Lightweight automatic loop invariant selection
Bachelor Thesis Project Description

Pavel Pozdnyakov
Supervised by Marco Eilers, Prof. Dr. Peter Müller
Department of Computer Science
ETH Zürich
Zürich, Switzerland

April 23, 2019

1 Introduction.

Modern software gradually tends to get more and more complex. Using some sort of computational devices in almost every aspect of human life becomes the norm. Utilizing computers also in life critical situations make the necessity of program verification evident, and, because of the complexity of modern software, automatic verification is desirable.

There are plenty of tools tackling the problem of automatic program verification available today. Some examples are Viper [1], Boogie [2], Why3 [3], ESC/Java [4] etc. Many of these tools rely on some kind of pre-specified annotations to the program text. Among these annotations are pre-, postconditions and loop invariants.

The process of specifying pre-, postconditions and loop invariants in a program text by hand is tedious, error prone and time consuming. Many such annotations, being simple enough, may be inferred using a "guess and check" approach as it is used, for example, in Houdini [5]. Houdini generates a set of annotations according to heuristics based on inspection of annotated programs. Houdini then repeatedly calls a verifier on the annotated program and deletes any annotations refuted by it. The process continues until no further annotation gets deleted. It is not always obvious, however, which invariants are required to prove the given code to be correct with respect to the given specification. Finding some non-trivial annotations may therefore still require human involvement.

2 Project.

In this project we focus our attention on loop invariants. We want to find a better way, compared to currently existing methods that employ the "guess and check" approach for selecting loop invariants from a given set of candidates for some program, whose correctness we want to prove.

One problem with existing approaches to invariant selection process is the need to run the verifier with the same program as input many times. This affects the performance and, hence, limits the scalability of these approaches, since increasing the cardinality of the candidate set increases the potential number of invariants to be refuted. In the worst case we are able to refute one invariant per run of the verifier, i.e. the number of runs, and, hence, the verification time, grows linearly with the cardinality of the candidate invariants set.

In this project we want to find ways to produce a result similar to that of the methods exploiting the "guess and check" approach while keeping the number of interactions with the verifier bounded by some constant. Ideally, we want to be able to reduce this number to one.

We want to encode the problem in a way that all the candidate invariants, both guessed by some tool and specified by the human user, will be given to the verifier at once in a way that enables it to perform the checking in one go. To accomplish this task we want to represent the information about invariants which currently hold in ghost state and subsequently use this information for further verification steps. An example of the problem with one possible way, illustrating the idea behind encoding, to handle it is shown in part 3.

Solving this problem may, however, lead to the need of considering each possible subset of the candidate invariants set. This means in the worst case we have to consider the number of candidate invariants subsets that is exponentially large in the number of invariants in the input set. Hence, we are interested not only in finding ways of performing invariant selection in one user query, but also in making the process of invariant selection efficient in terms of running time. Solving this problem may require, for example, finding dependencies between different candidate invariants and exploiting them in the algorithm or finding ways to parallelize the verification process.

3 Illustration.

We now want to illustrate the Problem with the help of the following toy example. Consider the method below written in Viper.

```
1   method repeat_n(n: Int) returns (res: Int)
2       requires n >= 0
3       ensures res == n
4   {
5       res := 0
6       while (res < n)
7       {
8           res := res + 1
9       }
10  }
```

Example 1.

Suppose we have guessed two candidate loop invariants: $res \leq n$ and $res > n$. We now would like to know which of these invariants, if any, will allow us to prove the correctness of the method. The existing approach, as implemented for example in Houdini, would run the verifier once to refute $res > n$ and, then, once again to conclude that the method is indeed correct and $res \leq n$ is the invariant we are looking for.

It is however possible to encode this method together with corresponding candidate invariants in such a way that only one run of the verifier will be sufficient to prove the correctness of the method.

Below is the encoding. To make it look clearer, we omit the declaration of Boolean variables as well as declarations and definitions of helper functions and methods.

The intention is that *le_on* represents whether the candidate invariant $res \leq n$ holds, *g_on* represents whether the candidate invariant $res > n$ holds. *le_on_b* should be *true* iff the candidate invariant holds before the loop. Analogously, *le_on_a* should be *true* iff the candidate invariant is guaranteed to be satisfied after an arbitrary iteration of the loop.

In Part I of the method we store whether a candidate invariant holds immediately before the loop. We then simulate an arbitrary iteration and check, whether the corresponding candidate invariant still holds after it. If the candidate invariant holds before the loop and after an arbitrary iteration, we set the corresponding variable, *le_on* in our case, to *true*.

Part II is the encoding of the loop we actually want to verify, which we do with the flags *le_on*, *g_on* set in the Part I.

```
1  method repeat_n(n: Int, m: Int) returns (res: Int, ans: Int)
2      requires n >= 0
3      requires m >= 0
4      ensures res == n
5  {
6      res := 0
7      ans := 0
8
9      // Part I
10     assume ans <= m <==> le_on_b
11     assume ans > m <==> g_on_b
12     havoc ans
13     assume (le_on_a ==> ans <= m) && (g_on_a ==> ans > m)
14     assume ans < m
15     ans := ans + 1
16     assume (le_on_b && (le_on_a ==> (le_on_a ==> ans <= m)))
17             ==> le_on
18     assume (g_on_b && (g_on_a ==> (g_on_a ==> ans > m)))
19             ==> g_on
20
21     // Part II
22     if(*) {
23         havoc res
24         assume (le_on ==> res <= n) && (g_on ==> res > n)
25         assume res < n
26         res := res + 1
27         assume false
28     }
29     else {
30         havoc res
31         assume (le_on ==> res <= n) && (g_on ==> res > n)
32         assume !(res < n)
33     }
34 }
```

Example 2.

As we see, it is possible in this case to select the right invariant and verify the method while running the verifier only once. We want to find a way to generalize such kind of encodings to an arbitrary method with arbitrary candidate loop invariants.

4 Core goals.

a) Collecting examples.

To better understand the problem we want to find or write ourselves examples of rather simple programs, which can benefit from our approach and show different challenges for the encoding, e.g. nested and consecutive loops, assertions and failing statements inside loops. For these constructs we want to find appropriate encoding possibilities.

For the sake of improving user convenience of our method, we also want to find ways to model the possibility for the user to specify additional invariants during the verification process, i.e., if the given candidates is not sufficient to prove the correctness of the program, the user should be able to manually add their own invariants. These user-given invariants should be treated differently from the candidate invariants in the sense that an error should always be reported if they do not hold.

b) Encoding.

After getting a deeper understanding of how our invariants selection method should work and identifying possible encoding ways, we want to formalize the encoding for some simple imperative language, e.g. IMP [6, 7].

c) Implementation.

We want to implement the encoding for some platform. Possible choices include Boogie [2], Dafny [8] etc. The implementation itself should be a program which takes the original method we want to verify, without invariants or with some user-given invariants that have to hold, and a set of candidate invariants as input. The program then creates a new method, encoding the candidate invariants into the method as defined before, and subsequently runs the verifier on the transformed method. Possible output may include not only the result of verification, but also the lists of refuted and chosen invariants.

d) Evaluation.

We want to evaluate our implementation against an existing one that uses the "guess and check" approach and the same platform for which we have implemented our encoding. Possible comparison metrics include running time on individual example programs and scalability.

5 Possible extensions.

a) We want to find and exploit dependences between candidate invariants for invariants search optimization. This should allow us not only to improve the computation time, but also the scalability of our method, since the naive approach would be to consider all possible subsets if the input invariants set, which implies the computation time exponential in the number of invariants of the input.

b) We want to generalize our method of invariant selection to pre- and postconditions.

c) We want to generate candidate invariants over some abstract domain. Furthermore, we want to compare the performance of the "guess and check" approach and Abstract Interpretation using this domain.

d) We want to prove the soundness of our encoding.

6 Schedule.

Collecting program examples and understanding the problem	1 month
Encoding	1 month
Choosing the implementation platform	2 weeks
Implementation	1 month
Evaluation	2 weeks
Work on extensions	1 month
Writing the report	1 month

7 References.

1. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, Verification, Model Checking, and Abstract Interpretation (VMCAI), volume 9583 of LNCS, pages41–62. Springer-Verlag, 2016.

2. K. R. M. Leino. This is Boogie 2. Working draft; available at http://research.microsoft.com/en-us/um/people/leino/papers.html, 2008.

3. Francois Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In Boogie 2011: First International Work-shop on Intermediate Verification Languages, pages 53–64, Wroclaw, Poland, August 2011. https://hal.inria.fr/hal-00790310.

4. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002), New York. SIGPLAN, vol. 37(5), pp. 234–245. ACM Press, New York (2002).

5. Flanagan C., Leino K.R.M. (2001) Houdini, an Annotation Assistant for ESC/Java. In: Oliveira J.N., Zave P. (eds) FME 2001: Formal Methods for Increasing Software Productivity. FME 2001. Lecture Notes in Computer Science, vol 2021. Springer, Berlin, Heidelberg.

6. Irons, E. T.: Experience with an Extensible Language. In: Communications of the ACM, vol. 13(1) , pp. 31-40, January 1970.

7. Bilofsky, W.: Syntax extension and the IMP72 programming language. In: ACM SIGPLAN Notices, vol. 9(5), May 1974.

8. Leino K.R.M. (2010) Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke E.M., Voronkov A. (eds) Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2010. Lecture Notes in Computer Science, vol 6355. Springer, Berlin, Heidelberg.