TESTING IN LEARNING CONJUNCTIVE INVARIANTS

BY

PEYMAN MAHDIAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Associate Professor Madhusudan Parthasarathy

ABSTRACT

We show a new approach in learning conjunctive invariants using dynamic testing of the program. Coming up with correct set of loop invariant is the most challenging part of any verification methods. Although new methods tend to generate a large number of possible invariants hoping this set contains all required invariants needed to verify the program, this large number will cause a significant delay in verification which often ends up to a time out. Our approach introduce a new method in which we can solve this problem by reducing the number of generated candidate invariants.

We apply our method in a verification engine that uses natural proofs for heap verification. We implement our method by running tests for linked list data structures and evaluate it by comparing the results to the original approach without testing. We also use an existing GPU verification tool, called GPUVerify[1], and apply our method to it. Finally, we show that our approach can significantly improve the verification time and in some cases prove programs that were initially timed out.

To my parents, for their unconditional love and support.

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my adviser Professor Madhusudan Parthasarathy. His endless support and innumerable knowledge was beside me during my experience at University of Illinois Urbana-Champaign.

My special thanks goes to my previous Advisor Prof Tao Xie, who also taught me a many great lessons, and all of my colleagues and friends at Department of Computer Science who never stop supporting me.

I would like to thank the University of Illinois and specially department of Computer Science for giving me this unique and amazing opportunity to learn in one of the best schools worldwide.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Generating loop invariants is the most difficult and crucial task in verifying a program[2, 3, 4, 5, 6]. While there are many approaches to produce the sufficient loop invariant, many of them are not showing significant results. Learning is an approach in which a learner is used to come up with an invariant by having a teacher which tests the nominated invariant by the learner and gives feedback to learner[7, 8]. A learner can possibly generate any kind of loop invariant. However, a conjunctive invariant is much simpler to learn and generate. A learner which generates only invariants which are conjunctive of the basic predicates is called a conjunctive invariant learner.

A conjunctive invariant learner needs candidate invariants to learn the proper set of invariant for verifying the program. Since invariants can be very complicated and could possibly look very different, the number of candidate invariants needed to prove the program correct could be notable[9], and this number can always grow exponentially by the complexity of the program. For example, increasing the number of local variables inside the program can dramatically expand the number of candidate invariant.

Houdini[10] is an example of a conjunctive invariant learner. The problem with Houdini[10] other than that it will just learns conjunctive invariants, and it might not be sufficient as a loop invariant is that it takes considerable amount of time to remove a predicate, and when the number of predicates goes up it is not scalable. Therefore, it is typical to see a correct program to take a great amount of time to be verified or it times out and never get verified in the specified time[11].

Because of the fact that, when a conjunctive invariant is a loop invariant, all of the predicates that it includes, are also true in any configuration of the program, we know that a predicate which is false in at least one program configuration can never be a part of the conjunctive loop invariant. Therefore, we can use this fact to eliminate those predicates by testing. If we run the

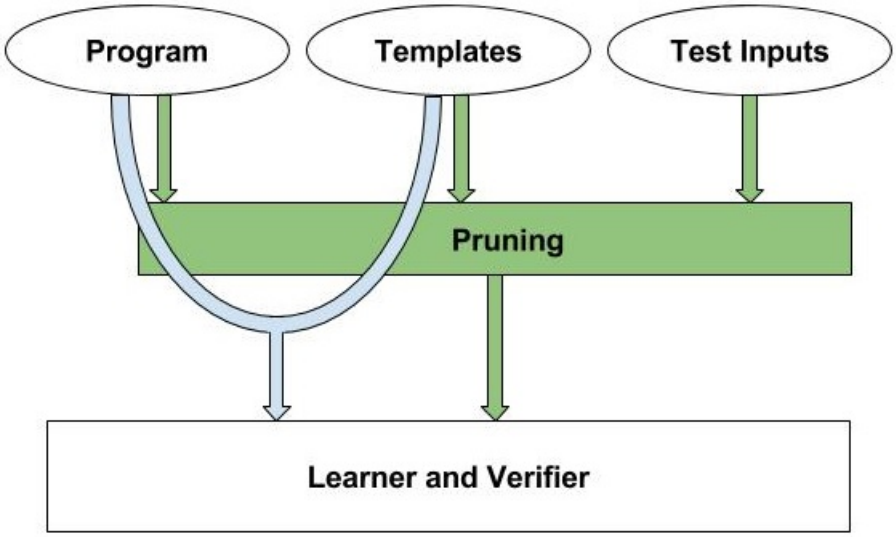

Figure 1.1: Two Different Approach. Green shows testing and blue shows without testing.

program, and evaluate each predicates where the loop invariant should be true, and the predicate is false, we can eliminate that predicate before feeding it to Houdini[10].

In our approach first we generate all the possible loop invariant candidates by using a template, and the number and type of variables inside the program. To prove the program correct first we simply feed all the candidate invariants to the conjunctive invariant learner and then try to prove it correct. All these candidate invariants are translated to Boogie[12] programming language for verification. On the second approach before we feed them to leaner we prune them by testing.

To prune candidate invariants by testing first we implemented the program and ran it by some random sample tests. The implemented program is modified to evaluate the candidate invariants. Since we are learning conjunctive invariants, we call these candidate invariants predicates. If a predicate is evaluated false in this process it will not be fed to learner for verification.

Figure 1.1 shows both original and our improved approach. The blue lines describe the basic way in which all the candidate invariants are passed to the leaner, while the green color highlights our new approach of using testing to prune the predicates.

Testing the predicates can significantly reduce the number of predicates

and it makes the Houdini[10] more scalable for larger number of predicates.

GPUVerify[1] is a tool for verifying race and divergence freedom of OpenCL and CUDA programming languages. The main difference between these programs and what we have discussed so far is that they are parallel programming languages. In other words, a GPU program is run by all GPU kernels at the same time, and this make them more difficult to analyze in terms of soundness of the pruning approach.

In the chapter 2 we explain how we inject the program with our evaluate function and we prove why this technique is sound. Chapter 3 shows an example of a program before and after injecting the evaluate function, and describes how the details are handled so there is no room for pruning a predicate by mistake. Set of template predicates and the way we came up with them are discussed in chapter 4. We talk about the detail implementations of our work in chapter 5 and we evaluate our approach in chapter 6. Future and related work are respectively discussed in chapters 7 and 8.

CHAPTER 2

APPROACH

In order to reduce the number of predicates, we tested the programs with a small set of inputs. We took the original program and injected it with a evaluate function which gets the state of the program (variables), and checks the validity of the predicates. Figure 2.1 shows where we injected the evaluate function calls.

```
S1;
evaluate(params);
while (b) {
    S2;
    evaluate(params);
}
S3;
```

Figure 2.1: Injected evaluate function calls.

Loop invariants always hold immediately before the loop, and after each iteration of the loop. Therefore, if the loop invariant is a conjunction between the predicates $p_1, p_2, ..., p_n$, these predicates should be true, when we are calling the evaluate function. The evaluate function removes any predicates which have been evaluated to false.

`S1`, `S2`, and `S3` are a set of a statements. `S2` should be a basic block while `S1` and `S3` are not required to be a basic block. A basic block is a block of statements which has only one entry point and one exit point, and when you execute first statement you will execute all statements. `S2` being a basic block is enough for our approach to be correct. However, if there is a inner loop inside `S2`, our approach will still be sound. To be precise the only way that `S2` not being a basic block can make our approach unsound would be when there is an entry point inside `S2` from either `S1` or `S2`, which is highly improbable in modern programming. Therefore, it will be easy to

make this injection to code automated without considering the branching and paths in the code.

`params` as arguments of `evaluate` includes all the predicates passed by reference which might be modified by `evaluate`. `params` also include all the local variables and arguments of the program which are used to generate the predicates.

Our approach for pruning GPU programs at core is the same as sequential programs. GPUVerify[1] takes a parallel program as an input, but turns it into a sequential program and then tries to prove the verification conditions. This makes it more interesting and at the same time more challenging with our approach, and the reason is we do not have the same program at the testing and proving stages. Thus, we cannot just use the same invariants.

Although a parallel programs at the testing stage becomes a sequential one at the verification time, the way GPUVerify[1] is designed makes it easy for us to translate the invarinats of the sequential program to their representation at the parallel stage. Moreover, this translation is sound if and only if when the parallel representation is false we can say the invariant of the sequential program is also false.

GPUVerify[1] translates a parallel pragram into a sequential one by duplicating all the variables and randomizing the value of a variable whenever a write happens. This means the sequential program is the cross product of only two threads running in parallel and the variable havoc represents the effect of other threads.

The sequential program includes all the states that a single-thread run of the parallel program can have. If we have an invariant in the sequential program consisting of variables only from one thread of the parallel program, then if we have a single-thread run of the parallel program showing the invariant is false, it means the invariant in the sequential programs would also be false. That's the key of our approach with GPUVerify programs. Figures 2.2 and 2.3 show a simple program in parallel and sequential stage.

Our approach for GPUVerify is to generate the invariants for sequential programs at the parallel level then run the program with a simple test and see which invariants are being falsified. For simplicity, we just use one type of GPUVerify invariants to test.

```
_kernel void foo(_local int* A, int idx) {
  int x;
  int y;

  x = A[tid + idx];

  y = A[tid];

  A[tid] = x + y;
}
```

Figure 2.2: Parallel GPU Program

```
//requires 0 <= tid$1 && tid$1 < N
//requires 0 <= tid$2 && tid$2 < N
//requires tid$1 != tid$2
//requires idx$1 != idx$2

void foo(int idx$1, int idx$2) {
  int x$1; int x$2;
  int y$1; int y$2;

  LOG_READ_A(tid$1 + idx$1);
  CHECK_READ_A(tid$2 + idx$2);
  havoc(x$1); havoc(x$2);

  LOG_READ_A(tid$1);
  CHECK_READ_A(tid$2);
  havoc(y$1); havoc(y$2);

  LOG_WRITE_A(tid$1)
  CHECK_WRITE_A(tid$2)
}
```

Figure 2.3: Sequential Program

CHAPTER 3

EXAMPLE

Figure 3.1 is an initial program without any modification. This specific program gets a `Node* h` as a head of list and an `int k`, removes `k` from the list if it exists. This program and so many of other programs use only two types of variables `Node*` and `int`.

Figure 3.2 shows the same `SLL_delete` program after we added the evaluate function. We added a `boolean` vector of predicates which are passed by reference to the `SLL_delete` function. This vector is initially `true` for all predicates but each time `evaluate` is called its values might get changed. Since all the variables are either `Node*` or `int`, `evaluate` takes two vectors of types `int` and `Node*` which represent the value of all variable in the function.

In the Figure 3.3 `evaluate` function modifies the value of each predicates in a way that if it changes to `false` once, it will remain `false` forever. Therefore, we capture the essence of the invariant which should be true in all possible situation.

Another important observation is that we preserved everything from the start of the `SLL_delete` to the beginning of the `while`, but we just simply removed everything after `while`, because we do not care about what happens afterwards. All we care is to get the state of program when we reach the `while` and in each iteration of `while`. Persevering everything before `while` is important because we need to know the initial value of variables before the first iteration. For example, in this program if `i == NULL`, we never reach the while, and we should keep it this way.

One important observation in Figure 3.2 is that we added a new variable `Node * P_h`. The reason behind this is that we need the old value of pointers in our invariant candidates. Some invariant candidates use both old value and new value of a pointer. That's why both index 0 and 1 of the `ptrs` are storing `Node * h`, but respectively old and new value. We added a

```
Node * SLL_delete(Node * h, int k)
  Node * i = h;
  Node * j = NULL;
  Node * t = NULL;
  if (i == NULL) {
    return h;
  }
  while(i != NULL && t == NULL)
  {
    if (i->key == k) {
      t = i;
    } else {
      j = i;
      i = i->next;
    }
  }

  if (i != NULL) {
    if (j == NULL){
      h = i->next;
      free(i);
    } else {
      t = i->next;
      free(i);
      j->next = t;
    }
  }
  return h;
}
```

Figure 3.1: Program Before Adding Evaluate Function Calls

```
void SLL_delete(vector<bool> &pred, Node* h, int k){
    Node * P_h = h;
    Node * i = h;
    Node * j = NULL;
    Node * t = NULL;
    vector<Node*> ptrs; ptrs.resize(5);
    vector<int> ints; ints.resize(1);

    if (i == NULL) {
        return;
    }

    ptrs[0] = P_h; ptrs[1] = h; ptrs[2] = i;
    ptrs[3] = j; ptrs[4] = t; ints[0] = k;
    evaluate(pred, ptrs, ints);

    while (i != NULL && t == NULL) {
        if (i->key == k) {
            t = i;
        }
        else {
            j = i;
            i = i->next;
        }
    ptrs[0] = P_h; ptrs[1] = h; ptrs[2] = i;
    ptrs[3] = j; ptrs[4] = t; ints[0] = k;
    evaluate(pred, ptrs, ints);
    }
}
```

Figure 3.2: Injected Code Around Loop to Evaluate the Predicates

```
void evaluate(vector<bool> &pred, vector<Node *> ptrs,
    vector<int> ints){
    Node * Plist = ptrs[0];
    Node * locallist = ptrs[1];
    pred[0] = pred[0] && sll(Plist);
    pred[1] = pred[1] && sll(locallist);
}
```

Figure 3.3: Evaluate Function

```
_kernel void scan_inter2_kernel(_global unsigned int* data,
    unsigned int iter)
{
  _requires(iter == 1);

    unsigned int stride = get_local_size(0)*2;

    for (unsigned int d = 1; d <= get_local_size(0); d *= 2) {
      stride >>= 1;

      barrier(CLK_LOCAL_MEM_FENCE );

      if (thid < d) {
        unsigned int i = 2*stride*thid;
        unsigned int ai = i + stride - 1;
        unsigned int bi = ai + stride;

        ai += CONFLICT_FREE_OFFSET(ai);
        bi += CONFLICT_FREE_OFFSET(bi);

        unsigned int t = s_data[ai];
        s_data[ai] = s_data[bi];
        s_data[bi] += t;
      }
    }
    barrier(CLK_LOCAL_MEM_FENCE );

    data[g_ai] = s_data[s_ai];
    data[g_bi] = s_data[s_bi];
}
```

Figure 3.4: GPU Kernel

new pointer variable for each pointer variable that is passed as argument to the program. This itself shows why increasing the number of variables can drastically increase the number of predicates. However, because we use testing here it would not be much effective.

Figure 3.4 shows a simplified GPU kernel code in OpenCL language. We added the `evaluate` function similarly here before the loop and at the end of all loop statements.

CHAPTER 4

PREDICATES

Our approach for coming up with the conjunctive loop invariant is simple. First, we generate so many predicates then we test them to remove the false ones, and finally we feed them to a learner (here Houdini[10]) to come up with the loop invariant.

Choosing what should be as the basic predicates were done by looking at the examples and finding those predicates which are common and then generalizing them. We call these generalized new predicates, *template predicates*. In order to avoid the increasing number of predicates, we chose to have three phases which each new phase adds new predicates templates, and if the program fails in one phase, we expand the number of templates and go to next phase till the program verification succeeds. Figure 4.2 shows the different predicate templates we used for each phase.

Figure 4.1 shows the procedure in which candidate invariants for each phase are added to the program in order to increase the chance of verification success and also reduce the time of verification. The complexity and number of templates in each phase increases, so moving forward to next phase will significantly add the number of predicates. However, because all these predicates can be pruned using testing, we can still add more complicated predicated and not be worried about time out. If we succeed to verify the program in each phase we halt the procedure.

We can also apply this three phase approach to any other type of programs not just linked lists. We also can add more phases if we are dealing with more complicated data structures. Generally, the idea of using more than one phase can help the process of verification in both time and simplicity.

All the templates are used with the program variables(local variables, arguments, our defined variables for old value) to generate all possible predicates. It is obvious here why if we add a new variable the number of predicates will expand dramatically. For example if a template uses two distinct variables

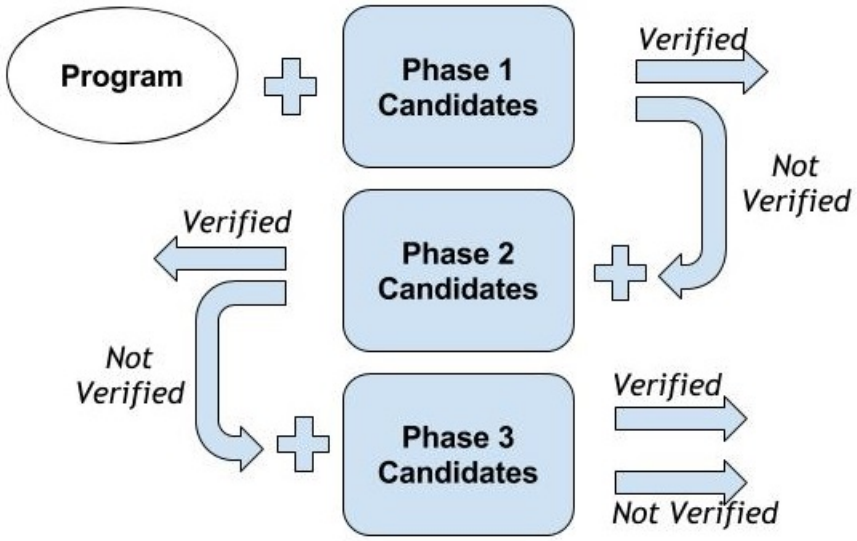

Figure 4.1: Three Phase Predicate Generation

and we have 3 such variables in our program the number of predicates generated by this templates would be $\binom{3}{2} = 3$. However, if we add one more variable of such type to the program, the number would be $\binom{4}{2} = 6$. This itself again shows the important of pruning while using the template method in loop invariant inference.

GPUVerify[1] has its own predicate generation. The only predicate type we focused was the power two type. Figure 4.3 describes this invariant template and when they are being generated. If we apply the described rule on the kernel we showed in figure 3.4, we will get the candidate invariants in the figure 4.4.

$x, y, x_1, x_2 \in PointerVars$
$\vec{x}, \vec{y}, \vec{z} \in PointerVars^*$
$i, j \in IntegerVars \cup \{0, \mathsf{IntMax}, \mathsf{IntMin}\}$
$pf \in PointerFields$
$key,\ df \in DataFields$

$$
\begin{array}{rcl}
list(\vec{x}) & := & \mathsf{LinkedList}(x_1) \mid \mathsf{DoublyLinkedList}(x_1) \mid \mathsf{SortedLinkedList}(x_1) \\
& \mid & \mathsf{LinkedListSeg}(x_1, x_2) \mid \mathsf{DoublyLinkedListSeg}(x_1, x_2) \\
& \mid & \mathsf{SortedLinkedListSeg}(x_1, x_2)
\end{array}
$$

$$
\begin{array}{rl}
 & list(\vec{x}) \qquad\qquad\qquad\qquad\qquad\qquad x = \mathsf{nil} \quad x = y \\
 & x \in list\_\mathsf{heaplet}(\vec{y}) \qquad\qquad\qquad\qquad x \neq \mathsf{nil} \quad x \neq y \\
 & x \notin list\_\mathsf{heaplet}(\vec{y}) \qquad\qquad\qquad\qquad x.pf = \mathsf{nil} \quad x.pf = y \\
Phase\ \ 1 & list\_\mathsf{heaplet}(\vec{x}) \ \cap\ list\_\mathsf{heaplet}(\vec{y}) = \emptyset \quad x.pf \neq \mathsf{nil} \quad x.pf \neq y \\
\hline
 & x.df \leq y.df \qquad\qquad x.df \leq i \\
 & x.df \geq y.df \qquad\qquad x.df \geq i \\
 & x.df = y.df \qquad\qquad x.df = i \\
Phase\ \ 2 & x.df \neq y.df \qquad\qquad x.df \neq i \\
 & i \in list\_key\_\mathsf{set}(\vec{x}) \qquad i \notin list\_key\_\mathsf{set}(\vec{x}) \\
 & list\_key\_\mathsf{set}(\vec{x}) = list\_key\_\mathsf{set}(\vec{y}) \\
 & list\_key\_\mathsf{set}(\vec{x}) = list\_key\_\mathsf{set}(\vec{y}) \ \cup\ list\_key\_\mathsf{set}(\vec{z}) \\
\hline
 & size(\vec{x}) = i \qquad\qquad size(\vec{x}) \geq i \\
 & size(\vec{x}) = i - j \qquad\qquad size(\vec{x}) \leq i \\
 & size(\vec{x}) - size(\vec{y}) = i \qquad\qquad size(\vec{x}) - size(\vec{y}) = i - j \\
Phase\ \ 3 & list\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{i\} \qquad\qquad list\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{i\} \\
 & list\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{y.df\} \qquad\qquad list\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{y.df\} \\
 & list\_key\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} list\_key\_\mathsf{set}(\vec{y}) \\
 & list\_key\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} shape\_key\_\mathsf{set}(\vec{y})
\end{array}
$$

Figure 4.2: Templates for predicates. The operator $\leq_{\mathsf{set}}$ denotes comparison between integer sets, where $A \leq_{\mathsf{set}} B$ if and only if $\forall x \in A. \forall y \in B.\ x \leq y$. The operator $\geq_{\mathsf{set}}$ is also similarly defined.

$$
\begin{array}{ll}
Conditions: & i = i * 2 \text{ or } i = i/2 \text{ occurs in the loop} \\
 & i \text{ is live at the loop head} \\
 & D \text{ is the smallest power of 2 with } D \leq SZ \\
 & \\
GeneratedCandidates: & i\&(i-1) = 0,\ i \neq 0,\ i < 1,\ i < 2,\ i < 4, ...,\ i < D
\end{array}
$$

Figure 4.3: GPUVerify Power Two Invariant Generation Rule

```
stride == 0 || (stride & (stride - 1) == 0
stride != 0
d == 0 || (d & (d - 1)) == 0
d != 0
(stride == 0 && d == 2) || stride * d = 1
(stride == 0 && d == 4) || stride * d = 2
(stride == 0 && d == 8) || stride * d = 4
(stride == 0 && d == 16) || stride * d = 8
(stride == 0 && d == 32) || stride * d = 16
(stride == 0 && d == 64) || stride * d = 32
(stride == 0 && d == 128) || stride * d = 64
(stride == 0 && d == 256) || stride * d = 128
(stride == 0 && d == 512) || stride * d = 256
(stride == 0 && d == 1024) || stride * d = 512
(stride == 0 && d == 2048) || stride * d = 1024
(stride == 0 && d == 4096) || stride * d = 2048
(stride == 0 && d == 8192) || stride * d = 4096
(stride == 0 && d == 16384) || stride * d = 8192
(stride == 0 && d == 32768) || stride * d = 16384
```

Figure 4.4: GPU Kernel Power Two Candidate Invariants

CHAPTER 5

IMPLEMENTATION

In this chapter we talk about the detail implementation, and discuss how we ran and executed the tests and knocked off the predicated before feeding it to Houdini[10].

The process of testing consists of two phases. In the first phase we generate a specific header file including the evaluate function that has all the generated predicates. This header file other than evaluate function which was shown in Figure 3.3, has another method called `print_pred` that prints the surviving predicates after pruning (Figure 5.1). For simplicity, when we generate the predicates we generated the Boogie[12] version of them too, and at the end we simply print out the the Boogie[12] version so it can be used directly in verification.

The whole first phase is done automatically. A JavaScript code permutes all the possible predicate patterns (Figure 4.2) by having the name and number of variables of each type (`Node*` and `int`), and generates this header file.

In the second phase, we run the injected program with evaluate function calls (Figure 3.2). Whenever the evaluate function gets called the value of predicates will be updated, and at the end the corresponding Boogie[12] version of invariants will be printed out.

Some of programs had unimplemented functions, but these functions did not have any pre- or post-conditions, so we safely replaced them with some simple functions. For example, `filter` function calls `to_remove` function which decides that if this `Node i` should be removed or not. We replaced this call with `i->key % 2 == 0` (Figure 5.2).

We ran each program with at most 5 different inputs, and compiled the surviving predicates. Our input set consists of singly linked lists (*sll*) with maximum length of 5. We gave each program a separated set of input to make sure this input satisfies the precondition of the program. Figure 5.3

```
void print_pred(vector<bool> pred) {
    vector<string> cpp_pred;
    cpp_pred.push_back(
        string("F#sll($s,$phys_ptr_cast(P#list,^s_node))"));
    cpp_pred.push_back(
        string("F#sll($s,$phys_ptr_cast(local.list,^s_node))"));

    for (int it = 0; it<pred.size(); it++)
        if (pred[it]) {
            cout << "\n" << cpp_pred[it];
        }
}
```

Figure 5.1: Print Surviving Predicates After Pruning in Boogie Language

```
while (i != NULL) {
    if (i->key % 2 == 0) { //to_remove(i)
        Node * nxt = i->next;
        j->next = nxt;
    } else {
        j = i;
    }
    i = i->next;

    ptrs[0] = h; ptrs[1] = i; ptrs[2] = j;
    evaluate(pred, ptrs, ints);
}
```

Figure 5.2: An Example of Unimplemented Function

```
//input 1
int k1 = 3;
int myints1[] = { 1, 2, 3, 4, 5 };
vector<int> keys1(myints1, myints1 + sizeof(myints1) /
    sizeof(int));
Node * list1 = gen_sll(keys1);
SLL_delete(pred, list1, k1);

//input 2
int k2 = 1;
int myints2[] = { 1 };
vector<int> keys2(myints2, myints2 + sizeof(myints2) /
    sizeof(int));
Node * list2 = gen_sll(keys2);
SLL_delete(pred, list2, k2);

//input 3
int k3 = 1;
int myints3[] = { 4 };
vector<int> keys3(myints3, myints3 + sizeof(myints3) /
    sizeof(int));
Node * list3 = gen_sll(keys3);
SLL_delete(pred, list3, k3);
```

Figure 5.3: Random Inputs for Each Program

shows how we run the program with different inputs. These inputs are chosen randomly from a predefined set that we constructed to achieve corner cases like empty list, single item list, and list with more than one item.

To evaluate template predicates, we needed to implement the list of function in Figure 5.4 for singly-linked list (*sll*).

In order to check that a pointer is indeed a singly linked list and not a cycle, when the $sll(x)$ is being evaluated, we keep track of every node by collecting them in a set. The concept of set here has a small difference. Here

sll	*sll_reach*	*sll_keys*	*sll_len*
sll_lseg	*sll_lseg_reach*	*sll_lseg_len*	*sll_lseg_keys*
intset_le	*intset_lt*	*intset_subset*	*oset_in*
oset_disjoint	*oset_union*	*oset_singleton*	*intset_in*
intset_disjoint	*intset_union*	*intset_singleton*	

Figure 5.4: Singly Linked List Functions

a set can be $Undefined$. The undefined concept is used when a pointer is not actually a list or a list segment, and in this case no matter what the variable is pointing to we consider it $Undefined$. To complete the definition, our `Boolean` type other than `True` and `False` has an extra constant which is also called `Undefined`. Any function on an $Undefined$ set returns `Undefined`, and any `Boolean` operation which has an `Undefined` operand will return `Undefined`. The concept of $Undefined$ will help us not to worry about what actually are pointers pointing to and easily evaluate the predicates by just calling *sll* functions. We used operator overloading to easily use operator on our new types with $Undefined$ value (Figure 3.3).

GPUVerify[1] benchmark kernels that we used for testing do not need more than some `int` inputs to be provided, so the tests for them is completely randomized and we just use random integers which satisfy the precondition of the loop. Therefore, we know for sure that we are going to execute the loop at least once.

CHAPTER 6

EVALUATION

We evaluated our approach with 27 different routines that manipulate linked lists. These programs are from GNU C library, and benchmarks used to evaluate separation logic based tools[13, 14].

We focused on linear data structure as a base because it is much simpler to implement and test and at the same time are one of the most significant unbounded data structures used in computer science[15].

Catagory	Method	#pred		NP + INV + Testing				No Testing
		before	after	verified	Phase	#conj	Time(s)	Time(s)
glib/gslist.c Singly-Linkedlist	g_slist_index	291	89	Yes	3	64	14	173
	g_slist_insert_before	265	47	Yes	1	44	4	45
	g_slist_insert	140	34	Yes	1	19	5	13
	g_slist_length	66	19	Yes	3	12	7	24
	g_slist_nth_data	414	115	Yes	3	73	21	276
	g_slist_nth	108	31	Yes	3	17	6	46
	g_slist_remove_all	452	71	No	3	–	TO	TO
	g_slist_remove_link	605	117	Yes	2	76	10	121
	g_slist_remove	140	35	Yes	1	27	2	9
	g_slist_sort_merge	2051	377	No	3	–	TO	TO
GRASS-hopper Singly-LinkedList	sl_concat	63	20	Yes	1	15	1	2
	sl_dispose	22	6	Yes	1	4	1	1
	sl_filter	140	22	Yes	1	8	5	13
	sl_insert	63	13	Yes	1	9	1	3
	sl_reverse	63	8	Yes	1	4	1	4
	sl_traverse	22	7	Yes	1	4	1	1
GRASS-hopper Sorted List	sls_double_all	140	37	Yes	1	37	2	TO
	sls_insert	117	24	Yes	2	24	18	457
	sls_pairwise_sum	450	91	Yes	1	91	2	TO
	sls_split	63	12	Yes	1	11	22	45
AFWP Singly-Linked and Doubly-LinkedList	SLL-create	5	3	Yes	1	1	1	1
	SLL-delete-all	22	3	Yes	1	2	1	1
	SLL-delete	265	47	Yes	1	44	2	20
	SLL-find	140	49	Yes	1	37	2	5
	SLL-insert	63	10	Yes	1	9	3	2
	SLL-last	63	11	Yes	1	8	1	3
	SLL-reverse	63	8	Yes	1	4	1	3

Table 6.1: Linked List Evaluation Results

Table 6.1 shows the results of using pruning with testing. Testing reduces the initial number of predicates up to between 2 to 8 times. This much speed up can make Houdini[10] more scalable. As you can see without testing total time are much larger. Total time with testing was up to 25 times shorter than without testing. Moreover, two programs that were timing out before, got verified with testing. This shows testing can actually make Houdini[10] more scalable when the number of predicates increases.

For GPUVerify[5], We used their benchmark and just focused on the loop invariant generation part and specifically tried all the programs that are generating power two invariants. This list as its shown in Table 6.2 consists of 16 different programs which 9 of them are in OpenCL and the rest are CUDA.

Catagory	Method	#pred		NP + INV + Testing			No Testing
		before	after	verified	#conj	Time(s)	Time(s)
AMD SDK	HistogramAtomics	48	2	No	–	TO	TO
	PrefixSum	74	14	Yes	14	17	200
	ScanLargeArrays2	74	14	Yes	14	19	213
	ScanLargeArrays3	74	14	Yes	14	25	231
parboil mri-gridding	scan_inter1	36	6	Yes	6	29	72
	scan_inter2	38	8	Yes	8	10	18
	scan_L1	74	14	Yes	14	1696	TO
	splitSort	84	24	No	–	TO	TO
shoc	scan	74	14	Yes	14	31	312
CUDA20	scan_best	76	16	Yes	16	47	475
	scan_workefficient	76	16	Yes	16	44	535
	scan_inter2	76	16	Yes	16	121	1046
CUDA50	mergeSort	40	8	Yes	8	280	1992
	bitonicSort0	40	8	Yes	8	1020	1475
	bitonicSort1	40	8	Yes	8	338	1299
	oddEvenMergeSort	40	8	Yes	8	362	1381

Table 6.2: GPUVerify Evaluation Results

Table 6.2 describes the result of out experiments with GPUVerify benchmark. Testing in this case can prune up to 95 percent of power two invariants and can speed up the verification up to 93 percent. We also managed to verify one program that were timed out without testing. Therefore, we can have similar indication that testing can make Houdini[10] more scalable.

CHAPTER 7

FUTURE WORK

We can extend and improve what we did in future in a few areas. One area would be to make our tool fully automatic. The main concerns for this purpose would be test input generation. However, we can assume when a developers wants to verify its program they can come up with some test cases. Another region for extension would be to apply the same approach on other types of verification.

In order to make the tool fully automated, we need to work on some points. We need to extract the name, type and number of variables of the program, detect the start and end of loops, check the entry points inside the loop, find unimplemented functions and replace them with safe functions, and generate the proper test inputs for the program. The latter is more difficult, but there are already some tools to automatically generate test inputs. The most famous one is Microsoft PEX[16] which uses symbolic execution to come up with the test inputs to achieve the most branch coverage. Another similar tool is KLEE[17] which also uses symbolic execution to achieve high code coverage. Between these two KLEE[17] could be potentially a better choice for the current tool since it is applicable for C programming language compared to PEX[16] which uses C#.

We can definitely extend the area of this approach from heap data structures to any type of verification using conjunctive loop invariant generations. A proper next project would be adding more GPUVerify[1] programs to the evaluation by focusing on different type of invariants.

The main weak point of our approach is that it is only applicable to conjunctive invariants. Although a great number of programs only need conjunctive invariants, lack of disjunctive invariants puts the weight on coming up with templates. Therefore, Another future work could be looking into finding a way to apply our approach for disjunctive invariant either by making the template generation smart or using new ways that change the problem

of disjunctive invariants inference to conjunctive ones[6].

The best scenario for invariant generation is when everything is fully automated including coming up with the templates. Thus, the most difficult and at the same time crucial future work would be automating the process of template generation. One way to approach this problem would be using other elements inside the programs like assumptions, assertions and type of predicates.

CHAPTER 8

RELATED WORK

Since invariant inference is the biggest challenge of verification, there are so many related works around it. We can simply divide all invariant inference methods to two main categories black-box and white-box. White-box method include abstract interpretation[18], interpolation[19, 20] and constraint solving[21, 22, 5]. However, the most prominent black-box tool is Daikon[23] which also uses a dynamic approach to find the proper invariant. Another new tool in black-box technique is Houdini[10] which uses learning.

Loop invariant inference using dynamic testing is become one of the greatest technique in computer science and specially in verification[24]. Another tool which works around this methodology is DySy[24]. This tool combines the concrete executions of tests on the program with the dynamic symbolic executions to come up with the lowest set of candidate invariant, therefore, speed up the process of verification.

While black-box techniques like Daikon[23] and Houdini[10] seem to be more applicable in finding loop invariants, the main challenge for them would be to come up with the most useful templates. This task is difficult because in real life programs, loop invariants can be dramatically complex. On the other hand, white-box approach can help in these situations. One way to infer the loop invariant would be to look at the already existing assertions in the program and come up with the proper invariant[11].

Using templates in generating loop invariants is drastically increasing in new invariant inference methodologies[1, 8]. Tools like GPUVerify[1] are showing how actually this approach can be useful in real life programs and can help developers to verify their programs in a relatively short amount of time without any extra work.

The main reason behind increasing use of templates in invariant generation is learning tools like Houdini[10] which use machine learning to provide a simple way to test a set of candidate invariants and give feedback. The

most important newly introduced category of learners is ICE[7] which uses example, counter-examples and implications to come up with the proper loop invariant to prove the program correct.

CHAPTER 9

CONCLUSION

In sum, testing approach in learning conjunctive loop invariant is highly effective and efficient. Our framework was semi-automatic because we needed to write down the test cases manually. However, testing can be easily automated by using tools like PEX, or other tools that provides test inputs, and because running time compared to learning and verification time is not considerable, it will be a great help to invariant learners. On the other hand, testing can be useful only when we are considering conjunctive invariants, and we cannot use testing, when we consider other type of invariants like disjunctive invariants, but if we use more sophisticated templates we can achieve that.

Three phase approach to generate invariants also is very useful. Not only it reduces the problem of complicated invariants for simple programs, but it also reduce the time of verification significantly by lowering the number of candidate invariants.

REFERENCES

[1] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[2] A. Bouajjani, C. Drăgoi, C. Enea, A. Rezine, and M. Sighireanu, "Invariant synthesis for programs manipulating lists with unbounded data," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 72–88.

[3] A. B. C. D. C. Enea and M. Sighireanu, "On inter-procedural analysis of programs with lists and data," 2011.

[4] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Learning universally quantified invariants of linear data structures," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 813–829.

[5] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 634–640.

[6] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 703–719.

[7] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 69–87.

[8] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi, "Automatically inferring quantified loop invariants by algorithmic learning from simple templates," in *ASIAN Symposium on Programming Languages and Systems*. Springer, 2010, pp. 328–343.

[9] Y.-F. Chen and B.-Y. Wang, "Learning boolean functions incrementally," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 55–70.

[10] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 500–517.

[11] M. Janota, "Assertion-based loop invariant generation," 2007.

[12] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[13] J. Berdine, C. Calcagno, and P. W. Ohearn, "Smallfoot: Modular automatic assertion checking with separation logic," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 115–137.

[14] E. Pek, X. Qiu, and P. Madhusudan, "Natural proofs for data structure manipulation in c using separation logic," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 440–451.

[15] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Quantified data automata for linear data structures: a register automaton model with applications to learning invariants of programs manipulating arrays and lists," *Formal Methods in System Design*, vol. 47, no. 1, pp. 120–157, 2015.

[16] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[17] C. Cadar, D. Dunbar, D. R. Engler et al., "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[18] 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*. ACM, 1977, pp. 238–252.

[19] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[20] R. Jhala and K. L. McMillan, "A practical and complete approach to predicate refinement," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2006, pp. 459–473.

[21] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 281–292, 2008.

[22] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 420–432.

[23] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[24] C. Csallner, N. Tillmann, and Y. Smaragdakis, "Dysy: Dynamic symbolic execution for invariant inference," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 281–290.