DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automatic Configuration of Goblint - Tuning for Efficient, Yet Precise Enough Analyses of Programs

Manuel Pietsch

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automatic Configuration of Goblint - Tuning for Efficient, Yet Precise Enough Analyses of Programs

Automatische Konfiguration von Goblint - Tuning für effiziente und hinreichend präzise Analyse von Programmen

Author:	Manuel Pietsch
Supervisor:	Prof. Dr. Helmut Seidl
Advisors:	Michael Schwarz, Julian Erhard
Submission Date:	15.09.2022

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

Munich, 12.09.2022 Manuel Pietsch

Abstract

Many configuration options of static analyzers are trade-offs between analysis time and precision. They require knowledge about the internal workings to select them optimally. This thesis implements an autotuner for Goblint that changes options based on the analyzed program. Further, the ability to enable some of these options for only specific parts of the code using attributes is added. The autotuner is evaluated with tests from the software verification competition SV-COMP and manages to increase the number of correct answers for the *No Overflow* and *Reachability* categories by 45% and 9.5%, respectively. At the same time we decrease the average time needed for the analysis by 3% and 16%, respectively. For the *Data Race* category, the autotuner was not able to improve Goblint's result.

Contents

1 Introduction

It is often useful to find out invariants of a program without having to execute it in every possible condition, which most of the time is infeasible. One, for example, might be interested in making sure that a certain part of the code never deadlocks or has a runtime error like dereferencing a null pointer. This is especially important in some areas, e.g. a flight controller in a plane, where losing control could have catastrophic consequences. Even if the result of a failure is not as extreme, most programs still profit from detecting bugs before they appear or from proving the absence of certain bugs. [17]

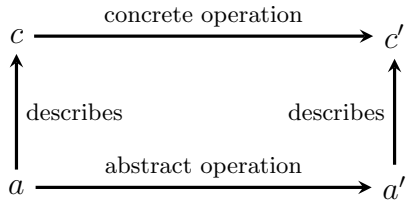
Tests are limited to the paths they execute and can only reveal bugs but never exclude the possibility of them. *Static analysis*, on the other hand, can be used to analyze the source code and show that no bugs of certain kinds exist.

The technique for static analysis this thesis deals with is *abstract interpretation*, which we will explain in sections 1.1 and 1.2. Instead of concrete values, abstract values that can stand for multiple values are used to condense many possibilities. As a result, many paths can be considered at once. Based on the choice of abstraction this also includes a different amount of additional paths and values never occurring in the original program. Selecting an abstraction is therefore often a trade-off between more precision and a reasonable runtime.

As there is no best choice for all programs, static analyzers offer the user many configuration settings. The amount of options as well as the domain-specific knowledge that is required to know which to use make it difficult to tune static analyzers towards a programs specific need.

The goal of this thesis is to transfer some of this work from users of the static analyzer *Goblint* (section 1.3) to Goblint itself. For this, we implement an autotuner that for some options tries to estimate which of them should be activated based on simple features of the analyzed file. To aid this, we increase the precision with which some of these options can be configured.

First we will present an overview of abstract interpretation, Goblint and the software verification competition SV-COMP. Chapter 2 describes existing approaches to tuning static analyzers depending on the analyzed file. In chapter 3 we explain the new features and heuristics we implemented. The impact of them is then measured and discussed in chapter 4. In the end, we conclude our work and outline what steps can be taken in the future.



(a) Relation between abstract and concrete values

```
1 int x = 0;
2 if (randInt() > 0) {
3     x = 2;
4 }
5 //x=[0,2]
6 assert (x != 3);//true, known
7 assert (x != 1);//true, unknown
```

(b) Example for precision loss when using intervals

Figure 1.1: Abstract interpretation

1.1 Overview of Abstract Interpretation

The concept of abstract interpretation was first introduced by Patrick and Radhia Cousot in [3] as a generalization of existing analysis techniques. When interpreting a program, the concrete values as well as the concrete operations on them are replaced by abstract ones. The important relationships between them are displayed in Figure 1.1a: an abstract value describes some concrete ones. The abstract operations have to be defined so that if a describes c, the result of applying the concrete operation to c has to be described by the outcome of applying the corresponding abstract operation to a.

For example, a common abstraction is to represent integers with intervals. An interval $[l, u]$ has a lower and an upper bound and describes all numbers x with $l \leq x \leq u$. When adding two intervals $[l_1, u_1]$ and $[l_2, u_2]$ the resulting interval has to encompass all possible results and thus at best is $[l_1 + l_2, u_1 + u_2]$.

We can easily see that this allows us to make some statements about the original program but loses some precision. Take for example the code in Figure 1.1b. As we do not know if the path of the if-statement is taken, the tightest representation of x in line 5 as an interval is $[0, 2]$. This information is still enough to conclude that the first assertion will always hold but for the assertion in line 7 we have lost the knowledge that x never will be 1.

Abstract interpretation is called an over-approximating method because it always encompasses all possible values as well as potentially additional ones. Furthermore, it means that in most cases properties can only be verified: if it is satisfied by all concrete values the abstract value describes, it is also satisfied in the real program. If not all of them fulfill it, no conclusion can be drawn because this could be a result of the abstraction including some values that are not possible in the actual program.

This trait allows a static analyzer to potentially be *sound*, meaning it finds all possible faults and therefore if no faults are reported it proves that there are none. This comes at the cost of including false positives.

The set of abstract values is called a domain and is required to be a complete lattice. That means it is a partial order with a comparison $\leq$ and every subset has to have a least upper bound, which is the smallest element greater or equal to every element in the subset. Fulfilling this also implies the existence of a greatest lower bound. The operation of finding these for two elements is commonly called *join* (upper bound) and *meet* (lower bound).

```
1 int i = 0;
2 while(i < 10) {
3     assert(i < 20);
4     i++;
5 }
```

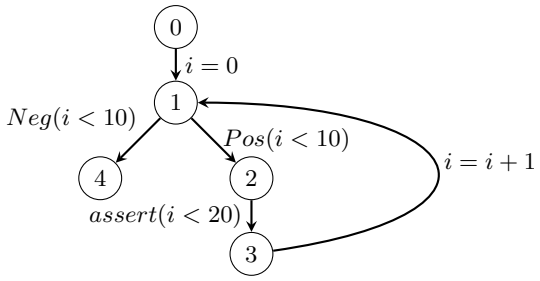


Figure 1.2: A simple while loop and its control flow graph

	1	2	3	4	...	12	13
0	$\bot$	$\bot$	$\bot$	$\bot$		$\bot$	$\bot$
1	$\bot$	$[0,0]$	$[0,1]$	$[0,2]$		$[0,10]$	$[0,10]$
2	$\bot$	$[0,0]$	$[0,1]$	$[0,2]$		$[0,9]$	$[0,9]$
3	$\bot$	$[0,0]$	$[0,1]$	$[0,2]$		$[0,9]$	$[0,9]$
4	$\bot$	$\bot$	$\bot$	$\bot$		$[10,10]$	$[10,10]$

(a) Without widening

	1	2	3	4
0	$\bot$	$\bot$	$\bot$	$\bot$
1	$\bot$	$[0,0]$	$[0,\infty]$	$[0,\infty]$
2	$\bot$	$[0,0]$	$[0,10]$	$[0,10]$
3	$\bot$	$[0,0]$	$[0,10]$	$[0,10]$
4	$\bot$	$\bot$	$[10,\infty]$	$[10,\infty]$

(b) With widening

Figure 1.3: Solving the constraint system for i of the program in Figure 1.2

Moreover, one element, called top, has to be greater or equal to every element in the domain and similarly a smallest element, called bot, must exist. The interval domain therefore would not be complete unless we add ∞ as possible bound (with $[-\infty, \infty]$ being top) and the special element $\bot$ as bot, which stands for uninitialized values.

1.2 Using Abstract Interpretation

To find out the values the variables can have, a constraint system is created based on the control flow graph. The constraint at every point where the control flow can be is defined as the least upper bound of all abstract elements that result from applying the abstract operations to the possible previous program states. For an example see Figure 1.2, which contains a simple program and the control flow graph of the same program. As a simplification, we now only consider the variable i because it is the only one. The actual objects the constraint system works with is either a map from variables to their abstract values or $\bot$ if the node is not reachable. At node 4, we have the possible values from node 1 that are not less than 10. At node 1, we come either from nodes 0 or 3. If we come from node 0, i is definitely 0. If we come from node 3, i is one more than it was there. This leads the constraint for i at node 1 to be the join of 0 and $i_3 + 1$.

To find a solution to the constraints we first set the values at all nodes to bot. One by one, every constraint gets applied to the known information to update it. This is repeated until the output no longer changes and a fixpoint is reached. Figure 1.3a shows this for our example, restricted to just the variable i and updating each constraint in increasing order every iteration. It is guaranteed that at the fixpoint the abstract values describe all possible real ones. In our example, we can use this information to check whether the assertion in line 3 holds.

The iteration count until the fixpoint is reached is obviously critical for the speed of the analysis. One way to optimize it is to intelligently choose which nodes to update. Most of the time, a more sophisticated algorithm than described here is used, but the basic principle remains the same. Another way of optimizing is *widening*. If at some point in the iteration a value increases, a larger upper bound is used instead of the least upper bound. This guarantees that a value can not change too often. In the case of intervals, instead of increasing the interval bounds to the larger value, they jump to infinity at once. In figure 1.3b we can see the solving of our example program when widening is used. In exchange for fewer iterations, we lose some information at node 4. Some of this can be recovered, e.g. by a technique called *narrowing*, or using widening thresholds, which we will explain in section 3.4.

1.3 Goblint

Goblint is a static analyzer for C written in OCaml. It is developed on GitHub at the *Laboratory for Software Science* at the University of Tartu and the *Chair of Formal Languages, Compiler Construction and Software Construction* at the Technical University of Munich.[1] While it was initially presented in 2007 in [18] as a tool for detecting data races and still focuses on that, it has been expanded with many further features, for example detecting overflows or reachability of certain points of interest.

One aim of Goblint is to be able to analyze large real-world programs in a reasonable time and with nearly all features of C. There are a few exceptions, e.g. inline assembler. Goblint is designed to be sound and report all errors of the categories it supports. It is mostly successful in doing so but currently has some bugs of its own.

Goblint provides more than one option for the abstract domain of some types such as integers and arrays. For integers, multiple domains can work together and profit from each other. The implemented analyses are modular and can be activated individually. The user can specify these and other settings either via the command line or a JSON-file. Some options may be more precisely configured by Goblint-specific annotations.

1.4 The Competition on Software Verification (SV-COMP)

The *Competition on Software Verification* is an annual competition that compares software-verification tools for C and Java. It aims to provide a technique-independent comparison. For this, it maintains a large number of tests which we utilize for our evaluation. These benchmarks range from ones requiring analysis of intricate conditions to large real live programs.

A test consists of a program file and a property that needs to be considered. For our purpose we are only interested in the following three from the categories that Goblint participates in: *Reachability* (Will a specified function never be called?), *No Overflow* (Are no overflows possible?) and *No Data Race* (Is there never a data race?).

[1] `https://github.com/goblint/analyzer`

An analyzer can answer true or false if it can also provide a *witness* that can verify the answer. For our case, we ignore the witness and only check whether the answer matches the expected one. The scoring system punishes wrong results heavily by subtracting sixteen times the points gained for being correct. A third answer, unknown, is available if no certain conclusion has been drawn or the analyzer runs out of time or memory. As the over-approximation method used by Goblint can only prove the properties but not disprove them, Goblint reports either true or unknown to prevent wrong results and therefore leaves a lot of points unobtainable. On the upside, Goblint is sound and the only tool that had no incorrect answers in 2022. [1]

2 Related Work

To our problem of tuning a static analyzer, there are two different kinds of relevant work. Some researchers try to automatically find the best configuration for a single tool based on the analyzed file as we do. Others are developing portfolio solvers that have access to different tools, potentially including an analyzer multiple times with different fixed configurations, and give a prediction of which one will be the best for a specific problem.

For simplification of the problem space, all mentioned literature first extracts some features from the program and uses those for determining their outcome. The concept of features was introduced in [13] and describes simpler information about the original problem that is still relevant for the performance of a strategy.

[16] focuses on device drivers for windows and selects between three software checkers from the *Static Driver Verifier* framework from the windows driver kit. Many different features are used for training a machine learning function for each tool which predicts the needed analysis time. The tool with the lowest estimate runs the actual analysis, which considerably decreased the amount of timeouts compared to using just one checker. Furthermore, they determine the features with the most relevance to the prediction, which are the numbers of variables of the different types, conditional statements, and function calls.

Machine learning is also used by [8] for their tool that can be trained to select a configuration for an arbitrary static analyzer. For this, they utilize the simulated annealing search algorithm. Starting from a provided base configuration, a small number of changes to options are proposed. If the learned estimate for the new configuration is better or only slightly worse, it replaces the current one. This process then is repeated a certain number of times and the analysis is executed with the final selection. Allowing a worse configuration is needed to avoid local minima and the limit of how much it can worsen decreases over time. For their features, they count the different instructions in the intermediate representation of the compiler (LLVM IR for C and WALA for Java). They evaluated this algorithm against 4 Java and C analyzers that participate in SV-COMP. When comparing the algorithm with a random initial configuration to the competition configuration, it archived an increase in the number of correct tests for three of the tools, but also more wrong answers for all of them. Running the algorithm after the experts' configurations failed improved the precision of three of the analyzers.

For a more complicated feature, [6] tries to determine which roles variables perform in the program. Examples of roles are loop iterators or indexing arrays. With this, they have some success in predicting which SV-COMP category a program belongs to and later use it in [5] to improve predicate abstraction, another technique for static analysis, as well as in [4] for a portfolio solver that has an improvement of about 66% over the winner of

SV-COMP 2016. For each tool, they trained a predictor for whether it answers correctly, incorrectly or unknown. Then they choose the tool for which the prediction is most sure of it being correct or, if no tool is predicted to be correct, a tool with the answer unknown. Further features are classifying loops, depending on e.g. whether they are bounded, and control flow, e.g. the amount of basic blocks.

The last two papers improve upon CPAChecker, a tool that at the time of publishing was the overall winner of the latest SV-COMP (2018). The most successful variant of it was CPA-SEQ, which changes to a different configuration when the current one did not find a confident solution in a limited time frame. [14] aims to optimize the originally fixed order of the configurations by predicting which will perform best. This prediction is again made with a machine learning method. For the features, they search for structures in the abstract syntax tree combined with information from control flow and dependence graphs. They succeed in reducing the runtime and false positives but also find a disadvantage of the machine learning approach. While performing better than CPA-SEQ in 2018, the improved version of the latter used at SV-COMP in 2019 was slightly better and the training was not directly transferable.

That even without machine learning and with simpler features a selection can outperform constant choices is shown in [2]. The 4 properties *hasLoop*, *hasFloat*, *hasArray* and *hasComposite* (structs or unions) and a simple decision between three strategies, one of them being CPA-SEQ, are enough to achieve better results than CPA-SEQ.

A result common to all the mentioned literature is that their methods did not end up choosing one configuration or one tool for every program. Moreover, the result of mixing of strategies is reported to be better than all constant methods. This strongly supports the intuition that there is no single best configuration for all programs.

3 Implemented Autotune Features

In this chapter we describe the options the autotuner tries to tune, based on which features of the analyzed file this is done, and how the options were implemented. The code written for this thesis can be found at `https://github.com/goblint/analyzer/pull/772`.

The autotuner is enabled by setting the option `ana.autotune.enabled` to true. It is possible to select which of the autotune features are actually used by changing the array `ana.autotune.activated`. By default, this contains all of the features.[1]

The features "octagon" and "wideningThresholds" are special in that even when activated they are not always used. Instead, they calculate an estimate for their cost and value. After we also estimate the base complexity of the file that is analyzed, they only get enabled if the total cost remains below a threshold. Choosing them is a variant of the knapsack problem and a simple greedy heuristic (highest $value/cost$) prioritizes between them. It is possible to add more options to this selection process by specifying the estimated cost and value and a function to activate it.

For these estimates, we collect multiple features from the abstract syntax tree. The first ones are the number of integral, array, and pointer variables the program uses, separated by global and local ones. Further, the amount of control flow statements, functions, function calls, assignments, and expressions are counted.

From this, the estimate of the file complexity was constructed by a combination of guessing how strong the influence of these factors on the performance is and refining these guesses based on testing. The amount of control flow instructions and loops hints at how many different paths there are and therefore the convergence speed of the fixpoint iteration. This is multiplied by an estimate, based on the instruction and expression count, of how many abstract operations need to be calculated for each iteration. A further multiplicative factor is how often a function is called on average because functions get analyzed separately for each context they are called in. Lastly, having a lot of variables, especially arrays, is also likely to cause a longer analysis.

3.1 C Intermediate Language (CIL)

Instead of directly working with C code, Goblint uses the **C** **I**ntermediate **L**anguage. The CIL library parses the file and provides a simplified abstract-syntax tree. The amount of constructs that need to be handled is less than if a complete abstract-syntax tree was used, while the representation is still close to C source code. For example, all loops are converted to `while(1)` loops. Figure 3.1 shows this transformation for a simple `for` loop.

[1] "congruence", "singleThreaded", "specification", "mallocWrappers", "noRecursiveIntervals", "enums", "loopUnrollHeuristic", "arrayDomain", "octagon", "wideningThresholds"

```
1 for(int i = 0; i < 10; i++){
2     a[i] = i;
3 }
```

(a) original

```
1  int i;
2  {
3      i = 0;
4      while(1) {
5          if(i < 10) {
6
7          } else {
8              goto while_break;
9          }
10         a[i] = i;
11         i = i + 1;
12     }
13     while_break:
14 }
```

(b) CIL representation translated back to C code

Figure 3.1: CIL converting a for loop to a while loop

CIL further separates the program into expressions (which do not have side effects), instructions (function calls or assignments), and statements (which influence control flow).

In our code, we often use the visitor pattern provided by CIL. Instead of traversing down the syntax tree ourselves, we can simply write a class that overrides the method for the part of the syntax tree we are interested in (e.g. the method `vexpr` gets called for every expression). Furthermore, with the return value we can choose for every node of the tree whether we want to replace it with a new one and whether the children should be skipped. In case we want to stop all further visits, we can throw an exception and catch it outside of the visitor.

CIL was originally developed by George C. Necula et al. [11]. Goblint maintains and uses its own fork because the original repository is no longer maintained. The fork also contains some improvements such as support for C11.

3.2 Selecting Array Domains ("arrayDomain")

Goblint currently offers three array domains: *trivial*, *unrolling*, and *partitioned*. We made it possible to choose the domain per array or type with annotations instead of having to select it globally. Further, we added some heuristics to automatically choose for some arrays.

In this section we first explain the array domains, their strengths, and what heuristics are used to select them. Then we show how to use the annotations in the source code and what limitations of our implementation exist. At last, the implementation is described.

3.2.1 Heuristics for the Array Domains

Keeping an abstract value for every index in an array is too inefficient because arrays can be very large or even have an unknown length. Therefore, an array domain has to somehow reduce the amount of values it differentiates.

domain	possible representation
trivial	[0,5]
unrolled ($k = 3$)	0: [0,0], 1: [1,1], 2: [2,2], rest: [3,5]
partitioned (i = [3,3])	index: i, xl: [0,2], xm: [3,3] xr:[4,5]
partitioned (i = [2,3])	index: i, xl: [0,2], xm: [2,3] xr:[3,5]

Figure 3.2: Possible representations of the array [0,1,2,3,4,5] in different domains, using the interval domain for all integers

```
1 int a[42];
2 a[0] = 0;
3 a[4] = 4;
4 a[10] = 10;
5 a[11] = 11;
6
7 assert(a[4] == 4);
8 //known if k > 10
9 assert(a[10] == 10);
```

(a) unrolling

```
1 int a[42];
2 int i = 0;
3
4 while (i < 30) {
5     a[i] = 1;
6     assert(a[i] == 1);
7     i++;
8 }
9 assert(a[i-10] == 1);
```

(b) partitioned

Figure 3.3: C code where either the *partitioned* or the *unrolling* domain is better than the other. Using the *trivial* domain would make all of the asserts unknown.

The simplest array domain *trivial* does this by keeping one value for all indices. This is an upper bound of all values ever written to the array and therefore often imprecise, but simple to implement and fast.

The *unrolling* domain represents the first k indices of each array individually and the rest as a single value. Figure 3.2 gives an example with $k = 3$. This unrolling factor is set by a global option. When assigning to an array, all indices that are possibly included are updated. If the index is exactly known and less than the factor, a strong update can be performed and the old value at this index is overwritten instead of joined with the new value. This domain distinguishes the most values of all implemented ones, which brings greater precision as well as greater cost. Further, it behaves the same as *trivial* for indices larger than k, which is bad if most accesses occur outside of that range.

The third domain partitions the array at some index and therefore is called *partitioned*. The index is not fixed and changes when values get written to the array. Further, it may depend on an expression with variables. The domain tracks three separate elements: the first element xm contains all values that could have been written at any index possibly described by the partition expression, and one element each for all values at indices that could be lower (xl) or higher (xr) than the partition expression. Figure 3.2 has two examples with i being the expression for the index. In the first one, i is known to be 3. In the second one, i can be 2 or 3. As a result, xm encompasses indices 2 and 3, xl 0 to 2 and xm 3 to 5. *partitioned* is most useful if arrays are accessed linearly. It also is the only domain that has the ability to be more precise for large indices.

Figure 3.3 shows examples of the strengths of the different domains. All three domains can also keep track of the length and perform out-of-bound checks. This is enabled by default.

```
1  int x[4] __attribute__((goblint_array_domain("unroll")));
2  __attribute__((goblint_array_domain("unroll"))) int x[4];
3
4  typedef int unrollInt __attribute__((goblint_array_domain("trivial")));
5  unrollInt x[4];
6
7  struct array {
8      int arr[5] __attribute__((goblint_array_domain("partitioned")));
9  };
10
11 void f(int * x __attribute__((goblint_array_domain("unroll")))){
12 }
13
14 void f(int x[4] __attribute__((goblint_array_domain("unroll")))){
15 }
```

Figure 3.4: Examples for annotating array domains

The autotuning has three heuristics for selecting the array domain:

- important types: If the type of an array is one of `pthread_mutex_t`, `spinlock_t`, or `pthread_t`, or has the attribute `mutex`, the values are especially important for the analysis of threads and deadlocks. Also, they are usually not very large. We therefore use the *unrolling* domain for these arrays.
- large arrays: If an array has a static size and is much larger than the unrolling factor, we use the *partitioned* domain.
- unrolled loop: The bachelor thesis [12], where loop unrolling and the *unrolling* domain was implemented in Goblint, notes that both features profit from each other because they can transform the code to be sequential. If a loop is unrolled (see 3.3), we therefore also unroll the arrays accessed in it.

3.2.2 Using the Array Attributes

The option `annotation.array` enables the user to decide the array domains used for each array individually. Arrays, parameters that are pointers, and types can be annotated by adding the attribute `__attribute__((goblint_array_domain("<domain>")))`, where `<domain>` is one of `"trivial"`, `"unroll"` or `"partitioned"`. For examples see Figure 3.4.

Lines 2 and 4-5 in the example annotate the type and not directly the variable x. All arrays declared with the type `unrollInt` will be unrolled but the attribute in line 2 will only apply to the array it is declared with. More specific annotations get prioritized. This means that type attributes override the global setting and variable attributes override those of the type.

Because C does not allow array parameters (the second function example in Figure 3.4 is transformed to the first one by CIL), we allow annotating the pointer parameters of a function and try to use that domain for arrays pointed at by that parameter.

This does not always work because of the way function calls are handled by Goblint. Instead of analyzing a function once, it is done for every context the function is called in.

If inside a function a call to another function occurs, the domain for an array could be changed during analyzing the latter. As all operators only work with two values in the same domain, we need to be able to revert the domains back to how they were before the call. For this, it is recorded which arrays each pointer parameter can point to in the first context the function is called.

Saving the array domains for the parameters from the first time a function gets called is of course not optimal, especially for functions called in many different contexts. If the pointers instead would be followed every time we return from a function call, a pointer may have changed which arrays it can point to. Doing it for every context at the start of the function would also not be trivial, because of recursive functions, and would potentially require a lot of memory but could be implemented in the future.

Another limitation of our implementation is that the unrolling factor of the *unrolling* domain is still set globally. This could be changed so that the attribute may contain an unrolling factor for the specific array. If, for example, the length of an array is known and small, it could be completely unrolled.

3.2.3 Implementation

The `FlagConfiguredArrayDomain` that dynamically switched between the domains based on the global flag was changed to `AttributeConfiguredArrayDomain`. When creating an array, the attributes of the variable and the variable type are given as additional parameters and determine the domain. All functions operating on two versions of an array, like e.g. *join*, can still always expect two arrays of the same domain and therefore remain unchanged.

Upon entering or returning from a function, an array could be represented by a different domain than we want in the current function because of pointer annotations. These arrays are projected to a new domain by reading the relevant values from the old one and creating a new array with them. Converting from *unroll* to *partitioned* results in an array partitioned at the index of the last unrolled value.

For remembering the change in domains, a map is maintained that maps each function to a map from array variables to the attributes of the pointer. If, when projecting, attributes for an array are found inside this mapping, they replace the variable attributes.

3.3 Loop Unrolling ("loopUnrollHeuristic")

Loop unrolling in the context of static analyzers increases the precision of the analysis by treating the first iterations of loops separately. Instead of having to work with abstract values that encompass all possible values of every possible iteration, the analyzer can be more specific the first times and find more exact results. This is especially true if the loop accesses the first elements of an unrolled array.

The existing implementation was created in [12] together with array unrolling and is activated by setting the unrolling factor (`exp.unrolling-factor`) to the amount of times loops should be unrolled. The unrolling is aggressive and unrolls every loop by this factor.

```
1  int i;
2  {
3      i = 0;
4
5      if(i < 10) {} else goto loop_end;
6
7      a[i] = i;
8      i = i + 1;
9
10     while (1) {
11         if(i < 10) {} else goto while_break;
12
13         a[i] = i;
14         i = i + 1;
15     }
16     while_break: ;
17     loop_end: ;
18 }
```

Figure 3.5: The loop of Figure 3.1, unrolled once

It is unproblematic for the correctness if a loop is unrolled more times than it executes, as the loop condition is still checked every iteration. The only disadvantage is an increased code size and with that analysis time.

Unrolling in Goblint's implementation is done by copying the loop body in the abstract syntax tree to before the loop. Some labels and gotos are added to simulate continue or break statements. As an example, the code in Figure 3.5 is the CIL representation of the result of unrolling the loop in Figure 3.1 once. The implementation also unrolls nested loops. As a result, the code size can grow a lot and slow down the analysis.

3.3.1 Loop Unrolling Heuristics

The "loopUnrollHeuristic" feature makes the unrolling factor of each loop depend on the size of the loop. For the size, only the amount of instructions, meaning function calls and assignments, is considered. A maximal number of 25 instructions after unrolling is targeted and the loop is unrolled until it reaches that. If in the loop a function of special interest is called, the target size increases to 50. As special we currently define functions working with heap allocations, threads, locks, and assertions. To obtain this specification, we use the existing interface provided by the module `LibraryFunctions` that classifies known library functions. Further, we try to detect some simple loops with fixed iteration counts and unroll them exactly that much. The maximal size is again increased here to 100.

3.3.2 Detecting Loops with Fixed Iterations

As unrolling is done before analysis starts, we have no information about the values of variables and therefore limited tools to determine the loop iterations. For example, if the upper bound is a variable, we cannot calculate the iterations even if at this point the variable has a fixed value.

To detect a simple loop with a fixed iteration count we use some criteria similar to those in [15], where Szécsi et al. describe a comparable heuristic they implemented for the LLVM Clang Static Analyzer. The loop has to fulfill the following properties to qualify:

1. There is one single break in an if statement where the condition compares a local variable v to a constant e.
2. The variable v is changed by a constant d at exactly one point inside the loop. This point also cannot be in a conditional path.
3. The last assignment to v before the loop is a constant s and not conditional.
4. There is no pointer to v. Otherwise, it would be harder to verify the other conditions.

Although limiting, these properties are general enough to match many simple loops, in particular most standard for-loops with a constant upper bound such as the example in Figure 3.1. The following formula then provides the iteration count if the break condition is $v < e$, with the other possible comparisons being similar:

$$\left\lfloor \frac{e - s}{d} \right\rfloor$$

To check property three we are not able to use a visitor, because the order of the assignments is important. Moreover, the unrolling must happen before the control flow information is created by `prepareCFG`, because this converts break and continue statements to gotos. As a result, we have to traverse the tree manually, starting at the function body containing the loop. We keep track of the last constant assignment to v. If an assignment happens in a conditional branch that does not also contain the currently considered loop, this value is invalidated. The same also happens if a statement has a label where a goto or a switch path could end up or the assigned value is not constant.

The detection is not always exact. Gotos inside the loop are largely ignored for the sake of simplicity, therefore can make the loop violate the second property and then behave differently than we predict. Ignoring loops that use gotos would disqualify all loops with a nested unrolled loop inside from this detection and as a result is not wanted. Being inexact is not a huge problem because, as mentioned above, unrolling too many times will still result in correct execution. If, however, we were exact, it would be possible to remove all checks of the exit condition.

3.4 Interval Widening Thresholds ("wideningThresholds")

Widening thresholds are a way to balance the faster iteration gained by widening of intervals (as explained in section 1.2) and the resulting reduced precision. Some numbers are selected as thresholds. Instead of always using ∞ when enlarging an interval bound, the next larger or lower widening threshold is selected, depending on whether the upper or lower bound is changed.

```
1 int i = 0;
2 while(i <= 10) {
3     i++;
4 }
```

```
1 int i = 20;
2 while(!(i < 10)) {
3     i--;
4 }
```

Figure 3.6: Similar loops to Figure 1.2 but the best widening threshold is different

The cost of enabling this option is estimated as a function of the number of widening constants, since they increase the number of iterations until the fixed point is reached, and the number of loops where widening is mainly used.

The existing implementation searches for constants occurring in the program and employs all of them as the widening constants. We added an alternative option that only collects constants occurring in conditions and processes them further.

When widening is used in a loop, we want to include the exit condition but at the same time still have as tight bounds as possible. In the example we used for widening (Figure 1.2), we want the upper bound of the interval for i to be widened to 10. Figure 3.6 contains similar loops where the constant that is useful as threshold changes. The loop on the left is exited when the upper bound is 11. In the right program, the same comparison is performed as in our original example, but the interesting bound is the lower one, with 9 being the best widening threshold for it.

We therefore choose the threshold depending on which comparison we find the constant in and collect separate upper and lower thresholds. To this some defaults, consisting of 0 and powers of 2, are added.

3.5 Integer Domains ("congruence", "noRecursiveIntervals", "enums")

Goblint currently offers four domains for integers that can work together to refine each other. They are normally enabled globally but it is possible to annotate functions to change which of them are activated in the function.

The values of the first domain, *def-exc*, are either a definite value or a set of values that are definitely excluded. There are almost no cases where we do not want this and it is enabled by default, so we do not change the configuration.

enums is useful for variables where no arithmetic operations are performed on but instead only constants are assigned to. As the name describes, its main purpose is to track enumerations. We therefore activate it globally if the program contains them.

With the *congruence* domain, Goblint keeps track of divisibility and remainders. A value is represented, if possible, as being equal to $c \mod m$. The biggest application of this is when the modulo operator % is used. We do not want to activate the domain only in functions containing %, because the values we are interested in might be modified in functions that get called or are calling this function. For this, we collect which function statically contains a call to which other function. Then, if we find a function using %, we annotate it as well as up to a certain depth the static callers and callees.

```
1 int i = 16;
2 int x = 1;
3 while (0 < i) {
4     x++;
5     i--;
6 }
7 assert(x + i == 17);
```

Figure 3.7: An example where the octagon domain adds precision

Lastly, the *interval* domain as described in section 1.1 is supported. We use the same collection of static function calls as above to disable the use of intervals as part of the contexts of functions that are directly or with one indirection recursive. This is done because in recursive functions the interval domain often leads to too many contexts, similar to how the fixpoint iteration of a loop takes a long time to converge without interval widening.

3.6 Octagons ("octagon")

Goblint utilizes the external library *Apron* [7] to include domains that track relationships between variables and another implementation of the interval domain.

With the *octagon* domain, we keep track of inequalities in the form of $\pm X \pm Y \leq C$, where X and Y are variables and C is a constant. For a simple example of where this is helpful see Figure 3.7. If only the interval domain is used, the abstract value for x after the loop is the interval $[1, 17]$ at best and possibly even $[1, \infty]$ because of widening. The octagon can show that after each loop iteration $x + i \leq 17$ as well as $-x - i \leq -17$, which implies $x + i = 17$.

The *polyhedra* domain can find general linear inequalities between variables. It has a high cost and most of the time little gain. This has, for example, been shown in [19], where the polyhedra domain only slightly increased precision but had the second worst impact of the tested options on the time needed fir the analysis. As a result, we ignored this domain for the autotuner.

The existing implementation tracks all integer variables. This is sub-optimal because most of the time only a few of them have interesting relationships and the worst time complexity of the abstract operations with n variables is $\mathcal{O}(n^3)$ [10].

If the option `annotation.track_apron` is set to true, we extended the filter that selects only the integral variables to also require the variables to have the attribute `goblint_apron_track`. As for global variables a local copy is created, we further make sure that the attributes get copied as well.

[9] describes a strategy the static analyzer ASTRÉE uses to automatically choose the variables for the octagon domain. Per syntactic block they determine for every assignment which variables occur, limited to the ones found in the expression syntax tree before a nonlinear operator. If two or more different variables like this are found in one assignment, they get added to the variables that get tracked in that block.

For our heuristic, we currently can only choose whether to include each variable in the whole program instead of block-wise but the method of selection is similar. Using a visitor, every variable gets assigned a score depending on how often it occurs in an expression. If the expression has an operator that is not `-`, `+`, `*`, `&&`, `||` or a comparison, we skip all child expressions. Also, if an expression is of the form $\pm X \pm Y$ inside a comparison, a bigger score gets added. In every function, we then take the 8 local variables with the highest score and add the attribute to them as well as the two best global ones. This means that at all times the octagon domain works with at most 10 variables.

When widening, the octagon domain uses thresholds to increase C, similar to how the interval domain does. As a result, the threshold collection described in section 3.4 is also used here. Because the inequalities also include those with two times the same variable (in particular $+X + X \leq C$), we additionally use twice the value of the constants as threshold.

3.7 Others ("singleThreaded", "specification", "mallocWrappers")

The following three options only do some small changes. "singleThreaded" checks if there exists a function that gets statically called and is classified by `LibraryFunctions` as creating a thread. Is this not the case, the whole program is single-threaded and we can deactivate most thread analyses.

Malloc wrappers are used by some programs to add some special features to malloc. With "mallocWrappers" we try to detect them according to two properties: They contain one (static) call to malloc and are called comparatively often.

Lastly, we consider the specification of a property used by SV-COMP, if present, and focus on the features that are relevant to it. For *Reachability*, we additionally unroll loops containing the function for which we have to decide if it is called. The thread analyses are especially important for *No Data Race* and we therefore activate them all. When trying to prove *No Overflow*, we enable integer domains and allow more variables in the octagon domain.

4 Evaluation

In this chapter, we evaluate the impact of our autotuner. For this, we compare running Goblint with our new features to doing so with a static configuration.

There are two main success indicators: how many programs we can prove to be correct and the time needed for the analysis. If the first one increases, we manage to activate features for programs where they are helping. The second one suggests whether we are enabling them efficiently because activating features that do not increase the precision for a file leads to unnecessarily taking more time.

4.1 Evaluation setup

For the evaluation we run all test from the SV-COMP benchmark repository at `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks` where the relevant property is *Reachability*, *No Overflow*, or *No Data Race*. The tests are executed with `benchexec` on a server with two Intel Xeon Platinum 8260 CPUs running at 2.40 GHz. Each test was limited to 60 seconds on one CPU core with 1GB memory. This is less than the official SV-COMP resource limit of 15 minutes and 15 GB memory to keep the runtime reasonable. Our baseline is the configuration that was used for SV-COMP 2022. The other runs use the same configuration with autotune enabled and, unless specified otherwise, the unrolling factor for arrays set to 5, which was determined by [12] to be a good balance between precision and performance. Additionally to no autotuner and all features, we separately enable each feature to measure the impacts of them on their own. Some further variations of features are evaluated to get a better idea of their behavior.

4.2 Evaluation results

Figures 4.1, 4.2 and 4.3 show the results separately for each category. The tables contain the number of correct and wrong answers and the average time it took to complete the analysis. If Goblint produced no result, we further distinguish whether it was because of the resource limits or other errors. The latter includes all cases where Goblint crashed because of e.g. stack overflows or throwing exceptions.

Although Goblint is supposed to be sound, there currently are some bugs that cause a few incorrect answers. Our autotuner does at points reveal more of the bugs but is not the source of them. We therefore mostly ignore them in this evaluation.

In the *No Data Race* category the correct answers do not change. This is caused most likely by Goblint already being close to the maximum number of tests it can solve because of its limitation of never answering false, and few of the implemented features helping with

Configuration	Correct	Wrong	average time in s	Resource limited		Errors
				Time	Memory	
baseline	60	0	0.52	1	0	0
loopUnrollHeuristic	60	0	0.70	1	0	0
arrayDomain	60	0	0.53	1	0	0
congruence	60	0	0.53	1	0	0
noRecursiveIntervals	60	0	0.53	1	0	0
enums	60	0	0.53	1	0	0
specification	60	0	0.53	1	0	0
singleThreaded	60	0	0.53	1	0	0
mallocWrappers	60	0	0.53	1	0	0
wideningThresholds	60	0	0.53	1	0	0
octagon	60	0	0.56	1	0	0
all	60	0	0.90	1	0	0
octagon + loop	60	0	0.87	1	0	0

Figure 4.1: Results for 162 tests (82 expecting true) checking *No Data Race*

Configuration	Correct	Wrong	average time in s	Resource limited		Errors
				Time	Memory	
baseline	1286	3	20.0	2192	281	316
loopUnrollHeuristic	1272	3	21.1	2245	281	375
arrayDomain	1287	3	20.1	2207	281	315
congruence	1281	3	20.7	2163	378	301
noRecursiveIntervals	1288	3	20.0	2187	281	323
enums	1293	3	20.2	2194	281	304
loops + specification	1278	3	21.1	2245	281	369
singleThreaded	1360	4	15.8	1417	301	203
mallocWrappers	1290	4	20.0	2192	281	324
wideningThresholds	1294	3	20.2	2209	281	296
octagon	1316	3	21.6	2248	281	293
all	1376	6	18.7	1659	222	329
unroll all loops (factor 5)	1276	3	21.8	2276	281	381
octagon, all variables	1214	2	28.6	3228	301	185
always with new thresholds	1287	4	20.0	2196	281	319
always with old thresholds	1287	4	20.0	2195	281	317
array (factor 15)	1287	3	20.1	2205	281	314
array + loop	1270	3	21.2	2253	281	375
array (factor 15) + loop	1271	3	21.2	2256	281	367
positive only	1408	7	16.8	1520	286	213

Figure 4.2: Results for 7995 tests (5797 expecting true) checking *Reachability*.

Configuration	Correct	Wrong	average time in s	Resource limited		Errors
				Time	Memory	
baseline	81	1	7.23	35	27	51
loopUnrollHeuristic	85	1	8.55	44	27	52
arrayDomain	81	1	7.22	35	27	49
congruence	81	1	7.45	35	27	49
noRecursiveIntervals	86	1	6.79	35	26	42
enums	81	1	7.26	34	27	51
specification	81	1	7.21	35	27	50
singleThreaded	98	1	6.19	25	28	32
mallocWrappers	81	1	7.21	35	27	51
wideningThresholds	82	1	7.30	35	27	52
octagon	94	1	8.41	41	19	51
all	118	1	8.40	37	20	31
unroll all loops (factor 5)	83	1	20.2	169	27	51
octagon, all variables	94	1	15.6	102	19	47
always with new thresholds	82	1	9.39	55	27	48
always with old thresholds	82	1	12.4	86	26	48
arrayDomain (factor 15)	81	1	7.23	35	27	49
array + loop	85	1	8.12	38	27	58
array (factor 15) + loop	85	1	8.20	40	27	56
octagon + specification	94	1	8.44	41	19	49
positive only	118	1	8.14	35	20	22

Figure 4.3: Results of 635 tests (370 expecting true) checking *No Overflow*

analyzing threads. The worse time is mostly caused by loop unrolling and, if combined with it, the octagon domain.

In the rest of this chapter, we only look at the other two categories. Both profit the most from "singleThreaded". Additionally to greatly reducing the time needed by deactivating unneeded analyses, it manages to reduce the amount of files that lead to the exception `Errormsg.Error` from 189 to 60 in *Reachability* and from 23 to 0 in *No Overflow*.

The octagon domain has the second highest increase in correct tests. This comes with a clearly visible time penalty but only leads to few tests that the baseline could solve now timing out. That our selection of variables is better than using the octagon domain for all of them can be seen in a further test, where the amount of timeouts and the average time drastically increase while at the same time fewer or the same number of tests are solved correctly.

The widening thresholds have a smaller impact. Additionally, we tested activating them at all times. When looking at what tests get solved, one can see that our estimate for activating this option still misses some files that could use the thresholds. Interestingly, some tests from *Reachability* (`pthread-deagle/airline-*`) that Goblint can solve without thresholds it cannot solve if they are activated, even though it is not timing out. In the *No Overflow* category, we can see a clearly longer analysis time and with it more timeouts. With the old method of using all constants as widening thresholds this effect is even stronger. The new one manages to solve exactly the same tests as the old one in both categories. In summary, our estimate performs better than always activating the thresholds and the described method of selecting the constants brings further improvement.

The estimates for the whole file need some further work. Most of the tests where the total estimate was high enough to not activate more options did not time out. Even though the average time of programs with such a high estimate is a lot higher (about 34 compared to 2 seconds), it is not precise enough yet. One reason for this is that some files include libraries and only use a small part of them. The estimates do not exclude unused code, while the analysis mostly does.

Surprisingly, loop unrolling leads to fewer correct answers in the *Reachability* category. The main root cause of this is not that correct tests from the baseline now reach a resource limit but instead an increase of errors. To be precise, Goblint with the loop unrolling heuristic can prove the reachability property 29 times where it could not without unrolling, loses three correct answers to timeouts and 40 to the errors `ERROR (verify)` and `EXCEPTION (Not_found)`, both of which originate in code concerning witnesses. All of the new errors occur in files from the folder `ldv-linux-3.4-simple`. If these bugs will get fixed, a clear benefit can be expected from these two options. At the time of writing, a pull request that fixes all `ERROR (verify)` errors is already being worked on.

Unrolling all loops in *Reachability* performs a bit better than our loop heuristic but still worse than the baseline. It is not immediately clear how much this is caused by our unrolling not being aggressive enough and how much by it changing some condition that triggers the errors. The total amount of errors approximately stays the same, which points towards the first option, but we cannot conclude whether the heuristic is good. There is a

time penalty for this fixed loop unrolling in both categories but it is especially strong in the *No Overflow* category, meaning that here we definitely have an improvement by our heuristics.

In the *Reachability* category, annotating with the congruence domain has the same problem as loop unrolling but to a lower extent. 14 formerly correct answers fail with `ERROR (verify)`, decreasing the total amount. Further, it leads to more tasks running out of memory, which does not change any of the correct answers.

On its own and even together with loop unrolling, the array domain heuristics have nearly no impact. This does not change when using a larger unrolling factor of 15, which suggests, together with the small increase in memory consumption, that not a lot of arrays get unrolled and more heuristics are needed.

"specification" also seems to have no clear impact and could profit from changing more options, for example by having different base configurations for each category. In *Reachability* nothing changes unless we are also unrolling loops, which is why we only test it with this combination. Here, we improve the amount of correct answers but it is difficult to differentiate between it having a benefit and it not triggering the same errors of loop unrolling by coincidence.

Some options reducing the total amount of correct answers is the reason why using all features is currently not optimal. Activating only the features leading to an increase for the *Reachability* category ("singleThreaded", "mallocWrappers", "noRecursiveIntervals", "enums", "arrayDomain", "octagon" and "wideningThresholds") results in the highest number of right answers. Compared to the baseline, this combination has an increase of about 9.5% of correct answers while lowering the average analysis time by approximately 16%. Even when subtracting the higher penalty for more incorrect answers, we still improve the score for SV-COMP.

In the *No Overflow* category, we archive around 45% more correct answers. Using only features that bring a benefit by either increasing the number of correct answer or decreasing the time needed for the analysis ("singleThreaded", "noRecursiveIntervals", "loopUnrollHeuristic", "octagon" and "wideningThresholds"), we can get a lower time compared to using all and even 3% less than the baseline.

5 Conclusions

The goal of this thesis was to implement an autotuner that improves the static analyzer Goblint by automatically changing its configuration based on the analyzed file. For this, we designed heuristics for selecting array and integer domains, widening thresholds, and variables for the octagon domain, as well as loops unrolling and some other options. To aid this, we added the ability to specify some of these choices more precisely with variable attributes instead of just globally.

The evaluation of these features with the SV-COMP tests showed that the autotuner is able to clearly improve the amount of tests Goblint can correctly solve as well as the time needed for this analysis in the categories *Reachability* and *No Overflow*. The greatest change was achieved by disabling most thread analyses if the program is single threaded. Some other implemented features had no impact or a minor negative one. In parts, this is caused by an increase of crashes. Better results for these features can be expected when the bugs are fixed. In the *No Data Race* category, we were unable to improve Goblint's result.

6 Future Work

There are many ways the autotuner can be extended to further improve its adaptability to a program. First of all, for some of the features, e.g. choosing array domains and which loops to unroll, more heuristics can be added. Furthermore, there are options that can have a great impact on precision and analysis time that the autotuner does not change yet. An example would be the domain used for structs.

To achieve better results at SV-COMP, it would be useful to have different base configurations that are loaded depending on the property that needs to be proven. This would also improve the autotuner, as different features have a positive impact in the different categories.

Estimating the total file complexity and the coupled activation of some options is another possible area of future work. As there are many relevant features and therefore many factors to tune, it is difficult to do this by hand even with a more systematic approach. As multiple works described in chapter 2 showed, machine learning could do this successfully.

List of Figures

Bibliography

[1] Dirk Beyer. Progress on Software Verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402. Springer International Publishing, 2022.

[2] Dirk Beyer and Matthias Dangl. Strategy Selection for Software Verification Based on Boolean Features. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 144–159. Springer International Publishing, 2018.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*. ACM Press, 1977.

[4] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. *Formal Methods in System Design*, 50(2-3):289–316, January 2017.

[5] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic Predicate Abstraction Using Variable Roles. In *Lecture Notes in Computer Science*, pages 265–281. Springer International Publishing, 2017.

[6] Yulia Demyanova, Helmut Veith, and Florian Zuleger. On the concept of variable roles and its use in software analysis. In *2013 Formal Methods in Computer-Aided Design*, pages 226–230. IEEE, 2013.

[7] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*, pages 661–667. Springer Berlin Heidelberg, 2009.

[8] Ugur Koc. *Improving the Usability of Static Analysis Tools Using Machine Learning*. PhD thesis, University of Maryland, 2019.

[9] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, March 2006.

[10] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Normale Supérieure, December 2004.

[11] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2002.

[12] Mireia Cano Pujol. Increasing the Precision of the Static Analyzer Goblint by Loop Unrolling. Bachelor Thesis, February 2022.

[13] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[14] Cedric Richter and Heike Wehrheim. PeSCo: Predicting Sequential Combinations of Verifiers. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–233. Springer International Publishing, 2019.

[15] Péter György Szécsi, Gábor Horváth, and Zoltán Porkoláb. Improved Loop Execution Modeling in the Clang Static Analyzer. *Acta Cybernetica*, October 2020.

[16] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V. Nori. MUX: Algorithm Selection for Software Model Checkers. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 132–141, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10(2):203–232, 2003.

[18] Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, 30:141–155, 2009.

[19] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *Programming Languages and Systems*, pages 653–682. Springer International Publishing, 2018.