A brief survey of Formal Verification

Report submitted by the Formal Verification Research Group, ISI Kolkata to SQAD, ASL

March 2016

Contents

Chapter 1
Software Verification

No Author Given

1.1 Introduction

Software Verification has become a must necessary step of Software engineering. All we need to have a code that is bug free. Now what is a bug ? when a code gives unwanted result we can say that there is a bug in the code for which we are not getting desired result. How can we detect that there is a bug in the code. During the Software development life cycle after development of the code the code undergoes testing phase. That basically validates the code for some input set. But it is not possible to test all input set for a particular code. Let us consider the following code:–

```
void main()
{
  int x, y;

  assert((x-y>0) == (x>y));

}
```

In the above code we are concluding that if the difference of any two integer number be greater than 0 then subtrahend is greater than the minuend. While testing this small code we will take some integer value and check but it is very unlikely that we can consider x = 1935052801 and Y = -229208062, for this condition our conclusion is not correct. Software Validation does not guarantee bug free code. Because Exhaustive testing is not possible by this traditional methods.

Now-a-days it is considered that if a code goes through exhaustive testing that is for all possible inputs (that can be the total range of the data type) and for each case it gives correct result (gives result according to the specification) then only it is said bug free code. This is the aim of Formal Verification.

Software Verification gives sounds result but is undecidable. That is if it says that a code is correct then there cannot be any bug in that code as it checks for all possible inputs. But for undecidable problems Formal verification cannot give correct result in a finite amount of time. It can find bug up to a definite range. Bugs that are beyond that range are not found.

1.2 Bounded Program Analysis

An execution path consists of sequence of instructions executed during a single run of the program. The states observed along the execution path are called trace. For different inputs many such different traces can be there in the program. We can represent each path with formula: for particular inputs assignments along that trace will be satisfied. These formula are called symbolic simulation. The useful application of symbolic simulation can be automatic test generation, detection of dead code and verification of asserted properties.

with a view to avoiding undecidability, we can impose bounds on the loops. Thus we can restrict the unwinding of the loop upto our desirable depth. Then we can construct a logical formula that gives a clear picture of the relationship between input and output variables. Here comes static single assignment that is a variable with time stamp that signifies a variable used once in the code and says the value of that variable at that particular time.

let us consider an example of loop, calculation of factorial of a number:–
1 void factorial (int n)

2 {

3 int i = 1;

4 int product = 1;

5 while (i ¡= n)

6 {

7 product = product * i;

8 i = i + 1;

9 }

10 }

The following table shows the sequence of instruction that comes through the execution of the above code. It has 3 Columns, First Column says the line number, second Column says the type of the instruction and the third Column shows the instruction or condition:–

Line No.	Type	statement or condition
3	assignment	i = 1;
4	assignment	product = 1;
5	branch	i < n;
7	assignment	product = product * i;
8	assignment	i = i + 1;

The above statements can be represented using single static assignment in the following way.

For this we are considering that n = 2.

Line No.	Type	statement or condition
3	assignment	$i_0 = 1;$
4	assignment	$product_0 = 1;$
5	branch	$i_0 < n;$
7	assignment	$product_1 = product_0 * i_0;$
8	assignment	$i_1 = i_0 + 1;$
5	branch	$i_1 < n;$
7	assignment	$product_2 = product_1 * i_1;$
8	assignment	$i_2 = i_1 + 1;$
5	branch	$i_2 < n;$

In the above table assignments are written using single static assignment. When i is first used in line number 3, it is denoted as i_0. From line 3 to 7 of the code the value remain same. In line number 8 the value of i is incremented to 1. now this new value of i is assigned in i_1. So it is called static single assignment where each variable will be assigned only once.

This SSA form is now translated into a logical formula called the **path constraint**. The formula is shown below ssa $\Longleftrightarrow i_0 = 1; \wedge$
$product_0 = 1; \wedge$
$i_0 < n; \wedge$
$product_1 = product_0 * i_0; \wedge$
$i_1 = i_0 + 1; \wedge$
$i_1 < n; \wedge$
$product_2 = product_1 * i_1; \wedge$
$i_2 = i_1 + 1; \wedge$
$i_2 < n; \wedge$

1.3 Assertion

While we write a program we want to perform some functionality. We can consider that a big task or function is an integration of small task. If these small small tasks can be handled successfully then the big task can be solved confidently. These small tasks or situation can be checked using Assertion.

If we could handle that situation properly in the code and we invoke assertions for that situation it will pass that means we handled that situation properly otherwise the assertion will fail. In case assertion, each run of the code is explored to check whether the properties hold or not.

For example , Let us consider the following assert:

```
int a, b;
assume(a = 6);
assert(b < a);
```

The above written assertions are valid Because as the range of b is not specified it can be any number less than 6. But consider the following assertion :

```
int a, b;
assume(a = 6);
assume(b > 10);
assert(a > b);
```

This assertion is not satisfiable as b cannot be greater than 10 and less than 6 at the same time.

1.4 Methods for Software Verification

1.4.1 Theorem Proving

When we want to write a program we want to get desired output from that program. Or when we have a program we want to know what will be the output of this program. Taking some considered value we go through the code and come to some conclusion. Now how we can we understand that our inferred result is correct or not. Or for for any possible inputs, the outputs can be predicted. Suppose If I have a code to make square of some Integer non negative number , we can say that for any non-negative integer value the output will be the square of the input number. So there are techniques called theorem prover that says how to prove a computer program.

By using Theorem prover we want to have a mathematical proof of an expression. This expression is called conjecture and stated in formal language. to prove such an expression we need to have Axioms and Inference rules.Axioms are conjecture which are considered to be true. Inference rules says that if I have fact 1, fact2..factn to be true then we can infer New fact from them.

For example

```
a * (b * c) = (a * b) * c : Axioms

a * e = b * e : Inference Rules

then we can infer that  a = b.  : New Fact
```

theorem prover works in the following way:

a) we have a sets of axioms A

b) Given a conjecture P.

c) To prove whether A $\models$ P ,i.e. prove A $\cup \sim P$ unsatisfiable

A and P are written in formal language that is First Order language. Now the derived formula is deduced by Inferring from the sequence of axioms. As Axioms are true and valid and if If the expression to prove can be derived by inferring from the sequence of axioms is also valid.

we can precisely formulate this as HOARE TRIPLES. where we have

S_o P S_final

Where S_0 is the sets of initial condition P can be an expression , function call or any block of code S_final gives the sets of the final result

In Hoar's calculus Axioms and Inference rules are used to derive S_final based on the S_0 and P. S_0 and S_final basically validates the program. If only P gets changed S_final has to change but S_0 can remain unchanged. For different S_0 initial condition for a particular P we will get different S_final. If S_0 kept same and P gets changed , we will get different S_final.

Application

Theorem prover was developed to help the Mathematicians. It can search equivalent theorem from a database of mathematical theorems. Moder application of theorem prover can be in the field of hardware and software verification. Automated theorem provers can be applied in other field also like network security.

1.4.2 Software Model Checking

Software Model Checking is the algorithmic analysis of a program to determine if the statement of the program are correctly designed to give the expected result. A computer program consists of some finite number of lines of code. Say if a program has n number of lines of code then it requires n

number of program counter locations. Program counter stores the address of the instructions to be executed next. Valuation of the variables is called the states of the program. Now a model of a program consists of the control flow of the program through the combination of states and program counter. Configuration transition graph describes how the state changes in course of the execution of the program. Each node gives a combination of state and program counter. The edge depicts the movement from one state to another. Configuration graph must show every state of the program. The model checking algorithms determines exhaustively the reachability of the states. If the graph has a finite number of configuration then it says that the program will terminate. consider the following code :

```
1: x := 3;

2:  while(x >= 0)

3:     if(x >= 1) then 4: x := -5;

                  else 5: x := 2;
6:
```

The configuration transition graph is depicted below

```
<s,1> --> <x = 3, 2> --> <x = 3, 3> --> <x = 3, 4>
--> <x = -5, 5> --> <X = -5, 6>
```

If the configuration graph reaches such a state where there is no instructions to execute then the program will terminate. If such a state is not reachable then there will be infinite state transition.

Model checking tool checks for such reachable state. If the tool checks that this snippet of code is unable to produce expected result then it gives a counter example to illustrate a situation when such circumstance can occur. model checking tool depicts an execution trace which says for which state configuration the program cannot produce expected result.

1.4.3 Bounded Model Checking

Background: Bounded Model checking was first proposed by Biere et al in 1994. It follows the principle of model checking but upto a depth say k. So it is called bounded model checking. It explores the reachable states within that depth k. If a bug is found within K depth BMC gives a counterexample

of the trace of the states. BMC cannot report if a bug could be present in greater than K depth. This depth value is called Completeness threshold beyond which BMC is unable to check.

Workings BMC unwinds the path k times and formulate Linear Temporal Logic for each path. It then checks whether there exists such a trace that contradicts them, gives the counterexample. Such a trace is called witness for the property.

We are considering here two path quantifiers E and A E denotes the expectation of correctness of the LTL formula over all the paths And A denotes the expectation of correctness of the LTL formula over some paths.

M $\models$ Af means , M satisfies f for all of the paths and M $\models$ Ef means there exist at least one path that satisfies f. The formula are presented in negated normal form(NNF).

The basic idea of bounded model checking is to consider only a finite length of the path, restricted to some bound k. Though the length of the path is finite, it can represent an infinite path, the back loop is present in the path from the last state to any of its previous state. If there is no such loop, then the path is finite within bound k.

We call a path π, a k-loop if there exists k >= l >= 0 , for which π is a (k,l) loop. Here, k loop signifies that, the semantics of model checking is defined under bounded traces. In the bounded semantics, we consider a finite length of a path. Precisely we only explore first k+1 states (S_0,..S_k)of a path to establish the validity of the LTL formula along the path.

Bounded Semantics for a loop

Let k > 0 and π be a k-loop. Then an LTL formula f is valid along the path π with bound k (in symbol $\pi \models \pi$k) iff $\pi \models$ f.

Now let us consider π is not a k-loop. Then the formula f := F_p is valid along the path π in the unbounded semantics if we can find an index i > 0, for which p is not valid along the suffix πi of π.

Bounded Semantics without a loop

Let k > 0, and let π be path that is not a k-loop. Then an LTL formula f is valid along π with bound k.

Reducing bounded model checking to SAT

In this section we will discuss how to reduce bounded model checking to propositional satisfiability. Precisely this technique illustrate how propositional SAT solver can be used efficiently to perform bounded model checking.

Let us consider, a kripke structure M, an LTL formula f and a bound k and want to construct a propositional formula $[[M, f]]_k$.

S_0,...S_k be a finite sequence of states on a path Π .

Each S_i represents a state at time stamp i and consists of an assignment of truth values from the set of state variables. The formula $[[M, f]]_k$ can be defined as the 3 separate formula.

The first component defines a propositional formula $[[M]]_k$, that contains S_0...S_k states to form a vital path starting from an initial state.

For a Kripke structure M, k >= 0.

$$[[M]]_k := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_i + 1) \tag{1.1}$$

The shape of the path Π plays a vital role as the translation of an LTL formula depends on the shape. The propositional formula is considered to be true if and only if a transition occurs from state s_k to state S_1.

the second component is a loop condition, that is propositional formula evaluates to true only if the path π contains a loop.

LOOP Condition

The Loop condition L_k is true if and only if there exists a back loop from state S_k to a previous state or to itself.

$$L_k := \bigvee_{l=0}^{k} {}_l L_k \tag{1.2}$$

The third component defines a propositional formula that constraints π to satisfy P.

1.4.4 CBMC

The bounded model checking Implementations fro C programs is CBMC developed at CMU. CBMC formally verifies ANSI-C programs and checked the properties of C including pointer safety, array bounds and user-provided assertions. As CBMC architecture is based on the BMC model,CBMC forms a transition relation for a complex state machine with its specifications are unwound to reduce a boolean formula which is then fed to a SAT solver. The SAT solver checks for the counter example where the specifications contradicts with the boolean formula implementations. If no such counter example obtained within the data type range bound then the verification is successful otherwise a counterexample showing the trace of states with violating specifications is depicted.

CBMC behaves like a compiler and takes .c files on the command line. Then like a linker merges the function definition from various .c files and performs symbolic simulation on the program. Let us consider the following C program, named Test1.c

```
#include <stdio.h>

int main()
{
  int a[4] = {1, 2, 3, 4, 5};

   printf("%d", a[3]);
```

```
}
```

Now we can run this program using CBMC in the following way.

cbmc Test1.c

CBMC transforms this design into CNF and passes it to a SAT solver. If SAT solver detects that the design is not valid then there is a bug in the program. Then it prints a counterexample that shows a trace where the property is violated. In the above code, cbmc will show that verification is successful and if it was not successful it would show us a counterexample.

Verifying Modules

Generally programs starts with main function but in embedded software, there are different entry points in the program.. Consider the following code

```
#include <stdio.h>

int main()
{
}

int product()
{
}
```

Here is a main functions and one sub function product. we want to set entry point to the product function, then we can write

cbmc Test2.c –function product

Loop Unwinding

CBMC unwinds the loop. As CBMC follows Bounded Model Checking architecture, all loops are unwinds upto certain depth, within which it guarantees that all bugs are found. Consider the following code, Prime.c :

```
int prime()
{
   int i = 2;
  while(1)
  {
     checkPrime(i);
      i = i + 1;
  }
}
```

CBMC cannot stop unwinding this design. the built-in simplifier cannot determine the upper bound to stop. But the unwinding bound can be given as the command line argument:

cbmc prime.c –function prime –unwind 8 –decide

Basic data types

CBMC supports the scalar data types including Bool. CBMC allows int 32 bits wide, short int 16 bit wides and char is 8 bits wide. Char is by default signed but using a command line options char an be changed to unsigned. CBMC also supports floating point data types like float, double and long double. CBMC uses fixed point representation for floating point type. By default float is 32 bits wide, double and long double are 64 bits wide.

Operators:

Bololean operators

Cbmc supports and, or, not boolean operators.

Integer Arithmetic Operators

CBMC supports all integer Arithmetic operators. Cbmc also checks Arithmetic overflow in case of signed operands and division by zero for division and remainder operators.

Floating Point Arithmetic operators

cbmc supports addition, subtraction, multiplication and relational operations on float, double and long double variables.

Type casts

CBMC supports Arithmetic type casts. For unsigned data types overflow is not checked but for signed data types the overflow condition is checked.

Control Flow Statement

CBMC supports the use of all conditional Statement as supported by ANSI-C standard. CBMC checks that a function having non-void return type must return a value according to the return type .

NON-Determinism

CBMC supports non-deterministic choice of functions and is defined them with the prefix nondet. For example,

```
int nondet_int();
```

This statements enable cbmc to return any non-deterministically taken int value. CBMC exhaustively checks all int value from this nondeterministic set.

Assumptions and Assertions

CBMC assumptions restricts the program traces that follows the assumptions. It takes boolean expression Consider the following example

```
int eval = nondet_int;

__CPROVER_assume(eval > 1 && eval <= 100);
```

In the course of execution only the value of eval will be between 1 to 100.

The assert statement also takes boolean Statement and CBMC checks whether this condition is true for each run of the designed code. If the condition is true for each runs then only the assertions can pass. Consider the following example:

```
int eval = nondet_int;

__CPROVER_assume(eval == 110);
assert(eval > 50);
```

The above assertion will pass but the following example

```
int eval = nondet_int;

__CPROVER_assume(eval == 10);
assert(eval > 50);
```

will fail as there is no such path that satisfies this property.

When using the assume statement, it is ensured that at least one program trace is supported by this assume conditions but when we use assert we want to establish that the property is true for each program trace.

Arrays

CBMC supports Arrays as defined by the ANSI-C standard including multidimensional and dynamically allocated array. CBMC checks for upper bound and lower bound of the array to warn the occurrence of overflow and underflow condition.

Apart from this cases CBMC many more features as shown in the following table.

OPTION	DESCRIPTION
– –show–parse–tree	show parse tree
– –show–symbol–table	show symbol table
– –show–goto–functions	show goto program
– –bounds–check	enable array bounds checks
– –div–by–zero–check	enable division by zero checks
– –pointer–check	enable pointer checks
– –memory–leak–check	enable memory leak checks
– –signed–overflow–check	enable arithmetic over– and underflow checks
– –unsigned–overflow–check	enable arithmetic over– and underflow checks
– –float–overflow–check	check floating-point for +/-Inf
– –nan–check	check floating–point for NaN
– –no–assertions	ignore user assertions
– –no–assumptions	ignore user assumptions
– –error–label label	check that label is unreachable
– –cover CC	create test-suite with coverage criterion CC
– –mm MM	memory consistency model for concurrent programs
– –function name	set main function name
– –property id	only check one specific property
– –program–only	only show program expression
– –show–loops	show the loops in the program
– –depth nr	limit search depth
– –unwind nr	unwind nr times
– –unwindset L:B,...	unwind loop L with a bound of B (use –show-loops to get the loop IDs)
– –show–vcc	show the verification conditions
– –slice–formula	remove assignments unrelated to property
– –unwinding–assertions	generate unwinding assertions
– –partial–loops	permit paths with partial loops

References

1. *ARM AMBA Specification Rev 2.0*, http://www.arm.com
2. Das, S. et al, Formal Methods for Analyzing the Completeness of an Assertion Suite against a High-Level Fault model, In VLSI Design, 2005.
3. Dasgupta, P., A Roadmap for Formal Property Verification, Springer 2006.
4. Lily, http://www.ist.tugraz.at/staff/jobstmann/lily/
5. Open Core Protocol, http://www.ocpip.org