DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Increasing the Precision of the Static Analyzer Goblint by Loop Unrolling

Mireia Cano Pujol

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Increasing the Precision of the Static Analyzer Goblint by Loop Unrolling

Verbessern der Präzision des Statischen Analysators Goblint durch Loop Unrolling

Author:	Mireia Cano Pujol
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Michael Schwarz
Submission Date:	15th of February 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15th of February 2022 Mireia Cano Pujol

Abstract

Goblint is a source code analysis tool for C programs. It is based on abstract interpretation, and aims for a sound analysis. For loop representation and analysis, Goblint currently performs a simple CIL transformation to the input C code and can be refined by choosing one of two different abstract array domains. These domains are quite imprecise, as all elements of an array are represented by either one or three abstract values, depending on the domain used. In other known static analyzers, the analysis of loops is made more precise by treating the first k iterations of the loop separately from the following ones. This transformation is known as *loop unrolling*. In general, the larger the k, the more precise the analysis, and the longer it takes to perform.
The goal of this thesis is to design and implement a limited unrolling of loops of the analyzed program, and an abstract domain for arrays that can express precise information for the values at the first k index of the array. Together, those modifications optimize the precision of the analysis results that Goblint produces at loops.
The approach proposed is powerful enough to transform all types of loops supported in C programs, even complex structures like nested loops. It also handles complete initialization of arrays and updates after initialization. The overhead of the analysis for real-world programs (about 50%) after integration into Goblint is also reported.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Goblint and CIL . . . 2
1.2 Loops in CIL . . . 3
1.3 Loop Unrolling . . . 4
1.4 Abstract Interpretation . . . 6
1.5 Array Abstraction Currently Used in Goblint . . . 7

2 Related Work **9**
2.1 Loop Unrolling . . . 9
2.2 Arrays in Abstract Interpretation . . . 11

3 Design and Implementation of Loop Unrolling in CIL **13**
3.1 Implementation . . . 14
3.1.1 Unrolling the Loop Statement . . . 14
3.1.2 Filtering the Loop Statement . . . 17
3.1.3 Helpers for the Loop Unrolling . . . 18
3.2 Simple Loop Example . . . 19

4 Design and Implementation of a New Abstract Array Domain in Goblint **21**
4.1 Lattice . . . 21
4.2 Array initialization . . . 22
4.3 Less or Equal Relation . . . 23
4.4 Least Upper Bound . . . 23
4.5 Writing to an array . . . 24
4.6 Reading from an array . . . 25
4.7 Simple Example . . . 25

5 Combining Both Techniques **28**
5.1 Limitations . . . 28
5.2 Simple Example . . . 29

6 Evaluation **31**
6.1 Precision . . . 31
6.2 Performance . . . 32

7 Conclusions and Future Work **34**
7.1 Conclusions 34
7.2 Future Work 34

List of Figures **36**

Bibliography **38**

1 Introduction

Static analysis refers to the techniques used to analyze programs without executing them. Thus, the results obtained hold for any possible input scenario and any possible program execution. These techniques are used to check whether a given program fulfills certain properties and to optimize programs. They are especially relevant when it comes to large and complicated programs and can be used to inspire trust in an implementation or point to potential problems, such as races in concurrent code. One of the techniques for static analysis is abstract interpretation. This thesis deals with improving the precision in loops in the static analyzer Goblint that analyzes C programs.

When representing a loop in abstract interpretation, it contains several program points, which accumulate everything that happens at that point in all iterations of the loop. That is done by applying widening operators, which guarantee the termination in loops in abstract interpretation. By simply unrolling these loops, new program points are created for the unrolled iterations and the widening operators don't need to be applied. Therefore, precision is gained.

Currently, Goblint performs a simple CIL transformation to the input C code (explained in details in Section 1.2 for the case of loops) and uses two possible array domains (explained in detail in section 1.5). These implementations are quite imprecise, as all elements of an array are represented by one or three abstract values, depending on the domain used, as it will be explained later.

The goal of this thesis is to optimize the precision of the analysis results that Goblint produces at loops. This is achieved by designing and implementing:

1. A limited unrolling of loops of the analyzed program.
2. An abstract domain for arrays that can express precise information for values at the first k index of the array.

This way, Goblint can consider the first k iterations of the loop more precisely, leading to better analysis results.

The sections immediately following give an introduction to Goblint and CIL, outline the behaviour of loops and arrays in C and the C subset used by CIL, give the basic principles on abstract interpretation, and finally describe the array domains that are currently available

in Goblint. Section 2 on related work describes approaches and techniques that have been applied by others to implement the loop unrolling transformation, as well as to the analysis of arrays. Section 3 describes the loop unrolling transformation applied on CIL approach taken for this thesis. Section 4 details the new abstract array domain added on the analyzer part of Goblint. Section 5 combines both techniques and ends laying out the limitations of both. Section 6 presents an evaluation of the implemented approach. Finally, Section 7 provides a conclusion and outlines possible future work.

1.1 Goblint and CIL

Goblint [1] is a static analyzer for C programs based on abstract interpretation (see section 1.4). It is developed and actively maintained at the Chair of Formal Languages, Compiler Construction and Software Construction at the Technical University of Munich as well as the University of Tartu's Laboratory for Software Science and can freely be downloaded, used and inspected on the respective GitHub repository [2].

It is specialized on detecting potential data races in the multi-threaded C code. In order to prove the absence of bugs or vulnerabilities, the analysis must be sound - so it must always contain all real instances of the issues it is looking for, but may also include false alarms. The analysis will only err on the safe side: if the analyzer does not detect any error, the absence of errors has been proven. The goal behind Goblint is to be completely sound on all set of C, and efficient enough to, on standard PC configurations, analyze software of about 25 thousand lines of code in a few minutes [1].

As seen in Figure 1.1, Goblint doesn't analyze the input C code directly. Instead, it first uses CIL [3] to transform the C program into a program from a subset of C that is easier to analyze. CIL (or C Intermediate Language) is a high level representation which simplifies the analysis and source-to-source transformation of C programs. It offers a representation that makes it easy to analyze and manipulate C programs, and emits them in a form that resembles the original source. Goblint does not use the original CIL implementation, but a forked one [4]. When we refer to CIL during this thesis, it will always be in referenced to the forked CIL.

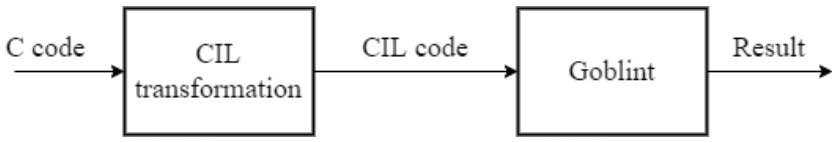


Figure 1.1: Representation of how Goblint and CIL work together.

In addition, given the *–html* flag when performing an analysis, Goblint automatically generates a visual representation of the control flow graph. This can help one understand the program better. The visual control flow graph will be specially helpful and visual when implementing the loop unrolling.

```
int main(void) {
    int a[5];
    int i = 0;
    while(i < 5){
        a[i] = i;
        ++i;
    }
    return 0;
}
```

(a) Function implementing a while loop in C

```
...
#line 4
  while (1) {
    while_continue: /* CIL Label */ ;
#line 4
    if (! (i < 5)) {
#line 4
      goto while_break;
    }
#line 5
    a[i] = i;
#line 6
    i ++;
  }
  while_break: /* CIL Label */ ;
  }
...
```

(b) CIL transformation of the while loop

Figure 1.2: Basic *while* loop and its CIL transformation.

1.2 Loops in CIL

The CIL syntax has three basic concepts[3]: expressions (which represent functional computation with no side-effect nor control flow); instructions (which express side effects without control flow); and statements (which capture control flow). On this thesis, our main concern are statements, as loops fall into that category.

The program structure is captured in a recursive structure of statements, each of which is annotated with labels, source location information and its successor and predecessor's control-flow information.

Loops specifically, are a kind of statement with the form *Loop (stmt list)*. CIL has only infinite looping constructs, and always uses a *Break* statement to exit from such loop. By way of illustration, Figure 1.2 shows the input C code of a basic *while* loop, and its CIL transformation into a *while(1)* loop. In order to transform the basic loop into a loop-forever looping construct, CIL uses the *goto* jump statement together with the *while_continue* and *while_break* labels (marked with the comment */* CIL Label */* to distinguish it from labels from the input source program), to simulate a *Break* statement.

The CIL transformation seen in Figure 1.2 is the most basic one. In other types of loops (e.g., do-while loops), the transformation to *while(1)* loop is maintained but the instruction

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
a[3] = 3;
```

(a) Fully unrolled loop

```
for(i = 0; i < 4; i+=2){
    a[i] = i;
    a[i+1] = i+1;
```

(b) Unrolled loop by a factor of 2

```
if (i<4){
    a[i] = i;
    i++;
    if (i<4){
        a[i] = i;
        i++;
        while(i<4){
            a[i] = i;
            i++;
        }
    }
}
```

(c) Unrolled loop by a factor of 2 (Another approach)

Figure 1.3: Different loop unrolling transformations on the same loop.

containing the *break* instruction moves accordingly.

Another key element from CIL's behaviour is that it moves the type declarations to the beginning of the program, and to substitute some calls to function for temporal variables, which are moved up as much as possible as well. That means that if the loop condition includes a call to a function, that call will be assigned to a temporal variable, and calculated right before the *if* condition in Figure 1.2.

In the proposed loop unrolling transformation, all the existing CIL alterations will be taken into consideration, and a general unrolling methodology will be presented for all of them.

1.3 Loop Unrolling

Loop unrolling [5] is a transformation applied to loops in order to either increase precision or reduce execution time, as explained in Section 2.1. It is ideally suited for sequential-array-processing loops in which the number of loop iterations is known before the execution of the loop.

The general idea of loop unrolling is to replicate the code inside a loop body a certain number of times and to adjust the loop-control accordingly. This transformation can be implemented in several ways, which we will explore shortly. The number of replications is called the *unrolling factor*, and the original loop is called the *rolled loop*. It is either necessary to add additional exit conditions to the unrolled loop body, or to handle those left-over iterations in a separate loop (*remainder loop*), if the loop iteration count of a rolled loop does not correspond

to an integral multiple of the unrolling factor [6]. If the number of iterations is known at compile time, then the compiler can add extra iterations after the unrolled loop. Otherwise, the insertion of additional code calculating the iteration counts during the program execution is required [6]. Nevertheless, loop unrolling is most effective when the loop counts are known at compile time, otherwise it can lead to performance degradation, which is not always worth it [7].

Loop unrolling can be done two ways; either manually (i.e., by hand) or automatically (i.e., by the compiler) [7]. The first option is rarely used; only when the compiler doesn't support the transformation. It is prone to errors, since decisions and modifications must be made by hand, like replicating the loop body, adjusting the loop count and maybe adding a remainder loop. The second option -the compiler performs the loop unrolling-, can be done in different phases of the compiling and optimizing process. The sooner, less information we have about the loop but future optimizations will benefit from the transformation [8]. The latter, we might have more information on the loop count, and it results in a lower growth in compilation time [7] .

To illustrate, Figure 1.3 shows some loop unrolling options for the loop in the source code below:

```
for(i = 0; i < 4; i++){
    a[i] = i;
}
```

Loop unrolling has many benefits. A. Koseki et al. [9] define it as one of the most promising parallelization techniques, as many programs spend most of the processing time in loops. Also, the unrolled code establishes additional potential for other compiler optimizations [6]. In addition, depending on the loop unrolling implementation chosen, the number of Jumps and Compares is reduced.

Loop unrolling also has negative side-effects. It is crucial to know the loop iteration counts for the transformation to succeed, and their absence diminishes the optimization potential. It is for that reason, that an effective loop unroller must have access to this parameter [7]. In addition, if the optimization is not applied elaborately it can also have an adverse impact on the program's performance. Because loop unrolling is a code-expanding transformation, an aggressive loop unrolling may overflow the I-cache (instruction cache). A typical solution is to use an unroll heuristic, which restricts the increase of the code size and allows loop unrolling as long as the loop size doesn't exceed a constant boundary [6]. However, with this approach optimization potential is missed.

As explained here and as it will be explained in the next section (Related Work 7), the loop unrolling transformation is used both by static analyzers in order to increase the precision they offer at loops, and also used in compilers in order to reduce execution time, improve parallelism, and so on.

1.4 Abstract Interpretation

The abstract interpretation methodology builds an approximate semantic understanding of programs that can be used to gather information about programs and to provide sound answers to questions about their run-time behaviours [10]. Using these abstract semantics, one can design manual proof methods or specify automatic program analyzers like Goblint. Abstract interpretation is useful when designing analysis, since it provides proof of the soundness of such program analysis methods.

Patrick Cousot and Radhia Cousot invented and formalized abstract interpretation in their article [11]. They formalized the concept in the following way:

> Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations [11].

In the introduction, an easy example to understand abstract interpretation is given. It consists on the abstraction of a trivial multiplication, where the abstract semantics only consider the signs.

The example can be generalized to:

$$(+A) * (-B) = X \tag{1}$$

To now abstract this simple equation with the rule of signs as base, the concrete domain must be replaced with the abstract one. It can therefore be mapped to:

$$(+) * (-) = (\pm) \tag{2}$$

Using this simple example, it can be seen now that while specific information might be lost, it also allows for accurate assumptions about the outcome of a process without performing it. Naturally, abstraction is usually more complicated than that, and finding an appropriate abstract domain is a challenge of choosing between precision and execution time. If the abstract semantics are too precise, they also become unrealizable like concrete semantics, and would potentially not terminate.

Patrick Cousot and Radhia Cousot [12] also created a visual representation of this concept, shown in Figure 1.4.

It can be seen that the lines marked as *Possible trajectories* represent the potential paths or iterations of a program, while the green area around them represents their abstraction. Red areas in the middle and the so-called "Forbidden Zone" are viewed as potential attack vectors. It is possible to prove, that these attack vectors cannot be realized, by abstracting the software process to a much larger area (the green area) and thus proving it does not reach any red zones, which have been defined beforehand.

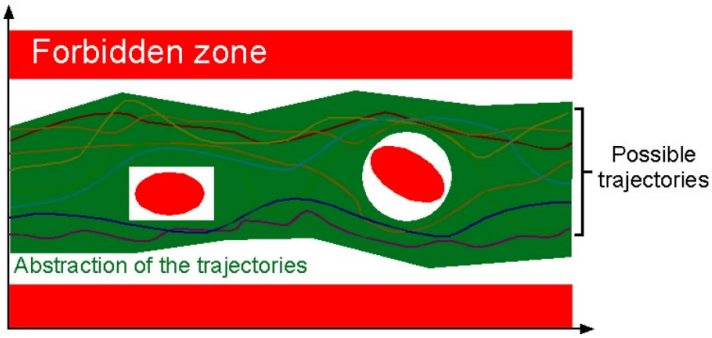


Figure 1.4: Visualization of Abstract Interpretation [12].

Every abstract interpretation requires a defined collection of abstract values. An abstract value is just a set of concrete values. When speaking about integers, the abstract values are:

- $\top$: (called *top*). Is the set of all integer values.
- 1,2,... : The singleton sets of integer values.
- $\bot$: (called *bottom*). Is the empty set.

The mathematical structure which controls the level of approximation of the abstract interpretation is called a lattice. The lattice is key since it tells us how to merge two abstract values, among other things.

1.5 Array Abstraction Currently Used in Goblint

Since arrays are an important data structure, an abstract interpreter like Goblint needs to have at least a domain dedicated to them. Although this domain needs to be precise to maintain useful information, it should not significantly slow down analysis.

At the moment, Goblint offers four abstract domains for arrays: two basic domains and two more complex ones based on the basic domains.

The most basic domain (*Trivial*), represents all elements of an array by a single abstract value. This domain is an implementation of the smashing approach (see Section 2.2). *Trivial* is parametric in the domain (*Val*) for this abstract value. It is also parametric in another domain (index domain *Idx*), an integer domain used to represent indices.

The other basic domain (*Partitioned*), was introduced in [13]. This domain is built around the idea of keeping 3 abstract values at every point, plus a symbolic index e. The 3 abstract values are:

- *xl* : An abstract value which depicts all values to the left of *e*.
- *xm* : An abstract value of the position *e*.
- *xr* : An abstract value which depicts all values to the right of *e*.

This domain is more precise than *Trivial*, and changes the *e* index depending on the value being accessed. However, it only works properly when the array is iterated in a linear way, either from the front or from the back. *Partitioned* is an implementation of the partitioning array approach explained in Section 2.2.

The domain Goblint employs by default is *TrivialWithLength*, which takes *Trivial* as the base domain and combines it with an integer domain representing the length of the array if it's known.

The fourth domain (*PartitionedWithLength*) can be chosen alternatively to *TrivialWithLength*. Similarly, it takes *Partition* and it combines it with an integer domain representing the length of the array if it is known.

PartitionedWithLength and *TrivialWithLength* are essentially the same as *Partitioned* and *Trivial*. The main and only difference is that they keep track of the length of the array, and perform an *out of bounds* check when reading from the array. For simplicity, we will refer to *Partitioned* and *Trivial*, when refering to *PartitionedWithLength* and *TrivialWithLength*.

2 Related Work

A variety of different approaches have been proposed in literature for dealing with loop unrolling and arrays in abstract interpretation. The ones we considered more relevant are described in Section 2.1 for the loop unrolling, and in Section 2.2 for abstract array domains.

2.1 Loop Unrolling

When it comes to loop unrolling implementations or approaches, two main distinct goals exist. On one hand, papers like [6, 7, 8, 14] implement the loop unrolling optimization at compiler level, with the objective of reducing the execution time. Some of those [6, 14] make use of a static analyzer in order to have more information about the code previous to the loop unrolling (e.g., to calculate the iteration counts). On the other hand, other implementations [15, 16, 17] include the loop unrolling as part of the tool-set or parametrization options that a static analyzer offers. That allows for a more precise analysis. The approach of this thesis has this second goal in mind. However, all implementations have interesting points which can be related to the loop unrolling realized on this thesis, and which help improve it in the future.

In [6], loop unrolling optimization is applied at source code level, which allows for a number of other compiler optimizations to benefit from it. Another relevant aspect of their implementation are their prediction mechanisms. Those compute the unrolling profit for each loop that allows the determination of the most promising unrolling factor. In addition, they integrate standard heuristics (like avoidance of I-cache overflows) which have been shown to be highly effective. Using their profit calculation, they can predict the adverse effects of unrolling a loop in advance.

In [7], Jack W. Davidson and Sanjay Jinturkar claim that when loop unrolling is only applied to loops which consist of a single basic block and whose iteration count can be determined only at compile time, a lot of optimization potential is lost. They implement an aggressive application of loop unrolling in order to prove that it is worth it. An interesting distinction they do, is that between *compile-time counting loop* (Figure 2.1a) and *execution-time counting loop* (Figure 2.1b). In the first one, the number of iterations is known at compile time, as opposite to the second one.

Litong Song and Krishna Kavi [8] present a loop unfolding technique which serves as a first transformation or pre-optimization, as they call it, to ease future optimizations by removing

```
for (i = 0; i < 12; i++)
    a[i] = i;
```

(a) Compile-time counting loop

```
for (i = 0; i < n; i++)
    a[i] = i;
```

(b) Execution-time counting loop

Figure 2.1: Counting loops, example from [7].

anti-dependences as much as possible. This is necessary because many loop-optimization techniques only work effectively on limited cases, when the loops are "well-structured". In their paper, they present generalized loop-unrolling methods that allow the transformation of even badly structured loops. The technique described relies on unfolding the loops for several initial iterations, in a similar way to our thesis' proposed loop unrolling, such that more opportunities may be exposed for many other existing compiler optimization techniques.

Marcelino Rodriguez-Cancio et al. [14] introduce a compiler loop optimization called approximate loop unrolling, which transforms loops similarly to loop unrolling. Instead of unrolling loops by adding exact copies of the loop's body, it adds code that interpolates the results of previous iterations. For that, it exploits the fact that some computations can be made less precise and still produce good results; if the function's computations is expensive, it is substituted by a less costly interpolation of the values assigned to nearby array values. Comparing it to loop unrolling, approximate unrolling's gains are much higher as about half of the operations are removed, but at the cost of reducing the system's accuracy.

As previously stated, some methodologies include making use of a static analysis of the loop before applying the optimization. In [14], Marcelino Rodriguez-Cancio et al. use this analysis to determine if the loop's structure fits the transformation to apply, and if that would actually provide some performance improvements. In the case of [6], it is used to compute safe and input-invariant loop bounds that are valid for all input data, which will serve to determine a suitable unrolling factor.

Clang Static Analyzer is a source code analysis tool which aims to find bugs in C, C++ and Objective-C programs by simulating the possible executions of the code. The loop unrolling implementation they currently have in use is efficient but could result in a loss of coverage in various cases. It consisted in unrolling the loops 4 times by default, and then cut the analysis. This way, from the point of view of the analyzer, all loops have only 4 iterations. In [17], Szécsi, Péter György et al. propose two solutions to resolve the limitations explained. One of those solutions being to include loop unrolling heuristics and patterns in order to find specific loops which are worth to be completely unrolled.

Bruno Blanchet et al. [16] and Daniel Kästner et al. [15] explain that the refinement of a general purpose static analyzer to adapt it to particular programs is done through parameterization. On [16], one of those strategies focuses on iterations and is done by loop unrolling. In many cases, the analysis of loops is made more precise by treating the first iterations of the loop separately from the following ones. This approach is the one we follow in this thesis, where

the loop is expanded the following way:

```
if (condition) { body; while(condition) {body} }
```

On [15], a similar approach is followed and mixed with *variable smashing* (explained in the next section). This way they can analyze critical program parts with high precision, and improve speed by lowering the precision for uncritical program parts. For the first part (analyzing critical program parts), the tool unrolls loops n times, and individual invariants are computed for the first n iterations. With this, the analysis will become more precise, in general, and the analysis time will increase. In a similar way as this thesis, users can specify a default unrolling factor, yet they also contain a heuristic loop unrolling which can override the default value for individual loops.

2.2 Arrays in Abstract Interpretation

A variety of approaches have been proposed in literature for dealing with arrays in abstract interpretation. In this thesis, three main ones will be considered: smashing, expansion and partitioning.

In array smashing [18], the whole array is represented by one abstract element which encompasses all the possible values for every index of the array. Is it especially useful when dealing with large arrays, since smashing them results in a smaller memory consumption. However, the simplicity of such an approach comes at a price, since its precision is relatively low. All updates on the array, for instance, will be performed as weak updates (i.e., if a specific element is updated, the whole array representation will just add this value to the smashed abstraction).

Whenever no information about the contents of an array is available at the start of analysis, its abstract value always remains $\top$. This is specially relevant in a language such as C, where local arrays are not initialized to default values. A possible solution to maintain more information is assuming that arrays are never read at indices where they are not initialized. Then, it is sound to start with $\bot$. However, as soon as this assumption is violated, the analysis becomes unsound.

The *Trivial* and *TrivialWithLength* array domains used in Goblint make use of that technique (i.e the whole array is represented by one unique abstract value). Since we are not interested in preserving precision on the totality of the array, a smashing approach will be used for a range of indexes on the array abstract domain implemented in this thesis.

Another approach on the other end of array smashing, is array expansion (also described in [18]). It consists of having one abstract element for each index in the array. As opposed to smashed arrays, expanded ones are much more precise. They also result in less weak updates,

since each value can be more accurately calculated. A detail which is also mentioned in [18], is that the precision gain of expanded arrays is especially interesting when combined with loop unrolling, which is the approach chosen for this thesis for the first k indexes of the array.

The analyzer used in [18] combines both previous approaches. The user has to decide which representation to use, by either providing an array size bound (arrays of size smaller than the bound are expanded) and/or by giving an exhaustive list of all arrays that should be expanded.

Another approach for abstract array domain implementation is array partitioning. It consists of splitting the arrays in some kind of partition and then maintaining only one abstract value for each of those partitions.

In [19], Patrick Cousot et al. present FunArray, a parametric segmentation abstract domain functor for the fully automatic and scalable analysis of array content properties. The analysis with FunArray automatically divides the array into a sequence of possibly empty segments delimited by a set of segment bounds. The values in each of such segments are then represented by one abstract value.

[20] partitions the array in groups, instead of segments. Its objective is to maintain precise information about the single-element groups, and less precise information about the other elements (e.g., if an array element is accessed, the analysis tries to isolate that element in a separate group so that a strong update may be performed). Such a partitioning heuristic allows the analysis to automatically discover constraints on the values of array elements after simple initialization loops. In Goblint, the *Partitioned* and *PartitionWithLength* array domains use this approach.

3 Design and Implementation of Loop Unrolling in CIL

As said in the introduction, the implementation of this precision-increase transformation was divided into two parts. The first was done on the CIL side of Goblint, and it's the loop unrolling itself. Even though this transformation is implemented in the Goblint executable, it conceptually fits within the *CIL transformation* step shown in Figure 1.1. The loop unrolling proposed is similar to the loop unfolding done in [15, 16], in that it does not replicate the body inside of the loop itself, but before it. There were some limitations that forged the path in which the transformation had to be implemented:

- Since the loop unrolling had to be placed inside the *analyzer* project, we would not be able to read the C input source code, but only the CIL transformation of it. That means, the loop unrolling is applied on top of the CIL code.
- We had no access to the number of loop iterations, for that reason :
  - We did not filter the loops in which the transformation could be applied to, depending on the information we had on the iterations. Instead, an aggressive loop unrolling approach was implemented, more similar to [7].
  - It was not possible to know how many times the loop needed to be unrolled. For that, the loop unrolling factor is entered as a parameter. In addition, flow control structures had to be added to the unrolled iterations.

To implement the loop unrolling we made use of the *visitor pattern*, which is a behavioral design pattern that lets you separate algorithms from the objects on which they operate. There already is a visitor pattern defined in *cil.ml* which takes care of traversing the Abstract Syntax Tree (AST). To use it, we inherit from *nopCilVisitor*, which simply walks the AST without doing anything, and then we override the base class's methods to do something other than nothing. We specifically overrode the *vstmt* method, which traverses all the control-flow statements, allowing us to find all the loops in the code.

Each of the visitor's methods end telling the visitor how to proceed next. There are four options for that, but the following ones are the only ones being used in the implementation:

- **DoChildren** : It directs the visitor to recurse into child AST nodes.

- **ChangeDoChildrenPost(x, f)** : It replaces the AST node with x, and runs f on the result of rebuilding x with the result of the visitor running on x's children.

3.1 Implementation

The unrolling transformation happens mainly inside the *vstmt* method. In there, an statement *s* is received. This statement is checked to be a Loop, case in which the loop unrolling takes place. Otherwise, the visitor continues examining the rest of the statements. The loop unrolling implementation can be divided into three main components or parts: Unrolling, Filtering and Helpers. Next, we will explain all three components.

3.1.1 Unrolling the Loop Statement

The first and most important part of the implementation is the unrolling itself.

As said before, there is no way (in our implementation) to know the number of iterations the loop is going to perform. Not even when dealing with a *compile-time counting loop* [7]. For that reason, the unrolling factor will be set with the flag ***exp.unrolling-factor*** and that will be applied indistinctly to all loops. For a better understanding, the loops to be unrolled are separated into two different cases:

- **Simple case - unrolling of individual loops** : As a general explanation, the loop unrolling is produced by simply copying the entire body of the loop, and putting those replicas before the loop itself. We will refer to the original loop, which will now be placed at the end of all the unrolled iterations, as *remainder loop*.

  That works because of CIL's internal processing of the loops; the condition to continue iterating is not inside the *while* loop condition anymore. Instead, the CIL source code has now a *while(1)* loop, and a *break* instruction inside of an *if* instruction, which holds the original condition to exit the loop. Due to that, the body can be just replicated and the control flow structures are already there to make sure there are no extra iterations performed.

```
FUNCTION unroll (base: STMT_LIST) (factor: INT)
    IF factor is 0, end recursion.
    ELSE
        X = concatenate loop_body and base
        CALL unroll X factor-1
```

Figure 3.1: Pseudo-code of the unroll function.

This implementation has the form of the pseudo-code in Figure 3.1. In there, the recursive function *unroll* occurs. On the first iteration, *unroll* receives the original loop statement *s*, and the factor previously set in *exp.unrolling-factor*. It then proceeds to replicate the loop's body in front of it, and repeat the whole operation. Thus, *base* will always include loop unrolled $exp.unrolling\text{-}factor - factor$ times. Each execution of unroll then builds the following structure, which can also be seen in a more graphic representation in Figure 3.2:

```
{ if(condition) {body} } @ base
```

The above transformation is iterated *k* times, where this factor is user-defined in the *exp.unrolling-factor* parameter. As a general norm, the larger this *k* is, the longer the analysis time, yet the more precise the analysis.

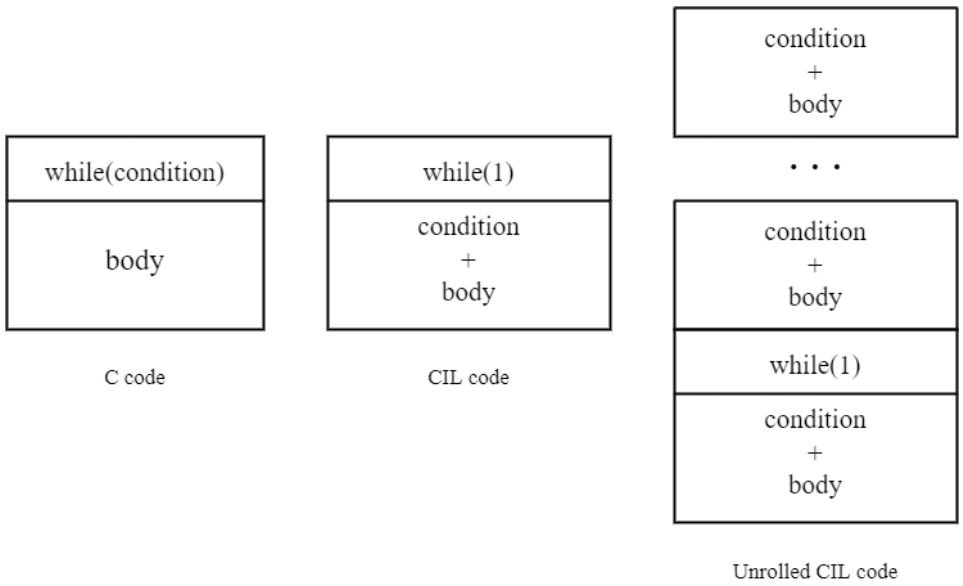


Figure 3.2: Graphic representation of a simple while loop in C code, CIL, and unrolled.

This simple yet effective implementation works for almost every loop possibility.

- **do-while loop**: In this case, the CIL source code is generated the same way as a regular loop, with the difference that the *if* instruction containing the *break* to exit the loop is at the end of the while loop, after the body. The structure build during the unroll would maintain all do-while loop properties:

  ```
  { {body} if(condition) } @ base
  ```

- **call to function in loop condition**: An interesting thing done by CIL is to move

the type declarations up, and to substitute some calls to function for temporal variables, which are moved up as much as possible. Thus, that means that if the loop condition includes a call to a function, that call will be assigned to a temporal variable, and calculated right before the *if* instruction. The built structure would then be:

```
{ {tmp=call} if(condition) {body} } @ base
```

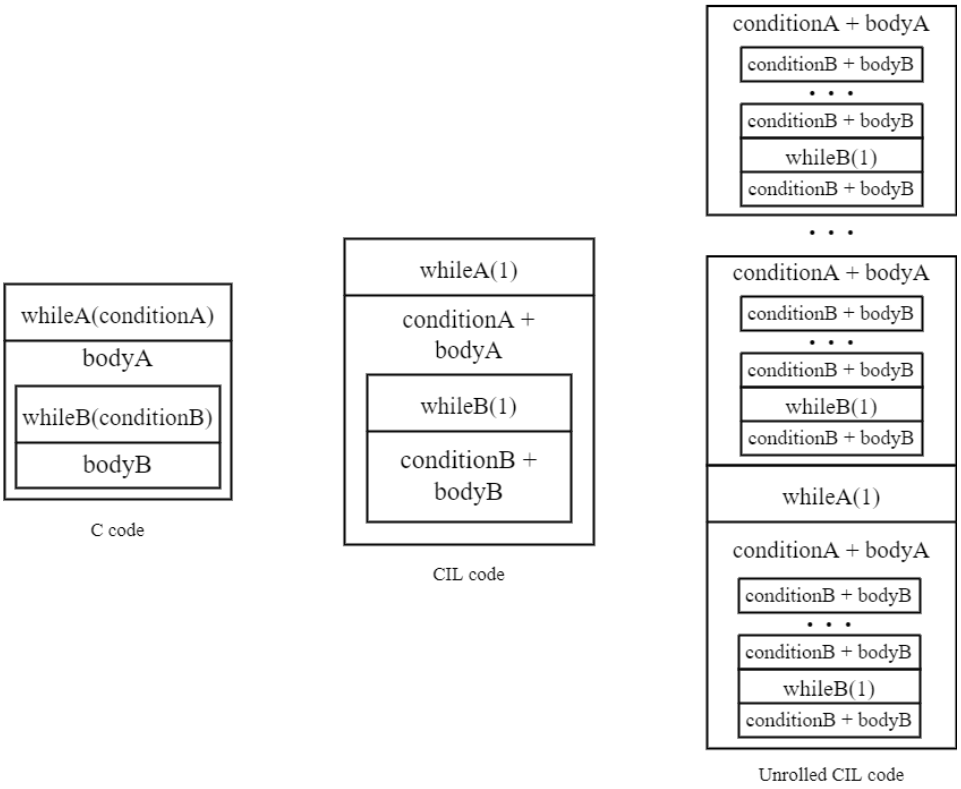


Figure 3.3: Graphic representation of nested loops in C code, CIL, and unrolled.

- **Loops with only one iteration**: CIL does not see that the loop will only have one iteration, so it generates the control flow graph in the same way as any other loop (i.e., it does not remove the while loop and leaves only the loop's body). Similarly, the loop unrolling does not have a specific behaviour for this case, and unrolls the loop as many times as the factor indicates. It is later, when the analyzer realises that one or multiple branches on the control flow graph are useless and will never be iterated or visited. This can be extended to any loop with less iterations than the factor *f*.
- **while(1) loops**: It can also occur that the original loop in C was implemented in the same way CIL would do it (i.e., *while(1)* loop plus *break*). In that case, the *break*

condition to exit the loop could be anywhere in the loop's body. Due to the nature of our implementation, though, that is handled properly:

```
{ {body} if(condition) {body} } @ base
```

- **General case - Unrolling of nested loops** : In our implementation, nested loops were also decided to be unrolled. The graphic representation of our nested loops implementation can bee seen in Figure 3.3. In order to support this, some extra implementation was required.

  Once the loop's children statements start being visited in order to find potential nested loops, *vstmt* is always going to find the *remainder loop*. That is the original loop which we unrolled, and that is now at the end of the statement, since all unrolled iterations where concatenated before it. Without any additional bounds, *vstmt* will detect that as unrolling material, and the loop would then be unrolled an infinite number of times.

  In order to solve this problem, it was decided to use labels. A *remainder_loop* label is added before the loop is unrolled. In Figure 3.4, a simplified example of the CIL code found inside of an already unrolled loop can be seen, which allows the analyzer to identify it. This way, when Goblint is iterating through the the children nodes, it will be able to distinguish the loops that have already been unrolled and skip them.

```
remainder_loop: /* CIL Label */
  while (1) {
    while_continue: /* CIL Label */ ;
    if (! (i < 5)) {
      goto while_break;
    }
    ...
  }
  while_break: /* CIL Label */ ;
  }
```

Figure 3.4: Simplified view of an already unrolled loop.

3.1.2 Filtering the Loop Statement

The second part of the implementation is the filter, which detects what type of loop we are dealing with, and indicates if it must be unrolled.

As explained in Section 3.1.1, we are able to easily and effectively unroll all loops, regardless of their type. This means that all loops can aggressively be unrolled the same way. Therefore,

in this particular case, the filter is a helper for the specific case of nested loops, which was explained previously, and it is currently designed so it can be easily extended. This way, if in the future there are specific cases for which we might not want to unroll a loop, it can be easily indicated.

If this filter returns *false* (meaning the loop is not unrollable), the result given to the visitor will be *DoChildren*. As previously explained, that result is indicating that the current statement (the loop) should not be modified, but that the children should be visited, in case there is something else to unroll inside the current loop. The visitor, then, will proceed with the loop's body with the goal of finding new loop statements.

In the case of the filter returning *true* (meaning the loop is unrollable), the result given to the visitor is *ChangeDoChildrenPost*, which indicates that the current node (the original loop) should be substituted by the indicated value. In this case, the substituting value will be the result of unrolling the loop. The second parameter is a function, which is applied to the visited node and children. However, in this case the function indicated does not do anything (since nothing needs to be applied).

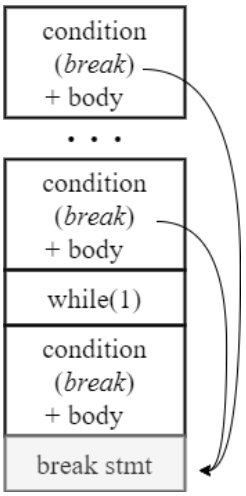


Figure 3.5: Manual handling of *break* instructions.

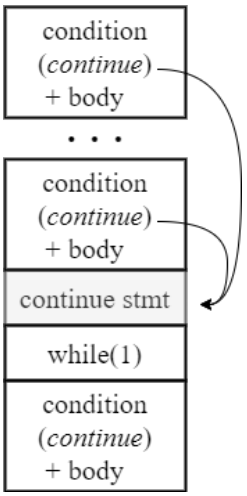


Figure 3.6: Manual handling of *continue* instructions.

3.1.3 Helpers for the Loop Unrolling

In order to integrate this loop unrolling implementation to the existing Goblint infrastructure, some modifications were required.

First, the loop unrolling is performed before *prepareCFG*, function which does not allow *break* nor *continue* instructions to be performed outside loops, as it loses reference of what should the control flow graph point to. For this reason, all *Break* and *Continue* statements

are manually converted into *Goto*s, which reference empty instructions also placed manually. Figures 3.5 and 3.6 display how *breaks* and *continues* are handled, respectively. *Goto*s replacing *break* instructions point to a statement added at the end of the loop. This way, when Goblint replicates the body of the loop, those iterations won't try to break out of a loop, but instead to go at the end of it. *Goto*s replacing *continue* instructions point to an statement added at the beginning of the loop. This way, when the loop's body is replicated, it won't try to go to the next iteration of a non-existent loop, but to the beginning of the remainder loop.

Another important concern when implementing the unrolling, was that Goblint does not allow for replicated statements. In order to replicate the body of the loop, it must first be duplicated. Those duplicates are shallow copies of all the statements inside the loop's body.

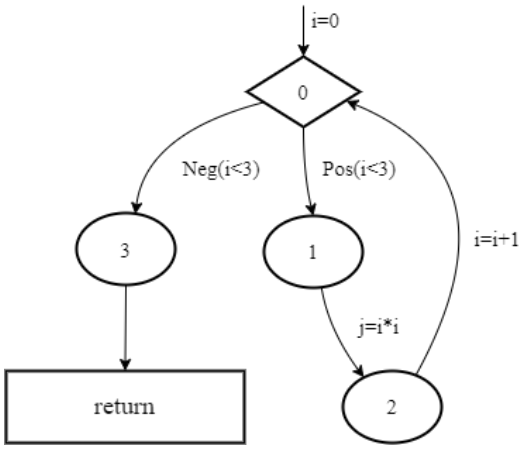
3.2 Simple Loop Example

In order to illustrate the aspects mentioned on the previous sections, an example of the program in Figure 3.7a will be given. Figure 3.7b displays the control flow graph generated by Goblint without unrolling enabled.

```
//Assume factor is set to 3
//Assume j initialized to 0
int i = 0;

while(i<3){
    j = i*i;
    i++;
}
```

(a) Program



(b) Control flow graph

Figure 3.7: Example program containing a simple loop.

Once the loop unrolling transformation is enabled (by setting *exp.unrolling-factor* to 3, in this case), the control flow graph seen in Figure 3.8 is generated. Figure 3.9 shows the corresponding program-point information for each node on the control flow graph.

Since the loop has been fully unrolled (i.e., the factor is equal to the number of iterations

of the loop), the remainder loop will never be accessed. This is the reason why there is no information about (11) and (12). The loop being fully unrolled also provides higher precision. The development of variable *j* can be precisely tracked and a final value is provided. Otherwise, the final result of *j* at the end of the loop would be *[0,4]*, instead of *[4,4]*.

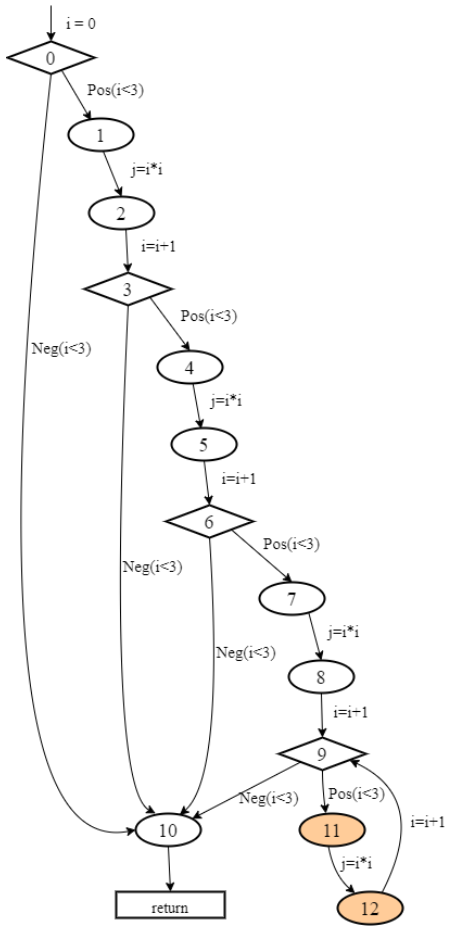


Figure 3.8: Control flow graph of the unrolled code in Figure 3.7a.

	i	j
0	[0,0]	[0,0]
1	[0,0]	[0,0]
2	[0,0]	[0,0]
3	[1,1]	[0,0]
4	[1,1]	[0,0]
5	[1,1]	[1,1]
6	[2,2]	[1,1]
7	[2,2]	[1,1]
8	[2,2]	[4,4]
9	[3,3]	[4,4]
10	[3,3]	[4,4]

Figure 3.9: Program points and states.

An additional interesting thing which can be appreciated on the unrolled control flow graph, is the need for the control flow structures when unrolling the loop. Due to the ignorance on the iteration count, even with a low factor number like 3, there is no guarantee that the loop will not perform less iterations. In that case, it would be possible to break from the loop at (0), (3) or (6).

4 Design and Implementation of a New Abstract Array Domain in Goblint

The second task was to add a new abstract array domain on the analyzer part of Goblint (Figure 1.1). The abstract domain proposed in this section uses a combination of the smashing and the extension techniques, while dividing the array to decide in which segment each of the first two techniques is applied. This domain is designed to work with the previously explained loop unrolling, yet it can also be used on its own.

This domain must be manually given a factor f. For this, the user must set the *ana.base.arrays.unrolling-factor* to a value greater than zero. Otherwise, an error is raised since the *Trivial* array should be used instead.

In this approach, an array A is divided into two different parts according to the given f. The parts are:

- *xl* : Consists of a list of length f, which contains an abstract value for each of the elements $A_{<f}$. This part of the array implements an expanding approach.
- *xr* : Consists of a unique abstract value representing all the rest of the elements on the array $A_{\geq f}$. This part of the array implements an smashing approach.

An example of an array divided according to a given factor can be seen in Figure 4.1. The elements contained in each part and the approach they implement is also illustrated.

Ideally, this index f used for dividing the array is chosen so that it is the same as the selected factor on the loop unrolling. If a higher or lesser value is chosen, precision optimization potential is lost and either extra iterations are unrolled or extra abstract values are computed.

While this approach is not novel, it is a simple and effective way to potentiate the loop unrolling optimization and shortly improve precision on its own.

4.1 Lattice

The abstract value for an array always has the same form, which is $(A_{<f}, A_{\geq f})$. $A_{\geq f}$ is an abstract value that describes the contents of the entire array. A is from $\mathbb{V}al$, the lattice

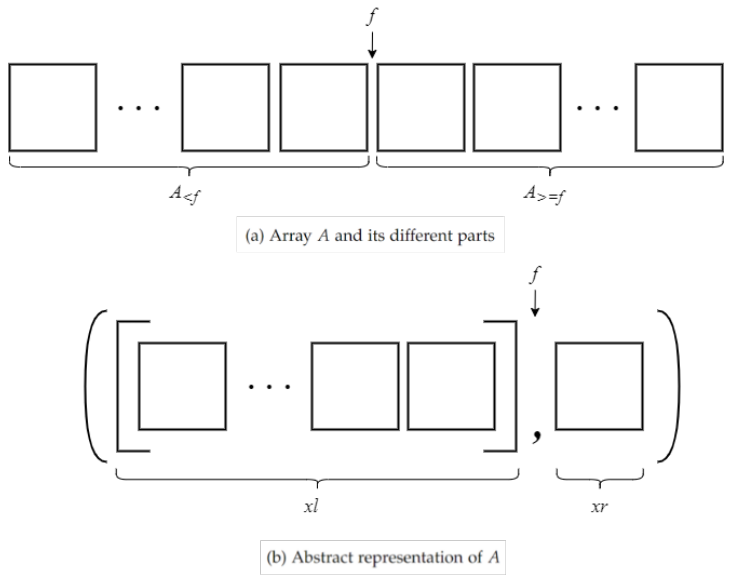


Figure 4.1: Graphic representation of an array A and its abstract representation.

employed for array contents. In the case of Goblint, $\mathbb{V}al$ is the *compound domain* and is the same domain as the one used for top-level variables. This means that arrays can contain anything a top-level variable can contain (i.e. not just scalar values but also structures, unions, or arrays, allowing for representing array contents of types that are an arbitrarily deep nesting of all of them). $A_{<f}$ is a list of abstract values, which are also from $\mathbb{V}al$.

Even though accessing to specific index on the *xl* list will be necessary, using arrays was not considered a good option since they are mutable structures. That means that Goblint would need to create a copy when modifying the array. Otherwise it would be modifying the original one. Since we assume that the factor f should not be too big, the cost of iterating the list each time is considered acceptable.

4.2 Array initialization

Initially, Goblint used only the *Trivial* domain, which uses an smashing approach and is initialized with $\bot$. As explained in Section 2.2, this in only sound as long as the arrays are never read at indices where they are not initialized. In order to solve this, the *Partitioned* domain was created. It initializes the array to $\top$, which combined with the partitioning approach provides Goblint with sound array abstraction.

For our new array domain implementation, it makes sense to translate the combination between the smashing and the expansion approaches we implement to the initialization of the array. For that, each value on the first element of the domain (*xl*), will be initialized to

$\top$. Since each abstract value represents a single index of the array, it is reasonable to have this value until specifically accessed. However, for the second element of the domain (*xr*), initializing it to $\top$ determines that Goblint will not find out anything about the summarized part of the array. For that reason, it is initialized to $\bot$, following the same approach as *Trivial*.

4.3 Less or Equal Relation

The less or equal relation for abstract values for arrays is shown in Figure 4.2.

For two abstract values A and B, both representing arrays implemented in our new domain, A is less or equal to B if and only if one condition holds: first, the smashing of all values of the list in A needs to be less or equal to the smashing of all the values of the list in B. Second, the second value on A need to be less or equal to the second value of B. Both abstract values will necessarily have the same unrolling factor f, since it is unique for the whole analysis.

$$(A_{<f}, A_{\geq f}) \sqsubseteq (B_{<f}, B_{\geq f}) \Leftrightarrow$$
$$(A_i \sqsubseteq B_i,\ \forall\, i \in \{0, 1, ..., f-1\}) \wedge (A_{\geq f} \sqsubseteq B_{\geq f})$$

Figure 4.2: $\sqsubseteq$ relation for abstract values for two arrays A and B.

4.4 Least Upper Bound

The least upper bound operation for abstract values representing arrays is given in Figure 4.3.

Both abstract values will necessarily have the same unrolling factor f, since it is unique for the whole analysis. The least upper bound for two values on the domain will keep the same structure. The values in indexes lower than f, which are on the *xl* list, are obtained applying the least upper bound on elements from both arrays with the same index (e.g., $A_1 \sqcup B_1$). The second element (*xr*), is obtained with the least upper bound of $A_{\geq f}$ and $B_{\geq f}$.

$$(A_{<f}, A_{\geq f}) \sqcup (B_{<f}, B_{\geq f}) =$$
$$([A_1 \sqcup B_1, A_2 \sqcup B_2, ..., A_{f-1} \sqcup B_{f-1}], A_{\geq f} \sqcup B_{\geq f})$$

Figure 4.3: The least upper bound of two abstract values for arrays.

4.5 Writing to an array

When writing to an array, what the *set* function receives is an index i to be updated, and a value v to be used for the update. The index i has the form of an interval, and it contains two values *min* and *max*, which are the ones being used for the update. The operation description is given in Figure 4.4.

An important distinction needs to be made between *weak updates* and *strong updates*. When the program is not sure where the update is taking place, it will return an interval index i with a range of indexes of the array. In this case, a weak update will be performed; for each index of the array, the value v will be joined to abstract values in that index. There is also the case in which *min* is equal to *max*. In this case, a strong update will be performed; the value v will override what the index i contained before, since Goblint knows for sure that is the index being accessed. That distinction is only relevant on the expanded side of the array. Since xr only contains one value representing multiple elements, we can never strongly update it.

The following situations can occur:

- *min* is equal or greater than the factor f: In this case, the update is only taking part on the second part of the array (xr), the smashed part. Since there is only one value for this whole range, the action performed is to join the current values of xr with v.
- *max* is lower than the factor f: In this case, the update is only taking place on the first part of the array (xl), the expanded part. To update it, Goblint will check if a strong or a weak update is necessary, and perform it.
- *min* is lower and max is equal or greater than the factor f: In this case, actions need to be performed in both parts of the array. That means, the value v will be joined with xr and also with the values from xl contained in the index i.

$$
[\![A[i] = x]\!]^\# D = \begin{cases}
D \oplus \{A \mapsto (A_{<f}, xr)\} & if\ min^\# i \geq f \\
\quad where & \\
\quad\quad xr = A_{\geq f} \sqcup [\![x]\!]^\# D & \\
D \oplus \{A \mapsto (xl, A_{\geq f})\} & if\ max^\# i < f \wedge min^\# i = max^\# i \\
\quad where & \\
\quad\quad xl = \{[A_1, A_2, ..., A_{f-1}] | A_{min^\# i} = [\![x]\!]^\# D\} & \\
D \oplus \{A \mapsto (xl, A_{\geq f})\} & if\ max^\# i < f \wedge min^\# i \neq max^\# i \\
\quad where & \\
\quad\quad xl = \{[A_1, A_2, ..., A_{f-1}] | ((m \geq min^\# i \wedge m \leq max^\# i) ? A_m \sqcup [\![x]\!]^\# D : A_m)\} & \\
D \oplus \{A \mapsto (xl, xr)\} & otherwise \\
\quad where & \\
\quad\quad xl = \{[A_1, A_2, ..., A_{f-1}] | ((m \geq min^\# i \wedge m \leq max^\# i) ? A_m \sqcup [\![x]\!]^\# D : A_m)\} & \\
\quad\quad xr = A_{\geq f} \sqcup [\![x]\!]^\# D &
\end{cases}
$$

Figure 4.4: Writing to an unrolled array A at index expression i.

4.6 Reading from an array

The operation when reading from an array is given in Figure 4.5

When reading from the array, what the *get* function receives is an index i to be read. In the same way as when writing to an array, the index i has the form of an interval, and it contains two values *min* and *max*, which are the ones being used for the reading.

Differently to when writing to an array, there is only one possible result to return; a single abstract value containing all joined values of indexes indicated in i. The way to get this result changes depending on the specific case:

- *min* is equal or greater than the factor f: In this case, returning xr is enough.
- *max* is lower than the factor f: In this case, the values in xl which are in the range of i must be joined and returned.
- *min* is lower and max is equal or greater than the factor f: In this case, the values in xl which are in the range of i must be joined and also joined with xr.

$$[\![A[i]]\!]^{\#}D = \begin{cases} A_{\geq f} & if\ min^{\#}i \geq f \\ \bigsqcup_{j=min^{\#}i}^{max^{\#}i} A_j & if\ max^{\#}i < f \\ (\bigsqcup_{j=min^{\#}i}^{f-1} A_j) \sqcup A_{\geq f} & otherwise \end{cases}$$

Figure 4.5: Reading from an unrolled array A at index expression i.

As explained in Section 1.5, both array domains currently in use have a *WithLength* alternative. This was also implemented for our new domain. This way, we ensure that no out-of-bounds accesses are occurring.

4.7 Simple Example

In order to illustrate the aspects mentioned on the previous sections, an example of the program in Figure 4.6 will be given. Figure 4.7 shows the corresponding control flow graph. Assume that intervals are used to represent integers. As seen in Figure 4.8, at program point

```
// Assume factor is set to 3
int A[10];

A[0] = 0;
A[2] = 2;
A[5] = 5;
A[7] = 7;
```

Figure 4.6: Example program without loops.

(0), no information about the array is available and therefor all elements on *xl* have value $\top$ and *xr* has value $\bot$, as explained previously on Section 4.2.

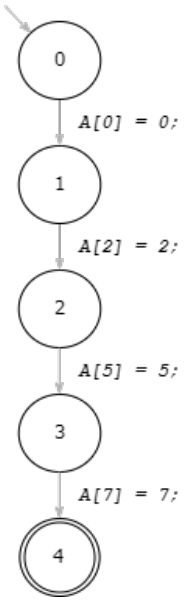


Figure 4.7: Control flow graph.

	A
0	([$\top$, $\top$, $\top$], $\bot$)
1	([[0,0], $\top$, $\top$], $\bot$)
2	([[0,0], $\top$, [2,2]], $\bot$)
3	([[0,0], $\top$, [2,2]], [5,5])
4	([[0,0], $\top$, [2,2]], [5,7])

Figure 4.8: Program points and states.

Between points (0) and (2), the array is accessed and written on elements with index lower than the factor f chosen. That means that Goblint will be able to precisely determine which value it needs to update, and therefore perform a strong update, which overrides the $\top$ value that was there previously. Thus, in point (2) the program would be able to identify an access to $A[1]$ as unknown, while precisely returning $A[0]$ and $A[2]$'s values.

At point (3), $A[5]$ is accessed. Due to 5 being greater than the factor f (3), the value is simply joined with the values *xr* already contained; $\bot$ in this case. The table on picture 4.8 shows then the result of $\bot \sqcup [5,5] = [5,5]$. A similar operation takes place at point (4); since the value accessed is greater than the factor, it is joined with the previous value on *xr*, which is now $[5,5]$. That results in: $[5,5] \sqcup [7,7] = [5,7]$.

From points (3) to (4), it is important to mention that Goblint can join the values the way it does due to initializing *xr* to $\bot$. Contrarily, if it were initialized to $\top$, it could not find out anything about *xr*, since $\top \sqcup [5, 5] = \top$.

5 Combining Both Techniques

As said in the introduction, the implementation of this precision-increase transformation was divided into two parts. The first was done on the CIL side of Goblint, and it's the loop unrolling itself. The second task was to add a new abstract array domain on the analyzer part of Goblint, so that it worked better together with the new loop unrolling implemented on CIL.

As previously illustrated, both techniques slightly increase precision on their own. However, it is when brought together that the desired loop precision increase takes place. In this section we will explain why joining both techniques is necessary, and benefits of this implementation, as well as limitations found. Some simple and more complex examples are given as mode of illustration.

It has been seen in Section 3.2 that the loop unrolling implementation on its own increases precision when the loop is fully unrolled, since variable paths can be precisely tracked. Section 4.7 displayed the precision increase on array tracking in sequential programs. The two techniques work great together since enabling them at the same time, translates to iterative code being transformed to sequential code. At the same time, the variables which will are being represented precisely (e.g. the first k index of the array), will ideally only be updated during this sequential code. Essentially, more strong updates will be performed on the array, since the level of abstraction is lower.

5.1 Limitations

This implementation deals with some limitations, explained in this section.

First, it is only possible to precisely track the first k elements of the array if those are also the first k elements to be accessed. In any other case, if the first k accessed elements on an array are not the first k ones, but the last ones or a subset in the middle, it is not possible anymore to track them precisely.

Second, not knowing the loops iteration count is reducing optimization potential and wasting resources. In loop with far more iterations than the ones unrolled, the extra effort of unrolling them and applying the new abstract array domain, might be not worth it. In other cases, where the loop iteration count is shorter than anticipated, the extra unrolled iterations are

pointless.

	i	A
0	[0,0]	([⊤, ⊤, ⊤], ⊥)
1	[0,0]	([⊤, ⊤, ⊤], ⊥)
2	[0,0]	([[0,0], ⊤, ⊤], ⊥)
3	[1,1]	([[0,0], ⊤, ⊤], ⊥)
4	[1,1]	([[0,0], ⊤, ⊤], ⊥)
5	[1,1]	([[0,0], [1,1], ⊤], bot)
6	[2,2]	([[0,0], [1,1], ⊤], ⊥)

	i	A
7	[2,2]	([[0,0], [1,1], ⊤], ⊥)
8	[2,2]	([[0,0], [1,1], [2,2]], ⊥)
9	[3,10]	([[0,0], [1,1], [2,2]], [3,∞])
10	[3,9]	([[0,0], [1,1], [2,2]], [3,∞])
11	[3,9]	([[0,0], [1,1], [2,2]], [3,∞])
12	[10,10]	([[0,0], [1,1], [2,2]], [3,∞])

Figure 5.1: Program points and states

Last, due to the way *xr* is initialized in the new abstract array domain, that second element of the array has the same problem as the *Trivial* domain; unsoundness. When the arrays are read at indices where they have not been initialized, the program is not sound.

5.2 Simple Example

By way of illustration, an example with a simple loop will be given. Consider the code in Figure 5.2. Figure 5.3 shows the generated control flow graph by Goblint, without using unrolling, and Figure 5.4 the control flow graph with unrolling and factor 3. Assume that intervals are used to represent integers.

```
// Assume factor set to 3
int A[10];
int i = 0;

while(i<10){
    A[i] = i;
    i++;
}
```

Figure 5.2: Example program with a simple loop

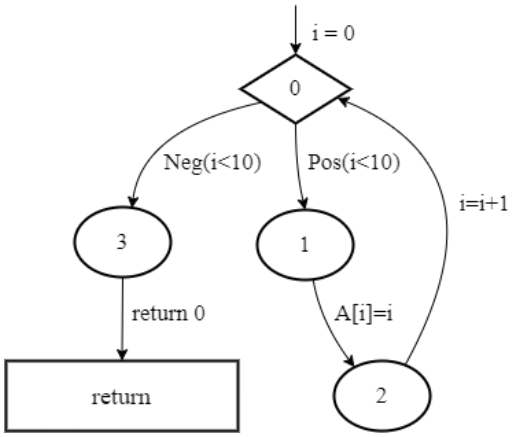


Figure 5.3: CFG generated by Goblint of the given loop

As seen in Figure 5.1, until program point (2), no information about the array is available and it therefore has the value ⊤ for all elements of *xl*, and ⊥ for *xr*. Right before program points

(2), (5), and (8) the array is written at indexes lower than the factor k given (3). Due to those instructions being written on sequential code, and to having precise abstract representation of each of the indexes accessed, the array precisely describes the values assigned.

From point (9) until (11), since each program point needs to summarize all iterations left, as well as the right part of the array representation, the abstraction level increases. In (12), after the loop execution, the first k values of the array are still visible for future accesses.

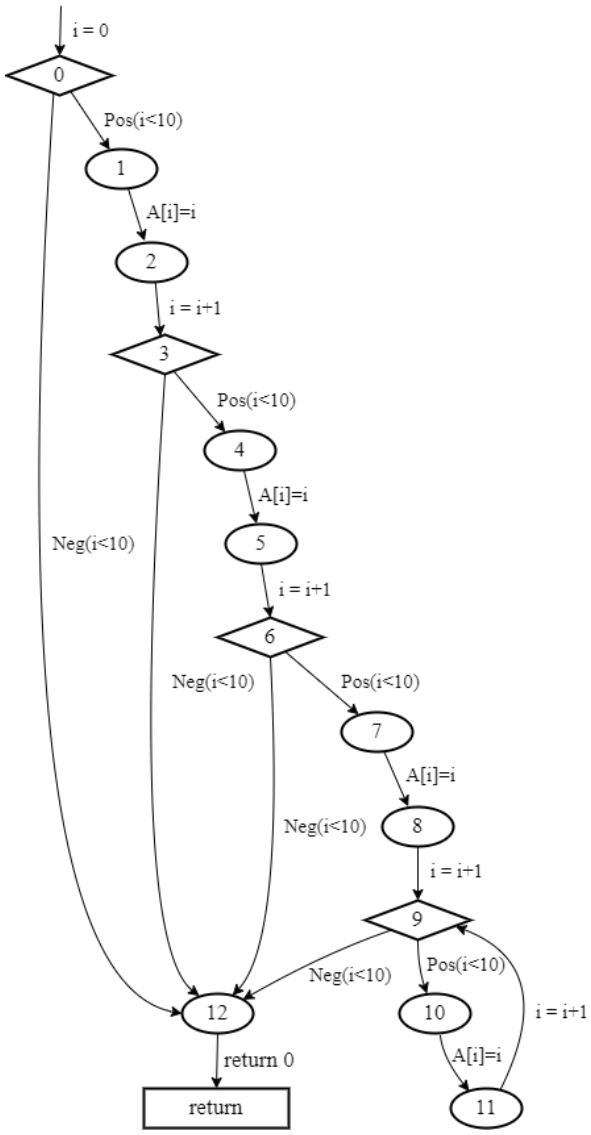


Figure 5.4: control flow graph of the unrolled code given

6 Evaluation

For evaluation, the approach presented in this thesis is compared to the current state of Goblint, i.e. loops are not unrolled and they are performed with the *TrivialWithLength* or the *PartitionedWithLength* array domains. In addition, different values of the unrolling factor, in order to also evaluate various performances, will also be tested.

There are two interesting dimensions along which this new approach will be compared. First of all, it is interesting to see how precision improved. For this, in the section immediately following (6.1), some of the cases the domain can handle are presented using examples. Section 6.2 answers the question what costs these improvements were obtained by reporting on the performance overhead for real-world programs.

6.1 Precision

Apart from the running examples already shown thought this thesis (Figures 3.7a, 4.6 and 5.2), the approach presented in this thesis is also powerful enough to deal with a variety of other interesting cases. In all examples in this section, the combination of loop unrolling and the new abstract array domain, both with same given factors, will be used. Although both techniques increase precision when used individually, this thesis' goal was to increase precision in loops, which is mainly done with the combination of both. For all these examples, it is once more assumed that intervals are used to represent integers. Also, consider factor f 5.

Figure 6.1a shows that this approach is also able to deal with iterating over the same array once more to modify values after it has been completely initialized before. After the first loop, the abstract value for A is $([\,[2,2],[2,2],[2,2],[2,2],[2,2]\,],[2,2])$. The final abstract value after both loops is $([\,[4,4],[4,4],[4,4],[4,4],[4,4]\,],\top)$.

Figure 6.1b demonstrates once more that this approach is able to precisely identify different initializations on the first k (5 in this case) of the array. The final abstract value after the loop is $([\,[0,0],[1,1],[0,0],[1,1],[0,0]\,],[0,1])$.

The ability to deal with arrays in nested loops is demonstrated in Figure 6.1c. Since the unrolling factor is the same as the iteration count of the internal loop, it is fully unrolled. This means, that on the first 5 iterations of the external loop, the elements of B can be precisely estimated. At the end of the loop unrolling, right before the remainder loop, the value of A is

$([\,[0,0],[1,1],[2,2],[3,3],[4,4]\,],\bot)$, and B is $([\,[0,0],[10,10],[20,20],[30,30],[40,40]\,],\bot)$. Once the analysis enters the external remainder loop, the abstraction is higher and so precision is lost. The final abstract values for A and B are $([\,[0,0],[1,1],[2,2],[3,3],[4,4]\,],[5,\infty])$ and $([\,[0,0],\top,\top,\top,\top\,],\bot)$, respectively. Even thought not so high, the precision for this domain is higher than for the two current domains in use (*TrivialWithLength* and *PartitionedWithLength*).

```
int A[20];
int i = 0;
int j = 0;

while(i<20) {
    A[i] = 2;
    i++;
}

while(j<10) {
    A[j] = A[j]*2;
    j++;
}
```

(a) Several iterations

```
int A[20];
int i = 0;

while(i<20){
    A[i] = 0;
    i++;
    A[i] = 1;
    i++;
}
```

(b) Different Initializations

```
while(i < 20){
    A[i] = i;
    int j = 0;
    while(j < 5){
        B[j] += A[i] * j;
        j++;
    }
    ++i;
}
```

(c) Nested loops

Figure 6.1: Examples showing various cases.

6.2 Performance

To investigate how expensive the loop unrolling and array analysis is, its run-time is compared to the non-unrolled one, while using the *PartitionedWithLength* array domain.

#	Name	f=0 [s]	f=2 [s]	f=5 [s]	f=10 [s]
1	array-examples/data_structures_set_multi_proc_trivial_ground.c	.146	.193	.402	1.85
2	array-examples/sanfoundry_43_ground.c	.126	.142	.213	.692
3	array-examples/standard_copyInit_ground.c	.130	.147	.234	.745
4	array-examples/standard_init1_ground-2.c	.114	.127	.204	.699
5	array-tiling/pnr2.c	.249	.365	.565	1.28
6	array-tiling/pnr3.c	.347	.505	.793	1.67
7	array-tiling/pnr4.c	.423	.667	1.11	2.20
8	array-tiling/pnr5.c	.584	.884	1.40	2.83
9	array-cav19/array_init_pair_sum_const.c	.157	.188	.286	.934
10	array-lopstr16/flag_loopdep.c	.235	.184	.277	.825

Figure 6.2: Run-time of the new approach with various values for factor f.

The test cases are a set of programs from the SV-COMP (Competition on Software Verification). SV-COMP [21] is an annual comparative evaluation of fully automatic software verifiers for C and Java programs. In order to choose a suitable factor f for the comparison, and to see

how the analysis acts with different values of f, Figure 6.2 shows how the run-time increases depending on different values of factor f.

When factor f is 10, the overhead of the analysis is too high. On the other side, when f is 2 the overhead obtained is acceptable, yet the precision gained by unfolding loops 2 times may not be worth it. When f is 5, a balance between both is obtained. For this reason, the factor f used for the rest of the evaluation will be 5.

Figure 6.3 shows the run-time of both the current version of Goblint using the *PartitionedWithLength* array domain (*Old*), and the new approach using factor f 5 for the loop unrolling as well as for the new array domain (*New*).

#	Name	Old [s]	New[s]	Difference [s]	
1	array-examples/data_structures_set_multi_proc_trivial_ground.c	.146	.402	+ .256	+63%
2	array-examples/sanfoundry_43_ground.c	.126	.213	+ .087	+40%
3	array-examples/standard_copyInit_ground.c	.130	.234	+ .104	+40%
4	array-examples/standard_init1_ground-2.c	.114	.204	+ .09	+4%
5	array-tiling/pnr2.c	.249	.565	+ .316	+55%
6	array-tiling/pnr3.c	.347	.793	+ .446	+56%
7	array-tiling/pnr4.c	.423	1.11	+ .687	+61%
8	array-tiling/pnr5.c	.584	1.40	+ .816	+58%
9	array-cav19/array_init_pair_sum_const.c	.157	.286	+ .129	+45%
10	array-lopstr16/flag_loopdep.c	.235	.277	+ .042	+15%

Figure 6.3: Run-time of the new and old approach for the analysis of various real-world programs.

As expected, the new approach takes longer to analyze the programs. While the overhead is just 4 percent for one program, it ranges between 40 and 60 percent in most cases.

7 Conclusions and Future Work

7.1 Conclusions

As seen in the previous sections, Goblint currently processes loops in a general-way-constructed graph, and then also provides two options for the representation of the array domain. This may be enough in most cases, yet it might lack precision in some others. For that reason, this project was aimed to precisely calculate the first k iterations of the loop, leading to better analysis results.

The loop unrolling and abstraction for arrays approach presented in this thesis manages to increase precision on loops. This simple idea supports different kinds of loops as well as different array initialization, access and updates.

Even thought this approach adds a considerable overhead to the analysis, loop unrolling is an extra parametrization element which is useful to have in a static analyzer like Goblint. It is not meant to always be used, but to experiment with it and use it as an additional tool. At some later point, other more intricate analyses such as those presented in the related work (Section 7) or future work (next section) of this thesis may be used to further improve precision and reduce the overhead.

7.2 Future Work

The most obvious direction for further improvements to this loop unrolling is to apply different kinds of unrolling for the different loops, depending on their necessities or circumstances. Currently, an aggressive approach is being implemented, but all loops are unrolled following the same scheme and the same unrolling factor. This is probably denying the optimization some of its potential. Various mechanisms could be implemented for this purpose.

First, knowing how many iterations the loop is performing would help unroll loops in a more accurate and personalized way (e.g., loops could unroll X% of their iterations, so that no excessive unrolls are performed). Precisely knowing the loop iteration counts will help determine the suitable unrolling factor, which will allow to exploit maximal optimization potential [6].

Another possible improvement could lie in developing some better way to predict the number of times the loop should be unrolled, considering the iterations and also some internal properties of the loop. For that, having some heuristic loop unrolling which automatically determines suitable unrolling factors, or some machine learning high level insight would be useful. It could also help in telling which loops are worth unrolling, and which ones should be left alone.

On top of this, a further way to increase precision is to improve the unroll array domain. So that not always the first k values of the array are the ones precisely depicted, but those first k to be accessed (e.g., the domain could unroll the last elements of the array instead of the first ones, or some subset in the middle).

Finally, another avenue worth investigating is to perform the unroll on the Goblint side, instead of in CIL. This implementation would be really interesting, since it would not require a modification of the input program like our implementation does. However, it would potentially be more expensive. It could be implemented in loops by identifying a counter value (e.g., an i in the loop condition), which would be used for path sensitivity. This way, for each value of the counter value, there would be a different state.

List of Figures

1.1 Representation of how Goblint and CIL work together. 2
1.2 Basic *while* loop and its CIL transformation. 3
1.3 Different loop unrolling transformations on the same loop. 4
1.4 Visualization of Abstract Interpretation [12]. 7

2.1 Counting loops, example from [7]. 10

3.1 Pseudo-code of the unroll function. 14
3.2 Graphic representation of a simple while loop in C code, CIL, and unrolled. . 15
3.3 Graphic representation of nested loops in C code, CIL, and unrolled. 16
3.4 Simplified view of an already unrolled loop. 17
3.5 Manual handling of *break* instructions. 18
3.6 Manual handling of *continue* instructions. 18
3.7 Example program containing a simple loop. 19
3.8 Control flow graph of the unrolled code in Figure 3.7a. 20
3.9 Program points and states. 20

4.1 Graphic representation of an array A and its abstract representation. 22
4.2 $\sqsubseteq$ relation for abstract values for two arrays A and B. 23
4.3 The least upper bound of two abstract values for arrays. 23
4.4 Writing to an unrolled array A at index expression i. 24
4.5 Reading from an unrolled array A at index expression i. 25
4.6 Example program without loops. 26
4.7 Control flow graph. 26
4.8 Program points and states. 26

5.1 Program points and states 29
5.2 Example program with a simple loop 29
5.3 CFG generated by Goblint of the given loop 29
5.4 control flow graph of the unrolled code given 30

6.1 Examples showing various cases. 32
6.2 Run-time of the new approach with various values for factor f. 32

6.3 Run-time of the new and old approach for the analysis of various real-world programs. 33

Bibliography

[1] V. Vojdani and V. Vene. "Goblint: Path-sensitive data race analysis". In: *Annales Univ. Sci. Budapest., Sect. Comp*. Vol. 30. Citeseer. 2009, pp. 141–155.

[2] *Goblint - GitHub page*. `https://github.com/goblint/analyzer`. [Online; Last Accessed: 27-December-2021].

[3] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs". In: *International Conference on Compiler Construction*. Springer. 2002, pp. 213–228.

[4] *CIL Goblint - GitHub page*. `https://github.com/goblint/cil`. [Online; Last Accessed: 5-October-2021].

[5] S. Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[6] P. Lokuciejewski and P. Marwedel. "Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization". In: *2009 21st Euromicro Conference on Real-Time Systems*. IEEE. 2009, pp. 35–44.

[7] J. W. Davidson and S. Jinturkar. *An aggressive approach to loop unrolling*. Tech. rep. Citeseer, 1995.

[8] L. Song and K. Kavi. "What can we gain by unfolding loops?" In: *ACM SIGPLAN Notices* 39.2 (2004), pp. 26–33.

[9] A. Koseki, H. Komastu, and Y. Fukazawa. "A method for estimating optimal unrolling times for nested loops". In: *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*. IEEE. 1997, pp. 376–382.

[10] P. Cousot and R. Cousot. "Abstract interpretation frameworks". In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[11] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[12] P. Cousot. "Abstract interpretation". MIT course 16.399, `http://web.mit.edu/16.399/www/`. Feb. 2005.

[13] M. B. Schwarz. "Sound Yet Precise Modelling of Arrays in Abstract Interpretation". In: *Mast*. 2019.

[14] M. Rodriguez-Cancio, B. Combemale, and B. Baudry. "Approximate loop unrolling". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 2019, pp. 94–105.

[15] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, et al. "Astrée: Proving the absence of runtime errors". In: *Proc. of Embedded Real Time Software and Systems (ERTS2 2010)* (2010), p. 9.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 2003, pp. 196–207.

[17] P. G. Szécsi, G. Horváth, and Z. Porkoláb. "Improved Loop Execution Modeling in the Clang Static Analyzer". In: *Acta Cybernetica* (2020).

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software". In: *The essence of computation*. Springer, 2002, pp. 85–108.

[19] P. Cousot, R. Cousot, and F. Logozzo. "A parametric segmentation functor for fully automatic and scalable array content analysis". In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 105–118.

[20] D. Gopan, T. Reps, and M. Sagiv. "A framework for numeric analysis of array operations". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 338–350.

[21] *SV-COMP 2022*. `https://sv-comp.sosy-lab.org/2022/`. [Online; Last Accessed: 7-February-2022].