Bootcamp Problems

Natarajan Shankar

May 2022

The goal of the Bootcamp is to put the knowledge gained during the summer school to use in solving a big verification problem. Randy Bryant has already prepared a challenge for verifying bit-twiddling operations using CBMC.

Here's a list of problems that should be solvable in five hours or less with teams of two or three students. Pick the tool of your choice. Some recommendations include:

1. Dafny (`https://github.com/dafny-lang/dafny`)
2. Viper (`https://www.pm.inf.ethz.ch/research/viper.html`)
3. Why3 (`https://why3.lri.fr/`)
4. CBMC (`https://www.cprover.org/cbmc/`)
5. Creusot (`https://github.com/xldenis/creusot`)
6. SeaHorn (`https://seahorn.github.io/`)
7. Frama-C (`https://frama-c.com/`)
8. Symbolic Pathfinder (`https://github.com/SymbolicPathFinder/jpf-symbc`)
9. Coq (`https://coq.inria.fr/`)
10. Isabelle/HOL (`https://isabelle.in.tum.de/overview.html`)
11. LEAN (`https://leanprover.github.io/`)
12. SAL/SALLY (`http://sal.csl.sri.com`, `https://sri-csl.github.io/sally/`)
13. PVS (`http://pvs.csl.sri.com`, get VSCode-PVS from VSCode Marketplace)
14. UCLID5 (`https://github.com/uclid-org/uclid`)
15. Scallop (`https://scallop-lang.github.io/ssft22/index.html`)
16. CVC5 (`https://cvc5.github.io/`)

17. Yices (`http://yices.csl.sri.com`)

18. Z3 (`https://github.com/Z3Prover/z3`)

You can be as abstract or concrete as you wish, but you get more points if you have efficiently executable code. Feel free to attempt more than one problem.

1 VERIFYING BIT TWIDDLING HACKS [Randal E. Bryant (Randy.Bryant@cs.cmu.edu)]

The CBMC project developed a verifier for C programs based on Bounded Model Checking. The idea is to symbolically simulate the program, using only bounded unrollings of loops and bounded recursions. The program converts C code into a low-level bit representation by "bit blasting." It then calls a SAT solver to check whether the desired property (typically specified as an assertion in the program) holds by showing that the Boolean formula encoding its negation is unsatisfiable.

CBMC can be obtained at: `https://www.cprover.org/cbmc/`.

Here's an example of a simple C program that verifies that a tricky way of implementing absolute value, matches a "reference version" i.e., a version that can serve as the specification:

```
int abs_ref(int x) {

    return x < 0 ? -x : x;

}

int abs_bits(int x) {

  int m = x>>31;

  return (x^m) + ~m + 1;
}

int main() {

    int t = rand();

    int ar = abs_ref(t);

    int ab = abs_bits(t);

    int err = ar != ab;
```

```
    assert(!err);
}
```

Here's an interesting application of CBMC: In the early 2000s, Sean Anderson, a Stanford PhD student, published a website with "bit twiddling hacks" It's still available at: `https://graphics.stanford.edu/~seander/bithacks.html`

This website gathered a collection of clever tricks people had devised over the years to implement standard operations using low-level bit manipulation tricks. Back in 2005, Bryant checked out the entire collection by translating Anderson's code into a series of functions, writing reference versions of them, and then using his own bit-blasting verifier based on BDDs. The following file contains both Anderson's code and the reference functions: `https://github.com/rebryant/unsat-tutorial/blob/main/project/bryant-bit-hack-2005.c`

It would be interesting to try verifying these functions with CBMC. There will be cases where SAT solvers work very well, but others where they scale exponentially. Also, it would be good to verify the functions that Anderson added after 2005.

2 Normal Forms

Define a representation for Boolean formulas constructed from Boolean variables using negation, implication, disjunction, and conjunction. Transform Boolean formulas to Conjunctive Normal Form. Prove that the transformation preserves satisfiability. See if you can do this as an inference system so that you go from a single formula to a set of clauses. For example, if the formula you are checking for satisfiability is say $(p \wedge q) \vee (\neg q \wedge r)$ becomes $u = v \vee w, w = p \wedge q, x = \neg q \wedge r$, which can be expanded as $\neg u \vee v \vee w, u \vee \neg v, u \vee \neg w, \neg w \vee p, \neg w \vee q, \neg p \vee \neg q \vee w, \neg x \vee \neg q, \neg x \vee r, q \vee \neg r \vee x$. Your code should be verified and executable.

3 Boolean constraint propagation with two-watched literals

BCP is a step in the CDCL SAT solving algorithm. Variables V are positive integers up to $2^{20} - 1$. The state consists of a partial assignment M mapping V to pairs of truth values and together with the decision level. For each variable, we also maintain two lists of watched clauses, i.e., clauses where a literal in the variable occurs as a watched literal. There is one list where the positive literal is watched, and another where the negative literal is watched. The goal is to propagate a new assignment to these watched literals to either:

1. Find a conflict clause if the watched literal is the only watched literal in the clause, i.e., the other watched literal has already been falsified.

2. Pick another unassigned literal in the clause to watch.

3. Propagate the other watched literal if there is no unwatched, unassigned literal in the clause.

Show that

1. BCP only introduces literals into the partial assignment that are implied by the existing assignments and the original set of clauses.

2. BCP does not change the set of satisfying assignments for the input set of clauses.

3. BCP ensures that the watched literals are both unassigned, or contains the only unassigned literal in the clause, if any.

4 Database Join

A database consists of a set of tables. Each table consists of rows that assign fields to values. Define a join operation and verify it. To keep things simple, assume that you have an array P of records with type $[\#a : uint32, b : uint32\#]$ and another array Q of type $[\#b : uint32, c : uint32\#]$, return an array PQ with element type $[\#a, b, c : uint32\#]$ containing all and only those elements $(\#a := x, b := y, c := z\#)$ where $(\#a := x, b := y\#)$ is in P and $(\#b := y, c := z\#)$ is in Q.

5 Shortest Path Algorithm

Dijkstra's shortest path algorithm considers a directed graph with non-negative edge-weights and computes the shortest path from a source vertex to all of the vertices. Write and verify Dijkstra's algorithm. To make things a little more explicit, you are given a graph with N vertices (numbered 0 to $N-1$) and an $N \times N$ edge matrix with non-negative (possibly ∞) weights), and a source vertex is s. You are to construct an array P of the smallest path weight from s to each vertex t. Each entry in P includes the index of a vertex, say t, and the smallest path weight from s to t. The P array is partitioned into a prefix with the dead vertices D and a suffix L of live vertices organized in a heap. An index i labels the location of the cursor separating D from L. As noted in the Speaking Logic slides (117 and 118), the shortest path from s to any t is given by a vector P such that $P(s) = 0$ and for any $v \neq s$, $P(v) = min_u(P(u) + W(u, v))$. For each vertex v in D, $P(v)$ is the shortest path. For each vertex v in L, $P(v)$ is the shortest path to v through an edge from D. For the minimal vertex $v \in L$, no path through an edge from a vertex u in L is going to be smaller than $P(v)$. Hence v can be moved to D and the weights of the remaining vertices in L can be updated to maintain the invariant.

6 Soundness of Separation Hoare Logic

The lectures from Prof. Ruzica Piskac contain the proof rules and semantics for Separation Logic. Formalize the semantics of separation logic and prove that the rules are sound. If you are using a proof assistant, you have the choice of capturing the state of the computation as a mapping S of variables to (integer and reference) values, and the heap (store) M as mapping references to arrays containing (integer or reference) values. A heap fragment is just a (possibly empty) subset of references in the domain of the heap. To model separating conjunction $P * Q$ over a heap fragment H, you need to partition H into H_1 and H_2 so that P is an assertion over heap fragment H_1 and Q is over heap fragment H_2. Recall that the assertions are of the form $acc(r)$, $P * Q$, $P \wedge Q$, and logical assertions (implicitly over empty heaps). The semantics is given by $S, M, H \models A$ for stack S, store M, heap fragment H, and assertion H. The Hoare rules specific to separation logic constructs are:

1. $\vdash \{acc(x)\}x[i] := y; \{x[i] \mapsto y\}$
2. $\vdash \{x[i] \mapsto z\}y := x[i]; \{x[i] \mapsto z * y == z\}$
3. $\vdash \{emp\}x := newT; \{acc(x)\}$
4. $\vdash \{acc(x)\}free(x); \{emp\}$

Prove these rules sound with respect to semantics. One tricky aspect of the formalization is the state. The easiest way to do this is to employ the de Bruijn representation so that each variable is numbered by its position from the top of the stack. The value type is a disjoint union between integers and references. The type of references is also just the natural numbers (just so we have an unbounded supply of references). Modeling the right-hand side expression of an assignment is another challenge. It is best to view this as some function of the state. The left-hand side of an assignment is either a variable or a dereference.

7 MaxSeg Sum

The Speaking Logic lectures contain a treatment of the maximum segment sum problem. Do your own definition of the algorithm and construct a proof for it.

8 IF-Normalization

IF-expressions can be defined by a datatype that has a constructor for IF-expressions with three accessors: condition, thenBranch, and elseBranch, and the truth values: True and False, and variables. An IF-expression is in normal form if the condition part of any IF-expression is always a variable, and there are no redundant IF-expressions of the form $IF(x, A, A)$. Show that any IF-expression can be converted into normal form. Write a simplifier for normal-form IF-expressions so that no variable is repeated along any branch.

9 BDD Construction

Write a function that implements the Apply operation on BDDs. We have a table of V variables that returns an ordering on the variables, i.e., an injection from V to $|V|$, and a BDD table T that maps each node N to $vars(N)$, $left(N)$ and $right(N)$. Show that any satisfying assignment for the equalities in the new table T' is a satisfying assignment for the original table and the result of the Apply operation. Recall that the Apply operation applies $\vee$ or $\wedge$ to two BDDs. To compute $M \odot N$, where $M = ITE(x, M_1, M_2)$ and $N = ITE(y, N_1, N_2)$ where $x > y$, we compute $MN = ITE(x, M_1 \odot N, M_2 \odot N)$. Otherwise, we just compute the $\odot$ operations on the constants 0 and 1.

10 Checker for RUP proofs

The LRAT proof format for Reverse Unit Propagating (RUP) proofs requires lemma clauses C that are obtained from the input clauses and prior lemma clauses by Reverse Unit Propagation. This requires negating the literals in C and propagating the conjunction of these literals through the given clauses in order to derive new literals until the final clause is falsified. A RUP proof consists of a sequence of clause additions and deletions (which we ignore). The final lemma should be the empty clause which follows. Prove that RUP inference is sound.

11 Checking Stratification

The problem of stratification came up in the lectures on Datalog. Negation and aggregation can only be applied to predicates in a prior stratum. For this to work, the dependencies between predicates have to be stratified. Given a dependency matrix, can you check that the graph is stratified. The dependency matrix shows whether a predicate depends positively or negatively on another predicate so that D_{ij} is 0 if the predicate i depends on predicate j positively, and 1 if it depends negatively, and -1, otherwise. Check that a dependency matrix is stratified. This could be done by generating inequalities $x_j + D_{ij} \leq x_j$, if $D_{ij} \geq 0$. Any solution for this system of inequalities would yield a stratification of the predicates so that no predicate depends on itself through one or more negations.

12 Simple Concurrency Problem

This problem comes from Andreas Podelski (who got it from a student). You have a fixed but unspecified number of threads `numthreads`. Each thread executes the following steps in an unbounded loop:

```
while * do
```

```
if  global + numthreads > maxint
  then skip
  else local = global; global++;
assert global > local
```

Check if the above program has a race condition. Fix the problems if any by adding atomicity annotations. Prove that the assertion is maintained and that never triggers an arithmetic overflow. Be careful, this problem might not be as simple as it appears.

13 Brzozowski Derivatives

Look at the Generalized Regular Expression (GRE) formalism described in `https://en.wikipedia.org/wiki/Brzozowski_derivative`. Define the syntax and semantics of GREs. Define a function that takes the derivative $\sigma^{-1}R$ of a GRE R relative to a token. Define the iterated version that operates over a string, and show that a string ρ is accepted by a regular expression exactly when $\varepsilon \in \rho^{-1}R$.

14 Two-Process Bakery Algorithm using Predicate Abstraction

The algorithm consists of two processes P and Q with control variables *pcp* and *pcq*, respectively, and shared variables x and y. The control states of these processes are either `sleeping`, `trying`, or `critical`. Initially, *pcp* and *pcq* are both set to `sleeping` and the control variables satisfy $x = y = 0$. The transitions for P are

$$\begin{array}{rrcl} & \texttt{pcp} = \mathit{sleeping} & \longrightarrow & x' = y + 1;\ \mathit{pcp}' = \texttt{trying} \\ [] & \mathit{pcp} = \texttt{trying} \wedge (y = 0 \vee x < y) & \longrightarrow & \mathit{pcp}' = \texttt{critical} \\ [] & \mathit{pcp} = \texttt{critical} & \longrightarrow & x' = 0;\ \texttt{pcp}' = \texttt{sleeping} \end{array}$$

Similarly, the transitions for Q are

$$\begin{array}{rrcl} & \mathit{pcq} = \texttt{sleeping} & \longrightarrow & y' = x + 1;\ \mathit{pcq}' = \texttt{trying} \\ [] & \mathit{pcq} = \texttt{trying} \wedge (x = 0 \vee y \leq x) & \longrightarrow & \mathit{pc}' = \texttt{critical} \\ [] & \mathit{pcq} = \texttt{critical} & \longrightarrow & y' = 0;\ \mathit{pcq}' = \texttt{sleeping} \end{array}$$

Establish for $P \,[]\, Q$, the interleaving composition of P and Q, the invariant $\neg(\mathit{pcp} = \texttt{critical} \wedge \mathit{pcq} = \texttt{critical}$.

15 Interval Analysis by Abstract Interpretation

The interval domain consists of intervals of the form $[l, u]$, where l and u range over the extended integers. Given a program that operates on integer variables

x, y, etc., your analysis should return a sound approximation of the interval ranges for each variable at each program point. You can assume that you have a program defined on the integer variables:

1. Using addition and subtraction
2. **assume** b, where condition b is an equality or inequality comparisons between variables or between variables and constants
3. Iteration S^*, where the exit condition is an equality/inequality comparison
4. Choice $S_1[]S_2$
5. Sequencing $S_1; S_2$

- Construct a transfer function from intervals to intervals for the operators $+$ and $-$.
- Construct a transfer function for statements from an abstract state to an abstract state.
- Define a widening operation ∇, where $I_1 \nabla I_2$ returns an interval such that $I_1 \sqsubseteq I$ and $I_2 \sqsubseteq I$ so that there are no infinite increasing chains of widenings.
- Implement an abstract interpreter.
- Bonus: Prove that the abstract intepreter is sound.