CHENNAI MATHEMATICAL INSTITUTE

CHENNAI MATHEMATICAL INSTITUTE

FACULTY OF COMPUTER SCIENCE

Verifying Array-Manipulating Programs

Author:
Arijit Shaw

Supervisor:
Prof. Mandayam Srivas

A thesis submitted for the degree of

MSc Computer Science

June 16, 2019

Abstract

Producing bug-free codes is a major challenge to computer scientists today and with a huge growth of the field of Software Verification, there has been a sufficient progress in it. Despite all successes the annual Verification competition SV-COMP hasn't seen much good results in a few sections and array-manipulating programs are one of them.

Synthesizing useful loop invariant helps verification process to a great extent. Therefore we aim to generate loop invariant for array-manipulating program.

Max-Strategy Iteration is a well-aquinted method that uses SMT solvers to synthesize loop invariant and for numerical programs, it can give such invariant that can help verifying a large set of problems.

We hereby propose a novel method to generate loop invariants for a subclass of array-manipulating programs which is an extension to the Max-Strategy Iteration algorithm. With growth of *array decision procedure* in SMT solvers, our method is supposed to generate loop invariants fast.

Acknowledgements

I would like to thank my supervisor, Mandayam Srivas, for introducing me to the field of Software Verification, and for all the helpful discussions that we have had. I also wish to thank my senior Johanan Wahlang, who was always there in all my doubts and queries and without whom, this thesis would not have been completed.

Contents

Chapter 1

Introduction

Software systems are ubiquitous, and their use in mission critical systems means that we need to have high degree of assurance that our software works according to the given specification. Getting this high assurance for software has proven to be much harder than for the hardware systems. There are several accounts of disasters caused by software flaws. Such a list can be found here (Der).

Manual inspection of complex software is infeasible, and consequently a significant body of computer science research has been focused on building tools to facilitate automatic verification of software systems. While testing provides us with a cheaper way to find bugs, it doesn't show that our software is bug free, so we can't just rely on testing for mission critical systems. Formal verification has been a huge success in the hardware industry, but the same is not true for the software industry, we rarely see software practitioners use formal verification tools.

Despite all its successes, verification research lack in many a fields, concurrency and arrays being two major of blocks in it. In this thesis, we'll look at the programs with arrays and some ways to analyze that for simpler cases.

This thesis deals with the verification of safety properties about programs with arrays. Verification of such properties amounts to checking whether the reachable state space is contained in the invariant specified by the property.

Arrays are an important language feature in many imperative languages. They provide the programmer a notion of contiguous blocks of memory which can be accessed using array indices. Programmers using these languages often use some kind of looping construct to access or modify regions of interest within the array. While it's flexible to use these constructs, it also makes it hard to reason about these programs statically hard. Naturally, we'd like to know if we can infer things about array accesses and use that information to simplify the verification of programs using these constructs.

The problem for arrays being difficult to analyze, we concentrate over some subclass for it. We wanted to solve the problem for programs where the array variable are changed over a loop with a simple loop counter. The aim being analyzing these programs with some precision, where other programs will be analyzed correctly, but they may not result in some good results. Our analysis is supposed to generate some invariant over these variables which will “cover” the property that we want to prove or disprove.

1.1 Summary of Contribution

1. **Max Strategy iteration over the proposed domain:** In strategy iteration method, we solve the fixed point equation by iteratively approximating the the least fixed point of $S = F(S)$ by fixed points of some easily computable equations $S = F^{(i)}(S)$. These equations are induced by some so-called "strategies". These strategies guarantees that the fixed-point will be found after a finite number of strategies.

   These techniques are applied to template domains, that is why we were needed to define an domain for arrays.

2. **Design of an Abstract Domain for Arrays:** To our observation, programs of above class may be analyzed well by segmenting the array into two parts, segmented by some scalar expression of the loop counter.

   We design a domain for this, as Cousot et. al. designed one for abstract interpretation (CCL11).

   Details of the domain is discussed in section 4, where we have discussed which type of program our scheme will work well.

3. **Implementation in 2LS:** We have started implementing our proposed scheme within 2LS. 2LS ("tools") is a verification tool for C programs. It is built upon the CPROVER framework. It does array blasting for arrays. We are replacing this with implementing our theoretical procedure.

1.2 Organization

The remainder of this thesis is organized as follows. Chapter 2 presents notation and describes preliminaries needed for the subsequent discussion.

Chapter 3 discusses the problem of verification of array-manipulating program and some work that has been already done in the purpose.

Chapter 4 discusses a domain that is needed to continue working with our algorithm.

In Chapter 5 we culminate with discussing our proposed algorithm for array invariant generation.

Chapter 6 discusses about a tool called 2LS and the design architecture needed on that to implement our algorithm.

Chapter 2

Background

2.1 Software Verification

According to wikipedia, Software verification is a discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements. In general cases, a piece of code is given with a number of assertions. The assertions being of safety or invariant properties. The task is to check, in all situations, whatever values may the variables take, whether the values are satisfied or not.

2.2 Invariant and Fixed Point

Invariants are logical formulas which is true for all states reachable from an initial set of states.

Inductive invariants are invariant which are inductive with respect to transition relation. That means the formula is true for all reachable states from any arbitrary initial state. Here we will compute inductive invariants. Particularly, we will compute inductive invariants of interval shape.

Fixpoint of a function is an element of the function's domain that is mapped to itself by the function. A formula(represents set of states) that includes set of initial states and by applying transition to each state of the set goes to some state inside the set. That means the set is closed under transition relation.

Invariants are actually fixpoints. Our algorithm runs until a fixpoint is found.

2.3 Program Model

We consider programs modeled as symbolic control flow graphs over a state space Σ. A symbolic control flow graph (CFG) is a directed graph $\langle L, R, l_0 \rangle$, where

- L is a finite set of locations
- $l_0 \in L$ is the initial location
- $(l, R, l`) \in R$ define a finite number of arcs from locations $l \in L$ to $l` \in L$

An execution of a CFG is a possibly infinite sequence

$$(l_0, s_0) \longrightarrow^R (l_1, s_1) \longrightarrow \dots$$

2.4 Max-strategy Iteration

As we stated earlier, In strategy iteration method, we solve the fixed point equation by iteratively approximating the the least fixed point of $S = F(S)$ by fixed points of some easily computable equations $S = F^{(i)}(S)$. These equations are induced by some so-called "strategies". These strategies guarantees that the fixed-point will be found after a finite number of strategies. The fixed point may be apporached from above or below, and that will decide, whether we call it a min- or max-strategy iteration

Though both of these techniques are formal techniques to synthesize inductive invariants, there is a sharp distinction among these two. In Abstract Interpretation, an abstract transformer is constructed by abstracting operators in program. Then, static analysis (Kleene iteration) is done using abstract transformers to reach the fixed point. It uses widening (for faster convergence) at the loss of precision. On the contrary, in Max SI, there is no need to abstract program operations. The algorithm searches abstract domain for fixpoints using constraint solving, mostly using SMT solvers. It does not use widening, hence no case for loss of precision.

The definitions at section 2.3 and section 2.4 are taken from (SS13).

2.5 Programs with Arrays

Array-like properties are properties present in array-manipulating programs. These are distinct with their characteristics. Those include :

- The properties that are required to satisfy are generally quantified over indices. E.g., all elements of the array are initilized to 0.
- A segmentation is often sufficient to generate loop invariant. E.g., all elements that have been seen in one segment, others being in other domain.

There may be variations from this pattern, but this pattern is quite common in array-manipulating programs.

Chapter 3

Related Works

For most of the approaches to intuitively solve array verification problem, we may categorize them into three categories.

Array expansion is the first and most precise approach for proving some properties that we may take for programs with arrays. The arrays are blasted and each variable at each index are treated as separate variables. The speciality that same operation is done for a good number of array indices is not reflected in this case. We can't use the SAT solvers that can solve theories of arrays in this case.

Array smashing stays at the opposite side of the precision spectrum where there is the smashing of all the array elements into one summary location. It is immediate that this is not going to work in many a cases, where there are difference in properties for different segment of arrays.

Array Partitioning approach separates the task of array partitioning from that of establishing array properties. Given a partition of the array into slices, the analysis populates the slices with some abstract value. The partitioning is done either syntactically or by some pre-analysis.

We work in this array partitioning approach. Some related works in array partitioning are discussed here.

3.1 Parametric Segmentation Functor

Gawlitza et. al. (GS07) used abstract interpretation over some abstracted segmented array domain. Over the abstracted domain, they defined the Galois connection between the abstract and concrete domain and did analysis using accelaration, widening and narrowing.

3.2 Tiling

Chakraborty et. al. (CGU17) used a pattern called tiling where tiles are defined as following. **Tile** : LoopCounter × Indices → {**tt**,**ff**} for loop L. With the theorem, if Tile satisfies some properties and if Pre → Inv holds then the Hoare triple $\{Pre\}L\{Post\}$ holds for a tile, the tiles are put to SMT solver to check whether these properties hold. The major challenge here is to find the right tile.

3.3 Cell morphing

Monniaux et. al.(MG16) expresses the blocks of cells as horn clauses and tries to infer some property from it. Array programs are turned to array-free Horn clauses and fed to SMT-solver to check. Here, Abstract *a* of array type into a couple $(k, ak = a[k])$ and to each program point attach, instead of a set I of concrete states $(x_1, \ldots, x_m, a)$, a set $I^\sharp$ of abstract states $(x_1, \ldots, x_m, k, ak)$.

Chapter 4

An Abstract Domain for Arrays

To do Max SI, we need a domain. For numerical programs, an interval domain or a template domain is used heavily for both max-strategy iteration and abstract interpretation. A detailed note on these domain are available here (Sch12).

For synthesizing the invariants for arrays, the domain was needed to contain information regarding the segmentation and the properties each segment holds. We therefore discuss a design for the domain here. This domain is contributed to (CCL11).

Our approach automatically divides the array into a sequence of possibly empty segments delimited by a set of segment bounds. The content of each segment is uniformly abstracted. The array analysis can be combined via a reduced product with an abstraction for scalar variables.

Therefore the abstract domain has three main parameters:

(i) the expressions used to describe the segment bounds;

(ii) the abstract domain used to abstract the segment values

(iii) the abstract domain used to abstract scalar variables.

When the three parameters above are chosen to be:

(i) simple expressions in the form k or $x + k$ where x is a variable and k is an integer

(ii) intervals or template bounds;

(iii) intervals or template bounds;

We start with an simple example where an array is being initialized. We want to prove at last that all elements are initialized to 0.

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

At line 3, the invariant that contains all the property is :

$(A.Length = 0 \wedge i = 0) \vee (A.Length \geq 1 \wedge 0 \leq i \leq A.Length, \forall j \in [0, i) : A[j] = 0)$

In our abstract domain, we want to encode this property as :

`A: {0} [0 0] {i}?  T {A.Length}?`

Here, `{0}`, `{i}` and `{A.length}` denote segment bounds whereas, `[0 0]` denote interval, within which values of all elements of index 0 to i of array A resides.

The `?` after `{i}` denotes the segment `{0} [0 0] {i}` is possibly empty. The `T` in the second segment denotes the value for the elements within this segment can range in $(-\infty, +\infty)$.

4.1 Details of Domain

As we discussed above, the proposed domain has three parts to abstract the array.

4.1.1 Segment Bounds

The initial and ending bounds being `{0}` and `{A.length}`. It denotes the complete array is being covered by the abstraction.

When the array `A` is modified within a loop having a loop counter, say i, the loop counter is added to segment the domain into two as

`A: {0} T {i}?  T {A.Length}?`

The `T` in both sections denote the values can range anything.

4.1.2 Segment Values

The segment values are represented by some abstract domain. Initially we choose interval domain for this. We may use different domains for our analysis, and it may be the case as well that analysis of different programs behave well under different domains. We may use cardinal power of one domain by other domain also.

Later we'll show that reduced cardinal power of intervals by conditional domain does a good analysis for a large number of programs.

4.1.3 Scalar Variables

There will be scalars associated with the abstract array domain, for example, the loop counter variable being used as segment bound. As some major information can be availed by analysis of this variable, we put this in our abstract domain.

This is also initialized as `T`.

Therefore, the abstract domain looks like

```
A: {0} T {i}?  T {A.Length}?
i :  T
```

Chapter 5

Strategy Iteration for Array Domain

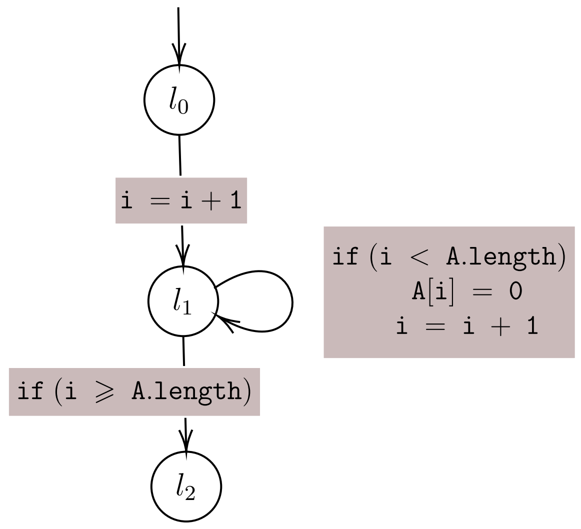
Max-strategy iteration(SS13; GS07) is a method for computing the least solution of a system of equations M of the form $\delta = F(\delta)$ where δ are the template bounds, and $F_i, 0 \leq i \leq n$ is a finite maximum of monotonic and concave operators $R^n \longrightarrow R$; in our case they are affine functions. The max-strategy improvement algorithm for affine programs is guaranteed to compute the least fixed point of F , and it has to perform at most exponentially many improvement steps, each of which takes polynomial time.

5.1 Algorithm

Semantic equations. The equation system M is constructed from the abstract semantics of the program's transitions:

for each $l' \in L : \delta_{l'} = max\left(\{d_l^0\} \cup \{R(\delta_l)|(l, R, l') \in R\}\right)$

Exmample for Semantic Equations.



The abstract domain looks like:

$\texttt{A} : \{0\}\mathtt{d}_1^{\mathtt{A}}\{\mathtt{i}\}?\mathtt{d}_2^{\mathtt{A}}\{\mathtt{A.Length}\}?, \mathtt{i} :$

Where we denote the variables at different segments like d_1^A and d_2^A

Consider the CFG above. The equations for this will look like following:

$$\delta_{1,1} = \begin{cases} -\infty \\ sup\{d_1^{A'} | d_1^A \leq \delta_{1,1} \wedge -d_1^A \leq \delta_{1,2} \wedge i < A.length \wedge d_1^A = 0\} \end{cases}$$

$$\delta_{1,2} = \begin{cases} -\infty \\ sup\{-d_1^{A'} | d_1^A \leq \delta_{1,1} \wedge -d_1^A \leq \delta_{1,2} \wedge i < A.length \wedge d_1^A = 0\} \end{cases}$$

Strategies. A strategy μ induces a ″subsystem″ $\delta = F(\delta)$ of M in the sense that exactly one argument F_i of the max operator on the right-hand side of each equation $\delta_i = max(\dots, F_i(\delta), \dots)$ is chosen. Intuitively, this means that a strategy selects exactly one ″incoming transition″ for each variable or array segment in each location l'.

Max-strategy improvement. $lfp(M)$ is computed with the help of the max-strategy improvement algorithm which iteratively improves strategies μ until the least fixed point $lfp(\mu)$ of a strategy equals $lfp(M)$.

Algorithm 1 Max-strategy iteration algorithm

```
while not $d$ is a solution of $M$ do
    $\mu := max\_improve_M(\mu, d)$
    $d := lfp(\mu)$
end while
return d
```

The least fixed point $lfp(\mu)$ of a strategy μ can be computed by solving the LP / SAT problem with the constraint system :

$$\text{for each } \delta_{l'} = R(\delta_l) \text{ in } \mu : \delta_{l'} \leq T_{l'}x' \wedge T_l x \wedge R(x, x')$$

where x and x′ are auxiliary variables and the objective function max i d i , i.e. the sum of all bounds d.

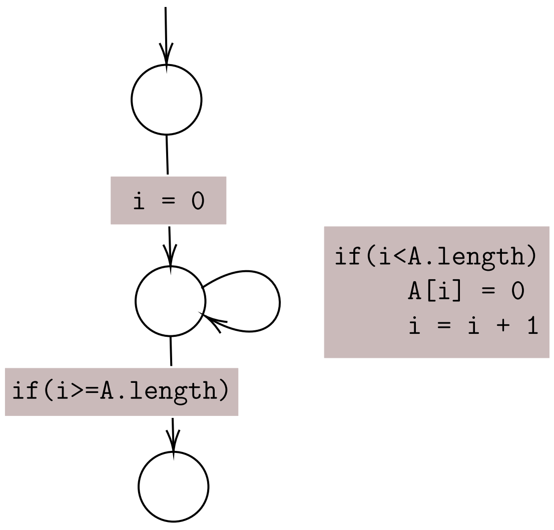
$\mu := max\_improve_M(\mu, d)$ iff:

- μ' is "at least as good" as μ
- μ' is "strictly better for the changed equations"

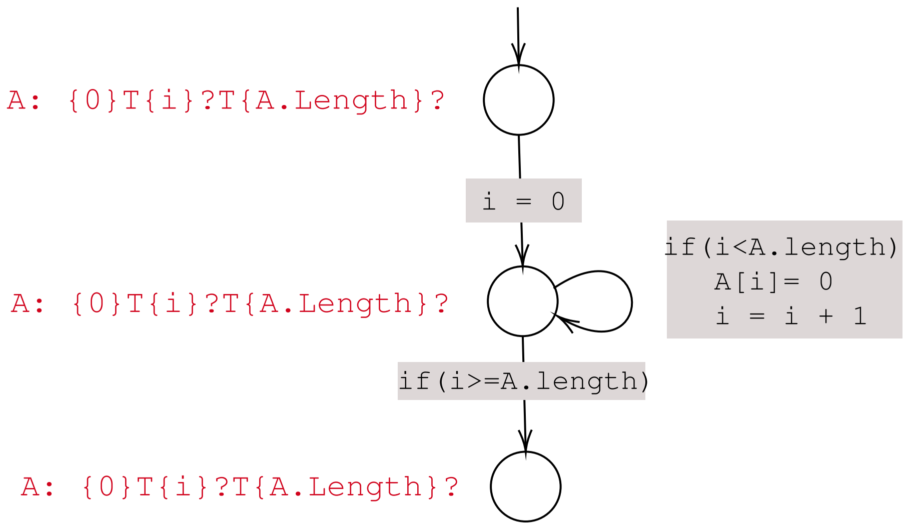
These definitions and algorithms are taken from (SS13).

5.2 An Worked-out Example

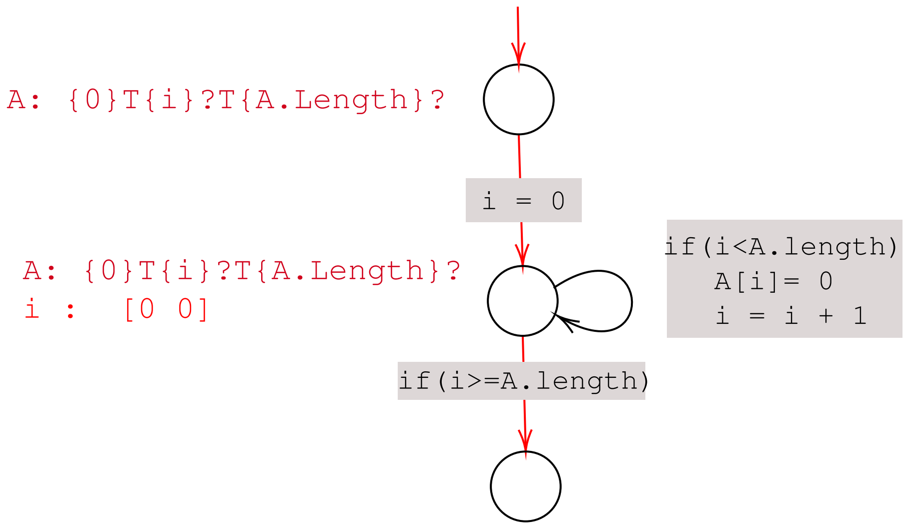
We'll show for the array initializatiion example once again.



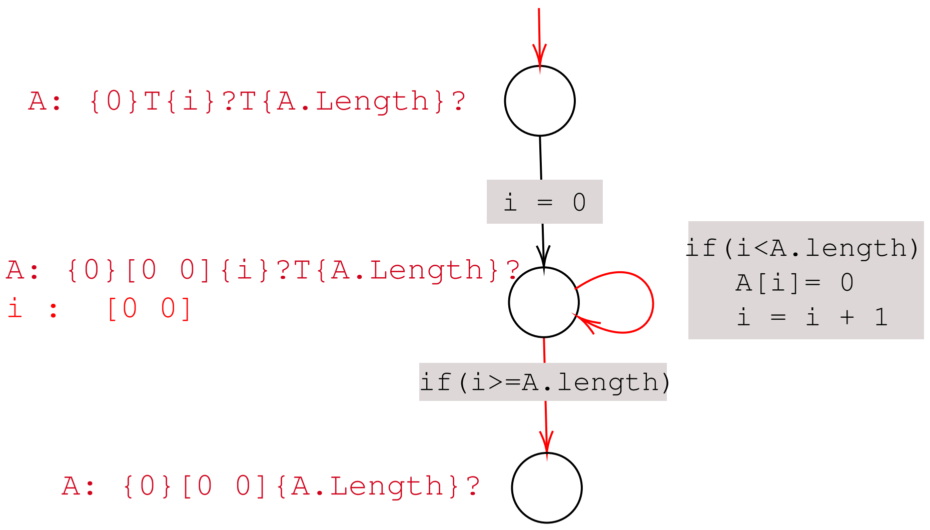
After initialization:



Choose a strategy: The strategy is shown in red.



Choose a different strategy.



This is how our algorithm works well in the problem.

5.3 Properties

Theorem 5.3.1 (Termination)**.** *The max-strategy algorithm for arrays terminates after a finite number of iterations.*

Proof. There is a finite number of strategies and each strategy is visited at most once. Therefore, max-strategy iteration terminates after a finite number of improvement steps. At this point max-SI has no more strategy to visit and returns a fixed-point.

Proof is contributed to (SS13). ■

Theorem 5.3.2 (Soundness)**.** *The max-strategy algorithm for arrays computes a fixed point.*

Proof. Each abstract domain is an over-approximation of the concrete domain, and fixedpoint computed using the Max SI enumerates all program paths. And till we reach a fixedpoint, we find an strategy to improve. Therefore, if termination is guaranteed, then soundness is also guaranteed. ■

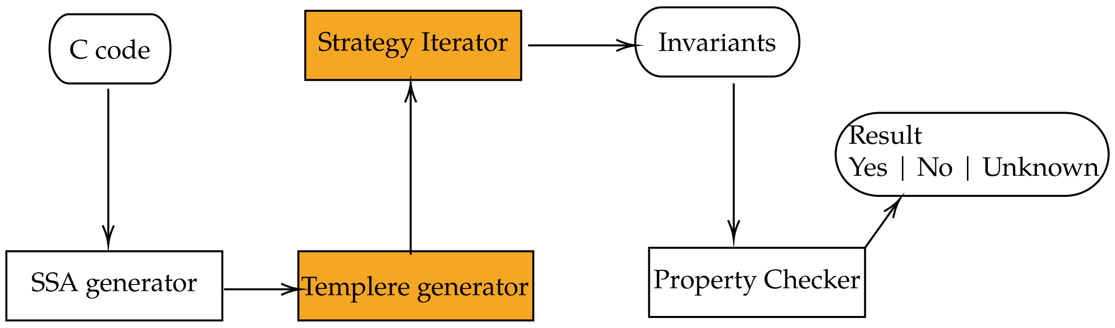
Chapter 6

Implementaion

Max Strategy iteration is smartly implemented in a tool called 2LS(SK16). To check the effectiveness of our method in real cases, we have chosen to pick the tool and implement the proposed method within that.

6.1 Components of 2LS

2ls takes a C program as input and send to SSA generator. SSA generator generates SSA of the entire function and stores into a local_SSAt object. Now SSA goes to template generator, where the guarded templates(see general technique section) for those queries are generated. Guarded templates are then goes to strategy iteration module. Here the queries are formed. Here the queries are solved. To improve d-constants(bounds) modified binary search is used to search over the domain. This module outputs invariants. Now property checker module checks desired properties by replacing functions. If the property satisfies, then return yes. If not satisfied, then returns no. If the generated invariant is not strong enough to prove the property, then returns unknown.



The shaded modules are modified to make 2ls handle the array segmentation.

We need to modify the template generator to generate templates for input and output invariants. We also need to generate queries differently from loops. Thats why strategy iteration module needs modification. Also, the solving techniques of those techniques are different.

6.2 Contribution

6.2.1 Defining a new Domain Data Structure

We were needed to define an data structure for this domain of arrays. And made the procedure to perform the strategy iteration over the domain.

The Abstract domain base classes are defined in `domain.h` header and classes inheriting that class are made to create specific domains. We have created a new domain called `array_segment_domain.cpp` that uses some classes from Template polyhedra domain `tpolyhedrea_domain.cpp` to represent things inside the array segments.

6.2.2 Performing Max-SI

We modified the code that does the max strategy iterator part.

Chapter 7

Experiments and Conclusion

7.1 Theoretical Experiments

In this section, we would like to take some examples from SV-COMP benchmarks, and some other sources and test whether they are provable within our proposed domain and running an strategy iteration over it.

7.1.1 Array Copy Programs

```
#define N 100000
int main( ) {
  int a1[N], a2[N], a, i, x;
  for ( i = 0 ; i < N ; i++ ) {
    a2[i] = a1[i];
  }
  for ( x = 0 ; x < N ; x++ ) {
    __VERIFIER_assert(a1[x] == a2[x]);
  }
  return 0;
}
```

Generating an invariant over `A - B`, e.g., `A - B : {0}[0,0]{i}`$\top$`{A.len}` would have sufficed to prove the assertion. Using template domain would have helped with this.

7.1.2 Initialization Modified

```
int n = 10, i = 0;
int[] A = new int[n];

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

Generating an invariant over `A`, e.g., `A : {0}[0,0]{i}⊤{n-i-1}[1,1]{A.len}` would have sufficed to prove the assertion. This shows a more class of program, that would have been solved with more than two domains, and is not solvable by less than that.

7.2 Conclusion

This thesis proposes a technique for generating array invariant for array-manipulating programs. For this purpose we have done a the following.

1. We followed an work by Cousot et. al.that defines an abstract domain for abstract interpretation (CCL11). And defined a domain to work with in Max-SI.
2. We demonstrated a process for generating invariant using Max-SI. The proposed methods works well with programs in a subclass of array-manipulating programs where the array is being manipulated within an loop and the loop counter has some relation with the index.
3. We finally proposed a design architecture for implementing our proposed scheme within 2LS.

7.3 Future Work

There are a few parts that we have skipped and assumed to be given. If these parts can be generated by some algorithm, then this would have been given optimal invariant for more set of programs.

Within the array segment domain, we are unable to generate the number of bounds or the expressions to be used in segment bounds. Any method in this area would come in great help.

In our belief, *Syntax Guided Synthesis* (SyGuS) may help us in this purpose.

Bibliography

[CCL11] Cousot, Patrick ; Cousot, Radhia ; Logozzo, Francesco: A parametric segmentation functor for fully automatic and scalable array content analysis. In: *ACM SIGPLAN Notices* 46 (2011), Nr. 1, S. 105–118

[CGU17] Chakraborty, Supratik ; Gupta, Ashutosh ; Unadkat, Divyesh: Verifying array manipulating programs by tiling. In: *International Static Analysis Symposium* Springer, 2017, S. 428–449

[Der] Dershowitz, Nachum: *SOFTWARE HORROR STORIES*. https://www.cs.tau.ac.il/~nachumd/horror.html

[GS07] Gawlitza, Thomas ; Seidl, Helmut: Precise fixpoint computation through strategy iteration. In: *European symposium on programming* Springer, 2007, S. 300–315

[MG16] Monniaux, David ; Gonnord, Laure: Cell morphing: from array programs to array-free Horn clauses. In: *International Static Analysis Symposium* Springer, 2016, S. 361–382

[Sch12] Schrammel, Peter: *Logico-Numerical Verification Methods for Discrete and Hybrid Systems*, Université de Grenoble, Theses, Oktober 2012. https://tel.archives-ouvertes.fr/tel-00809357

[SK16] Schrammel, Peter ; Kroening, Daniel: 2LS for program analysis. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2016, S. 905–907

[SS13] Schrammel, Peter ; Subotic, Pavle: Logico-numerical max-strategy iteration. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation* Springer, 2013, S. 414–433