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

Native pointer support in abstraction-based model checking of concurrent systems

Scientific Students' Association Report

Author:

Botond Sisák

Advisor:

Levente Bajczi
dr. András Vörös

2023

Contents

Kivonat

A biztonságkritikus rendszerek nagymértékben támaszkodnak szoftveres megoldásokra az elvárt működésük biztosítása érdekében. Ezen szoftveres megoldásoknak minden esetben megfelelően kell működniük, mivel meghibásodásuk veszélyes helyzetekhez vezethet. A rendszer tesztelésével találhatóak a szoftverben hibák; azonban a tesztelés nem kimerítő módszer, nem képes önmagában bizonyítani a szoftver hibamentességét. A formális verifikáció a szoftver lehetséges viselkedésformáinak kimerítő bejárására szolgáló technika, de a lehetséges állapotok nagy vagy akár végtelen száma megakadályozhatja a sikeres verifikációt.

A CEGAR (Counterexample Guided Abstraction Refinement) különböző absztrakciókat használ a komplex adatok kezelésére a szoftver verifikálásakor. A CEGAR számos felhasználási esetben bizonyította hatékonyságát. Eddig azonban nem volt hatékony pointer-támogatás.

A beágyazott rendszerekben használt programozási nyelvek gyakran erősen támaszkodnak pointerekre a hatékony memóriakezelés és adatmanipuláció érdekében. A pointerek használata azonban az indirekciók bevezetésével növeli a program bonyolultságát, ami nagyban megnehezíti az ilyen programok verifikációját. Annak ellenére, hogy a beágyazott és biztonságkritikus rendszerekben többnyire kerülik a pointerek használatát, valós programokban teljesítményi és funkcionalitási elvárások miatt mégis gyakran szükséges a használatuk.

A beágyazott rendszerekhez rendelkezésre álló többmagos processzorok elterjedésével a többszálas programok is egyre gyakoribbá válnak. Az ilyen párhuzamos programok verifikációja még nagyobb kihívást jelent a szálak nem-determinisztikus átlapolódása miatt. A részleges rendezés redukció (Partial order reduction, POR) hatékony módszer a párhuzamosság kezelésére a verifikáció során. Azonban a pointerek bevezetésével ezt a technikát át kell alakítani, hogy képes legyen kezelni az újonnan megjelenő lehetséges viselkedésformákat.

A korábbi verifikációs megoldások a pointer-analízist a megoldó számára biztosított problémába kódolták. Célunk, hogy a problémát a verifikációs algoritmus szintjére emeljük: újszerű megoldásunk a CEGAR ciklus állapottér-bejárási részében próbálja megoldani a pointer analízis problémáját, lehetővé téve egy pontosabb, az absztrakció-alapú verifikáció által támogatott módszert, mindezt integrálva a párhuzamosság kezelésére szolgáló részleges rendezés redukcióval. Kiértékeljük a javasolt módszer teljesítményét SV-COMP benchmarkokon, és célunk az SV-COMP 2023-as nevezés részeként szerepeltetni.

Abstract

Safety-critical systems rely heavily on software-based solutions to provide the required functionalities. Such software solutions have to operate correctly, as their failure might lead to dangerous situations. Testing is traditionally used to find bugs in software; however, testing is not an exhaustive method, it is not able to prove the absence of errors in software alone. Formal verification is a technique to exhaustively explore the possible behaviours of the software, but the large or even infinite number of possible states might prevent successful verification.

Counterexample Guided Abstraction Refinement (CEGAR) uses various abstractions to handle complex data when verifying software. CEGAR proved its efficiency in many use cases. However, there was no efficient pointer support so far.

Programming languages used in embedded systems often heavily rely on pointers for efficient memory management and data manipulation. However, the use of pointers introduces an additional level of indirection and complexity that makes verifying such programs challenging. Despite the preference to avoid pointers in embedded and safety-critical systems, real-world programs often necessitate the use of pointers for performance and functionality reasons.

With the spread of multi-core processors available for embedded systems, multi-threaded programs are becoming more common. The verification of such concurrent programs is even more challenging due to the non-deterministic interleaving of threads. Partial order reduction (POR) is an effective method to handle concurrency in verification. However, with the introduction of pointers, this technique must be adapted to handle the additional possible behaviours introduced.

Former solutions for the verification encoded the points-to analysis into the problem provided for the solver. Our goal is to lift the problem to the level of the verification algorithm: our novel solution tries to solve the pointer analysis problem at the state space traversal part of the CEGAR loop, enabling a more precise method supported by the abstraction-based verification, while also integrating it with partial order reduction to handle concurrency. We evaluate the performance of the proposed method on SV-COMP benchmarks, and our goal is to include it as part of a submission for SV-COMP 2023.

Chapter 1

Introduction

Our everyday life is surrounded by software-based solutions. Many times a day, we place our trust, and even our lives in the hands of software usually running on embedded systems. These systems are responsible for the correct operation of many safety-critical applications, such as medical devices, automotive systems, or aircraft.

With the spread of multi-core processors, multi-threaded programs are also becoming more common on embedded systems to take advantage of the increased performance.

The correct operation of these systems is crucial, as their failure might lead to catastrophic outcomes. Therefore, it is essential to ensure that these critical systems operate correctly at all times.

Pointers are an essential feature of low-level programming languages used in embedded systems. They are used for efficient memory management and data manipulation.

However, the design and implementation of programs using pointers and concurrency are challenging. The use of pointers introduces an additional level of indirection and complexity, while the non-deterministic interleaving of threads requires special attention from the programmer. It might become extremely challenging for them to consider all possible behaviours of a concurrent program with the added complexity of pointers.

Testing is traditionally used to find bugs in software; however, testing is not an exhaustive method, it is not able to prove the absence of errors in software alone. To ensure the correctness of software, all possible behaviours of the program must be explored.

Formal verification is a technique used to prove safety properties of software, for example, whether an error state is reachable from the initial state of the program. This is done by exhaustively exploring the possible states of the program. However, this quickly becomes infeasible due to the large or even infinite number of possible states and the exponentially growing possible executions of concurrent programs.

This problem is usually solved by using abstraction. Abstraction is a technique to reduce the complexity of the problem by removing unrelevant details. With the reduced complexity, there is a chance that the verification of the abstracted program becomes feasible.

Counterexample Guided Abstraction Refinement (CEGAR) is an efficient technique that uses abstraction and refinement in an iterative manner to come to a conclusion about the correctness of the program.

With the introduction of pointers, the verification process becomes more challenging. The use of pointers introduces additional possible behaviours to the program, which must be considered during the verification.

As pointers are an essential feature of low-level programming languages used in embedded systems, the need for a solution to handle pointers in verification is evident.

Verification of concurrent programs is even more challenging due to many possible interleavings of threads. Partial order reduction (POR) is an effective method to handle concurrency in verification. However, with the introduction of pointers inside concurrent programs, this technique must be adapted to handle the additional possible behaviours to ensure the correctness of the verification.

This work aims to develop a novel solution to handle pointers in a CEGAR-based verification algorithm. This solution solves the pointer analysis problem at the state space traversal part of the CEGAR loop, featuring a very precise, but efficient method of tracking pointers.

Additionally, this solution is integrated with partial order reduction to handle concurrency, enabling the verification of concurrent programs with pointers.

I have implemented this solution in the open-source verification tool THETA, and I have evaluated its performance on SV-COMP benchmarks. The contribution leads to a substantial increase in the number of successfully verified benchmarks.

The rest of this report is structured as follows. In Chapter 2, the essential background information is presented, basic concepts of formal verification, CEGAR, and partial order reduction are introduced. Chapter 3 presents the main contribution of this work, a dynamic pointer analysis method integrated into the CEGAR loop. Chapter 4 describes the integration of the pointer analysis method with partial order reduction. Chapter 5 evaluates the performance of the implemented solution on SV-COMP benchmarks. Finally, Chapter 6 concludes the report and discusses possible future work.

Chapter 2

Background

This chapter introduces the background knowledge required to understand the rest of this report.

2.1 Formal Software Verification

Formal verification is a method for ensuring the correctness of safety-critical software by using math and logic-based techniques. Formal verification can mathematically prove certain properties of a program, which is something that cannot be achieved through testing only.

Such properties to be verified can include: [3]

- *Memory safety*: detecting invalid memory accesses or other memory-allocation problems
- *Termination*: detecting if the program will terminate in all its executions
- *Reachability*: detecting if an unsafe state is reachable from the initial state (such as a division by zero error, a pre-specified assertion failure, etc.)

In this work, safety properties are always assumed to be reachability problems, i.e., whether an unsafe state is reachable during the execution of the program.

Model checking is a technique used in formal verification to exhaustively analyze system behaviours and ensure compliance with specified properties or requirements. By exploring all possible execution paths, model checking can detect design flaws and potential bugs in complex software systems.

Model checking involves constructing a formal model of the system capturing its behaviour and structure. Through the traversal of all possible states, model checking verifies the pre-specified properties. In practice, exploring the entire state space is often infeasible, and model checking tools must address this issue to provide a useful analysis. [3]

2.2 C Programming Language

The C programming language [11], created in the early 1970s, known for its speed, efficiency, and portability, stands as one of the most widely used programming languages in the world. It is a general-purpose, imperative language with a static type system. Its simplicity and low-level characteristics have established it as a popular option for performance-critical tasks, like embedded systems, rendering it appropriate for high-risk and safety-critical applications that require formal verification.

The language consists of many constructs. For simplicity, this work will only consider a subset of the language, consisting of variables of the primitive types (e.g., `int`, `float`, `char`, etc.), operations on these variables (e.g., arithmetic operations, assignment, etc.), basic control-flow constructs (e.g., `if`, `while`, `for`, etc.), function calls, and pointers, not including pointer arithmetic. This subset is sufficient to demonstrate the concepts discussed in this work.

2.2.1 Pointers in C

Pointers allow storing addresses of memory in variables. They are a powerful tool used widely in many programs. A pointer is essentially a variable, that is able to hold a memory address, usually the address of another variable or struct.

In the C language, pointers are denoted by the $*$ sign. Pointers also have a type, which corresponds to the type of variable stored at the address of the pointer.

A crucial operator when working with pointers is the & symbol, which is able to get the memory address of a variable. For example, the expression $\&i$ would yield the memory address of the variable i. A basic usage of this is shown in Listing 2.1.

```c
int i = 5;
int* p = &i;
```

Listing 2.1: Basic pointer usage in C. i is a normal integer variable, p is a pointer to an integer, now pointing to i.

The syntax for dereferencing pointers also uses the $*$ symbol. Dereferencing a pointer means accessing the value stored at the address of the pointer. For example, the expression $*p$ would yield the value stored at the address of the pointer p. Overwriting the value stored at the address of the pointer is also possible using the $*$ operator. For example, the expression $*p = 5$ would overwrite the value stored at the address of the pointer p with the value 5. A small example of dereferencing pointers is shown in Listing 2.2.

```c
1 int j = 5;
2 int* q = &j;
3 *q = 10;
```

Listing 2.2: Dereferencing pointers in C. The value of j will be 10 after line 3.

Pointers are also able to point to other pointers. For example, a pointer `int**` pp is a pointer that holds the address of a `int*` pointer, thus it can be dereferenced twice to get a value. This is called the *level of indirection* of the pointer.

Pointers are also an essential tool for dynamic memory allocation. The `malloc` function is used to allocate memory on the heap and returns a pointer to the allocated memory. The `free` function is used to free the allocated memory. *Supporting dynamic memory allocation is out of scope for this work.*

2.3 Pointer Analysis

The term *pointer analysis* [19] generally refers to the problem of collecting information about pointers in a program, either statically or dynamically. Pointer analysis techniques can be categorized based on several properties.

May and must analysis

In May analysis, if there exists at least one program path, where a points-to relation holds, it will be considered. In Must analysis, a relation is only considered if it holds at all paths of the program. Therefore, may analysis provides an over-approximation of the points-to relations, while must analysis provides an under-approximation.

Flow-insensitive and flow-sensitive analysis

Another aspect of imprecision is whether the analysis considers the control flow of the program. If it does, we call it flow-sensitive analysis, if it does not, meaning that it considers that the lines of the program could be executed in any order, we call it flow-insensitive analysis.

Intra-procedural and inter-procedural analysis

Intra-procedural analysis makes worst-call assumptions about function calls without evaluating them. In an inter-procedural analysis, all function calls are evaluated.

Context-insensitive and context-sensitive analysis

Context-insensitive analysis does not differentiate between different call sites of the same function, while context-sensitive analysis does. [19]

2.3.1 Andersen's Pointer Analysis

Andersen's pointer analysis [2] is a flow- and context-insensitive, over-estimating (may) approach. The algorithm first constructs a constraint system, consisting of subset constraints. Solving this constraint system yields the may-point-to sets.

The rules for constructing the constraint system are the following, where $pts(x)$ denotes a may-point-to set of x:

1. Referencing: p = &i $\Rightarrow i \in pts(p)$
2. Dereferencing read: p = *q $\Rightarrow \forall x \in pts(q) : pts(x) \subseteq pts(p)$
3. Dereferencing write: *p = q $\Rightarrow \forall x \in pts(p) : pts(q) \subseteq pts(x)$
4. Aliasing: p = q $\Rightarrow pts(q) \subseteq pts(p)$

Creating and solving the constraint system has a runtime complexity of $O(n^3)$ in the worst-case [1], where n is the size of the input program.

2.3.2 Steensgaard's Pointer Analysis

Steensgaard's pointer analysis [14] is also a flow- and context-insensitive, over-estimating (may) approach. It is based on equality constraints, and has a speed advantage over Andersen's algorithm, at the cost of higher imprecision. The algorithm runs in *almost* linear time, with a runtime complexity of $O(n\alpha(n, n))$ [1], where n is the size of the input program, α is the reverse Ackermann function, whose value is close to 1 for large values of n, hence the author's claim of *almost linear time* in the original publication.

The approach is based on the concept of abstract locations. First, every variable (including pointers) identifies an abstract location, but later an abstract location can contain multiple variables. The restriction is that an abstract location that contains a pointer can only point to one abstract location. Therefore, in a case where a pointer may point to two different locations in the memory during the execution of the program, those two corresponding abstract locations will be joined, thus the pointer's abstract location points to only one abstract location, which now includes (at least) two variables.

Abstract locations are to be implemented in a union-find or disjoint-set data structure, from which the complexity's α function comes. This data structure defines the *unify* method to merge two sets.

The rules for handling pointer operations are the following:

1. Referencing: `p = &i` $\Rightarrow join(*p, i)$
2. Dereferencing read: `p = *q` $\Rightarrow join(*p, **q)$
3. Dereferencing write: `*p = q` $\Rightarrow join(**p, *q)$
4. Aliasing: `p = q` $\Rightarrow join(*p, *q)$

The method *join* for the abstract locations $p1$ and $p2$ is as defined in Listing 2.3.

```
join(p1, p2)
  if (p1 == p2)
    return
  p1next = *p1;
  p2next = *p2;
  unify(p1, p2)
  join(p1next, p2next)
```

Listing 2.3: Pseudocode for Steensgaard's *join* method

2.4 Control-Flow Automata

The programs discussed in this work are in the form of source code, written in the C programming language. To support formal verification and pointer analysis on these programs, a formal representation must be defined. Control-Flow Automata (CFA) is a formalism widely used to model programs for verification purposes.

Definition 1 (Control-Flow Automata). A Control-Flow Automaton [10] is a tuple $CFA = (V, L, l_0, E)$, where

- $V = \{v_1, v_2, ..., v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, ..., D_{v_n}$
- L is a set of program *locations* modeling the program counter

- $l_0 \in L$ is the *initial* program location
- $E \subseteq L \times Ops \times L$ is a set of directed edges representing the *operations* that are executed when control flows from the source location to the target.

$op \in Ops$: An assumption of a predicate over V asserting its truth (i.e., an execution is only legal if the predicate is fulfilled), or an assignment of a new value to a $v \in V$. A special kind of assignment has the form *havoc* v, which assigns a non-deterministic value to v. [3] ▪

Example 1. Given a program in Listing 2.4, the corresponding CFA is shown in Figure 2.1. The distinguished location l_0 is the initial location, and l_E is the location corresponding to an assertion failure. Guards and assignments are shown on the edges of the CFA.

```
int x = 0;
int i = 0;
while (i < 100) {
if (x == 0) x = 1;
  else x = 0;
i++;
}
assert (x <= 1);
```

Listing 2.4: Example C program with various elements of structured programming. [10]

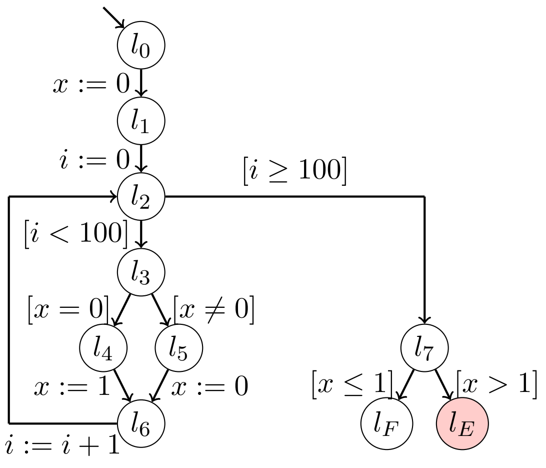

Figure 2.1: CFA representation of Listing 2.4. The distinguished location l_E corresponds to an assertion failure. [10]

2.5 State space

The execution of a program can be modeled as a sequence of operations that modify the state of the program (e.g., the values of the variables). The state space of a program is the set of all possible states that the program can be in during its execution.

Definition 2 (State space). The *state space* [18] is a tuple (S, t), where:

- S is a set of states.
- $t : S \times Ops \to S$ is a transition function, where Ops is a set of operations. ▪

In the context of a CFA, the set of states $S = L \times D_{v_1} \times D_{v_2} \times ... \times D_{v_n}$, where L is the set of locations, and $D_{v_1}, D_{v_2}, ..., D_{v_n}$ are the domains of the variables [18]. This means that the state space of a CFA is the set of all possible combinations of locations and variable values.

2.6 State space abstraction

It is easy to recognize that the state space of a program can be very large. For example, if we declare a 32-bit integer variable whose value is non-deterministic (e.g., user input), then the size of the state space grows by a factor of 2^{32}. With each additional variable, the size of the state space grows exponentially. This problem is known as the *state space explosion problem.*

Abstraction is a technique used to reduce the size of the state space. It is based on the idea that not all information might be relevant to the verification process, certain information can be *abstracted away*, i.e., ignored, thus creating an abstract state space. An *abstract state* can represent multiple concrete states, making it possibly much smaller than the original state space.

For example, if we have a 32-bit integer variable x, the original state space contains 2^{32} states, where each state is a possible value of x. However, if we only care if x is positive or negative, we can abstract away the exact value of x, and only keep track of whether x is smaller or greater than zero. This newly formed abstract state space now only contains two states, and thus is much smaller than the original state space.

Definition 3 (State space abstraction). A *state space abstraction* [18] is a tuple $(S_{abstract}, c, \Pi, t_{abstract})$, where:

- $S_{abstract}$ is a set of abstract states. An abstract state is a set of concrete states. Two special abstract states are $\top$ (top), which represents all possible concrete states, and $\bot$ (bottom), which represents no concrete states.
- $c : S_{abstract} \to 2^{S_{concrete}}$ is a concretization function, that maps an abstract state to a set of concrete states.
- Π is the precision of the abstraction. The exact definition of precision depends on the type of abstraction.
- $t_{abstract} : S_{abstract} \times Ops \to 2^{S_{abstract}}$ is an abstract transition function, that given an abstract state and an operation, returns the successor abstract states. ▪

Two frequently used types of abstraction are *explicit value abstraction* and *predicate abstraction.*

- Explicit value abstraction defines the current abstraction precision as a set of *tracked* variables, i.e., the variables that are considered relevant to the verification process. All untracked variables' values are unknown in all abstract states.
- In predicate abstraction, the precision is defined as a set of tracked predicates over the variables (e.g., $x > 5$, or $y = z$).

Definition 4 (Abstract error state). An abstract state $S_{abstract}$ is an *abstract error state* if $\exists s \in c(S_{abstract}) : s$ that violates the safety property. ▪

Given these definitions, we can define a partial order on the abstract states.

Definition 5 (Partial order on abstract states). $\preceq \subseteq S_{abstract} \times S_{abstract} : A \preceq B$ if $c(A) \subseteq c(B)$ [18] ▪

2.7 Abstract Reachability Graph

An *Abstract Reachability Graph* (ARG) [6, 18] is a graph representation of the execution of a program on the abstract state space.

Definition 6 (Abstract Reachability Graph). An *Abstract Reachability Graph* (ARG) is a graph $ARG = (N, L, E, C)$, where

- N: set of nodes in the graph
- $L : N \to S_{abstract}$: labeling that assigns an abstract state to a given node
- $E \subseteq N \times Ops \times \{1\} \times N$: set of directed edges in the graph. $E = \{(n, op, 1, m) | n \in N, op \in Ops, s \in t_{abstract}(L(n), op), m \in N, L(m) = s\}$
- $C \subseteq N \times \{0\} \times N$: set of *cover edges* in the graph. A cover edge can be inserted from node n to node m if $L(n) \preceq L(m)$. ▪

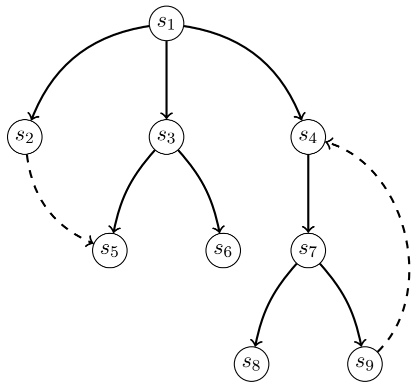

Figure 2.2: An example ARG. Cover edges are shown with dashed lines.

This means that an ARG is a directed graph, where the nodes represent abstract states, and the edges represent operations. Edges can also contain *guards*, which are predicates over the variables that only allow the edge to be taken if the predicate is true. A complete ARG represents all possible executions of the program on the abstract state space. It can contain traces (routes) to abstract error states, however, these traces may not be feasible in the concrete program, as the abstract state space is an over-approximation of the concrete state space. An example of an ARG is shown in Figure 2.2.

2.8 Boolean satisfiability problem (SAT)

The *Boolean satisfiability problem* (SAT) is the problem of determining whether there exists an assignment of boolean values to variables in a given boolean formula, so that the formula evaluates to true.

Example 2. Consider the formula $(a \lor b) \land (\neg a \lor \neg c)$. This formula is satisfiable, as the assignment $a = true,\ b = true,\ c = false$ satisfies the formula.

In contrast, the formula $(a \land \neg a)$ is unsatisfiable, as there is no assignment of values to a that would satisfy it.

The SAT problem was the first problem shown to be *NP-complete* [8], meaning that there is no known algorithm that can solve the problem efficiently (i.e., in polynomial time).

2.9 Satisfiability Modulo Theories (SMT)

An *SMT-problem (Satisfiability Modulo Theory)* [4] is a decision problem for logical formulas, in which, when given a first-order formula and the theories used in it, a solver can decide whether there exists a substitution of variables in the formula to concrete values so, after the substitution, the formula evaluates to true; or the formula is unsatisfiable. [7]

Example 3. Given the formula $(x < 5 \land x \geq 3 \land y > 7)$, where the symbols x and y have the domain of *integers* ($\mathbb{Z}$), an example *model* (a substitution of variables) would be $(x = 4);\ (y = 8)$, as substituting these values for x and y in the formula would result in a true statement. As there exists a model for the formula, the formula is satisfiable.

In contrast, the formula $(x < 4 \land x > 5)$ is unsatisfiable, as no substitution of x would result in a true statement.

SMT problems can be reduced to SAT problems, and thus are also NP-complete. However, specialized *SMT solver* software [13] exist that are usually able to solve SMT problems more efficiently than the theoretical worst-case complexity.

2.10 Counterexample Guided Abstraction Refinement

Counterexample Guided Abstraction Refinement (CEGAR) is an abstraction-based iterative model-checking technique.

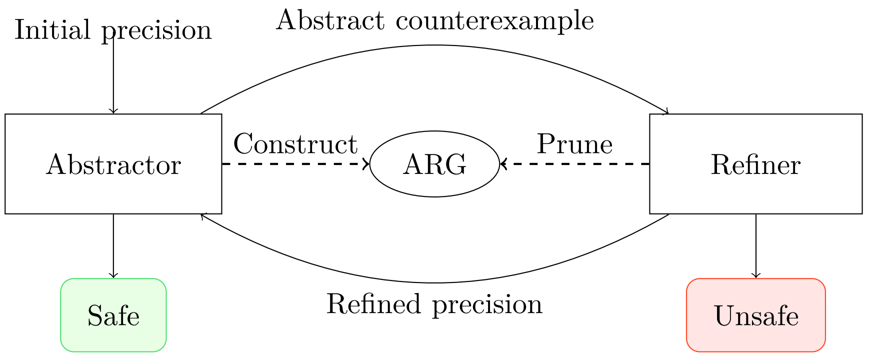

Figure 2.3: The CEGAR-loop [7]

The CEGAR algorithm consists of two main components: the *Abstractor* and the *Refiner*.

The Abstractor is responsible for constructing an ARG of the program along the current precision and finding an error path in the ARG. This error path is called an abstract counterexample. As the abstract state space is an over-approximation of the concrete state space, this abstract counterexample may not be feasible in the concrete program.

The Refiner is responsible for checking the feasibility of the abstract counterexample in the concrete program. This is achieved by transforming the operations on the trace of the abstract counterexample into a logical formula and checking its satisfiability with an SMT solver. If the counterexample is not feasible (or *spurious*), the Refiner is also responsible for reducing the amount of abstraction by refining the precision, so that the same error path cannot be found in the ARG again. This new precision is returned once again to the Abstractor, making it an iterative process, as shown in Figure 2.3.

The loop terminates either when:

a) the Abstractor is unable to find an error path in the ARG. As the ARG is an over-approximation of the concrete program, the program itself is safe.

b) the Refiner finds an abstract counterexample that is feasible in the concrete program. This means that an actual execution path of the program has been found that results in an error, thus the program is unsafe.

The abstraction of the state space is refined until the program is proven to be either safe or unsafe.

2.11 Concurrent Software

Concurrent software is able to execute multiple tasks at the same time. This is achieved by executing the tasks on multiple threads. Threads are independent sequences of instructions that can be executed concurrently, but threads of the same process share the same memory space, thus their execution can influence each other by accessing and modifying the same memory locations.

2.11.1 Control-Flow Automata for Concurrent Software

In order to also model and verify multi-threaded, concurrent programs, the CFA formalism has been extended, resulting in the eXtended Control-Flow Automata (XCFA). [3]

Definition 7 (eXtended Control Flow Automaton). An *eXtended Control Flow Automaton* (XCFA) [15] is a tuple $XCFA = (V_g, P)$, where:

- V_g is a set of global variables
- P is a set of *processes.* A process is a tuple $p = (V_l, CFA)$, where:
    - V_l is a set of local variables
    - CFA is a CFA (whose variables are $V \subseteq V_g \cup V_l$) extended with the following operations: *start thread* and *join thread*, *atomic begin* and *atomic end.* ▪

A *start thread* operation creates a new process p_{new} (and marks $p_{new} \in P$ as an active process) and starts the concurrent execution of the new process at its initial CFA location. A *join thread* operation is disabled until the specified process p terminates: after p has terminated, the join thread operation can be fired. *Atomic begin*, and *atomic end* operations mark atomic blocks: while the execution of a process is inside an atomic block, all other processes are disabled. [15]

```c
#include <pthread.h>

int x = 0;

void* thread1(void* arg) {
  x = 1;
}

void* thread2(void* arg) {
  x = 2;
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, thread1, NULL);
  pthread_create(&t2, NULL, thread2, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  assert(x == 2);
}
```

Listing 2.5: Example C program with pthreads. The program creates two threads, `t1` and `t2`, and each thread writes a different value to the global variable `x`. Then the termination of both threads is waited for with `pthread_join`. The assertion at the end of the program cannot be guaranteed to hold.

2.11.2 Concurrency in C Programs

The C programming language standard does not support concurrency. *POSIX threads* (or pthreads) is a standard for threads, and an implementation of this standard is available to be used in C programs with the *pthread* library.

The library provides the following functions for creating and managing threads:

- `pthread_create`: creates a new thread
- `pthread_join`: waits for a thread to terminate
- `pthread_mutex_lock`: locks a mutex
- `pthread_mutex_unlock`: unlocks a mutex

2.12 Partial Order Reduction

Concurrency adds a new level of complexity to the verification of software.

Consider the program in Listing 2.5. The program creates two threads, `t1` and `t2`, and each thread writes a different value to the global variable `x`. However, the order in which the threads are executed is non-deterministic, thus the value of `x` after joining the threads can be either 1 or 2, therefore the assertion at the end of the program can fail.

One way to deal with concurrency is using the *interleaving* semantics, which uses overlapping traces of the threads to model the execution of the program, adhering to the assumption that any of the threads may execute at any point in time. This means that to verify the program, all possible interleavings of the threads must be considered. Checking all possible interleavings could also quickly become infeasible as the number of options grows exponentially with the number of threads and operations. [3]

A technique to combat this problem widely used in the verification of concurrent software is *Partial Order Reduction* (POR).

Partial Order Reduction is a well-known technique for avoiding the exploration of redundant thread interleavings in the verification of a multi-threaded program [9]. Its key idea is to define an equivalence relation on traces and explore a single representative (or as few as possible) from each equivalence class. Traces are defined to be equivalent if they can be obtained from each other by successively swapping adjacent independent actions. An equivalence class is called a Mazurkiewicz trace [12]. Intuitively, if adjacent independent actions are swapped, the outcome will remain the same: by exploring a single trace from each equivalence class, we still cover all behaviours of the system.[16]

Essentially, with POR, we define a *dependency relation* between the operations of the threads. Actions can be *dependent* of each other, for example, if they write the same global variable. From these dependencies, the algorithm is able to deduce which thread interleavings are in the same equivalence class, thus reducing the number of interleavings that need to be actually checked for safety.

Two actions of different threads are dependent of each other if they access the same variable, and at least one of them is a write operation. Actions of the same thread are always dependent of each other.

$vars(\alpha)$ denotes the set of variables referenced by the action α.

Definition 8 (Syntactic Dependency Relation). A *Syntactic Dependency Relation*, denoted by D_S is, for two actions α and β: $(\alpha, \beta) \in D_S$ (α and β are dependent) iff:

a) α and β are actions of the same thread, or
b) $vars(\alpha) \cap vars(\beta) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta)$ is written by α or β. ▪

2.12.1 Abstraction-aware POR

The syntactic dependency relation defined in Definition 8 is a valid dependency relation in the concrete state space [9], but it may not be valid in the abstract state space [16], as e.g., a single term in the precision of a predicate abstraction may include two variables, making them dependent in the abstract state space, while they might be independent in the concrete state space. [16]

Therefore, a new dependency relation is introduced for the abstract state space. $vars(\Pi)$ denotes the set of variables that appear in the abstract state expression formulae, that is, for explicit value abstraction, the set of tracked variables, which is a subset of all variables of the program ($\Pi \subseteq V, vars(\Pi) = \Pi$), and for predicate abstraction, the set of variables that appear in the tracked predicates of the precision (e.g., for $\Pi = \{(x > 5), (y = z)\}$, $vars(\Pi) = \{x, y, z\}$). [16]

Definition 9 (Abstraction-Based Dependency Relation). Let us have an abstract state space built with precision Π, and let D_Π be a binary, reflexive, and symmetric relation. Two actions $(\alpha, \beta) \in D_\Pi$ (α and β are dependent with respect to precision Π) iff [16]:

a) α and β are actions of the same thread, or
b) $vars(\alpha) \cap vars(\beta) \cap vars(\Pi) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta) \cap vars(\Pi)$ is written by α or β. ▪

This also means that α and β may still be independent if they use common variables that are not in the precision.

It was shown that constructing a POR technique based on this dependency relation is still sound [16].

Chapter 3

Abstraction-based Pointer Analysis

The methods introduced in Chapter 2 are static analysis techniques. In this chapter, our goal is to construct a dynamic, flow-sensitive, and precise, but still efficient pointer analysis method as a part of the CEGAR loop.

3.1 Overview

In order to perform a precise pointer analysis, we need to keep track of the exact values of pointers.

In our approach, we maintain a points-to graph for each node while constructing the ARG. This points-to graph represents the exact values of pointers at a given state of the program execution. The points-to graph is updated based on the actions of the ARG edges and the points-to graph of the source node. Explicitly keeping track of pointers allows us to substitute a dereferenced pointer with the exact value it points to when needed, and we can perform the analysis on the concrete values of the program variables. An overview of the CEGAR-loop extended with the proposed approach is shown in Figure 3.1.

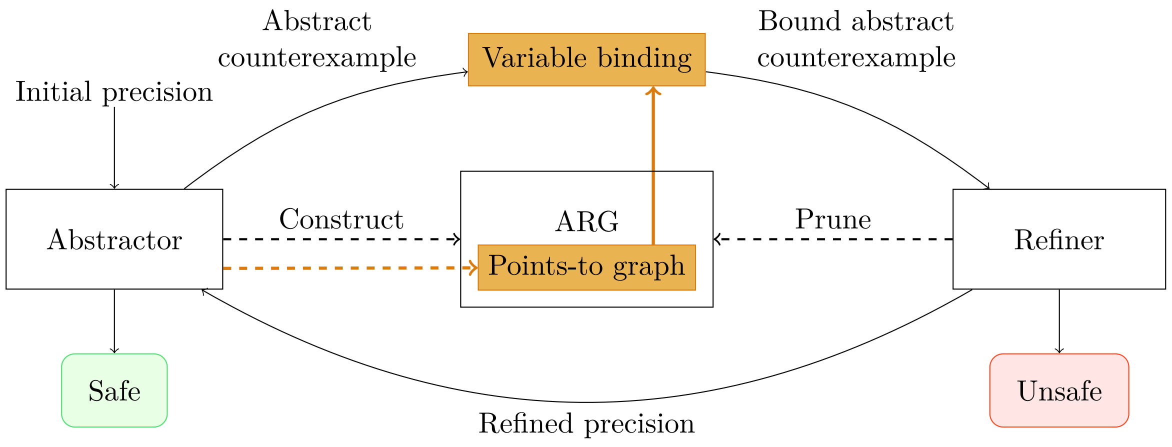

Figure 3.1: CEGAR loop with pointer support

To implement the approach, we only need to extend the Abstractor with the ability to compute and store the points-to graphs, and create a pre-procession step for the input of the Refiner. This has the major advantage that the Refiner does not need to be modified, and only the general Abstractor must be extended, thus the many different implementations of them already available can be used transparently.

3.2 The Points-to graph structure

To represent the points-to relations of pointers, we use a directed graph, where the nodes are the variables of the program (including pointers), and the edges represent the points-to relations.

Definition 10 (Points-to graph). A *Points-to graph* is a directed graph $PG = (V, E)$, where V is the set of variables of the program, and $E \subseteq V \times V$, where $(v_1, v_2) \in E$ if v_1 can point to v_2. ▪

The *points-to set* of a variable will be the set of variables it can point to directly.

Definition 11 (Points-to set). For a points-to graph $PG = (V, E)$, the points-to set of a variable $v \in V$ is the set of variables that v can point to: $pts(v, PG) = \{u \mid (v, u) \in E\}$. ▪

The *reachable set* of a variable will be the set of variables it can point to, including the variables that can be reached through pointers with higher levels of indirection.

Definition 12 (Reachable set). Let $PG = (V, E)$ be a points-to graph, and $v \in V$ be a node in the graph. The *reachable set* of v is defined as: $reachable(v, PG) = \{u \mid \exists n \in \mathbb{N} : u \in pts^n(v, PG)\}$, where $pts^n(v, PG)$ is the n-th iteration of the points-to set of v. ▪

Definition 10 does not specify whether the points-to graph is a static or dynamic representation of the program, nor if it is an under- or an over-approximation of the points-to relations. In our approach, the points-to graphs will be an exact representation of a given (abstract) state of the program. This means that in the points-to graph $PG = (V, E)$ at a state S, for any node $v \in V$, there exists at most one node $u \in V$ such that $(v, u) \in E$, as a pointer can only point to one location at a time.

Theorem 1. Given a Points-to graph $PG = (V, E)$, representing an exact points-to relation of a program state S in a CFA of a C program containing only primitive variable types[1] and pointers, PG is a directed acyclic graph.

Proof For a variable p with level of indirection i, the points-to set $pts(p, PG)$ can only contain variables with level of indirection $(i - 1)$. Therefore, for any given route in the points-to graph, the level of indirection of the variables in the route must be strictly decreasing, thus the graph cannot contain any cycles.

Theorem 1 implies that the points-to set and the reachable set of a variable are finite.

Example 4. Take the example program in Listing 3.1. The points-to graph for this program is shown in Figure 3.2, as an exact representation of the program state after line 4.

[1] In C, a `struct` is a user-defined type that is able to hold other variables of any type, including pointers. It is possible to create a cyclic data structure using `struct`s, however, strictly speaking, this does not result in a cyclic points-to graph. Still, supporting `struct`s are out of the scope of this work.

```c
    int i;
    int* p = &i;
    int* q = &i;
    int** u = &q;
```

Listing 3.1: An example C program with pointer operations.

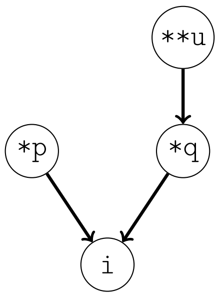

Figure 3.2: An example Points-to graph for the program in Listing 3.1.

The points-to sets and reachable sets based on the points-to graph are the following:

- $pts(i, PG) = \emptyset$, $reachable(i, PG) = \emptyset$
- $pts(p, PG) = \{i\}$, $reachable(p, PG) = \{i\}$
- $pts(q, PG) = \{i\}$, $reachable(q, PG) = \{i\}$
- $pts(u, PG) = \{q\}$, $reachable(u, PG) = \{i, q\}$

3.3 Extending the Abstractor

To facilitate a dynamic pointer analysis, the Abstractor needs to be extended to be able to store and compute a points-to graph of each abstract state.

3.3.1 Computing the points-to graph

When the Abstractor constructs the ARG, it computes possible successor states for each node by applying the operations (*Ops*) of the program on the current abstract state, starting from the initial state. These operations can include assignments, function calls, or returns from function calls.

Assignment operations can modify the points-to relations of pointers and variables, therefore we need to compute a new points-to graph for the successor node when encountering an assignment operation that involves pointers, as shown in Figure 3.3.

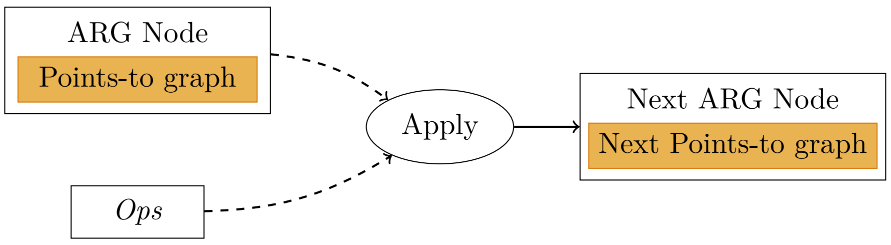

Figure 3.3: Applying operations on the ARG node results in a new state, including a new points-to graph.

For the initial ARG node, the points-to graph is initialized as an empty graph. For an ARG node N with points-to graph $PG = (V, E)$, the successor node N''s points-to graph $PG' = (V', E')$ is computed based on the type of the assignment action:

1. Referencing:
   `p = &i` $\Rightarrow E' = (E \setminus (p, j) \mid j \in V) \cup (p, i)$
   This operation removes all outgoing edges from p and adds a new edge from p to i, as the pointer p now points to the variable i, and nothing else.

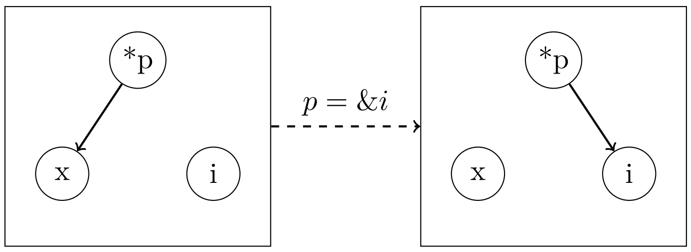

Figure 3.4: Example of changes in the points-to graph for a referencing operation. After the operation, p points to only i.

2. Dereferencing read:
   `p = *q` $\Rightarrow E' = (E \setminus (p, j) \mid j \in V) \cup ((p, j) \mid (q, r) \in E \wedge (r, j) \in E)$
   This operation removes all outgoing edges from p and adds new edges from p to all variables that $*q$ points to (as in $pts(pts(q, PG), PG)$).

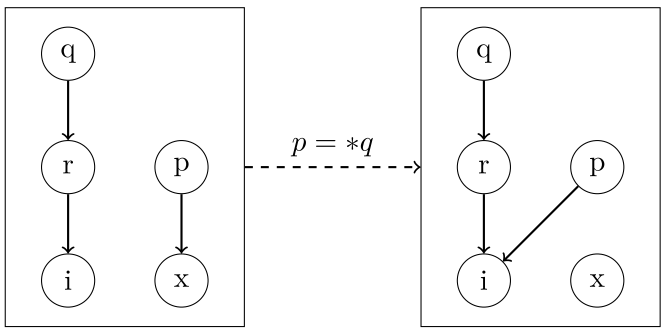

Figure 3.5: Example of changes for a dereferencing read operation. After the operation, p points to the same variables as $*q$, where $*q$ in this case is r.

3. Dereferencing write:

   `*p = q` $\Rightarrow E' = (E \setminus (p,j) \mid j \in V) \cup ((r,j) \mid (p,r) \in E \wedge (q,j) \in E)$
   This operation removes all outgoing edges from $*p$ and adds new edges from $*p$ to all variables that q points to.

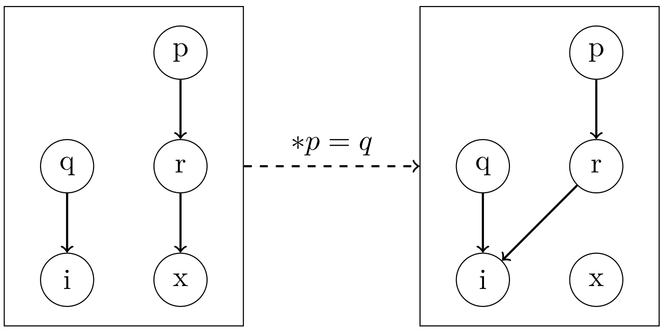

Figure 3.6: Example of changes for a dereferencing write operation. After the operation, $*p$, in this case r points to the same variables as q, in this case i.

4. Aliasing:
   `p = q` $\Rightarrow E' = (E \setminus (p,j) \mid j \in V) \cup ((p,j) \mid (q,j) \in E)$
   This operation removes all outgoing edges from p and adds new edges from p to all variables that q points to.

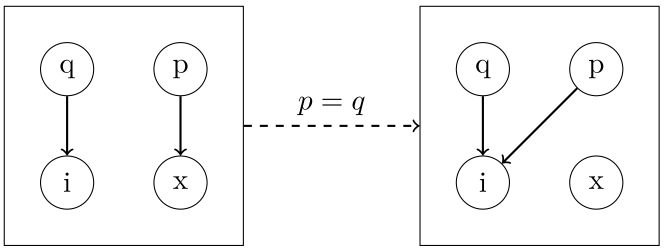

Figure 3.7: Example of changes for an aliasing operation. After the operation, p points to the same variables as q, in this case i.

Unlike the static analysis techniques, performing these actions does not require any additional steps, such as constraint solving; they are deterministic updates of the points-to graph, conforming to the rules and expected behaviour of the C programming language.

3.4 Variable binding for the Refiner

In the refinement step, we need to examine whether an abstract counterexample is feasible in the concrete program. This task is usually performed by an SMT solver, but an SMT solver is not able to reason about pointers.

To solve this problem, we will substitute dereferenced pointers with the values they point to. This method is referred to as *variable binding*. An overview of the variable binding step is shown in Figure 3.8.

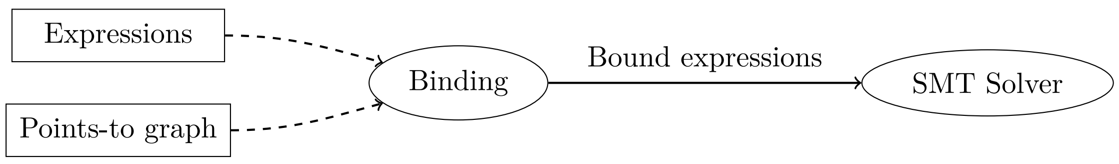

Figure 3.8: Binding variables from the points-to graph

The output of the Abstractor will be an abstract counterexample, a path in the ARG as a sequence of nodes and edges. The Refiner will then try to find a concrete counterexample, a path in the program that corresponds to the abstract counterexample. This is done by traversing the path in the ARG and transforming the operations and guards of the edges to SMT formulas.

This transformation step is extended with the variable binding, where any dereferenced pointer found inside an expression is bound or substituted with the variable it points to, based on the points-to graph of the source node.

As each variable in our points-to graph can only point to one other variable, the binding is deterministic.

The binding can be implemented in two ways:

a) Adding a binding term to the SMT formula for each dereferenced pointer. For example, in case of $(i = 1) \wedge (p = \&i) \wedge (*p \neq 1)$, a new term $(*p = i)$ is added to the formula.

b) Substituting the dereferenced pointer with the variable it points to. For example, in case of $(i = 1) \wedge (p = \&i) \wedge (*p \neq 1)$, the formula is transformed to $(i = 1) \wedge (p = \&i) \wedge (i \neq 1)$. (In this case, the term $(p = \&i)$ could be omitted.)

Example 5. Take the example program in Listing 3.2 with basic pointer operations.

```
int i = 1;
int* p = &i;

assert(*p == 1);
```

Listing 3.2: A program with pointer operations.

A possible ARG built by the Abstractor for this program is shown in Figure 3.9. The Abstractor is able to find an abstract counterexample, a path in the ARG that leads to a violation of the assertion. This path (shown in red) can be represented as the following formula: $(i = 1) \wedge (p = \&i) \wedge (*p \neq 1)$. The SMT solver's task is to deduce whether this formula is satisfiable or not, however, as discussed, it will not be able to reason about pointers.

The variable binding step substitutes the dereferenced pointer $*p$ with the variable it points to, i, resulting in the formula $(i = 1) \wedge (i \neq 1)$, which now can be easily declared unsatisfiable by the SMT solver.

3.5 Further Considerations

Our focus so far was on handling pointer dereferences. However, a case that also needs to be considered is when pointers are not dereferenced, but their exact value is used in an

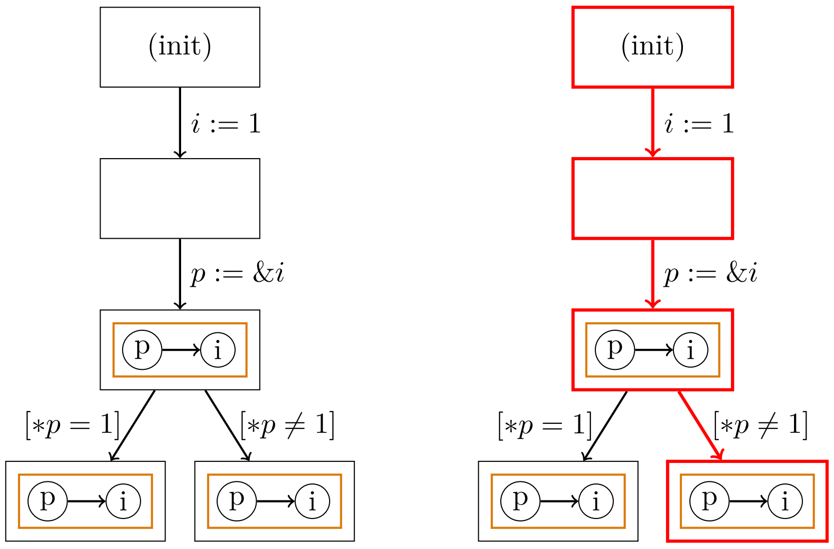

Figure 3.9: An example ARG that could be built by the Abstractor for the program in Listing 3.2. The points-to graphs of the nodes are shown in the orange boxes. The abstract counterexample found by the Abstractor is shown in red.

expression. Of course, the exact value (i.e., the memory address itself) after setting it to the address of a variable is non-deterministic and cannot be explicitly tracked. However, there are a few reasonable cases when the pointer itself does appear in an expression without dereferencing (other than an aliasing operation already discussed):

1. A pointer is checked whether it is a *null pointer*. This is a common check in C programs, as dereferencing a null pointer is undefined behaviour, most probably resulting in an error.
2. Two pointers are compared to each other. This is a less common, but still valid operation, as it is possible to compare pointers to determine whether they point to the same memory location.
3. A pointer is passed to a function as an argument.
4. A pointer is used in a *pointer arithmetic* operation. This is a common operation when dealing with arrays, as the elements of an array are stored in a contiguous memory block, and the elements can be accessed by adding an offset to the pointer to the first element.

To handle the first two cases, we simply need to declare to both the Abstractor and the Refiner that a pointer that has been assigned to the address of a variable can not be a null pointer, and that any two pointers that point to the same variable are equal, while two pointers that point to different variables cannot be equal.

These rules can be described formally as follows, given that the points-to graph is an exact representation of the program state:

- $pts(p, PG) \neq \emptyset \implies p \neq 0$
- $(pts(p, PG) = pts(q, PG)) \land (pts(p, PG) \neq \emptyset) \implies p = q$

- $pts(p, PG) \neq pts(q, PG) \implies p \neq q$

The third case, passing a pointer to a function, does not require any special handling, as this is inherently handled by the way function calls are modeled in the formal representation of the program along with the usage of the proposed techniques.

As for the last case, as stated before, we do not wish to support pointer arithmetic in our method.

Chapter 4

Pointer Support in Abstraction-Aware Partial Order Reduction

Verification becomes more difficult for concurrent programs. With the interleaving semantics of concurrent programs, the number of possible program executions grows exponentially with the number of threads and operations. This can easily make the analysis of concurrent programs infeasible in practice. To overcome this problem, partial order reduction (POR) techniques were introduced. These techniques reduce the number of interleavings needed to be explored by exploiting the independence of certain actions. For example, two independent actions adjacent in the program can be executed in any order without affecting the result of the program, therefore, only one of the possible interleavings needs to be explored.

With the introduction of pointers inside concurrent programs, the independence of actions and variables becomes more difficult to determine. For example, two pointers can point to the same variable, therefore the actions on these pointers are not independent anymore, but classical POR techniques might not be aware of this.

This raises the need to make POR techniques aware of the pointer operations and their effects on the program state for a successful analysis.

In this chapter, we extend the abstraction-aware POR technique with pointer support.

4.1 Dependency Relations with Pointer Support

To make the dependency relation aware of pointers, we need to extend the definition of the dependency relation defined in Definition 9. Generally, we also need to consider two actions dependent if they can access the same memory location, even through the use of pointers.

First, we need to define the set of variables that can be accessed by an action. This set will be the union of the variables directly found in the action and all the variables that can be reached through the pointers found in the action.

To compute this set, we need to acquire a points-to graph that is accurate in the context where the action is performed. According to our method described in Chapter 3, a precise points-to graph is now available for each state. Therefore, we can use the points-to graph of the state whose enabled action is the action we are considering.

However, in certain POR implementations (i.e., in *static POR*), an action considered for the dependency relation might not have a corresponding state [16], therefore no precise points-to graph is available for the action. For these cases, an alternate solution must be found.

A static pointer analysis method, such as the ones described in Section 2.3.1 and Section 2.3.2 can be run on the whole program, creating an over-approximation of the points-to graph, valid at all times. We run such an over-estimating pointer analysis before the verification process. This way, we can ensure that a points-to graph is available for every action by using this over-approximation when no precise information is available.

Definition 13 (Reachable variables of an action). Let α be an action, and PG_α be the points-to graph of the state whose enabled action is α, if available, or an over-approximating points-to graph of the whole program otherwise. The *reachable variables* set of α along with PG_α, is defined as:

$$reachablevars(\alpha, PG_\alpha) = vars(\alpha) \cup \bigcup_{v \in vars(\alpha)} reachable(v, PG_\alpha) \tag{4.1}$$

▪

With the definition of the reachable variables, we can provide a new definition for the dependency relation that also considers the variables that can be reached through pointers.

Definition 14 (Syntactic Dependency Relation with Pointer Support). A *Syntactic Dependency Relation with Pointer Support*, denoted by D_{SP}, for two actions α and β, and points-to graphs PG_α and PG_β being the points-to graphs of the state whose enabled action is α and β respectively, if available, or an over-approximating points-to graph of the whole program otherwise, $(\alpha, \beta) \in D_{SP}$ (α and β are dependent) iff:

a) α and β are actions of the same thread, or
b) $reachablevars(\alpha, PG_\alpha) \cap reachablevars(\beta, PG_\beta) \neq \emptyset$, and at least one variable in $reachablevars(\alpha, PG_\alpha) \cap reachablevars(\beta, PG_\beta)$ is written by α or β. ▪

This definition extends the original definition to not only consider the variables directly found in the actions, but also the variables that can be reached through the pointers found in the actions, by using the newly defined *reachablevars* instead of *vars*.

The same idea can also be applied to the abstraction-based dependency relation.

Definition 15 (Abstraction-Based Dependency Relation with Pointer Support). Let us have an abstract state space built with precision Π, and let $D_{P,\Pi}$ be a binary, reflexive, and symmetric relation. Two actions $(\alpha, \beta) \in D_{P,\Pi}$, with PG_α and PG_β being the points-to graphs of the state whose enabled action is α and β respectively, if available, or an over-approximating points-to graph of the whole program otherwise, (α and β are dependent with respect to precision Π) iff [16]:

a) α and β are actions of the same thread, or
b) $reachablevars(\alpha, PG_\alpha) \cap reachablevars(\beta, PG_\beta) \cap vars(\Pi) \neq \emptyset$, and at least one variable in $reachablevars(\alpha, PG_\alpha) \cap reachablevars(\beta, PG_\beta) \cap vars(\Pi)$ is written by α or β. ▪

Once again, this definition extends the original definition by using *reachablevars* instead of *vars*.

```c
#include <pthread.h>

int x = 0;
int* p = &x;

void* thread1(void* arg) {
  *p = 1;
}

void* thread2(void* arg) {
  x = 2;
}

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, thread1, NULL);
  pthread_create(&t2, NULL, thread2, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  assert(x == 2);
}
```

Listing 4.1: A modified version of the example program in Listing 2.5, with a pointer operation.

Example 6. Take the example program in Listing 4.1. The program has two threads, and both threads modify the variable x. However, the first thread modifies x through a pointer p, while the second thread modifies x directly.

Let the two actions $\alpha =$(`*p = 1`) and $\beta =$(`x = 2`) to be considered for the dependency relation.

With the classical dependency relation, α and β are considered independent, as they are actions of different threads, and they do not share any variables: $vars(\alpha) \cap vars(\beta) = \{p\} \cap \{x\} = \emptyset$. However, it is clear that they cannot be independent, as they both modify the same variable x.

With the newly introduced pointer-aware dependency relation, α and β are considered dependent, as PG_α will contain the information that p points to x, thus $reachablevars(\alpha, PG_\alpha) \cap reachablevars(\beta, PG_\beta) = \{p, x\} \cap \{x\} = \{x\} \neq \emptyset$, and they both write x.

4.2 Correctness of Pointer- and Abstraction-Aware POR

The non-trivial soundness of the abstraction-aware POR technique utilizing the abstraction-based dependency relation was proven in [16]. We must show that using the newly introduced pointer-aware dependency relation instead of the classical dependency relation in the abstraction-aware POR technique for a concurrent program containing pointers will result in a correct analysis.

First, we need to prove that the pointer-aware dependency relation is a superset of the classical dependency relation. This is trivial, as the pointer-aware dependency relation is defined to be the same as the classical dependency relation, except for using *reachablevars* instead of *vars*, which is a superset of *vars*, thus there cannot be a case where the classical dependency relation considers two actions dependent, but the pointer-aware dependency relation does not.

It is also easy to see that by using the *reachablevars* set, dependencies between actions that have the ability to access the same memory location are also considered dependent, given that one of the actions is a write action, correctly extending the main idea of dependency relations to pointer operations. When the over-approximating points-to graph must be used instead of a precise points-to graph, even though two actions might be unnecessarily considered dependent, the analysis will still be sound, as additional dependencies will not affect the correctness of the analysis, only its performance.

With these observations, we can conclude that the pointer-aware dependency relation used in the abstraction-aware POR technique will also be correct.

Chapter 5

Evaluation

The described approach has been implemented in the open-source model checker THETA[1] [17] and evaluated on a large set of benchmarks in the form of C programs [5].

The practical output of my work is a contribution to the THETA model checker, adding support for the verification of concurrent C programs containing pointers.

5.1 Theta

THETA is a configurable, CEGAR-based model checker, developed by the Fault Tolerant Systems Research Group (FTSRG) at the Budapest University of Technology and Economics. It is able to perform reachability analysis on several different formalisms with different abstraction domains and refinement strategies.

THETA is able to parse C programs through its C front-end, including concurrent programs, transforming them into an XCFA representation.

Abstraction-Aware Partial Order Reduction has been implemented in THETA as part of the abstractor component [15], and was shown to successfully verify concurrent programs.

THETA had no support for the verification of programs containing pointers and pointer operations so far. Its C front-end partially supported parsing programs containing pointers, but the verification itself could not be performed.

5.2 Implementation

The implementation has been completed according to the theories described in the previous chapters.

Firstly, I extended the C front-end to fully support parsing programs containing pointers, along with some adjustments to the formal representation of the program.

I implemented a generic points-to graph data structure (`PointerStore`), and extended the `XcfaState` class to be able to hold such a `PointerStore` instance, as described in Section 3.2 and Section 3.3.

I extended the Abstractor to be able to compute and store the points-to graph of each abstract state based on the specific actions of the program, as described in Section 3.3.

[1]`https://github.com/ftsrg/theta`

I also implemented the variable binding mechanism by fully substituting any dereferenced pointer with the variable it points to, as described in Section 3.4.

Lastly, I extended the already existing Abstraction-Aware Partial Order Reduction technique to also consider the variables a pointer can point to when determining the dependency relation between actions, as described in Chapter 3. For this, I also implemented two static pointer analysis methods (Andersen's and Steensgaard's) described in Section 2.3.1 and Section 2.3.2.

5.3 Evaluation on Benchmark C Programs

The implemented approach has been evaluated on a large set of benchmark C programs, provided by SoSy-Lab[2]. These programs are used in the SV-COMP competition, a prestigious competition for software verification tools.

5.3.1 Test Configuration

The changes were benchmarked and compared with a baseline configuration of THETA without pointer support in two different configurations. The first configuration did not use POR, only evaluating non-concurrent programs, and the second configuration used Abstraction-Aware POR (`AAPOR`), evaluating only benchmarks of concurrent programs.

Every benchmark used explicit-value domain abstraction (`EXPL`) and had a time limit of 900 seconds.

5.3.2 Results

As my contributions are an extension to the THETA model checker, the results of the benchmarks will be compared to a baseline version of it, which did not contain the pointer support, but is otherwise identical.

Specifically, we are interested in comparing the number of solved tasks, which is the number of benchmarks that were successfully deemed safe or unsafe by the model checker. As programs containing pointers were not supported by the baseline version, they were unsolved, thus we wish to see an increase in the number of solved tasks.

5.3.3 Results on non-concurrent benchmarks

The benchmarks of non-concurrent programs were evaluated using the explicit value domain abstraction (`EXPL`), first on a baseline version without pointer support, and then with the newly added pointer support. The results are shown in Table 5.1.

[2]`https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/`

	Baseline version	**New version**
Correct safe	256	266
Correct unsafe	274	284
Total correct	**530**	**550**
Incorrect	4	7

Table 5.1: Number of solved non-concurrent tasks.

The results show that with the newly added pointer support, THETA was able to solve 20 more tasks, increasing the number of solved tasks from 530 to 550.

The results also show that the number of incorrect results (as in, a safe program deemed unsafe, or an unsafe program deemed safe) increased from 4 to 7. After careful examination, it was found that these new incorrect results were caused by a bug in THETA unrelated to the newly added pointer support, and fixing this bug would result in these 3 incorrect benchmarks also being solved correctly, further increasing the number of solved tasks.

The fact that (practically) no new incorrect results were introduced suggests that the pointer support method described in Chapter 3 and the implementation of it is correct.

5.3.3.1 Results on concurrent benchmarks

The benchmarks containing concurrent programs, found in the `ConcurrencySafety` benchmark set, were evaluated using the `AAPOR` configuration, first on a baseline version without pointer support, and then with the newly added pointer support. The results are shown in Table 5.2.

	Baseline version	**New version**
Correct safe	54	54
Correct unsafe	193	238
Total correct	**247**	**292**
Incorrect	0	0

Table 5.2: Number of solved concurrent tasks.

The results show that the number of solved tasks increased from 247 to 292, which is a notable 18.2% increase. This shows that the implemented pointer-analysis approach along with the extension of the abstraction-aware POR technique can successfully verify significantly more concurrent programs.

The results also show that the number of incorrect results remained 0, which suggests that the extension of the abstraction-aware POR technique and its implementation is also correct.

5.3.4 Summary of results

Overall, we can conclude that the contributions of this work, the implementation of a pointer analysis method supporting concurrent programs for the THETA model checker, successfully increased the number of solved tasks significantly, in both non-concurrent and concurrent benchmarks, without introducing any new incorrect results.

Chapter 6

Summary and Future Work

This work aimed to broaden the applicability of formal verification to programs with pointers in concurrent programs motivated by the growing need for the verification of embedded systems.

In my work, I have developed a method to handle pointers from a C program in a CEGAR-based verification algorithm, as described in Chapter 3. The method allows a precise, but efficient tracking of pointers, allowing the verification of such programs. Additionally, in Chapter 4, this method has been integrated with abstraction-aware partial order reduction to handle concurrency, enabling the verification of concurrent programs with pointers.

Finally, in Chapter 5, I have implemented this method in the open-source verification tool Theta, and I have evaluated its performance on SV-COMP benchmarks. The contribution leads to a substantial increase in the number of successfully verified benchmarks.

The proposed method can be further improved in the future. A related field of research is the support of handling dynamic memory allocation, such as `malloc` and `free` operations, as their usage is closely tied to the use of pointers. This is also related to the support of arrays and pointer arithmetic, as arrays are a common usage of dynamically allocated memory, and pointer arithmetic is often used to access array elements. It is also a possible future direction to investigate the possibility of using the current precision of the abstraction during the pointer analysis as a performance optimization.

Acknowledgements

This work was partially supported by IncQuery Labs (incquery.io).

Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the partial support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

Bibliography

[1] Jonathan Aldrich. Pointer analysis. Lecture Notes for 15-819O: Program Analysis, 2023. URL `https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/pointer.pdf`.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. URL `https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf`.

[3] Levente Bajczi. Handling axiomatic memory models in abstraction-based model checking of concurrent and distributed systems. Master's thesis, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, 2022.

[4] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer International Publishing, 2018. ISBN 978-3-319-10575-8. DOI: `10.1007/978-3-319-10575-8_11`. URL `http://theory.stanford.edu/~barrett/pubs/BT18.pdf`.

[5] Dirk Beyer. Progress on software verification: Sv-comp 2022. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99527-0.

[6] 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, Oct 2007. ISSN 1433-2787. DOI: `10.1007/s10009-007-0044-z`. URL `https://doi.org/10.1007/s10009-007-0044-z`.

[7] Mihály Dobos-Kovács. On the verification of safety-critical embedded software systems. Master's thesis, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, 2021.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979. ISBN 0716710455. URL `http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455`.

[9] Patrice Godefroid, editor. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer Berlin Heidelberg, 1996. DOI: `10.1007/3-540-60761-7`. URL `https://doi.org/10.1007/3-540-60761-7`.

[10] Ákos Hajdu. *Effective Domain-Specific Formal Verification Techniques*. PhD thesis, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, 2020.

[11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011. URL `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`.

[12] Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324. Springer Berlin Heidelberg, 1987. DOI: `10.1007/3-540-17906-2_30`. URL `https://doi.org/10.1007/3-540-17906-2_30`.

[13] David Monniaux. A survey of satisfiability modulo theory, 2016. URL `https://arxiv.org/abs/1606.04786`.

[14] Bjarne Steensgaard. Points-to analysis in almost linear time. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 10 2001. DOI: `10.1145/237721.237727`.

[15] Csanád Telbisz. Partial order reduction for abstraction-based verification of concurrent software in the theta framework. Bachelor's thesis, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, 2022.

[16] Csanád Telbisz, Levente Bajczi, Dániel Szekeres, and András Vörös. Partial order reduction for abstraction-based verification of concurrent software.

[17] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: `10.23919/FMCAD.2017.8102257`.

[18] Asztrik Vörös. Informált keresési stratégiák absztrakció alapú modellellenőrzésben. Scientific students' association report, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Department of Measurement and Information Systems, 2022. URL `https://tdk.bme.hu/VIK/DownloadPaper/Informalt-keresesi-strategiak-absztrakcio`.

[19] Stefan Weinzierl. Configurable pointer-alias analysis for cpachecker, 2016. URL `https://www.sosy-lab.org/research/bsc/2016.Weinzierl.Configurable_Pointer-Alias_Analysis_for_CPAchecker.pdf`.