Crowdsolving Program Verification

Alex Kantchelian Rohit Sinha Nishant Totla

***Abstract*—In this effort, we consider the verification of properties in C (subset) programs. That is, we prove the validity of a pre/postcondition pair for a program, or demonstrate invalidity via an error trace. This is undecidable in general, and modern static analysis techniques struggle to reason about non-linear programs and programs with loops. To that end, we use abstraction for defining results of looping and non-linear computations. Furthermore, existing techniques do not take advantage of programmer's insight, since the programmer receives little feedback from failed proof attempts. To that end, we ask user for facts that are relevant to the proof. Our technique is sound, modulo the correctness of verification tools we rely on. As a first step, existing static analysis and invariant generation techniques are deployed on the program. We try to prove the postcondition using these facts. If that fails, we compute an abstraction of the original program, and query user for facts about the abstract program. We check the validity of each user fact given the set of known proven facts. We again try to prove the postcondition using this new fact. This process iterates until we either discharge the proof, or demonstrate a bug. In summary, we hope to involve a crowd of semi-expert users in solving a verification problem. With game mechanics and proper incentive mechanisms in place, crowdsolving can help scale program verification.**

I. Introduction

A particular challenge for formal verification of programs is reasoning about non-linear behaviour and looping computations. Non-linear operators such as multiply have exponentially large encoding in propositional logic. This makes solving especially difficult for SAT/SMT solvers. Fortunately, proving most properties do not require the entire formula describing a multiplier circuit. To that end, we use term-level abstraction in this effort to describe the results of non-linear operations such as multiply. Term-level modeling abstracts away the details of data representations, and models operations (such as multiply) as uninterpreted functions. The logicial encoding of an abstract program is significantly smaller than the original program, and is thus easier for solvers to reason about.

Similarly, loops present a problem for automatic program verification. One technique for reasoning about loops is to unroll them, until one proves there cannot be another iteration. However, this approach does not scale to programs where loops have many iterations. To combat this issue, program verifiers make use of loop invariants to discharge the proof. A loop invariant is a predicate that holds on entry into the loop, and is preserved in each iteration of the loop. The theorem prover can make use of loop invariants to prove the postconditions, without reasoning about each iteration of the loop. However, generating loop invariants is difficult for most class of programs, especially loops containing non-linear operations. To that end, we use abstraction to model program state after the loop. This allows us focus on predicates that are relevant to proving the final property.

Another drawback of current verification methodologies is that they do not leverage human insight. Humans, especially the programmer, has insight about the program that is not necessarily encoded in the program annotations or assumptions. We seek to leverage this insight in the verification process, by adding human in the loop. That is, we ask user for program facts that are relevant to the final property. First, we run existing static analysis and invariant generation techniques to accumulate program facts. We try to prove the postcondition using these facts. If that fails, we compute an abstraction of the original program, and query user for facts about the abstract program. We check the validity of each user fact given the set of known proven facts. We again try to prove the postcondition using this new fact. This process iterates until we either discharge the proof, or demonstrate a bug. Our technique is sound, but not complete.

II. Problem Statement

Our goal is to prove the validity of a pre/post condition pair, or prove invalidity via a buggy trace. The program P is expressed in an IMP like language with the following constructs:

- x := e
- assume p
- assert p
- if b do s1 else s2
- while b do s

Note that the rhs e of an assignment x := e is an expression using arithmetic operators and program variables.

Statements s1, s2, s are assignments. The argument to assume and assert is a predicate p expressed as a set of difference constraints.

We are given property φ as set of boolean formulas over difference constraints. Our goal is to prove that P satisfies φ. In addition, we want a sound methodology.

III. Background and tools

A. Sastifiability modulo theories and solvers

Satisfiability modulo theories (SMT) is a type of constraint satisfaction problem which generalizes the boolean satisfiability problem SAT.

Where SAT exclusively deals with boolean formulas, SMT extends expressiveness to general first order logic. As such, an SMT formula not only uses logical connectives and variables, but quantifiers, function and predicate symbols as well. The symbols' interpretation is defined by the appropriate set of background theories. For example, theories that are handled well by current solvers are, among others, the theories of equality with uninterpreted functions, of linear arithmetic over integers (mathematical or bit-vectors), arrays and more generally inductive data types. Unlike SAT, the quantification and the richness of the theories make SMT solving an undecidable problem in general. In practice, SMT solvers rely on the shallowness of their inputs via simple or more involved heuristics to efficiently come up with a model (e.g. a set of satisfying assignments to the variables) for the input or deem it unsatisfiable. In this work, we rely on SMT solvers to provide us with counter-examples in failed proof attempts. We make heavy use of the z3 solver [1], as well as Alt-Ergo [2], Yices [3] and CVC3 [4] as back-ends for the deductive program proving part.

B. Deductive program verification

Deductive program verification is a class of formal methods that aim at proving safety properties as well as behavioral correctness of code. Essentially, this approach relies on Hoare logic [5], a semantic axiomatization of program behavior. The vanilla Hoare formal system is defined by 2 axiom schemes, the skip/empty statement and the assignment statement, and 4 rules defining the behavior of composition (instruction successions), conditionals, loops, and consequence (allowing for weakening of pre-condition and strengthening of post-condition).

Using Hoare logic, one can turn a program and a property to be verified into a set of quantified logical propositions to be proved in some formal system, the so called verification conditions. These verification conditions can be either automatically proved by an SMT solver for example, or can be given to a proof assistant for manual proving when automated tools fail.

While it is possible to develop a Hoare-like logic for real-life programming languages like C, the resulting complexity of this approach have lead researchers to design small languages dedicated to program verification, along with compilers to it. The Boogie [6] and Why [7] tools are two examples of this approach.

In this work, we use the Frama-C [8] static code analysis platform in conjunction with Why to try automatically proving human-generated invariants. In particular, we first compile down the code along with all the proved invariants and the single unknown invariant to the Why language. Invariants are written in ACSL [9], a rich first-order logic specification language. From there, the tool will generate the proof obligations and delegate these to the SMT solvers. For example, a loop invariant will generate two proof obligations: one to ensure the invariant initially holds, and one to ensure it is inductive.

C. Bounded model checking

In bounded model checking, and more specifically in this work, our aim is to disprove that a certain user-supplied invariant holds. This is done by unrolling all the program loops k times for a fixed k, transforming the resulting code into static single assignment form, and converting it to a quantifier-free SMT format. We then add the negation of the user-supplied possibly quantified proposition to the SMT and feed the resulting formula to the solver. If there exists a model, we know the user supplied fact is not an invariant. If the solver claims unsatisfiability, we have no guarantee on the correctness of the invariant. BMC is fundamentally an unsound technique.

D. Automated loop invariant generation

Automated loop invariant generation is still an ongoing research effort. We are mainly interested in using off-the-shelf invariant generation tools for populating the program with the maximum number of loop invariants. This will both help the deductive theorem proving and spare some human effort.

All of the published loop invariant generation tools we found are based on execution traces. Starting from a set of program executions, the tools use various heuristics to come up with likely arithmetic invariants that hold on the loop variables.

Unfortunately, we were unable to run any of the published tools due to generalized obsolescence. The Daikon tool [10] ships with a modified version of an old Valgrind source which would not compile for Linux

kernels above 2.6[1]. InvGen [11] is only available in binary format, which unfortunately segfaults instantaneously under Linux.

IV. Methodology

We consider programs that have an `assert` statement, which contains the property that needs to be proved. Given program P, let this property be φ. We begin the analysis by running our program through a static analyzer, which allows us to infer some primary set of facts[2] $\mathcal{I}$ about P. To prove the correctness P, we must prove that $\mathcal{I} \models \varphi$. Often, $\mathcal{I}$ is not strong enough to discharge this proof, which is why we need to infer more facts about P. This can be done using Abductive Inference [12], to generate questions that humans can answer about the program. The method of [12] is unsound, since it trusts human input. We wish to use the power of the crowd to infer assertions and invariants about the program. At the same time, we want to guarantee soundness. Our approach can be outlined as follows:

1) Run static analysis to compute program facts and infer any loop invariants
2) Construct SMT encoding of the C program, and try to discharge the proof
3) If this proof fails, query humans for more program facts
4) Eventually, the program is verified or demonstrates a buggy execution trace

Figure IV shows an overview of the procedure that we adopt. $A(P)$ includes the original program facts $\mathcal{I}$ along with the SMT encoding of P (abstracted version). This is inferred at the beginning of the program, and is part of the preprocessing. At any stage, we have a set $U = \{u_1, u_2, \cdots, u_n\}$ of new facts about the program (that have been proven until this point). We try to prove φ by using an SMT solver to check the satisfiability of $\Phi = A(P) \wedge U \wedge \neg\varphi$. If Φ is unsatisfiable, we have proved the property φ, and the program is correct. If Φ is satisfiable, we get a counterexample (model of Φ). We check if it is spurious by executing the program and comparing the counterexample trace against the values in the real execution. If counterexample is valid, we have found a real bug. If spurious, it means some facts are missing from U and U needs to be strengthened. In this case, we ask humans to supply us with a new fact u_{n+1}. Alternatively, the human could also provide an inductive invariant for some loop of the program.

[1]The current kernel is 3.5

[2]Such as relationships between variables, loop invariants etc.

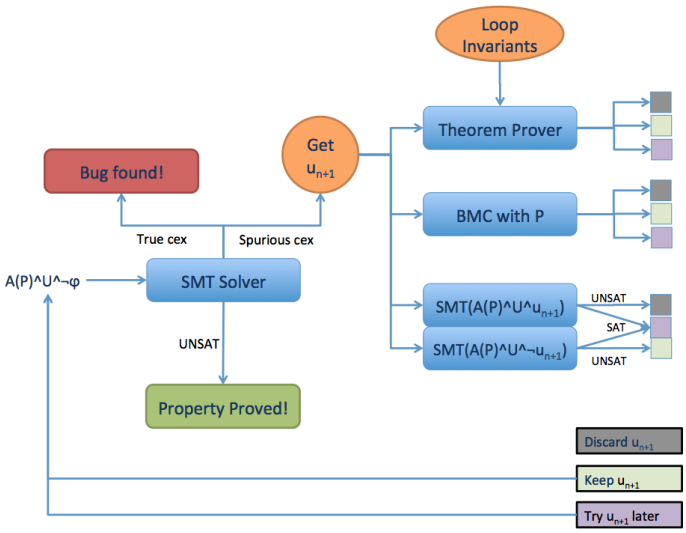

Fig. 1. Verification Flow. Orange denotes user input, blue denotes invocations of automatic solvers.

When we receive a property from a human, we have one of the three options:

1) Perform a simple check. If $A(P) \wedge U \wedge u_{n+1}$ is unsatisfiable, we discard u_{n+1}. Alternatively, if $A(P) \wedge U \wedge \neg u_{n+1}$ is unsatisfiable, we add u_{n+1} to U.
2) Input $U \cup \{u_{n+1}\}$ to a theorem prover which uses the loop invariants, tries to prove validity of new invariants, and eventually the validity of u_{n+1} subject to U and these invariants. We add u_{n+1} to U if the theorem prover is able to prove its validity, and discard u_{n+1} if the theorem prover gives a counter example.
3) Input $U \cup \{u_{n+1}\}$ into a Bounded Model Checker, that tests it against program P. Bounded Model Checking might provide a counter example, in which case, we discard u_{n+1}.

Note that all three techniques are independent, and one of them might work. In case none of them can validate u_{n+1} to be a valid fact about the program, we keep u_{n+1} around and try to check its validity at a later stage.

Furthermore, we may also receive a loop invariant from a human. We can use a theorem prover to check if that loop satisfies the invariant. This constitutes 2 checks:

1) the invariant holds on entry to the loop
2) each iteration of the loop inductively preserves the invariant

We now describe our procedure in greater detail, and demonstrate it with the help of a running example.

A. Static Analysis

Various static analysis tools exist, that can reason about code and infer properties. A major issue with static analysis is that of scaling. Yet, it can certainly help us automatically infer some properties about the program. These properties can then be used as initial facts to guide the generation of new facts.

B. Basic blocks

Consider the simple C program in listing 1, originally from [12]. We want to check that $\{k+i+j > 2n\}$ holds at the end of the program.

```
unsigned int n;
char flag;
int k = 1;

if(flag) k = n*n;

int i = 0;
int j = 0;
while(i <= n){
      i = i + 1;
      j = i + j;
}
//@ assert k+j+i>2*n;
```

Listing 1. Example toy problem from [12]. `n` and `flag` are inputs to the program and we wish to prove the property $\{k+i+j > 2n\}$.

The first step is to recognize basic blocks within the program. We identify straight-line code as single basic blocks, with the exception that we treat the entire loop as 1 basic block. The idea is to reason about basic blocks as a whole, and not split their internal structure. For instance, we will want to reason about functions in terms of their pre and post conditions, and loops in terms of invariants or the variables that the loops modify.

C. Separating paths in the CFG

Conditional statements introduce branches in the control flow graph (CFG) of the program. In order to prevent disjunctive program facts (which simplifies analysis), we consider each such path separately, and try to prove the property for each such branch independently. This might create a large number of splits, but that is not an issue since we are crowdsourcing the problem. Additionally, it becomes easier for humans to reason about programs if there are no conditionals, and programs are simply sequential. For the running example, which has a single conditional statement, we split the program into two instances, and prove the correctness of each instance separately.

The two branches above should both satisfy the desired property.

```
unsigned int n;
char flag;
int k = 1;

//@ assume flag==0;

int i = 0;
int j = 0;
while(i <= n){
      i = i + 1;
      j = i + j;
}
//@ assert k+j+i>2*n;
```

Listing 2. Path 1

D. Converting program paths to SMT

In the beginning, we abstract away all complex (non-linear etc.) operations, and introduce abstract variables instead. For example, if a program has the update $y = 2^x$, we just replace y with the updated instance y_i, and do not focus on the exact operation. More facts might be added about this operation, if needed later. In a similar way, we abstract loops away as well. This is done by introducing more abstract variables. For instance, if a loop updates variable z, we introduce a variable α_z to represent the value of z after the loop. Initially, α_z is unconstrained, since the loop has been ignored. If a loop does not make a difference to the property φ, more constraints on α_z may not be needed. If the loop does make a difference, more facts should be discovered about α_z later.

Once abstraction is done, we now have simple sequential programs, that can easily be encoded into an SMT formula. We demonstrate how this works for the first path of our running example, shown in Listing 2. Let α_i and α_j denote the values of i, j respectively, after the loop terminates. Using notation from Figure IV, suppose the static analyzer gave the following set of properties upon analyzing the code: $A(P) = (\alpha_i \geq 0) \wedge (\alpha_i > n) \wedge (n \geq 0)$. Now, the relationships between variables are clear, and we construct the SMT formula with the main property negated. Listing 3 shows the aforementioned SMT formula.

We test this on the Z3 SMT solver, and it returns a model, i.e. a counterexample.

```
(assert (= k_0 1))
(assert (= k_1 1))

(assert (= i_0 0))
(assert (= j_0 0))

(assert (>= alpha_i 0))
(assert (> alpha_i n_0))
(assert (>= n_0 0))

(assert (= z_0 (+ k_1
      (+ alpha_i alpha_j))))

(assert (<= z_0 (* 2 n_0)))
```

Listing 3. SMT formula for Path 1. We omitted the initial variables declarations for sake of brevity.

E. Analyzing the counterexample

The first counterexample assigns the following values: $n = 0, \alpha_i = 1, \alpha_j = -2$. We wish to find out if this counterexample is spurious or not. To do that, we simply run the program with initial value $n = 0$, and check if any variable in the real execution mismatches with the value assigned to it by the model. As we execute, we notice that the value of α_j should actually have been 1, but the model assigned it -2. Hence this counterexample is spurious, and we must ask the humans for more facts.

F. Abstraction Refinement

As a heuristic, we track the first variable whose value in the real execution does not match its value in the counterexample. This means that more facts about this variable might be needed to reason more effectively about the program.

G. Asking the crowd

Based on the counterexample, we ask the human user to supply a fact about α_j. Suppose the human user entered an additional fact that $\alpha_j \geq 0$, and one of the three techniques of validating new facts shows that this new assertion is indeed true. In that case, we add the fact to the SMT encoding of the program, and check satisfiability. In this example, we get a model again, which makes the following assignments: $n = 2, \alpha_i = 3, \alpha_j = 0$. Running with initial value $n = 2$ again establishes that the value of α_j is amiss. Like before, we ask for more facts from the crowd. If some smart human adds the fact $\alpha_j \geq \alpha_i$ and we are able to establish its validity, then it turns out that the SMT solver is able to prove the final property with this one extra fact.

Proof obligations	Alt-Ergo 0.94	Z3 4.3.1 (SS)	Yices 1.0.36 (SS)	CVC3 2.4.1 (SS)
Function f default behavior	✗			
1. postcondition	✂	✂	✓	✓
2. postcondition	✓	✓	✓	✓
3. postcondition	?	✂	?	✂

Fig. 2. Why output for initial run on f. The first two paths are cleared by Yices and CVC3. No solver can prove the result for the last path. A pair of scissors signifies a timeout (10 seconds), the question mark is an explicit "I give up" signal from the solver.

Notice that it is certainly possible for the solver to get stuck at a point, and there might be no way to progress until the human comes up with a brand new property. In the current version of our idea, humans must come up with the property themselves, but we eventually hope to be able to guide this process by asking specific queries about target variables.

H. Code splitting

An important idea for handling large programs (not demonstrated in this paper) is the splitting of code into convenient blocks that can be easily analyzed. The idea is to introduce abstract variables that propagate values between these splits of code. Additionally, reasoning should be easier for humans if the chunks of code they analyze are small.

V. Case Study

We report here on a representative case study, problem 3 from Dillig benchmark suite [12]. The ACSL contract header for function f reads as, for any integers a, b, f returns a positive integer. This is the property we are interested in proving. Listing 4 shows the code in question.

We begin by explicitely splitting the code into the 3 cases implied by the two `if` statements. Luckily, in the first two cases (namely paths corresponding to `if` and `if-if`, both Yices and CVC3 automatically prove the property (Alt-Ergo and Z3 fail). Figure 2 shows the state of the affairs at this point.

The remaining code we have to prove is presented in listing 5.

We start by abstracting this code to an SMT formula where the while loop assignments are abstracted away. The resulting formula is shown in listing 6.

The solver immediately outputs a counter-example which features a negative value of the abstract variable $\alpha_p = -2$ after the loop. This is automatically deemed

```
/*@ ensures \result > 0;
*/
int f(int a, int b){
      int r = -1;
      int c = 2*a + (a+1)%2;

      if(c == 0){
            int y = (b & ~(1 << 31) );
            int z = (y | (1 << 4));
            return z;
      } else {
            int c;
            if(b <= 0) c = -1;
            else {
                  int i = 1;
                  int p = 1;
                  while(i <= b){
                        i++;
                        p +=i;
                  }
                  c = p;
            }
            r = c+2;
      }
      return r;
}
```

Listing 4. Problem 3 from benchmark [12].

```
/*@ ensures \result > 0;
*/
int f(int a, int b){
      int r = -1;
      int c = 2*a + (a+1)%2;

      //@ assume c != 0;
      //@ assume b > 0;
      int i = 1;
      int p = 1;
      while(i <= b){
            i++;
            p +=i;
      }
      c = p;
      r = c+2;
      return r;
}
```

Listing 5. The hard path for problem 3.

as a spurious counter-example after an actual run of the code and the user is asked about a fact to rule it out.

In our case, we came up with the assertion that $\{p > 0\}$ after the while loop. At this point, Why finds no deductive proof of this user fact. Indeed, it requires a

```
;;;conditions
(assert (= r_0 (- 0 1)))
;branch choice condition
(assert (not (= c_0 0)))
;branch choice condition end
(assert (> b_0 0))
(assert (= i_0 1))
(assert (= p_0 1))
(assert (= cnew_1 alpha_p))
(assert (= r_1 (+ 2 cnew_1)))
(assert (= k_0 r_1))

;;;assertion - actually the negation of it
(assert (<= k_0 0))
```

Listing 6. First SMT abstraction.

loop invariant to prove this fact. In this case, a loop invariant that works is $\{i > 0 \wedge p > 0\}$, and the assertion that $\{p > 0\}$ is discharged by all solvers.

Finally, we add this new fact to our SMT model by introducing the following assertion

```
(assert (> alpha_p 0))
```

and run Z3, which claims the formula is unsatisfiable, thus proving correctness of f.

VI. Conclusion

In summary, we make the following contributions in this effort:

- Mechanism to leverage human insight for program verification by 1) querying users for facts about relevant program variables, and 2) providing feedback about their contributed facts
- Technique to combine theorem proving and SMT solving to incrementally construct proofs or find bugs
- Heuristics for computing and refining the program abstraction

References

[1] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[2] F. Bobot, S. Conchon, É. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout, "The alt-ergo automated theorem prover, 2008," *URL http://alt-ergo.lri.fr*.

[3] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, vol. 2, p. 2, 2006.

[4] C. Barrett and C. Tinelli, "Cvc3," in *Computer Aided Verification*. Springer, 2007, pp. 298–302.

[5] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[6] M. Barnett, B. Chang, R. DeLine, B. Jacobs, and K. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*. Springer, 2006, pp. 364–387.

[7] J. Filliâtre and C. Marché, "The why/krakatoa/caduceus platform for deductive program verification," in *Computer Aided Verification*. Springer, 2007, pp. 173–177.

[8] G. Canet, P. Cuoq, and B. Monate, "A value analysis for c programs," in *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. IEEE, 2009, pp. 123–124.

[9] P. Baudin, J. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "Acsl: Ansi," *ISO C specification language*, 2008.

[10] 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. 1–3, pp. 35–45, Dec. 2007.

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

[12] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference." in *PLDI*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 181–192.