable is not tracked by π. The CEGAR algorithm starts the analysis with an empty precision, tracking no program variables. This will allow the analysis to reach any specification violation reachable in the CFA, for example line 18 in figure 5. Once a violation is found, the abstract error path $\sigma = \langle (op_0, l_0), ..., (op_n, l_e) \rangle$ is checked for feasibility. This is done using a feasibility check that computes the strongest postcondition $\widehat{SP}_{\gamma\sigma}(\top)$ with the current precision π, which is empty, for all locations $\mathrm{l} \in \mathrm{L}$ on the path σ. $\widehat{SP}_{\gamma\sigma}(\top)$ can be thought of as executing the analysis along the abstract error path. The feasibility check confirms a counterexample only if its execution reaches the target state. If it does not reach the target state, e.g. through a contradiction $\widehat{SP}_{\gamma\sigma}(\top) = \bot$, the precision has to be refined. A new precision has to be found that tracks the program variables relevant for the violation. To do this, value domain interpolation can be used.

3.8.3 Value Domain Interpolation

Since no background solver is used, value domain interpolation is not based on Craig-interpolation, as predicate interpolation usually is, but on constraint sequences. Value domain interpolation uses two constraint sequences, γ^- and γ^+, and returns an interpolant Γ. The constraint sequences consists of operations op such that $\gamma^- = \{op_1, ..., op_m\}$ and $\gamma^+ = \{op_1, ..., op_n\}$ and their conjunction being a path to an error location. The interpolant is based on the notion that the strongest postcondition of the conjunction of γ^- and γ^+ is contradicting. Interpolant Γ removes all variable assignments not needed to prove the contradiction, returning only the needed assignments. The algorithm removes iteratively abstract variable assignments that

Algorithm 3: *Interpolate*(γ^-, γ^+) algorithm[16].

```
Input: two constraint sequences $\gamma^-$ and $\gamma^+$, with $\widehat{SP}_{\gamma^- \wedge \gamma^+}(\top) = \bot$
Output: a constraint sequence $\Gamma$, which is an interpolant for $\gamma^- and \gamma^+$
Variables: an abstract variable assignment $\upsilon$
1 $\upsilon := \widehat{SP}_{\gamma^-}(\top)$
2 foreach $x \in def(\upsilon)$ do
3     if $\widehat{SP}_{\gamma^+}(\upsilon_{|def(\upsilon)\setminus\{x\}}) = \bot$ then
4         $\upsilon := \upsilon_{|def(\upsilon)\setminus\{x\}}$ // x is not relevant and should not occur in the interpolant
5 $\Gamma := \langle\rangle$ // start assembling the interpolating constraint sequence
6 foreach $x \in def(\upsilon)$ do
7     $\Gamma := \Gamma \wedge \langle [x = \upsilon(x)] \rangle$ // construct an assume constraint for x
8 return $\Gamma$
```

are a result of $\widehat{SP}_{\gamma^-}(\top)$ from $\widehat{SP}_{\gamma^+}(\top)$, which can have two effects. Either it results again in a contradiction $\widehat{SP}_{\gamma^+}(\upsilon_{|def(\upsilon)\setminus\{x\}}) = \bot$, then the variable declaration was irrelevant and can be removed freely, or no contradiction can be found anymore. This means that the variable assignment that was just removed is needed for the contradiction and is therefore also necessary in the interpolant. The algorithm *Interpolate* tries this with every abstract variable assignment found through applying $\widehat{SP}_{\gamma^-}(\top)$. In the end, all necessary variable assignments are combined into the single interpolant Γ. Note that the exact definition for value domain interpolation, as well as the proof[16] that algorithm *Interpolate* fulfills it, are omitted from this thesis due to its scope. Further, the *Interpolate* algorithm has a runtime of $O((m + n)^3)$, because of the removal of the unnecessary variable assignments in the first loop, with m and n being the bounds of the used constraint sequences. This is acceptable however, as the algorithm has the potential to reduce the runtime later on in the analysis. The reason for this is that removed variable assignments can not longer lead to a state-space explosion as it does in figure 5. Furthermore, several heuristics to improve the interpolation procedure have been explored[16]. But since they only speed up the interpolation and don't change the general behavior of the algorithm, they are not discussed further.

3.8.4 Refinement of Precisions based on Value Domain Interpolation

The *Refine* algorithm uses interpolation on an infeasible error path to generate a new set of precisions Π for the program analyzed. This is achieved by splitting the entered infeasible error path at every program location into the two infeasible prefixes γ^- and γ^+. Those are then used by the just defined *Interpolate* to compute inductive interpolants that prove the remaining path infeasible. Inductive interpolants are needed in order to remove infeasible error paths from the analysis via the precision. An interpolation algorithm produces inductive interpolants[10], if each interpolant computed along the path ensures that it the remaining path does not become feasible. Since the interpolation as defined in *Interpolate* is only strong enough to eliminate infeasible error paths for a specific location, the application of *Interpolate*

Algorithm 4: *Refine* algorithm[16]

```
Input: an infeasible error path $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$
Output: precision $\Pi$
Variables: interpolating constraint sequence $\Gamma$
1 $\Gamma := \langle\rangle$
2 foreach $l \in L$ do
3     $\Pi(l) := \emptyset$
4 for $i := 1$ to $n - 1$ do
5     $\gamma^+ := \langle op_{i+1}, ..., op_n \rangle$
6     $\Gamma := Interpolate(\Gamma \wedge \langle op_i \rangle, \gamma^+)$ // inductive interpolation
7     // extract variables from variable assignment resulting from $\Gamma$
8     $\pi(l_i) := \{x | (x, z) \in \widehat{SP}_{\Gamma}(\top)\}$
9 return $\Pi$
```

```c
#include <assert.h>

void main() {
  int b = 0;
  int i = 0;
  for (int j = 0; j < nondet(); j++) {
    i++;
  }

  assert(i < 100000000);
  assert(b != 0);
}
```

Figure 6: Example program 5, that is unsafe based on two assertions that both fail.

along the path with respect to the previous precision fulfills the requirements for inductive interpolants. Precisions themselves are constructed out of the relevant assignments returned by *Interpolate*. A *Refine* procedure is used in the CEGAR algorithm of the value analysis.

This CEGAR and interpolation approach discussed in tends to perform only slightly better than the concrete-value analysis without CEGAR. For this reason, the algorithm has been improved over the years.

3.8.5 Improved CEGAR using Infeasible Sliced Prefixes

One such improvement is the usage of infeasible sliced prefixes. This technique is based on the notion that a infeasible path through a program can have more than one reason for its specification violation. In figure 6 for example, tracking variable *i* or *b* leads to an infeasible error path. The important takeaway is that tracking any one of the two variables would suffice to prove this. However, the variable *i* is a loop bound variable, and it is obvious that tracking it would either result in a lengthier analysis, or the need for a more complex loop analysis algorithm. Therefore, selecting the path only using the variable *b* is favorable.

To be able to select any infeasible sliced prefixes however, they first need to be extracted from

Algorithm 5: ExtractSlicedPrefixes(σ) algorithm[18]

```
Input: an infeasible path $\sigma = \langle (op_1, l_1), ..., (op_m, l_m) \rangle$
Output: a non-empty set $\Sigma = \{\sigma_1, ..., \sigma_n\}$ of infeasible sliced prefixes of $\sigma$
Variables: a path $\sigma_f$ that is always feasible
1  $\Sigma := \emptyset$
2  $\sigma_f := \langle\rangle$
3  foreach (op,l) $\in \sigma$ do
4      if $\widehat{SP}_{\sigma_f \wedge (op,l)}(\top) = \bot$ then
5          // add $\sigma_f \wedge (op, l)$ to set $\Sigma$ of infeasible sliced prefixes
6          $\Sigma := \Sigma \cup \{\sigma_f \wedge (op, l)\}$
7          // append no-op
8          $\sigma_f := \sigma_f \wedge ([true], l)$
9      else
10         // append original pair
11         $\sigma_f := \sigma_f \wedge (op, l)$
12 return $\Sigma$
```

our current infeasible path. This is done by the *ExtractSlicedPrefixes* algorithm depicted in algorithm 5. It takes the current infeasible error path and tries to build the set of all possible infeasible paths based on the original path. The decision about the feasibility is based on the strongest postcondition, similar to the feasibility check.

By iterating through the initial path for each location *l* and operation *op* pair, the algorithm can check which operations are necessary for the feasibility of the total path. If the path consisting of the current feasible path σ_f and the current operation is still feasible, the current operation is added to σ_f. In case that the path is no longer feasible $\widehat{SP}_{\gamma\sigma}(\top) = \bot$, a new infeasible sliced prefix is found and saved. Additionally, the current operation is replaced by a no-op in the feasible path, before continuing the algorithm. This algorithm per definition guaranteed to find at least one infeasible sliced error path, as the algorithm is started with one. The resulting sliced error paths can then be used individually in the *Refine*$^+$ algorithm as depicted in algorithm $Refine^+$.

Interpolation engines of SMT solvers tend to not be able to produce similar results. This is typically rooted in the fact that they do not give the user control about how the interpolants are produced and the multitude of used interpolation techniques behaving differently.

Now the question remains as to how to choose the best sliced prefixes, as the previous example program 5 clearly showed that the selection can have a huge influence on the analysis. Multiple non-trivial selection heuristics have been proposed, but this thesis will only cover two of them. The reason for this is that the combination of the two is currently regarded as the best heuristics available in CPAchecker. The first heuristic is *selection by domain-type score of path precision*, which inspects the type of variables used in the precision for refine-

Algorithm 6: *Refine*$^+$ algorithm[18]

```
Input: an infeasible error path $\sigma = \langle(op_1, l_1), ..., (op_n, l_n)\rangle$
Output: a precision $\pi \in \Pi$
Variables: a set $\Sigma$ of infeasible sliced prefixes of $\sigma$, a set $\tau$ of pairs of an infeasible
           sliced prefix and a precision
1 $\Sigma := ExtractSlicedPrefixes(\sigma)$
2 // compute refinement for each infeasible sliced prefix, using algorithm Refine
3 foreach $\varphi_j \in \Sigma$ do
4     $\tau := \tau \cup \{(\varphi_j, Refine(\varphi_j))\}$
5 // select a refinement based on the original path, the infeasible sliced prefix, and their
    respective precision
6 return $SelectRefinement(\sigma, \tau)$
```

ment. The types of used program variables are then labeled with a score. A smaller type by size or simpler types, can be associated with a smaller score. Booleans for example have a small score per default, but loop associated types have a larger score than their non-loop bound counterparts. Since loop counters are associated with loops, choosing a sliced error path with less scoring variables in this heuristic helps avoiding loops.

Selection by width of path precision is based on how long along the path precisions have to track information. This notion is called the width of a precision and is defined on the number of non-empty precisions of the distinct locations of a sliced infeasible error path. Choosing a path with a set of precisions that are empty, contributes to the reduction of the analyzed state-space. However, choosing small, non-empty precisions could lead to the selection of loop-bound variables, as their assignments tend to be located at the end of loop-exits.

All these heuristics and more can be combined in a composite heuristic, combining their characteristics based on the repeated application of them. A useful combination would for example be avoiding loops, while also choosing narrow precisions to reduce the amount of state-space-exploration through many different variables. Strengthening the low precision width heuristic with the low domain-type score heuristic would be one example. The chosen precisions would be loop counter prone but also tend to have lower levels of state space unfolding for the first heuristic, but loop counter based variables would be reduced by the second. Thus we can choose sliced error paths for our analysis more closely than using just one of those heuristics. Both, the analysis describes by this thesis and the value analysis use the just described heuristic. Many more heuristics and composite heuristics are possible of course.

3.9 Symbolic Memory Graphs (SMGs)

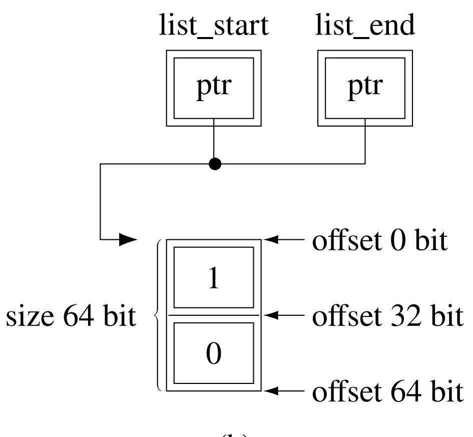
Symbolic memory graphs[27] are a more specialized version of informal memory graphs. The goal of SMGs is to represent memory as concrete as needed to verify memory safety issues. SMGs tend to perform well in regards to linked lists, which are abstractable, while retaining a lot of information about them. This section first describes SMGs in general, including read and write operations, before explaining how linked lists are abstracted. Lastly, the materialization of abstracted lists back to concrete list segments is discussed. Because of the complexity involved, the following sections describe SMGs only as much as needed to understand the rest of this thesis.

3.9.1 General Overview

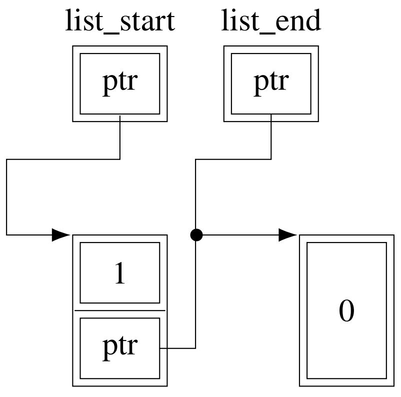
Symbolic Memory Graphs consists of objects and values. Objects can be seen as nodes with certain characteristics such as size and their type, which is either a *region* or a list segment. A region is a continues memory area, while linked lists consist of multiple regions. There is a special memory object null, which is invalid. Objects generally have a notion of validity that is used to remember if the memory is invalidated, e.g. through a memory freeing operation. Values are saved in objects using edges, called *has-value-edges*. Has-value-edges have an offset depending on their location in the memory and a size depending on their type. Values in SMGs are always symbolic, with the exception of the zero value. Values can be addresses that point to objects. This is another form of edge called *points-to-edge*, modeling pointers. Points-to-edges lead from a value to a target object and offset. Further, they have a target specifier depending on the target. These specifiers are mainly used in lists later, but non-list memory regions uses the region specifier *reg*. The special has-value-edge zero, leading to the invalid null object via a points-to-edge, is always present. A SMG consists of nodes, modeling objects, and edges, modeled by points-to-edges, originating from value edges inside of objects. All values are symbolic in nature, reducing equality checks to simple identity checks inside SMGs. Checking memory properties in SMGs breaks down to checking the validity of objects, offsets and sizes. For example, a points-to-edge pointing to an offset that is greater than the size of a memory region can not be valid. Neither can be a has-value-edge that exceeds the size of a memory region. If a has-value-edge with value zero is used as a pointer, the pointer is considered as an invalid null pointer. An example, including some code that could be used to create the different steps of the example, can be seen in figure 7. In figure 7 (b), three objects exist. Two objects, labeled with *list_start* and *list_end* respectively are of size 32 bits, and both have a has-value-edge covering the entire size. Values are symbolized by the smaller rectangles inside of the objects, with the height symbolizing the size. Both of these has-value-edges have points-to-edges associated with them, making them pointers, both pointing to the same offset zero in the third object. Values that are pointers are symbolized in all figures using *ptr*. This third object is of size 64 bits and has two values, one from offset 0 bits with size 32 bits and another at offset 32 bits, also sized 32 bits. The upper value is the concrete value one, while the lower value is of value zero, which means it is also a pointer to the zero object. Pointers to zero are omitted to increase readability. Also, while SMGs are designed to be used as symbolic values, to discern between values, they have a single value in the examples provided.

```
struct node {
  int value;
  struct node * next;
} List;

int main() {
  // (a)
  List * list_start  = malloc (...) ;
  List * list_end  =  list_start ;
  list_end ->value = 1;
  list_end ->next = 0;

  // (b)
  list_end ->next = malloc (...) ;
  list_end  = list_end  -> next;
  list_end ->next = 0;
  list_end ->value = 0;

  // (c)
  list_end ->value = 2;
}
```

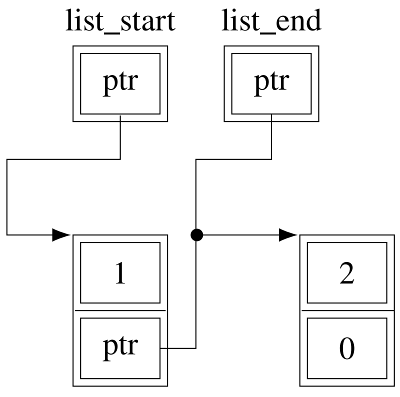
(a) Example code for the SMG at (b), (c), and (d).



(b)



(c)



(d)

Figure 7: A SMG showcasing objects, values and pointers, including their offsets and sizes, as well as write operations.

Read and write operations are emulated with so called read- and write-reinterpretations. Since SMGs use symbolic values only, with the exception of zero, the memory is only read and written such that the difference between zero and non-zero is important.

A read-reinterpretation can be described as follows:
The read-reinterpretation looks up existing has-value-edges with the object, offset and size given. The object can be a region or a list. If a value is found, return it. Otherwise, if the field read is covered by zero value edges, return zero. In case that none underlying value could be found, extend the SMG by a new symbolic value at the location read and return it.

A write-reinterpretation can be described as follows:
If the value one is trying to write to the SMG at a given offset, with a given size, is already present, do nothing. Otherwise, if it is not present, add the value to the SMG as follows. Remove all edges that are non-zero and overlap with the field of the write. If the written value is not zero, then cut overlapping zero value edges such that the new has-value-edge fits. This means that the zero edges end at the offset of the new value and start again after the edge of the new value ends. Finally, write the value into the SMG at the specified offset with the size given.

Figure 7 (c) shows a freshly created new SMG object. The pointer of *list_end* now points to the new object, as well as a new pointer that has overridden the zero at offset 32 bits in the region to the left. Also, the second region sized 64 bits was written to zero entirely. In figure 7 (d), we see a write occurring from offset 0 bits with size 32 bits that cut the has-value-edge zero down to only cover the second half of the object.
It is easy to see that over- and underread, as well as over- and underwrite, can be easily detected by checking the offsets and sizes of the operation in relation to the sizes of the objects. Overflows can not be detected based on SMGs, but have to be checked based on values and operations.

Since addresses are modeled as abstract points-to-edges with no true value, they can not be compared easily. But since pointer equality plays an important role in programming languages such as C, reasoning about the equality of addresses is needed. SMGs solve this issue by inspecting the memory graph that the points-to-edges lead to. If two pointers are equal, so are the values and the shapes of the memory they point to. This however also means that it is possible that checking the equality of two pointers in SMGs can be expensive, as all of the connected memory graphs have to be compared.

A SMG is encapsulated by an *symbolic program configuration* (SPC). The SPC additionally holds the mapping of program variables to their values. In the previous example of figure 7, this can be seen as *list_start* and *list_end*. This allows the SMG to reason about them in the same form as it does about heap memory.

3.9.2 Abstraction of SMGs

Symbolic memory graphs can be abstracted by merging them. Generally, as long as two SMGs, or sub-SMGs, are similar enough in shape, they may be mergeable. A sub-SMG of an object or value is defined as the set of objects and values reachable through connected points-to-edges. However, this work does not merge SMGs in general, but only in certain circumstances. The reasons for this will be discussed in chapter 4. The merging of SMGs reduces the amount of existing sub-SMGs, resulting in some loss of information, but also a speedup of operations and reduction of memory consumption. The most important merging of sub-SMGs for this thesis is the merging of list segments, which will be discussed in detail next.

3.9.3 Lists and List Abstraction

Lists, as well as all other dedicated memory structures can be constructed using SMGs. They use the tags dedicated to their type. A *singly linked list* (SLL) uses *singly linked list segments SLS* and a *doubly linked list* (DLL) consists of *doubly linked list segments DLS*. Pointers pointing to list and from list segments use the special target specifiers *fst*, *lst* and *all* for the pointers to the first, last and all other segments. Further, list segments need to be consistent with their offsets for next pointers or previous pointers. This is used to find and merge list segments based on their layout and shape. The pointer to the previous list segment needs to be directly following the pointer to the next segment for DLS. Also, the offsets for the head, next and previous pointers of list segments need to be consistent in all segments. They are labeled as *hfo*, *nfo* and *pfo* respectively.

SMGs can be used to abstract memory by merging SMGs, which is valuable for memory that has similar shape. This is especially helpful for continues sub-SMGs that have the exact same shape by design, as it is usually the case in lists or trees. Symbolic list segments retain as many characteristics as possible from the individual list segments that they were created from. The shape of the symbolic segment equals that of the previous individual segments in that its size and the offsets *hfo*, *nfo* and *pfo* match. It also retains information about the amount of merged list segments, which is visualized as 2+ SLS for example. 2+ SLS stands for singly linked list segment with at least two concrete segments, with the two being arbitrarily chosen for our example. An illustration can be seen in figure 8. Starting of with (a), we have three singly linked list segments. In (b) those are extended arbitrarily and in (c) all list segments, except for the last, are merged into a 3+SLS. Figure 8 also shows the tag *hfo* for the head offset of the list, the target specified *reg* for region, as well as *fst* for first list segment. One aspect not discussed up to this point is the nesting level of objects and values. This level keeps track of the position of the object or value in relation to its nesting in the (sub-)SMG, allowing the restoration of the most important features of the SMG after abstracting it. An example would be nested lists referenced by a pointer at the *head* offset of the list.

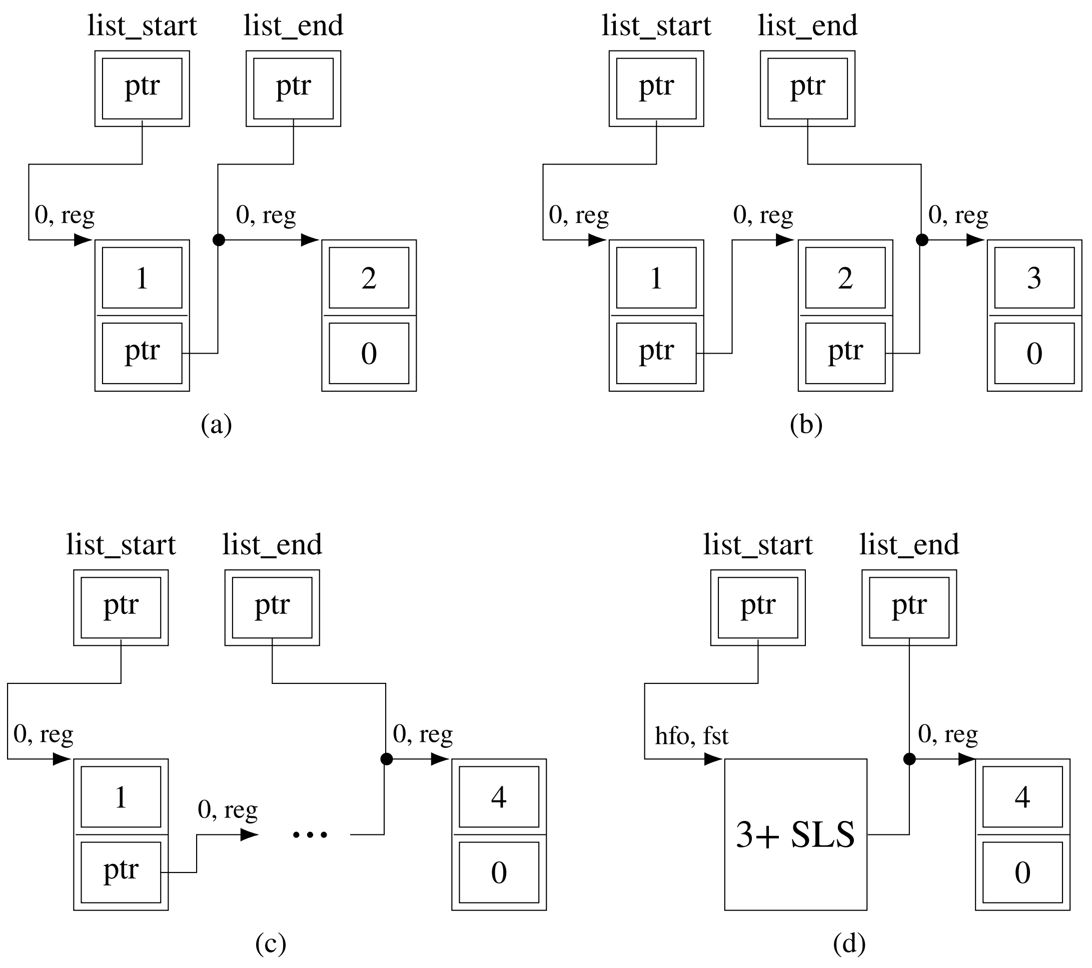


Figure 8: Extension of a singly linked list and subsequent abstraction using SMGs.

There are rules that list abstractions in SMGs have to abide by:
(1) List segments do not allow pointers to point outside of the list in between their segments.
(2) There is no cyclic path containing 0+ list segments.

Possible candidates for additional list segments can be found by inspecting valid heap objects and checking that the following characteristics hold:
(1) The possible new list segment needs to be valid.
(2) The possible new list segment needs to be either singly or doubly linked through the offsets *nfo* and for DLS also through *pfo*.
(3) Their size and nesting level has to be equal and their respective offsets *hfo*, *nfo* and *pfo* at the same position.
(4) SMGs[27] assume that the *pfo* offset always directly follows *nfo*.

If there are possible candidates identified, they may be merged. It is to note that the merging may fail if the candidates are not similar enough in their values or shape. Merging a list

of possible segments can be briefly describes as follows:
(1) Starting from the first segment, get the next segment via the pointer located at the *nfo* offset.
(2) Check that the sizes of the segments, as well as their offsets match and both are valid.
(3) If the values of the two segments are not similar enough, abort.
(4) Replace all has-value-edges that are located at the *nfo* or *pfo* offsets, in both candidates, by has-value-edges with value zero at the same location. Please note that this removes the pointers from and to the two segments temporarily.
(5) Extend the SMG with a fresh linked list segment of the correct type (SLS or DLS) and with the size and offsets of the segments merged.
(6) This new list has the minimum length of both segments added together.
(7) Now remove both segments from the SMG and let all points-to-edges that lead to them point to the new abstracted list segment. The nesting level of all objects and values that are part of a sub-SMG connected to the just changed pointers is increased based on their merging. All of the memory associated with the connected sub-SMG may be merged based on the more general join algorithm.
(8) Restore the dropped pointers at the offsets *pfo* and *nfo* with new target specifiers if necessary.
(9) Repeat for the next candidate and the newly created list segment.

This algorithm ensures that the values are not lost, but merged as well. Merging SMGs[27], or sub-SMGs, beyond that will not be used by this thesis. It is easy to imagine that merging only list segments with the same value results in a list with one concrete value. Also, the nesting levels ensure that the pointers to and from the list can be restored to their original position later on. Since lists in their abstracted form need to be read, written to or reasoned about, they might need to be unfolded again. This process is called materialization and is discussed next.

3.9.4 Materialization of Abstracted Lists

Abstracted memory needs to be reasoned about at some point, emulating a read operation on the abstracted memory. There are multiple different ideas about how this can be done, such as tracking of instrumentation predicates[14]. SMGs however solve this by materializing back list segments that have been merged. This can be thought of as pulling a concrete list segment out of the abstraction. This allows the concrete analysis of list segments with the properties from before the merging, which means no predicates or other information have to be tracked.

The materialization algorithm[27] for DLS is given informally as follows:
(1) Extend the current SMG with a copy of the memory $\widehat{G_d}$ nested below the abstracted linked list, if such a sub-SMG exists. This new sub-SMG $\widehat{G_r}$ has a nesting level that is decremented by one compared to $\widehat{G_d}$. Then, extend the SMG with a new, valid region r, whose size is

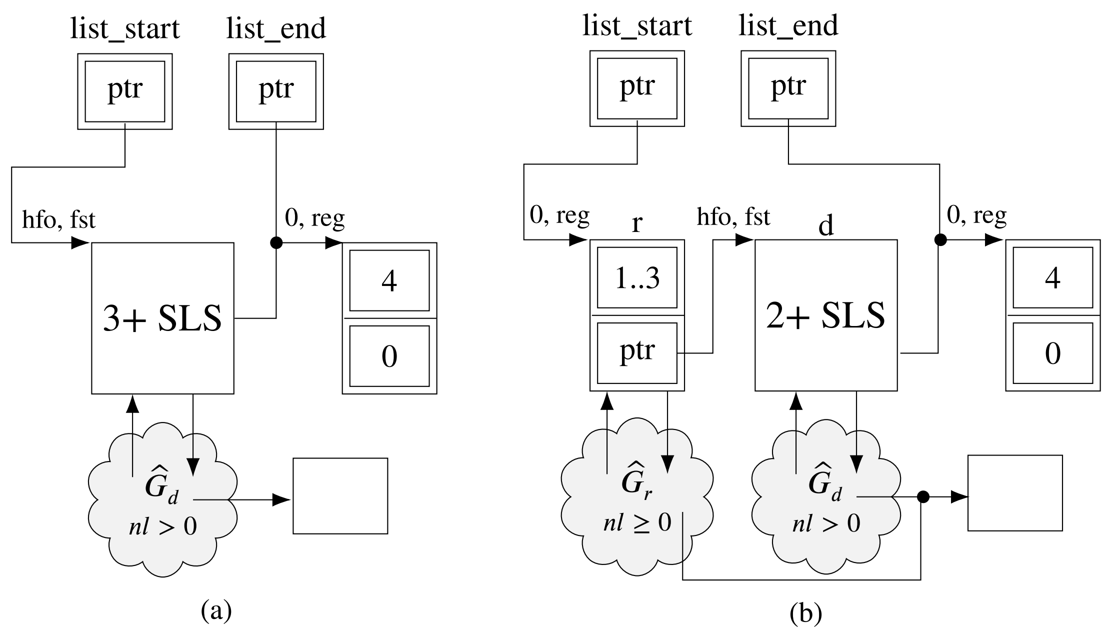


Figure 9: Materialization of a concrete list segment out of a abstracted list, including behavior of pointers to and from the abstracted list.

equal to that of the abstracted list. All pointers from $\widehat{G_r}$ to *d* are replaced with the target *r*.
(2) Add a new points-to-edge that points to the beginning of *r*. If it does not exists, add a new points-to-edge pointing to the beginning of *d* and save its value at offset *nfo* for *r*. Write the value for the points-to-edge pointing to the beginning of *r* to the offset *nfo* for a possible list segment that pointed to *d* as the next list segment previously. Also, add the value for the pointer pointing to the beginning of *r* to the *pfo* offset of *d*. The new region *r* retains the other values, especially the back pointer from *d*.
(3) Copy all has-value-edges of the $\widehat{G_d}$ region with nesting level 0 to their counterparts in $\widehat{G_r}$.
(4) Decrement the minimal length of *d* by one if it is larger than zero.
(5) If the minimal length of *d* is zero, remove it from the SMG and replace the has-value-edges such that the pointers from and to *r* directly connect to and from the objects that previously pointed to and from *d*.
(6) Delete the addresses that are no longer needed, as well as the sub-SMG $\hat{G}_d$ with its edges for a 0+ segment.

This informal algorithm is applicable for singly linked lists as well, by simply removing the sections about the pointers to the previous segments. An illustration is provided in figure 9 from (a) to (b). To illustrate the changes better, figure 9 (a) shows the previously abstracted linked list, with some additional external pointers and a new sub-SMG. Also note that since the list values were merged, the values of the materialized list segments are now equal to the range of merged values.

3.10 Lazy Shape Analysis

Using some form of memory graph with abstraction can help reasoning about tasks that would otherwise suffer from state-space explosion. An example can be seen in figure 10. A doubly linked list is created and a value is added in the first segment. Then, in a loop, values based on the loop control variable are added to new elements of the list and the length of the list is remembered. Following the loop, which terminates after a unknown amount of iterations, the final value is overridden. The program then resets to the first list element and loops through the list to the last list segment, before asserting the distinct last value. Then the loop is rewinded to assert the first value as well, if it is not already in the first list segment. Finally, the length of the list that was traversed is confirmed.

SMGs in the form described above would only be able to abstract the list shown by merging list segments and also their concrete values, creating a range of values. Since concrete-value analysis reasons only about concrete and symbolically unknown values, the value analysis would find the final assertion to be spurious if no concrete values are tracked. Tracking all concrete values would however result in a list that is not abstractable by the presented list abstraction algorithm, as we abstract only list segments with equal values inside.
Tracking only concrete values that are necessary to reason about the feasibility of the current path however is possible. This form of heap based CEGAR has been proposed previously for memory graphs[11]. The application on SMGs and lists in particular is just an adaption of this past work. Lazy shape analysis works mostly similarly to the already presented CEGAR approach for the value analysis. The differences are that the analysis has to detect missing heap information and then add the missing information to its precision. Restarting the CEGAR loop would then track the important values only. In the example above, the analysis would start with an empty precision, tracking no heap values. It would find the assertion with the distinct last value 3 and find the path to it spurious. Therefore the analysis would be restarted with the value 3 explicitly tracked. The analysis would run into the same problem again later on with the first value in the last assert and the list length. After restarting the analysis with these values tracked as well, the program is proven correct, if the shape of the abstracted list accurately reflected the doubly linked nature.

3.11 Memory Graph Refinement

It is obvious from the previous example that the shape of the abstracted list and its materialization have a direct impact on the verification capabilities of the analysis. The doubly linked list of program 6 could be treated as singly linked list when abstracting, but this would lead to a problem when the *prev* pointer is being used. As a result, the type of a list needs to be refined depending on the needs of the analysis, such that the abstracted list reflects all the information needed. Refinement[27] can be thought of as simply checking SMG objects for points-to-edges leading to other valid objects. This checking of potential candidates differentiates doubly linked lists from singly linked lists mainly by existence and validity of the

```
#include <assert.h>

typedef struct node {
  int data;
  struct node *next;
  struct node *prev;
} *List;

void main() {
  List curr = (List) malloc (...) ;
  if (curr == 0) exit(1);
  List old = curr;
  curr->data = -3;
  curr->prev = 0;
  curr->next = 0;
  int length;

  for (int i = 0; i < nondet_int(); i++) {
    curr->next = (List) malloc (...) ;
    if (curr->next == 0) exit(1);
    curr->next->prev = curr;
    curr = curr->next;
    curr->data = i;
    curr->next = 0;
    length = i + 1;
  }

  curr->data = 3;
  curr = old;

  while (curr->next != 0) {
    curr = curr->next;
  }
  assert(curr->data == 3);
  if (curr->prev != 0) {
    while (curr->prev != 0) {
      curr = curr->prev;
      length--;
    }
    assert(curr->data == -3);
    assert(length == 0);
  }
}
```

Figure 10: Example program 6, showing a safe program involving a doubly linked list.

pointer to the previous object. However, since refining the type of lists can be seen as equivalent to refining its shape, previous work on shape refinement[14] can be applied as well. The difference is that shape refinement, can be applied to general memory graphs, e.g. structures besides linked lists, such as trees. Furthermore, it can find instrumentation predicates, which can be explained as information about the behavior of the abstracted list. An example would be that the previous list segment is of the same shape as the current inside the abstracted list segment. These predicates are not needed by this thesis however, as SMGs are materialized when concrete information about them is required by the analysis, as described before.

Continuous shape refinement however can be used instead of individually checking each list candidate for its shape class.
Starting off with some heap memory, samples are gathered up to a threshold. Once that threshold is hit, the heap is analyzed for singly linked lists only. Since every DLL is also a SLL, an invariant can be used to find all SLL candidates and therefore all list sequences. Then, the shape can be refined based on the DLL invariant, in that every segment has a pointer to the previous segment except the first segment. Using this refinement, the type of the list can be found based on samples gathered before the abstraction procedure. If tree structures are possible, an invariant for trees needs to be checked independent of the success of the invariant for pointers to their previous segments. This is needed as there exist trees with pointers to their previous segment that fulfill the doubly linked list invariant.
Using example program 6 again as an concrete example, the loop would unroll several times, gathering concrete list segments. If a sequence is found to be sufficiently long and fulfilling the SLL invariant, it is checked for the DLL invariant. The shape information is then updated based on the findings. Finally, the abstraction procedure uses this shape information to abstract the list. While this refinement can be used for more than lists, the aforementioned trees for example, this work focuses only on the application on lists.

4 Value Analysis with Domain Extension using Symbolic Memory Graphs

This section describes the new concrete-value analysis with SMG domain and its refinement using CEGAR and interpolation.

4.1 Motivation

Value analysis possesses many favorable qualities. It is comparably fast, solves a lot of tasks despite its simplicity, and no background solver is needed to run the analysis. Furthermore, it can be combined with many other analyses via composite CPAs. A composite CPA that extends the domain, such as the CPAchecker Pointer-CPA[33], allowing tracking and dereferencing of pointers, is used per default in the state-of-the-art value analysis inside CPAchecker. However, it lacks heap memory tracking and memory safety detection. The program depicted in figure 10 for example, can not be analyzed correctly by the current state-of-the-art value analysis in CPAchecker that uses the pointer-CPA. It can also not be solved by using CEGAR to track only needed program variables, as will be shown later in this chapter. Symbolic memory graphs are capable of modeling and abstracting memory. They have been used before for fast shape analysis without background solver in the software verification tool PredatorHP. Hence the choice to combine the two into one analysis and then use proven CEGAR techniques to improve it. Because of the limited scope of this thesis, the abstraction and shape refinement concepts are presented using linked lists only.

The reason why it was chosen to implement this new analysis in a single CPA, and not a composite CPA, is rooted in the information tracked by both. The value analysis needs concrete or symbolic values to reason. Using a composite approach, the value analysis either tracks program variables itself, or delegates this to the SMG domain. In any case, the value analysis needs the values of the heap memory tracked by SMGs. This means that, either the memory would need to be split between SMGs and value analysis, or the SMGs track the entire memory anyway. Furthermore, the heap memory in SMGs does need to track concrete values for the value analysis. So instead of splitting the tracking of values, the entire memory, stack and heap, is modeled using SMGs. This also has the advantage, that some memory analysis techniques of SMGs can be used on the stack as well. Another reason for this is e.g. the ampersand address operator & that returns the address to any memory and enables manipulation and access to stack memory via pointers.

4.2 Value Analysis CPA with SMGs

The abstract value analysis CPA is explained in the following section. This includes some description about the used SMGs. The concepts used are mainly based on the background chapter.

4.2.1 SMGs

We use SMGs as described in background, but add some constraints to them. Lists, as well as any other merge operation, aborts if the concrete values inside the merging objects are not equal for all offsets. This constraint allows the analysis to retain as much information as possible, while still being able to merge based on equal concrete and all unknown values.

4.2.2 Abstract Domain

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete states, the semi-lattice $\mathcal{E}$ of abstract states and the concretization function $[\![\cdot]\!]$. Abstract states are assigned their meaning by the concretization function $[\![\cdot]\!] : E \to 2^C$ based on the semi-lattice. The semi-lattice (E, $\sqsubseteq$, $\sqcup$, $\top$) of this abstract domain is defined as follows. Lattice elements $e \in E$, representing abstract states, are defined by the SPC, holding all relevant information and the SMG, e.g. variable assignments. The top element $\top$ represents a state without any information. In the context of SMGs this means that the SPC holds only the null value that is also a pointer to the null memory region, as well as the null memory region, which is invalid. The bottom element $\bot$ represents unreachable states. The partial order $\sqsubseteq \subseteq E \times E$ is defined similarly to that of the value domain as e $\sqsubseteq$ e´, if $e = \bot$, or v´ $= \top$, or def(e´) $\subseteq$ def(e) and e(x) = e´(x) holds for all x $\in$ def(e´). The join $\sqcup$ of two abstract states yields the least upper bound.

4.2.3 Precision

The precision π tracks two different kinds of information. First are currently tracked program variables and second the values that are allowed to be used on the heap. If a variable is not tracked by the precision, it should not be tracked by the analysis. Similarly, concrete heap values are assumed to be unknown if the precision does not include them.

4.2.4 Transfer Relation

The transfer relation $\rightsquigarrow$ is defined as $e \overset{g}{\rightsquigarrow} (\widehat{SP}_{op}(e), \pi)$, with the strongest postcondition $e' = \widehat{SP}_{op}(e)$ and g = (l, op, l´). The strongest postcondition $\widehat{SP}$ is defined as the application of an operation $op \in Ops$ on an abstract state e, resulting in a new abstract state e'. Additionally to the execution of the operation, the strongest postcondition performs a shape analysis step for all operations involving heap memory.

4.2.5 Precision Adjustment Operator

The precision adjustment operator for this CPA computes a new abstract state e', based on the given state e, that only uses the program variables that are part of the current precision π. Furthermore, the precision-operator also enforces that only heap values that appear in the precision are used in the new state e'. Concrete values that are not part of the precision are replaced by unknown values.

The precision adjustment operator also computes the list abstraction and refinement for the new state after the precision is enforced. This ensures that abstracted lists use only the information needed for the analysis.

4.2.6 Merge Operator

The used merge operator is $merge_{sep}$, which means that no states are merged. A joining merge, i.e. $merge_{join}$, is not used for several reasons. Merging the concrete values inside abstract states makes only sense for equal values, as otherwise the analysis would lose all of its information about a value. As a result, most states that are mergeable are very similar and therefore likely to be subsumed in one way or another by the stop operator. Also, merging SMGs can be costly, especially if the merge does not succeed, which is expected to happen for states not subsumed by the stop operator.

4.2.7 Stop Operator

The used stop operator is $stop_{sep}$. $stop_{sep}$ checks for an abstract state if it is already covered by an element of the reached set. In the context of SMGs all of the available concrete memory has to be subsumed by the state of the reached set. This means that heap needs to be compared based on concrete values, pointers and also its shape. An example would be two abstracted singly linked lists 5+SLL and 4+SLL. As long as the size, the values and the offsets of the values are the equal in both, the larger list can be subsumed by the smaller list. This behavior is sound, as we only subsume lists such that the state with the larger list behaves the same as the state with the smaller list.

4.2.8 Value Analysis with CEGAR and Interpolation

The analysis uses CEGAR to track only necessary program variables, as described for the value analysis in the previous chapter. This includes value domain interpolation, which did not need adaption because of the used strongest postcondition. The only new addition is, that the removal of variable assignments inside the Interpolate algorithm also removes the heap memory associated with the removed variable iff the memory originates from the removed variable. This important detail makes sure that no subsequent pointers use heap memory that was never declared from the point of view of the interpolation algorithm.

4.2.9 Heap Value Refinement

Heap value refinement focuses on the used concrete values in heap memory. The interpolation and refinement itself is based on the value analysis CEGAR and interpolation approach.

Removing a value is equal to assuming it to be unknown. The feasibility checker alone however would find the previously infeasible path not to be feasible even if the value removed was essential. Since values are replaced by unknown values, they can be assumed to equal the same value as before. However, what happens is that a previously concrete path now branches for assumptions.

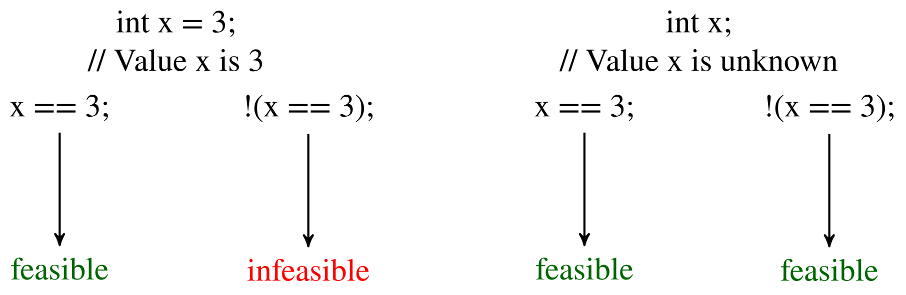


Figure 11: Illustration of the different (in-)feasible paths for essential values based on their value in an equality assumption and its negation.

An example can be seen in figure 11. An equality assumption is evaluated for the same variable *x* for a concretely known value and an unknown value. On the left hand side we see that tracking the value allows only one path, while on the right hand side all possible paths are feasible due to the removal of the value and its replacement with an unknown value. The reason is that the now unknown value can be assumed for assumptions and also their negations. As a result, a feasibility check alone does not suffice to proove the need for a concrete value in the analysis. Extending the feasibility check to check if a path branches is needed. A value that is necessary is also a value that removes this branching from the path. To avoid adding heap values that are not needed, an initial branching check is run before a value is removed. If a path is already branching, removing a value does not change that. Instead of only removing all program variables that are necessary, *Interpolate* is extended to also check all newly found heap values for branching and removes them if they are deemed not necessary. Algorithm *Refine* is extended to extract the found essential heap values and add them to the precision, similarly to program variables.

4.2.10 List Abstraction and Shape Refinement

List abstraction and shape refinement procedures follow the concepts lined out in the previous sections about list abstraction and shape refinement. During the execution of the analysis, information about possible linked list candidates are gathered. If an adequate amount of list segments is found during the execution, e.g. at loop heads to reduce the amount of redundant lookups, the shape of the structure is analyzed further. This is done by searching all of the heap samples for singly linked list sequences. Since doubly linked lists satisfy the SLL invariant, they are also found. Then the invariant for DLLs is checked upon the found sequence of candidates. If this invariant is found, the list is abstracted as DLL, else it is abstracted as SLL. The abstraction itself is similar to the procedure informally described *before*. Refinements on already abstracted lists or additional list segments is easier. Adding pointers to previous list segments to an already abstracted list saves the initial search for singly linked lists and new segments are identified by them just adhering to the already known shape information.

4.3 Resulting Analysis

The analysis resulting from the definition of this chapter is capable of solving all example programs, except for example program 3, which is not solvable by design. Using the example of figure 10, we show the inner workings of this new analysis. For simplicities sake, we assume that allocation functions always succeed in the following example. Furthermore, we use a specification that ensures that asserts, e.g. in line 34 and 40, always succeed. The analysis uses list abstraction with shape refinement and CEGAR as specified in this chapter.

An initial analysis run with an empty precision determines the assert in line 34 as a possible specification violation and the refinement algorithm would be started. Subsequent interpolation would learn that the program variables *curr* and *old*, as well as values *3* and *0* are relevant, and they are all added to the precision. Heap value *0* is only added as it is always present in the SPC. This is a side effect that always happens. Following this, the CEGAR procedure restarts the analysis with the new precision. Memory for the first list segment is created successfully and the pointer to it is saved in *curr*. The variable *old* saves the pointer to the first list segment as well and an unknown value is assumed for the *data* field, as the value *-3* is not tracked by the precision. Since the execution of the loop depends on some non-deterministic value, each time the loop head is checked, two states are assumed. Because it is easier, we follow the state that skips the loop entirely. Value *3*, which is in the precision, is saved to the *data* field of the current list segment in line 28, overriding the unknown value. The following while loop is not entered, as the condition is not fulfilled. The assert in line 34 succeeds and the condition for the if statement in line 35 is checked and found to be not fulfilled. As a result, the analysis ends without property violation for the state that skipped the for loop in line 18. We assume that in subsequent analysis runs the state with only the initial list segment always succeeds without property violations. Moving on to the state that entered the for loop. New memory is allocated and the pointer to it saved in the next field of the current list segment. The back pointer is set set properly and the variable *curr* is advanced to the new list segment. The field *data* of the new list segment is now filled with an unknown value, as the loop variable that should equal *0* is not tracked. Now the analysis checks the loop head and finds two possibilities, generating two states again. We follow the state that leaves the loop again. The analysis behaves the same up until the while loop in line 31, which is executed once. Following this, the assertion succeeds again and the condition of the if statement is fulfilled. The while loop in line 36 is executed once, rewinding the list to the first segment. The assertion in line 40 leads to a spurious counterexample as the value *-3* is not tracked. As a result, the refiner is started again, leading the interpolator to find that the heap value *-3* should be included in the precision. The analysis restarts and performs as before, but can now confirm that the first list segment has the correct value *-3* for lists greater than one element. Afterwards, the assertion in line 41 is found to be spurious as the variable length is not tracked. The interpolation algorithm is started by the refiner and finds that not only length, but also the loop variable *i* need to be tracked. Value *0* would also be found at this time if it was not already part of the precision. After a restart of the analysis with the new precision, one problem remains, the for loop in line 18. Since the analysis does not know when to stop the loop, it generates two states every time the loop head is checked. The state leaving the loop is confirmed to be correct every time, but the other state repeats the loop. This is where the list abstraction kicks in. As the loop variable *i* is now tracked, the values that are saved in the list would now all be concrete as well. But our current heap value precision removes all values that are not *-3*, *0*, or *3*. After several loop executions, the heap is checked for possible singly linked lists. Once a number of them is found, they are checked for their shape, in this case we find that the list is a doubly linked list, and that their values are equal. The abstraction algorithm finds, that there is an initial list segment with value *-3*, then a segment with value *0*, followed by two list segments with unknown values. These are tailed by one segment with value *3* and finally arbitrary many unknown values. The abstraction algorithm abstracts the tail of the list, starting from the first unknown value after the concrete value *3*. Then the abstracted list is again split by the loop head into two states. However, the next time the loop head is encountered, the list abstraction is performed again, increasing the minimum size of the abstracted list by one. The stop operator would then find that the two states with the abstracted lists subsume each other. This stops the execution of the loop, and since no property violation can be found, ends the analysis proofing the program safe.

5 Implementation

This chapter describes the implementation done for this thesis in the CPACHECKERframework. CPACHECKER, and this implementation are written in the programming language Java. Apache Subversion (SVN) has been used as version control system and the code is located in the *smgv2* branch[5] of the SVN repository of the SoSy-Lab. The used C standard, on which the software analysis is based, is C99 ISO/IEC 9899:201x. But some API additions by the popular C compiler *GNU GCC*[6] were included as well. The analysis can handle all standard C types, including floating-points and String literals, as well as all standard stack and heap memory operations. Since the ampersand operator & is supported, stack memory can be treated exactly the same way as heap, in terms of operations of the program. All newly created classes and interfaces for the CPA and CEGAR are found in the *org.sosy_lab.cpachecker.cpa.smg*2 package and its subpackages. The SMG and its components, except for the SPC, are found in the *org.sosy_lab.cpachecker.util.smg* package. Most of the classes in the *smg2* package inherit one or more necessary CPA component super-classes, e.g. the transfer relation inherits from the abstract class *ForwardingTransferRelation*. The used Java objects are implemented such that they are always immutable and a change always returns a copy with those changes, never changing the old object. The data structures used have been selected such that often times lazy copies are possible, to reduce memory and runtime complexity, and are also immutable in nature. Immutable objects mainly help to always reflect a concrete state of the CPA, without side effects by further usage. Also, since a lot of operations need to return tuples, each one of those are modeled with a helper class bundling the return values. Annotations are used in the rare cases that null is a possible input in a method. Generally null is not used as a value, except for when CPACHECKER demands it due to legacy reasons.

5.1 SMGs and SMG Abstraction

Symbolic Memory Graphs are generally implemented as described in the background chapter Their implementation is not part of this thesis, as they were already implemented prior to this work, with the exception of the abstraction process of lists, as well as some minor additions and changes.

Some implementation details are given to provide context for the later parts of this chapter. SMG-values are symbolic in nature and a static method inside the *SMGValue* class returns new, unique *SMGValues* on request. The only special value is zero. Here special caution has to be taken that zero always refers to a sequence of pure bit zeros, especially for floating-point types. As described in chapter 3, read operations return new, unique values for read memory regions where no prior value could be found. These are later used as symbolically unknown

[5] At the time of writing this thesis. The branch is expected to be included in the trunk at some time.
[6] https://gcc.gnu.org/, 21.9.2022

values. Doing this allows the comparison in regards to equality.

From this point onward only new additions and modifications to the SMGs are discussed. First, the read and write functions and all helper methods were extended from byte precise reads and writes to bit precise. This was needed as support for bit-fields. Also, the previously existing algorithm to determine the equality of pointers, based on the inspection of the memory they lead to, had to be modified. The modification was done to analyze not only pointer equality, but also given data structures with their concrete values derived from the SPC, discussed later in this section. This was needed to determine the equality of arbitrary memory sections to check the equality of SPCs for the stop operator.

While the join algorithm for SMGs has been implemented beforehand as described by Dudka[27], it is used in a slightly simplified and newly implemented form for this thesis. The reasons for this have been outlined by the section about the used merge operator. In terms of implementation, this simplified the joining of SMGs as we are only ever abstracting linked lists with the same value. Therefore the rules outlined in chapter 3 can be eased such that we expect to only ever merge sequences of linked lists with the same value. Also, the implementation detects the *next* and *prev* pointer offsets independent of their ordering. We also allow arbitrary many data fields in our lists. The rest of the outlined algorithms are implemented as described informally in chapter 3. The choice of which list to abstract is depended on the precision adjustment operator per default. List abstraction is possible independent of the usage of CEGAR. The difference is that without CEGAR, every heap element will by analyzed and every found list will be abstracted as much as possible. This is expected to be limited for lists with values that do not equal for an extended list. The materialization of lists is always dependent on the dereference of a pointer and is handled by the class *SMGCPAMaterializer*. List abstraction candidate search and shape refinement is handled by the class *SMGCPAAbstractionManager*, but the actual abstraction is performed on the current state.

5.2 Value analysis with SMGs as domain

The implementation of the value analysis with SMGs as domain extension has been aided by the previous concrete-value analysis implementation. The *Value* interface, and its implementations, represent the concrete and symbolic values used by this implementation. Except for the pointers, all of the concrete and symbolic values were already implemented by the value analysis. If not declared outright, the reader can expect that all of the rest of this implementation was done by the author of this thesis.

5.2.1 SMGState and Symbolic Program Configuration

The *SymbolicProgramConfiguration* class (SPC) holds the SMG and therefore the memory and its symbolic values. The class maps symbolic SMG-values to their concrete counterparts, that may be concrete or symbolic. This mapping is more difficult as it might seem. The value analysis part of the implementation needs concrete type information in its CEGAR procedure. However, SMGs only reason about the size of a type in their implementation. It was decided to remember the types of values that are part of variables separately, but not do so for values only used on the heap, as they only need to be removed und remembered in the used CEGAR approach, which would write them back to the heap if removed from it. The reason for the removal and adding back is the *Interpolate* algorithm of the refiner. Further, concrete-values know their Java equivalent type by being saved as them, wrapped in the *Value* interface. The exception to this are C types not present in Java. They are assumed via larger types in Java, e.g. *BigInteger*, and then translated to their C counterpart. Since zero has a special handling in SMGs, all values with only zero bits get mapped to the same SMG-value zero. Therefore, cautious handling of zero values in regards to floating point types is needed, which means a possible re-translation into floating-points or from them to other types based on the current C operation. The SPC also holds the stack frame and all mappings from variables to their memory. The stack frame consists of local variables depending on the current function, as well as variable arguments used in the function call that created the stack frame if needed. Besides that, the stack frame holds a return value if necessary, while the external and global variables are held by the SPC. The SPC also includes all methods to retrieve the variables memory for later reads and writes.

The *SMGState* class holds all relevant information for the abstract states of the analysis. But it is also a mediator between the SPC, holding the information about the memory and the values, and the rest of the analysis. This can be divided into three categories. First, the CPA related methods, then the SPC related tasks and finally procedures connected to CEGAR. The *SMGState* decides about how the stop operator is implemented. For SMGs, this means that the current memory state of two states have to be compared. While local and global variables are compared by their values, with symbolics always treated as equal, heap memory is more complex. It is to note that pointers, while being symbolic in that they do not have a concrete value, are treated separately. Pointers are always compared using the shape of the memory graphs that they are pointing towards. Heap memory, is compared using its shape in addition to its values. In general, a state is considered less or equal to another, if all the memory present in the smaller state is considered equal by the larger state.
The *SMGState* class also holds the error information of the shape analysis that is collected while executing the analysis. An example would be that if a null pointer is dereferenced, the state remembers information about that. The analysis is not stopped by this, as the CEGAR algorithm might execute the analysis as part of its procedure, which means that not all occurring critical errors are feasible. Only if a memory safety violation is in general possible, the state remembers the error and the stop operator would stop the analysis if the specification asks for it.
As a mediator between the analysis and the memory, the *SMGState* possesses the most general

write and read methods for the current memory model. Since both write and read have side effects in SMGs, the state needs to be changed as well after such an operation. Because of the employed immutability in the entire implementation, it is easier to do this in the state itself and return a new state object. Also, the write and read methods of the state are the mediators of the *Value* class that holds concrete information about values and its SMG-value counterpart that is symbolic. Some special writes and reads have been added to ease commonly used versions of the methods. For example a write to zero method or read return value method for stack frames. Lastly, the *SMGState* class uses the *ImmutableForgetfulState* interface that was created as part of this thesis. The interface can be seen as the extension of an already present mutable version that is part of the CEGAR procedure. Due to the interface, the state has to be able to forget a local or global variable or a single heap value, but not remember it again. Heap memory is in general only forgotten if the creating variable was removed via a forget operation. Since the state is immutable in nature, the old state can be re-used by the *Interpolate* procedure if a variable or heap value is deemed essential. The state also detects dereferences to abstracted heap memory and employs the materialization of lists.

5.2.2 Transfer Relation and Precision Adjustment

The transfer relation executes the entered operation on the entered abstract state. These operations reflect the execution of the C code. This execution includes adding new stack frames for function calls and making function arguments available on the new stack, including variable function arguments. When functions return, the stack frame is removed, and, depending on the options, all memory associated with the old stack is pruned. This pruning detects memory leaks by checking if a removed variable holds the last reference to a heap memory region. The transfer relation uses the *SMGCPAValueVisitor* to retrieve values for the operations from the state, depending on the current CFA edge. Similarly, memory is retrieved via the *SMGCPAAddressVisitor* class in an encapsulated form. The encapsulation is necessary as the address, which is either memory behind a pointer or a variable, could receive an additional offset before being read or dereferenced. After each execution of the transfer relation, the precision adjustment operator is used with the current *SMGCPAPrecision*. This can have multiple effects. First of all, all variable assignments for variables that are not tracked by the precision are removed using the forget method from the *SMGState*. Also, if a heap memory region uses a value that is not tracked, it is replaced by an unknown substitute value. Then, if the current CFA edge is a loop head, the precision adjustment operator searches for lists to abstract. Loop heads are used as lists tend to be extended in loops and CPACHECKER does not support endless recursion. If a list is built without a loop or recursion, it is finite and does not branch extensively and therefore no state-space-explosion occurs. The type of list, either SLL or DLL, is dependent on the available samples that have been gathered during execution of the analysis. If a length threshold for valid list segments is reached a list is abstracted to the found shape. The threshold is per default three, as preliminary tests did find very few false positives with that value, but it is changeable via an option.

5.2.3 Precision

The precision for the described SMG domain needs to track two kinds of information that the analysis is allowed to use. Global and local variables, as well as the heap values that are allowed to be used. This is implemented in the *SMGCPAPrecision* class that is an extension of the existing *VariableTrackingPrecision* class. The variable tracking is mostly taken care of by the inherited abstract *VariableTrackingPrecision* class. Heap value information is tracked by a set of allowed concrete values.

5.2.4 Value- and Addressvisitor

The *SMGCPAValueVisitor* class takes any expression and returns an evaluated value for it. Expressions can for example be the disassembled parts of an operation, e.g. *a* or *b* of *a == b;*, or the result of the evaluated operation. The value visitor can also evaluate arithmetic expressions, casts, pointer dereferences, array access and more. It is to note here that most of the arithmetic used, including arithmetic for symbolic values, is taken from the present concrete-value analysis and modified. The remaining operations, for example memory access operations or pointer equality, were all implemented by the author. The analysis supports all types available in standard C, including floating-point types and for example bit-fields and unions. Floating-point types need special care in regards to how they are treated as their arithmetic differs from usual integer based arithmetic. Pointer arithmetic and comparison operations on memory and memory addresses is supported as well and was newly added. Memory address comparison is based on the pointer comparison algorithm outlined in chapter 3, but extended to include the concrete values known into its verdict. It is also usable to check the equality of SPCs based on the shape of the memory and its values. A *SMGState* is always necessary to retrieve values from the current memory model. Since nearly all SMG operations can have side-effects, a new state is always returned in conjunction to the value. If an expression can not be evaluated, an unknown value is returned. For an expression that returns a pointer without dereferencing it, the result is the same as for the address visitor. All added built-in functions are also executed from this class. The class might also use the *SMGCPAAddressVisitor* class, as for example some array expressions or the ampersand & operator return the address to some memory. Since SMGs are used for the entire memory used, all the memory can be referenced by pointers. The implementation also supports pointers to functions.

The *SMGCPAAddressVisitor* class returns memory requested by the entered operation. This can be some address to a memory location, or a program variable, but it is always bundled with the current offset. The expression struct.field1.field2 with a struct on the stack that has a nested struct in field1 would be an example for the need of this class. Evaluating the variable struct with the offset for field1 would not suffice and might miss the additional offset of field2 if it is not zero. The same applies for pointer expressions. Since C supports copy operations for structs, those are detected based on the types of the operation in the transfer relation.

5.2.5 SMGCPABuiltins

Built-in functions are supported via the *SMGCPABuiltins* class. It handles the tasks of the functions requested, or decides if an unknown function is safe to use and might assume unknown values for it. Supported functions are for example malloc, alloca, strcmp, memset, memcpy and variable argument functions. The functions might return more than one resulting state or value. Malloc for example can be configured to fail and return a zero value. If a function can not be sufficiently executed, for example due to unknown values, an unknown value is assumed as the result of its operation. This can for example happen due to input parameters being unknown.

5.2.6 SMGCPAExpressionEvaluator

Writing to and reading from the SMG back-end is assisted by the *SMGCPAExpressionEvaluator* class. The evaluator provides useful helper methods for writing and reading, depending on the used expression. Reading a value is rather easy, as only either a variable or a memory address and an offset needs to be known. Writing is however dependent on the current value and type that is written, as well as the target type and the operation used. For example, writing to heap memory always involves dereferencing a pointer and then writing a value to the SMG via the state. Writing to a local variable usually needs the variable mapping of the SPC and an offset. Please note that local or global memory can be complex as well, as all types used in C are supported by the analysis. Especially Unions can have type reinterpretation behavior and are handled with that in mind. The evaluator also creates and maps new pointers based on the needs of the analysis. No two pointers to the exact same memory location exist, as this would increase the runtime of SMG operations.

5.2.7 Assigning Value Visitor

A core component for a value analysis is that values can be assumed in assumptions. This is managed in the *SMGCPAAssigningValueVisitor* class, that is only used if an assumption is not concretely evaluatable. Independent of if the assumption is in a negation or not, the expression is always evaluated such that the current assumption is fulfilled. If one of the separate parts of an operation possesses a concrete value, the other value is either equal, or not equal to that value. For equal values, the other value is replaced in the memory model with the concrete value that it equals, iff an equality operation is assumed to be fulfilled. In the that case they are not equal, this can not be done, except for boolean values, in which case the negated integer can be assumed. Generally, using the assigning value visitor results in at least two distinct states that cover all possible results of the assumption. Soundness is preserved by this, as possible changes are made such that they only effect the followup state for the next CFA location.

5.3 CEGAR and Interpolation

This work uses the generic refiner, path-slicer, feasibility checker and interpolator classes of the already present value analysis as much as possible. However, most methods and classes needed overrides to change their behavior from a mutable state with the *ForgetfulState* interface, to a new, *ImmutableForgetfulState* interface. Both interfaces allow the removal of a program variable, but the immutable version does not change the current state and returns a new one with the variable removed. All the additional classes created for this thesis are implemented in the *org.sosy_lab.cpachecker.cpa.smg2.refiner* package. The interpolation for heap values is based on branching of paths. This was implemented as an extension of the feasibilty checker. The same procedure that executes the path based on the strongest post-condition is used, but all assumption statements that may lead to additional branching are checked for their negations. If the current state allows this branching based on the previously concrete path, a branching path is detected. This check is used twice in the interpolator for heap values. Once before removing the values to detect branching before removing values and once after the removal of a heap value. If the path branches differently then before, a necessary heap value is found and inserted back.

5.3.1 Refiner

Most of the *SMGRefiner* class is using the *GenericRefiner* super-class that was already present. The *GenericRefiner* class implements the $Refine^{+}$ algorithm and as part of that delegates to the rest of the classes needed. Since this thesis needs to gather the heap variables needed for an execution, the $Refine^{+}$ algorithm has been extended. It now also extracts heap value information necessary for the execution from interpolants. Another addition made for this thesis is that, now the refiner can be used with immutable objects that return an updated copy of themselves after operations.

5.3.2 Slicer

The *PathInterpolator* class performs the slicing of the given abstract error paths as described in *ExtractSlicedPrefixes*. However, the generic slicer method did not perform the slicing according to the described algorithm, but used a shortcut. It removed basically all CFA edges per default, and rebuilt the CFA only for the necessary parts. The sliced path did not include function calls or returns, as they are handled by the strongest post operator of the value analysis. The strongest post operator is not to be confused with just the execution of the transfer relation here. Also, variable declarations and assignments, except for the last assignment before the first branching of the path that relied on that program variable, were sliced. Since the variables can be created on-the-fly, as CPAchecker remembers their declaration, this works for non-heap memory. This on-the-fly creation did create only needed parts of memory however, for example just a single element of an array, but not the whole array. The SMG domain has other dependencies however. Firstly, memory in SMGs is declared using the total size of the memory. Secondly, function calls and returns may only be sliced if the function call already returned and the return value is irrelevant and no heap is created or manipulated in that call. Function calls that are not yet returned may not be sliced, as the stack frame is closely tracked. And most importantly, since we are working on heap memory, we can't ever slice heap memory operations, as we don't know how long that heap memory persists from the CFA alone. Therefore, the slicer had to be changed to reflect the original algorithm more closely and abide by these rules.

5.3.3 Strongest Postcondition Operator

The *StrongestPostOperator* class is similarly to most other classes inherited by a generic counterpart. It hands the operation and precision for a CFA location to the transfer relation to get the strongest post-condition. Substantial changes needed to be made however, as most overhead of the value analysis could be removed. The generic strongest postcondition operator used a stack of states to remember the states from before function calls. It was then used to drop a stack frame and remember the previous stack frame in the concrete-value analysis. This is not feasible for the SMG domain as heap is tracked. Changes in the heap performed in functions need to be reflected correctly afterwards. The new implementation discards this stack of states and employs the transfer relation for function calls and function return statements, as well as all other operations.

5.4 Violation Witnesses

The implemented analysis is capable of producing witnesses for programs that have been found unsafe. This is done by simply iterating along the found error path and extracting all known values from the states based on their location. If a value represents an address to some memory, the memory is also extracted and the values and relationships of the memory via pointers is copied as well. As a result, the variable assignments and also heap of each program location is returned as proof for the path taken. Unknown values are not extracted, as they would not be of any help since they may equal any concrete value.

5.5 Tests

A suite of unit-tests was added to test the implementation. The transfer relation, the visitors and abstraction implementation have dedicated classes to test them. While the SMG and join algorithms are also tested, these are not part of this thesis. The tests are designed to be as generic as possible. For example the transfer relation is tested by creating memory for e.g. structs that are built from a subset of all types and then consequently read or written to. The tested subsets are of varying sizes, use nested types and try to cover as many combinations as possible. Since all major C types are used in all tests, all operations on them, for example pointer access and arithmetic, subscript array access, nested struct access, are also tested.

5.6 Limits

The analysis has certain limits due to the used SMGs, the scope of this thesis or per design. Unknown values are generally a problem for the analysis. An example is that if an offset in an array subscript expression or pointer expression is now known, the expression can not be evaluated. As a result, unknown values are assumed for expressions or functions that can not be evaluated. If an array is declared with an variable length, the analysis stops. This is necessary as they can not be assumed if the concrete size is not known with SMGs. The same problem occurs for symbolic memory allocation in general. CEGAR helps to avoid this if possible. However, CEGAR itself is hindered by this, as will be described in the next chapter. A series of assertions and preconditions ensure the consistency of the implementation. As a result, critical errors in the implementation also stop the analysis. The following chapter however shows that no errors of this kind occur on the basis of a large benchmark set.

6 Evaluation

In this chapter, the newly defined concrete-value analysis with SMG domain and CEGAR is evaluated. Firstly the used benchmarking tool and environment, including the used configurations is discussed. Then the set of tasks is described, that is used for the evaluation. Finally, the results are compared to PREDATORHP, as well as the state-of-the-art value analysis inside CPACHECKER , before being critically discussed.

6.1 Evaluation Setup and Research Questions

The resources of the task executions is measured using BENCHEXEC[19], version 3.10. BENCHEXEC is a reliable benchmarking tool with support for CPACHECKER and many other software verification tools. The tasks were distributed to a cluster of 167 dedicated machines. The same cluster has also been used for the SV-COMP 22, enabling direct comparisons to results of the SV-COMP. Every machine on the cluster uses a Intel Xeon E3-1230 v5 CPUs with 8 processing units each with a frequency of 3.4GHz and 33GB of RAM. GNU/Linux Ubuntu 20.04 was used as the operating system of the units.

The tasks themselves are executed using BENCHEXEC, ensuring not only accurate measurements for the used resources, but also guarantees that the resources of each task are isolated from each other. The limits are similar to those of the SV-COMP 22, i.e. 15 minutes of CPU time, with 15GB of memory and 8 processing units available, for each task.

All tasks used are taken from the international competition on software verification 2022 benchmark set[7]. The SV-COMP benchmark set is a large set of example programs of varying difficulty in several categories used in the annual international competition on software verification, making it ideal to be used as a reference for soundness and comparison of analyses. Besides the complete *MemSafety* benchmark set of the SV-COMP 22, several subsets are evaluated in detail. Used benchmark subsets are *MemSafety-Heap* and *MemSafety-LinkedList* of the *MemSafety* category, as well as *ReachSafety-Heap* from the *ReachSafety* category. The goal of the ReachSafety category is to determine if an error state is reachable, while the MemSafety category provides tasks to check for invalid memory operations. Both categories reflect the capabilities of the new SMG based values analysis, while the chosen subsets reflect the focus on general heap and linked lists. It is to note that the tasks of both categories overlap and only differ because of the used specifications. The tasks themselves reflect a wide variety of test cases with varying difficulties. This includes, for example, singly and doubly linked lists, that are generated up to a symbolic bound and then traversed, manipulated and checked via assertions. But also pointer-, struct-, array- and union-related tasks in the heap subcategories.

This ensures that the implementation of this work is tested thoroughly.

Also, the results of the analysis presented in this thesis will be compared against PREDATORHP

[7]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, 15.09.2022

in the MemSafety category and the state-of-the-art value analysis of CPAchecker in the ReachSafety category. Since Predator is also based on SMGs and had good results in the past years of the SV-COMP, the new analysis can be evaluated more closely. The results for Predator are taken directly from the SV-COMP 22 repository, as the used setup is equivalent to that of the SV-COMP 22. The the state-of-the-art value analysis is run with the same specifications as presented above, and with the same options as the SMG based value analysis. This includes, for example, the used CEGAR thresholds and strategies presented in chapter 3. It also utilizes a Pointer-CPA[33], enabling the value analysis to reason about pointers, as well as predicate analysis[16] as a counterexample check.

6.2 Experimental Results for ReachSafety-Heap

	ValueAnalysis			SMG-ValueAnalysis			SMG-ValueAnalysis-CEGAR		
	status	cputime (s)	memory (MB)	status	cputime (s)	memory (MB)	status	cputime (s)	memory (MB)
total	241	9380	105000	241	30100	78200	241	16100	75900
correct results	60	**472**	**12200**	177	981	31700	**184**	1360	39900
correct true	27	**188**	**4700**	104	517	18100	**119**	918	26600
correct false	33	**284**	**7530**	**73**	464	13700	65	437	13300
incorrect results	**0**	-	-	28	145	5000	16	89	2950
incorrect true	**0**	-	-	**0**	-	-	**0**	-	-
incorrect false	**0**	-	-	28	145	5000	16	**89**	**2950**

Table 1: Experimental results of the newly presented value analysis with SMGs, with and without CEGAR approach, and the state-of-the-art value analysis in CPAchecker on the ReachSafety-Heap SV-COMP 22 subset.

The results of the state-of-the-art value analysis and the new SMG based value analysis for the ReachSafety-Heap benchmark sets can be seen in table 1. The experimental results show that our new approach performs better than the compared value analysis with value domain independent of the implemented CEGAR approach. However, CEGAR performs only slightly better than the non-CEGAR analysis. Comparing the time and memory consumption is difficult, as the state-of-the-art value analysis could only solve very few tasks compared to the new analysis. While the total CPU time and memory consumption of the value domain based analysis was less than both versions of the new analysis, compared to per solved task, the new analysis performed better in both categories. Also, the CEGAR approach seems to consume less time and also less memory in total compared to the non-CEGAR version of the analysis. However, if only the correctly solved tasks are compared, we can see that this is not the case. Since the incorrectly solved tasks contribute only little to the total time and

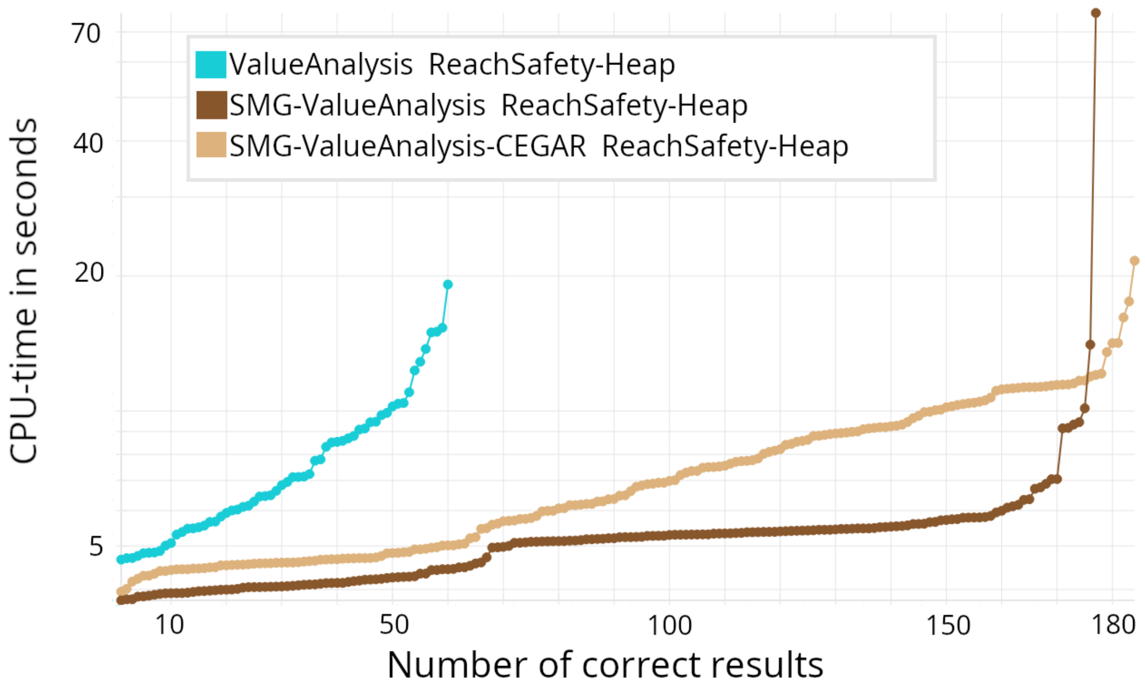


Figure 12: Comparison of the the newly presented value analysis with SMGs, with and without CEGAR approach, and the state-of-the-art value analysis in CPAchecker on the ReachSafety-Heap SV-COMP 22 subset.

memory consumption, the speedup has to be a result of less tasks having a timeout. This could be confirmed by an inspection of the benchmark results.

A visualization of the results is provided as a quantile plot in figure 12. The n-th fastest correct results are ordered along the x-axis, while the y-axis depicts the CPU time consumed logarithmically. It can be seen that the new analysis seems to be faster than the value domain value analysis. But since the tasks are not compared in an ordered fashion, this statement can not be made. However, the new analysis with CEGAR approach seems to consume more time than the non-CEGAR version for correctly solved tasks. This claim can be backed up by the data in table 1.

6.3 Experimental Results for MemSafety-LinkedList

The results of the MemSafety-LinkedList benchmark sets can be seen in table 2. The experimental results show that PredatorHP performs better in every category. PredatorHP does not produce any incorrect results at all, while our new value analysis approach solved 8 tasks incorrectly. However, the analysis of this thesis solved roughly 77% as many programs correct as PredatorHP. In terms of memory consumption, it is evident that PredatorHP performs significantly better than our analysis. The memory consumption of our approach was more than two times as high as PredatorHP. Furthermore, our approach consumed more than twice as much time as PredatorHP overall.

The quantile plot in figure 13 visualizes the results of the LinkedList category. Both anal-

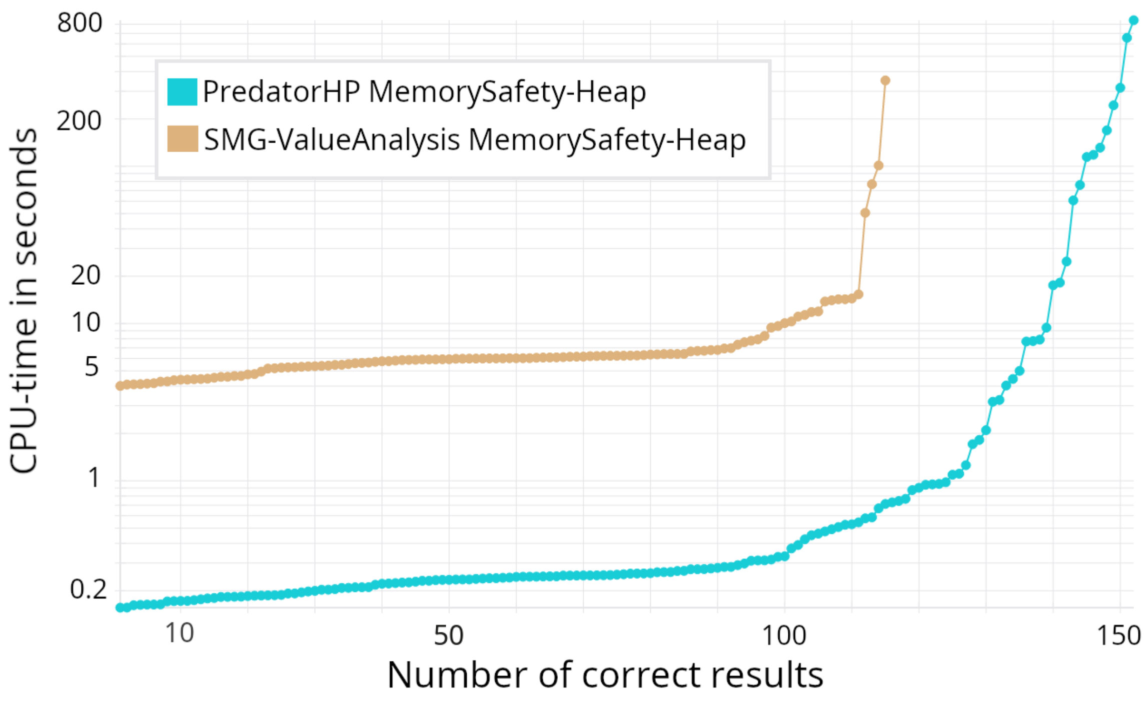


Figure 13: Comparison of the newly presented value analysis with SMGs and PredatorHP on the MemorySafety-LinkedLists SV-COMP 22 subset.

yses can solve most tasks in a few seconds, with our new approach ramp-up time for each of the tasks. This static time increase is partially rooted in the Java Virtual Machine windup time. A later section in this chapter will discuss this in detail however.

6.4 Experimental Results for MemSafety-Heap

Table 3 shows the results for the MemorySafety-Heap category. The MemorySafety-Heap category results are comparable to the results of the linked list category. Our new value anal-

	SMG-ValueAnalysis			PredatorHP		
	status	cputime (s)	memory (MB)	status	cputime (s)	memory (MB)
total	103	22300	45800	103	9090	21400
correct results	68	444	13800	**88**	**72**	**3350**
correct true	42	249	7910	**62**	**34**	**2200**
correct false	**26**	195	5880	**26**	**37**	**1150**
incorrect results	8	80	2000	**0**	-	-
incorrect true	**0**	-	-	**0**	-	-
incorrect false	8	80	2000	**0**	-	-

Table 2: Experimental results of the newly presented value analysis with SMGs and PredatorHP on the MemorySafety-LinkedLists SV-COMP 22 subset.

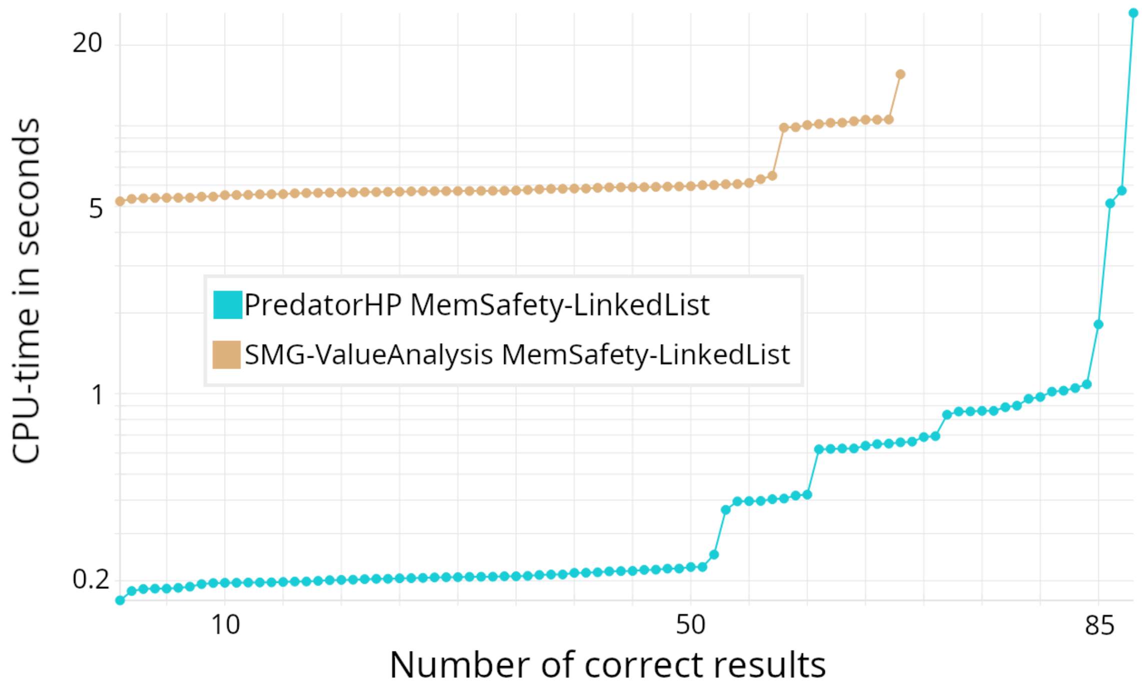


Figure 14: Comparison of the newly presented value analysis with SMGs and PREDATORHP on the MemSafety-Heap SV-COMP 22 subset.

ysis with SMGs solved roughly 76% as many tasks correct as PREDATORHP solved correctly. Also, PREDATORHP did again not produce a single incorrect result, while our new approach produced 28 bad results, with 2 programs incorrectly found to be safe. While the value analysis with SMGs consumed more memory than PREDATORHP, our new approach consumed significantly less CPU time for correctly solved tasks.

Looking at the quantile plot in figure 14 the results of both analyses are shown with the n-th fastest correct results ordered along the x-axis. The y-axis depicts the time consumed logarithmically. It is evident that both analyses are relatively fast for the tasks that they can

	SMG-ValueAnalysis			PREDATORHP		
	status	cputime (s)	memory (MB)	status	cputime (s)	memory (MB)
total	184	27000	78600	184	18300	48700
correct results	115	**1310**	31600	**152**	2900	**16800**
correct true	37	**698**	12200	**68**	2500	**11400**
correct false	78	612	19400	**84**	**399**	**5360**
incorrect results	28	358	7120	**0**	-	-
incorrect true	2	205	2090	**0**	-	-
incorrect false	26	153	5030	**0**	-	-

Table 3: Experimental results of the newly presented value analysis with SMGs and PREDATORHP on the MemorySafety-Heap SV-COMP 22 subset.

solve correctly in this category as well. A ramp-up time of our new analysis, similar to that of the LinkedList category, can be observed.

6.5 Experimental Results for MemSafety Overall

The results for the entire MemSafety benchmark set can be seen in table 4. Since PredatorHP competes only in the two subsets already presented, no comparison is made. The set however shows that the new analysis presented in this paper could solve 85% of the memory-safety tasks correctly. Less then 200 tasks were solved incorrectly. Of remaining roughly 300 tasks, 11 could not be evaluated as the verdict in the SV-COMP benchmarks was missing. 62 tasks could not be solved because a memory allocation was requested with an unknown memory size. Seven tasks failed with errors, which all were caused by CPAchecker, independent of the implementation of the analysis. The remaining tasks all timed out. The table also adds two new CPU time measurements, both measured by CPAchecker internally. CPAchecker CPU time displays the time from the start of CPAchecker to its termination and can be seen as the CPU time independent of the Java Virtual Machine (JVM). The analysis CPU time is the internal measurement of the time spent on the analysis. This does exclude setup necessary for the analysis however, e.g. the creation of the CFA. What can be observed is that the time spent on the analysis alone is much lower than the total CPU time. Also, the mean difference between total CPU time and CPAchecker CPU time is between 1.4 and 1.5 seconds for correctly and incorrectly solved tasks, while the remaining time difference for the tasks that timed out or stopped due to an error is much greater.

	SMG-ValueAnalysis				
	status	cputime (s)	CPAchecker cputime (s)	analysis cputime (s)	memory (MB)
total	3439	261000	250140	237476	983000
correct results	2900	21800	17536	6722	659000
correct true	1503	12100	9872	4270	347000
correct false	1397	9680	7664	2452	312000
incorrect results	196	1500	1198	468	42300
incorrect true	121	960	774	321	26200
incorrect false	75	535	424	147	16100

Table 4: Experimental results of the newly presented value analysis with SMGs on the complete MemorySafety SV-COMP 22 benchmark set.

```
int main(int x) {

int array [10];
  if (x >= 0) {
    array [x] = x;
  }
}
```

Figure 15: Example program 7, showcasing an unsafe program that the value analysis with SMGs determines as safe.

6.6 Discussion of the Evaluation

It is evident that the newly presented approach for memory safety analysis by this thesis did not perform as good as the mature tool PREDATORHP in its results and also resources spent on the analysis. However, our new approach was able to solve around 61% of the correct and 93% of the incorrect tasks that PREDATORHP could solve. One point for why PREDATORHP performs better is that it uses symbolic execution, which is able to abstract more lists compared to our approach, as they may be merged for more than one concrete value. The reasons for the large memory consumption, especially in the LinkedLists category, is most likely rooted in the way linked lists are abstracted, but also the fact that PREDATORHP abstracts sub-SMGs, while our approach does not. Also, the memory consumption gradually increases for linked list tasks if unbounded or symbolically bounded loops are used. Since the analysis splits into at least two abstract states on most loop heads, not counting assumption statements inside loops, a lot of states are created during loop unrolling. At some point the lists used in these tasks are abstracted, leading to the stop operator finally stopping the process of loop unrolling. If this does not happen, time and memory consumption increase if the states are not merged. However, the complete MemSafety benchmark set results showcased that the analysis performs well overall. A problem that could be seen is that the new approach solves a lot of tasks incorrectly however. This problem had already existed in the previous value domain for incorrect unsafe results and was taken over to this analysis. An example for this is provided in figure 4. The reason is the over-approximation of unknown values and as a result state-space exploration that finds violations that are infeasible. For heap memory, this problem extends to include incorrect safe results, as can be seen in figure 15, where the unknown value *x* can not detect the invalid array access for values greater or equal to the size of the array.

Furthermore, the new analysis was compared against a mature tool with many years of active development. Therefore, the overall results of this newly presented value analysis approach using SMGs and CEGAR are promising. The goal of analyzing memory related safety issues fast and without a solver has been reached, even if it does not perform as well as a state-of-the-art tool.

In the ReachSafety category, the new analysis performed significantly better than the state-of-the-art value analysis of CPACHECKER. Still, especially the CEGAR approach did not perform much better than the analysis without it. The reason for this is that the analysis can not deal with symbolic or unknown memory sizes, e.g. *malloc(x);* or *int array[x];* for an unknown value *x*. Since the CEGAR approach starts with an empty precision, tracking no program variables at all, all memory allocation attempts fail. It would be possible to accept allocation of memory with unknown values and simply return an unknown value for all memory operations in the ReachSafety category. As a consequence, the analysis would however also overapproximate all memory related to unknown memory allocation in general, independent of the used variable tracking CEGAR. Also, no values would be able to be saved, as no SMG exists in such a case. Hence why this solution was determined to be infeasible. Since the value analysis does not allow comparisons of symbolic values besides equality, this problem regarding symbolic memory allocation is currently not resolvable for the combination of SMGs and value analysis.

Table 4 lists additional measurements for the spent CPU time per category. The total CPU time depicted in table 2 and table 3, as well as the quantile plots showed that our implementation seems to be slower than the implementation of PREDATORHP. However, there are multiple things to consider, as those tables and graphs compare the total execution time of the programs. The difference in time could be rooted in other causes than differences in the implementation, for example the used programming languages. The JVM, for example, is known to take some time to start the program execution. The provided CPACHECKER CPU time in table 4, measured internally by CPACHECKER, measures the CPU time of the execution of CPACHECKER more accurately. Comparing this time against the total CPU time, we find the time needed to start the JVM. The mean of this difference is 3.2 seconds across all tasks, but only between 1.4 and 1.5 seconds for correctly and incorrectly solved tasks. This means that the tasks that timeout, or end with an error, have a larger impact. A quick investigation of the log-files confirm that tasks with timeouts tend to have much larger differences in CPU times. This however means that the JVM startup time only partially explains the static time difference between our implementation and PREDATORHP. The remaining difference again could be a consequence of the used programming languages, C for PREDATORHP and Java for CPACHECKER, the implementation itself, or both. CPACHECKER provides another internal statistics, in the time spent per analysis, excluding CPACHECKER specific setup, e.g. CFA creation. Here we can see that the analysis itself, which includes the creation and management of the entire SPC and its SMG, accounts for less than a third of the total time spent on correctly or incorrectly solved tasks. As a result, the time difference between our implementation and PREDATORHP would be reduced substantially if only the analysis time would be compared. This comparison is however only partially helpful, as the CPA requires the now excluded setup to run. For this reason, more investigation is required to answer the question as to what causes the time difference between both programs exactly.

7 Conclusion and Future Work

This chapter first sums up the work done, before discussing future work directions.

7.1 Conclusion

This thesis explored a new combination of techniques in value analysis with symbolic memory graphs, with the goal of analyzing C programs. The symbolic memory graphs are used to model the complete memory of an analyzed C program and analyze it for memory errors. Furthermore, CEGAR is used to enhance the analysis by tracking only necessary program variables and heap values. Also, linked lists are abstracted, based on the used SMGs. The analysis is implemented in the CPAchecker framework, and subsequently tested using the benchmarking tool BenchExec. The tests are run using subsets of benchmarks for memory-safety and reach-safety, that are both used in the international competition on software verification 2022. Furthermore, the results for reach-safety are evaluated against the state-of-the-art value analysis of CPAchecker, whilst the results for memory-safety are evaluated against PredatorHP, a comparable memory analysis tool that also uses SMGs and no background solver. While the results in the reach-safety category outperformed the state-of-the-art value analysis in regards to heap memory tasks, problems held back the new analysis, especially in tasks related to memory allocation with non-concrete values. These problems could be identified as a weakness of the combination of value analysis with CEGAR and SMGs. Also, the results showed that the analysis could not outperform PredatorHP, while still solving the majority of memory-safety tasks correctly.

7.2 Future Work

The abstraction of linked lists using memory graphs, seems promising despite the little reasoning capabilities provided by a value analysis. Memory graphs can however abstract other data structures as well. One example would be trees, such as binary trees with or without back-pointers. The current analysis is able to use tree structures and even track values based on the needs of the verification via the presented CEGAR approach. However, the analysis is currently unable to abstract tree structures. Their similarities to lists could be used to extend the analysis not only by their abstraction, but also a refinement based on the used shape refinement approach.

Another possibility to extend the analysis would be with respect to arrays. The current analysis fails to accurately model arrays with symbolic size and also can not abstract arrays at all. Both problems have been tackled before, but most techniques use external SMT solvers. Adapting for example SMT bound techniques[1, 22] similarly to the adaption of interpolation or SMTs from separation logic could yield a new research direction.

Furthermore, the performance of the software analysis as a whole could be improved via the usage of block-abstraction memoization (BAM). Using this technique, the currently analyzed program is divided into blocks that are analyzed separately. Functions or loops are typical examples for those blocks, which would fit with the abstraction of SMGs that typically happens in those blocks. Using the performance gains by incorporating BAM, the analysis might not only perform faster, but could use these performance gains, for example, to use more costly analysis extensions.

Literaturverzeichnis

[1] Mohammad Afzal, A. Asia, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Advaita Datar, Shrawan Kumar, and R. Venkatesh. Veriabs : Verification by abstraction and test generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1138–1141, 2019. `doi:10.1109/ASE.2019.00121`.

[2] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, pages 178–192. Springer, 2007. `doi:10.1007/978-3-540-73368-3_22`.

[3] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *International Conference on Computer Aided Verification*, pages 178–183. Springer, 2011. `doi:10.1007/978-3-642-22110-1_15`.

[4] Dirk Beyer. Competition on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012. `doi:10.1007/978-3-642-28756-5_38`.

[5] Dirk Beyer. Status report on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 373–388. Springer, 2014. `doi:10.1007/978-3-642-54862-8_25`.

[6] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016. `doi:10.1007/978-3-662-49674-9_55`.

[7] Dirk Beyer. Progress on software verification: Sv-comp 2022. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402. Springer, 2022. `doi:10.1007/978-3-030-99527-0_20`.

[8] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 721–733, 2015. `doi:10.1145/2786805.2786867`.

[9] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A unifying view on smt-based software verification. *Journal of automated reasoning*, 60(3):299–335, 2018. `doi:10.1007/s10817-017-9432-6`.

[10] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007. `doi:10.1007/s10009-007-0044-z`.

[11] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *International Conference on Computer Aided Verification*, pages 532–546. Springer, 2006. `doi:10.1007/11817963_48`.

[12] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007. `doi:10.1007/978-3-540-73368-3_51`.

[13] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008. `doi:10.1109/ASE.2008.13`.

[14] Dirk Beyer, Thomas A Henzinger, Grégory Théoduloz, and Damien Zufferey. Shape refinement through explicit heap analysis. In *International Conference on Fundamental Approaches to Software Engineering*, pages 263–277. Springer, 2010. `doi:10.1007/978-3-642-12029-9_19`.

[15] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, LNCS 6806, pages 184–190. Springer-Verlag, Heidelberg, 2011. URL: `https://cpachecker.sosy-lab.org`, `doi:10.1007/978-3-642-22110-1_16`.

[16] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013. `doi:10.1007/978-3-642-37057-1_11`.

[17] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In *International SPIN Workshop on Model Checking of Software*, pages 20–38. Springer, 2015. `doi:10.1007/978-3-319-23404-5_3`.

[18] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Sliced path prefixes: An effective method to enable refinement selection. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 228–243. Springer, 2015. `doi:10.1007/978-3-319-19195-9_15`.

[19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. `doi:https://doi.org/10.1007/s10009-017-0469-y`.

[20] Thomas Bunk. LTL software model checking in CPAchecker. Master's Thesis, LMU Munich, Software Systems Lab, 2019.

[21] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. `doi:10.1109/ICIEA.2016.7603701`.

[22] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 22–39. Springer, 2020. `doi:10.1007/978-3-030-45190-5_2`.

[23] Marek Chalupa, Tomáš Jašek, Jakub Novák, Anna Řechtáčková, Veronika Šoková, and Jan Strejček. Symbiotic 8: beyond symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 453–457. Springer, 2021. `doi:10.1007/978-3-030-72013-1_31`.

[24] Marek Chalupa, Jan Strejček, and Martina Vitovská. Joint forces for memory safety checking. In *International Symposium on Model Checking Software*, pages 115–132. Springer, 2018. `doi:10.1007/978-3-319-94111-0_7`.

[25] David R Chase, Mark Wegman, and F Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, 1990. `doi:10.1145/93548.93585`.

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

[27] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *International Static Analysis Symposium*, pages 215–237. Springer, 2013. `doi:10.1007/978-3-642-38856-9_13`.

[28] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A shape analyzer based on symbolic memory graphs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 412–414. Springer, 2014. `doi:10.1007/978-3-642-54862-8_33`.

[29] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully automated shape analysis based on forest automata. In *International Conference on Computer Aided Verification*, pages 740–755. Springer, 2013. `doi:10.1007/978-3-642-39799-8_52`.

[30] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009. `doi:10.1145/1592434.1592438`.

[31] Neil D Jones and Steven S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 66–74, 1982. `doi:10.1145/582153.582161`.

[32] Petr Muller, Petr Peringer, and Tomáš Vojnar. Predator hunting party (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–446. Springer, 2015. `doi:10.1007/978-3-662-46681-0_40`.

[33] Stefan Weinzierl. Configurable pointer-alias analysis for cpachecker.

[34] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In *International Conference on Compiler Construction*, pages 1–17. Springer, 2000. `doi:10.1007/3-540-46423-9_1`.

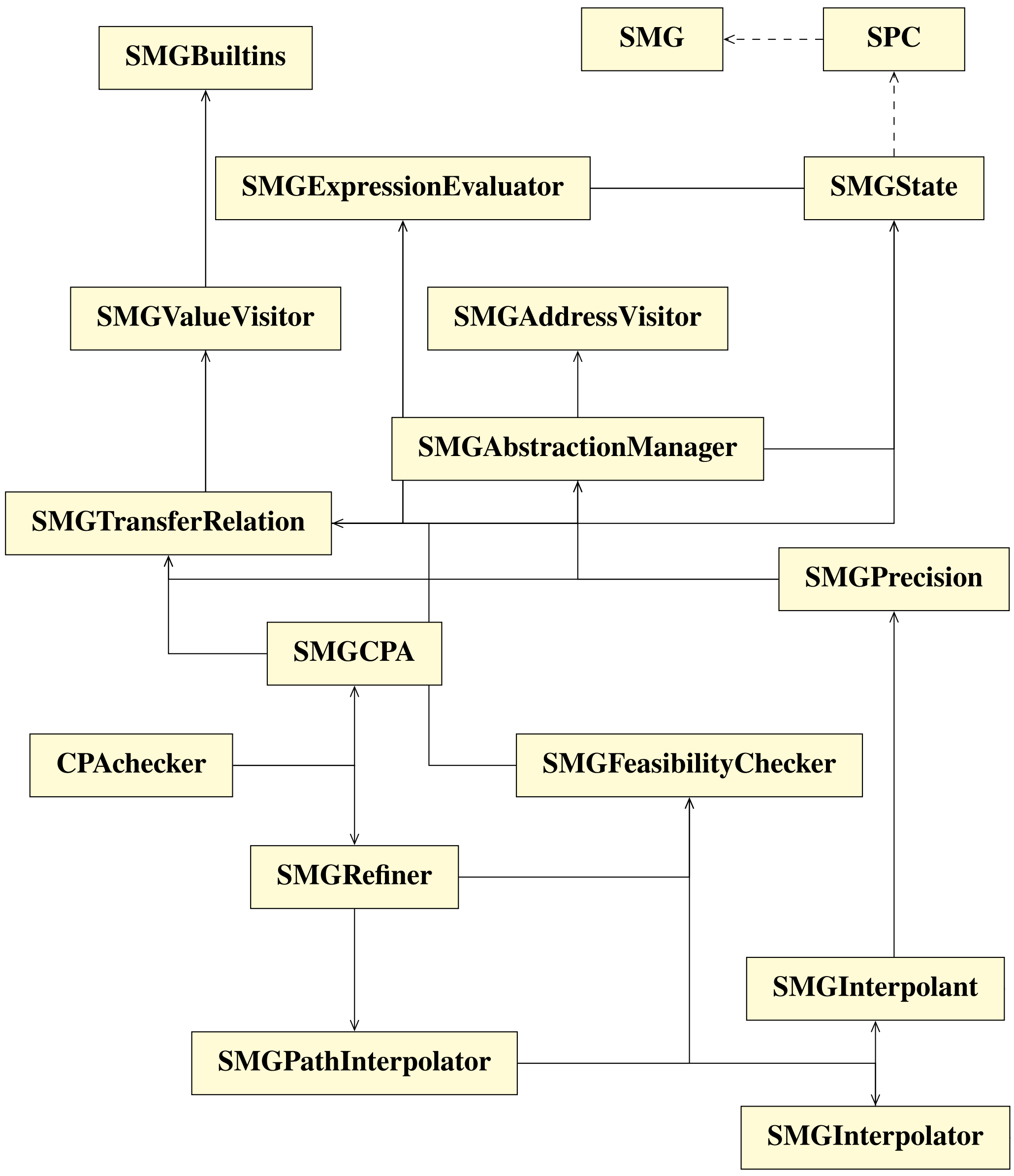


Figure 16: The Abstract Class Diagram of the Analysis and the CEGAR Refiner.

Appendix

7.1 Overall Implementation Structure

An abstract overview of the implementation in CPAchecker is provided in figure 16. Helper classes, including, for example, tuple classes that bundle return objects such as a value and a new state, are omitted. The class *SMG* abstractly bundles all SMG components, including values and edges. It is used by the symbolic program configuration, mediating between the SMG and the rest of the analysis. The intersection of SMG related classes and operations to the value analysis part is held at a minimum. The classes *SMGRefiner*, *SMGPathInterpolator*, *SMGInterpolant*, *SMGFeasibilityChecker* and *SMGInterpolator* are used as a simple representation of the CEGAR and interpolation approach. Their super-classes, generic versions of the classes that are part of the value analysis, are not shown. CPAchecker itself is symbolized as a class, because of its complexity. The CPACHECKER class also includes the CEGAR procedure that restarts the analysis and delegates to the refiner.

Danksagung

Zunächst möchte ich mich bei Prof. Dirk Beyer für die Chance bedanken, meine Masterarbeit am Software and Computational Systems Lab der Ludwig-Maximilians-Universität München schreiben zu dürfen. Aber auch dafür, dass Prof. Dr. Beyer immer ein offenes Ohr hatte und sich die Zeit nahm mich zu unterstützen.
Einen besonderen Dank möchte ich Thomas Bunk aussprechen. Auf seine Unterstützung und seinen guten Rat konnte ich mich immer verlassen.
Weiterhin möchte ich mich bei Philipp Wendler, Thomas Lemberger, Gidon Ernst und allen anderen für ihre Unterstützung, Motivation und Ausdauer meinen Redestrom zu ertragen bedanken. Bei Karlheinz Friedberger bedanke ich mich dafür, dass er mich für diese Arbeit empfahl, sowie zusätzlich bei Luisa Mauerer für ihre seelische Unterstützung.
Zuletzt will ich noch Jutta und Roland Baier danken, deren Unterstützung mir ermöglicht hat erfolgreich zu studieren und letztendlich diese Arbeit zu verfassen.