Inferring Annotations: Presentation Notes

Ana Nora Evans

February 6, 2017

These notes are prepared for the Software Engineering Research Group meeting discussing the ASE 2016 paper, *Inferring annotations for device drivers from verification histories* [1].

1 PL Background

1.1 Axiomatic Semantics

The theory commonly used to reason about program correctness is "Axiomatic Semantics". It consists of a language for stating assertions about programs, usually first order predicate logic involving program variables, and rules for establishing the truth of assertions.

The notation used is: $[A]c[B]$. A is a logical predicate and it is called **precondition**. B is also a predicate and it is called **postcondition**. c is a command or instruction.

The triple $[A]c[B]$ is called a Hoare triple. Its meaning is: "If the the predicate A is true, then the command c terminates and the predicate B is true after its execution".

For example, $[x < y]x := x + 1; [x \leq y]$.

How hard is it to prove program termination?

We may be able to prove termination for some toy programs, and if we work very hard even for some programs with loops, but this is an undecidable problem.

Hence, we downgrade our goals a little bit. We remove the termination proof burden by moving the termination condition in the assumptions. We introduce a new notation $\{A\}c\{B\}$, which means that "If the predicate A is true and the command c terminates, then the predicate B is true after c terminates. If the command c does not terminate the assertion is valid.".

What kind of assertions can we make?

- $x > 0$
- this program is memory safe
- the array accesses are within the array bounds
- this program is deadlock free
- this program is race free
- this program does not leak memory

For more interesting properties the language of assertions would use other logics like separation logic or temporal logics.

These are all things we can say about a program, but that means nothing if we can't actually prove them. We need a set of rules for establishing the truth of assertions:

- usual rules from first order logic like

$$\frac{A \qquad B}{A \wedge B}$$

- rule of consequence

$$\frac{A' \Rightarrow A \qquad \{A\}c\{B\} \qquad B \Rightarrow B'}{\{A'\}c\{B'\}}$$

- one rule for each command

$$\frac{\{A\}c_1\{B\} \qquad \{B\}c_2\{C\}}{\{A\}c_1; c_2\{C\}}$$

$$\frac{\{A \wedge b\}c_1\{B\} \qquad \{A \wedge \neg b\}c_2\{B\}}{\{A\}if\ b\ then\ c_1\ else\ c_2\{B\}}$$

$$\frac{A \wedge b \Rightarrow C \qquad \{C\}c\{A\} \qquad A \wedge \neg b \Rightarrow B}{\{A\}while\ b\ do\ c\{B\}}$$

  C is the **loop invariant**.

Here is an example:

$$\{x \leq 0\}while\ x \leq 5\ do\ x := x + 1\{x = 6\}$$

with invariant $x \leq 6$.

Trick: Prove first

$$\{x \leq 6\} while\ x \leq 5\ do\ x := x + 1 \{x = 6\}$$

and finish with consequence rule.

Axiomatic semantics is:

- **sound**: we can only prove true predicates
- **complete**: we can prove all the true predicates (only if the underlying logic is complete).

Where are the difficulties?

1. Application of the rule of consequence
2. Proving the implications

   With modern SMT solvers, the two above are possible to solve in most cases. Cook's theorem proves that boolean satisfiability if NP-complete, but, luckily, the hard instances are not dense in the input space.
3. loop invariants
4. preconditions and postconditions for functions

   Solution for the last two: let's call them **annotations** and make the programmers write them.

This is not impossible. In the world of formal methods and verification, one uses proof assistants like Coq, Isabelle, PVS, to combine code and proofs about code. Notably, Xavier Leroy and his team implemented a verified C compiler [2] in Coq.

1.2 Static Program Checkers

In the world of C and Java, pointers, inheritance, legacy code and libraries complicate the analysis. Usually, the tools are neither sound nor complete.That means they will allow buggy programs and will give spurious warnings. Tools like ESC/-Java [3] are not really verifiers, in the sense that they prove the analyzed program is correct, but they will find some categories of programming errors.

For example, ESC, can give static warnings about null deferences, array bounds errors, type cast errors, race conditions, and deadlocks. These are all undecidable problems in general, but the kinds of programs that occur in practice are not the one from the undecidability proofs.

Here is an example (adapted from [4]) of an annotated Java class implementing linear search. The method f takes a parameter the element it's searched for and returns an index where it is found.

```
public abstract class LinearSearch
{
    /** The function that describes what is being sought. */
    //@ requires j >= 0;
    public abstract /*@ pure @*/ boolean f(int j);

    /** The last integer in the search space, this describes the
     * domain of f, which goes from 0 to the result.
     */
    //@ ensures 0 <= \result;
    //@ ensures (\exists int j; 0 <= j && j <= \result; f(j));
    public abstract /*@ pure @*/ int limit();

    /** Find a solution to the searching problem. */
    /*@ public normal_behavior
      @   requires (\exists int i; 0 <= i && i <= limit(); f(i));
      @   ensures \result == (\min int i; 0 <= i && f(i); i);
      @*/
    public int find() {
        int x = 0;
        //@ maintaining 0 <= x &&
        //@ (\forall int i; 0 <= i && i < x; !f(i));
        while (!f(x)) {
            /*@ assert 0 <= x && !f(x)
              @    && (\forall int i; 0 <= i && i < x; !f(i));
              @*/
            x = x + 1;
        }
        /*@ assert 0 <= x && f(x)
          @     && (\forall int i; 0 <= i && i < x; !f(i));
          @*/
        //@ hence_by (* definition of \min *);
        //@ assert x == (\min int i; 0 <= i && f(i); i);
        return x;
    }
}
```

requires is the notation for the precondition and *ensures* for the postcondition. This is the loop invariant:

```
/*@ assert 0 <= x && !f(x) &&
(\forall int i; 0 <= i && i < x; !f(i)); @*/
```

This is great, but how many times does it happen to write functions that do not call other functions? What does ESC/Java do then?

ESC/Java performs modular checking. It assumes that the called functions are annotated with preconditions and postconditions and the implementation satisfies them. This is called **assume-guarantee** reasoning.

There may be some hope one can make every programmer involved in the project to write annotations, but there is not much to do about legacy code and external libraries.

Why not run ESC/Java on unannotated programs? It produces an enormous number of false alarms.

1.3 Houdini

It would be very useful if we could automatically generate invariants. The invariants are necessary for legacy code and they are very hard to write.

One of the early annotation assistants is Houdini [5]. The key insight use of the ESC/Java as an oracle. Houdini generates a set of candidate annotations and inserts them in the input program. Then it runs the ESC/Java and removes the "bad invariants", the ones for which the checker found counterexamples, until quiescence.

The interesting part is candidate annotation generation. If the initial set is too small, the tool fails to find interesting bugs or remove enough warnings, and if it's too big, it's too slow to be practical. The authors used their expertise and the inspection of a variety of hand-annotated programs to develop the following heuristics:

1. if f is a field of type
   (a) integral $f\ cmp\ exp$ where cmp is one of $<, \leq, ==, \neq, \geq, >$ and $expr$ is an integral field or an interesting constant.
   (b) reference $f \neq null$
   (c) array $f \neq null$, $notnullelements(f)$, $\forall 0 \leq i < expr \Rightarrow f[i] \neq null$, $f.length\ cmp\ expr$
   (d) boolean $f == true$, $f == false$
2. if f is a method then preconditions and postconditions are generated similarly as for fields. Candidate preconditions may include expressions involv-

ing formal parameters and fields. Candidate postconditions may relate the special variable $result$ to either formal parameters or fields.

Note that inconsistent guesses will not cause problems, as ESC/Java will refute the incorrect ones.

How well does this work in practice?

How many annotation did you write in your last Java implementation?

1.4 Reality

The last questions were a bit sarcastic. The reality is that, in practice, in certain cases it works very well. The SLAM project [6], which later became the Static Driver Verifier [7] is the reason the "blue screen of death" does not make so much sense to you as it did to programmers in the late nineties.

Microsoft programmers realized that the vast majority of the Windows crashes were due to device drivers. The Static Driver Verifier (SDV) [8] systematically analyzes the source code of Windows device drivers against a *set of rules* that define what it means for a device driver to properly interact with the Windows operating system kernel. The rules are generated from a manually-tunned set of heuristics [1].

Why do you think it works in this case?

2 The Paper: Inferring Annotations For Device Drivers From Verification Histories

This paper takes the ideas from the Houdini [5] paper a step further. Instead of relying on humans to come up with a set of templates it learns the set of templates to be tried by the Houdini algorithm from prior verification runs.

2.1 Key Insight

Programs that use the same API probably require similar annotations for verifying contracts of that API. By using information from prior verification runs (i.e. invariants already checked) of the tool on potentially different sources, annotations will be generated and checked.

This approach needs a large repository of annotated sources.

2.2 Key Challenges

Invariants and annotations are formulas over program variables, using local variables that will be out of scope on a different location.

The size of inferred annotations must be small. In the current architecture, the annotations are passed to Houdini [5] to generate invariants for the program verifier, Corral [9]. Houdini's performance is not acceptable on very large inputs, but its is very fast on small and medium ones.

The lower bound on how small the annotation set can be is set by the goal to be able to generate all the invariants from the previous runs.

2.3 Example

Recall, an annotation is a candidate formula for an invariant. An invariant is a formula that is always true at a point in the program.

A postcondition ϕ of a function f is denoted by $[\phi]@f$. $[\phi]@f@L$ denotes ϕ is an invariant for the loop labeled L in function f. The predicate old gives the previous value of the variable.

```
var depth: int;

procedure init() {
        depth := 0;
}
procedure Acquire() {
        depth := depth + 1;
}
procedure Release() {
        depth := depth - 1;
}
procedure d_exit() {
        assert depth == 0;
}
procedure P_n()
{ call init(); call dispatchP_n(); }

procedure dispatchP_n() {
        [call Acquire()]^n;
L1: while(*)
        { call Acquire(); call Release(); }
        [call Release()]^n; call d_exit();
}
```

The predicates are:

$$\psi_i \equiv depth - old(depth)$$

$$\eta \equiv (opld(depth) == 0 \Rightarrow ok)$$

Possible proof of P_n:

$$[depth == 0]@init, [\psi_1]@Acquire, [\psi_{-1}]@Release, [\eta]@d\_exit$$

$$[\psi_0]@dispatchP_n@L1, [\eta]@dispatchP_n$$

Houdini can reconstruct this proof using the annotations: $depth == 0, \psi_{-1}, \psi_1, \psi_0, \eta$. Notice that the function names are not present anymore.

The algorithm collects the annotations from all the proofs in the repository. Next it applies an abstraction function to the annotations. Using a minimization algorithm, it creates a minimal set of abstract annotations.

2.4 The Abstraction Function

The set of global variables common to all the programs in the repository is called the *shared vocabulary*. It is assumed that these variables will serve a similar role in all programs.

The program specific variables are mapped to a generic variable of one of the types LOCAL, GLOBAL, FORMALIN, FORMALOUT.

2.5 The Minimization Algorithm

The minimization algorithm uses a cost metric which sets the cost to ∞ if the abstract annotation set can not prove the set of programs, and it reflects the running time of the verifier otherwise.

When an abstract annotation does not increase the running time of the verifier or it increases it by a tolerable amount, it is dropped from the set.

If an annotation can not be dropped from the set, the algorithm will try to simplify it, perhaps breaking it into the conjunctive components.

2.6 Evaluation

The training is done on a dedicated test suite containing 304 drivers. The testing is done on 66 real. From the hundreds of rules in SDV, the evaluation is done on 14 rules chosen rules, and 14 random.

3 Names and Projects to Remember

1. SLAM[6]: Thomas Ball, Sriram Rajamani
2. ESC/Java [3] and Houdini[5]: Cormac Flanagan and Rustan Leino
3. Microsoft Static Driver Verifier [7], Corral [9]: Thomas Ball, Akash Lal.

References

[1] Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. Inferring annotations for device drivers from verification histories. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 450–460, New York, NY, USA, 2016. ACM.

[2] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, New York, NY, USA, 2006. ACM.

[3] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[4] Jml linear search example.

[5] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[6] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, January 2002.

[7] Thomas Ball, Ella Bounimova, Vladimir Levin, Rahul Kumar, and Jakob Lichtenberg. The static driver verifier research platform. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 119–122, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Thomas Ball, Byron Cook, Vladimir Levin, Sriram Rajamani, and Tom Ball. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. Technical report, January 2004.

[9] Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 202–212, New York, NY, USA, 2014. ACM.