The Automatic Generation of Loop Invariants using SMT Solvers

Haoran Ni
Nanjing University
Nanjing, Jiangsu province, China

Abstract

Loop invariant is a key and fundamental concept in the field of programming languages and formal verification. While the majority of programmers are not familiar with it. In this survey, we will introduce the concept of loop invariant and how to generate loop invariants automatically using SMT solvers.

ACM Reference Format:
Haoran Ni. 2023. The Automatic Generation of Loop Invariants using SMT Solvers. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

A loop invariant is a condition or property that holds true before and after each iteration of a loop.

Loop invariants are of fundamental importance to the field of programming languages. They are used in the design of programming languages, compilers, and static analysis tools. They are also used in the verification of programs, which is a major area of research in programming languages.

Understanding loop invariants is also beneficial to programmers. Just as Dijkstra had suggested (he called it "constructive approach", see [1]), it helps them write correct programs and reason about the behavior of their code.

However, the concept and idea of loop invariants are not widespread beyond the field of programming languages and formal verification. This survey aims to give a brief introduction to loop invariants and generate loop invariants automatically using SMT solvers.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Prerequisites and Motivation

2.1 Loop Invariant

Formal definition of loop invariant is as follows:

$$\frac{\{C \land I\}\ \text{body}\ \{I\}}{\{I\}\ \texttt{while}\ (C)\ \text{body}\ \{\neg C \land I\}}$$

It is presented in Hoare Logic, and its meaning is that if the condition C and the invariant I hold before the loop, and if the body of the loop preserves the invariant, then the invariant will hold after the loop. The invariant I is called a loop invariant if it holds before and after each iteration of the loop.

Loop invariant is a major tool to reason about and formalize the behaviour of loops. Just as in semi-automatic program verification tools such as Dafny [4], only by annotating the loop invariant can Dafny prove the correctness of the program.

2.2 SMT Solver

SMT solvers are widely used in the field of programming languages and formal verification. They are used to solve the satisfiability of logical formulas.

We can regard SMT solvers as declarative programming languages. We can write logical formulas in SMT solvers, and then the SMT solvers will tell us whether the logical formulas are satisfiable or not. We just need to focus on what we want to do, and we don't need to care about how to solve the logical formulas.

2.3 Motivation of Automatic Generation of Loop Invariants

When we debug a program, we tend to take a big-to-small approach, i.e., we generally consider whether some large part of this program is working as expected. However, we find that many times the problem arises in the handling of a very small detail. As a result, it is also of great significance to garantee the correctness of some basic properties of a program (for example, an array index is always in the range of the array). They are simple, but they exist in almost every part of the program. It is definitely a waste of time to verify them manually(that is, manually type in the pre-/post-condition and the loop invariant for every loop) because of the simplicity and the large quantity.

So, it naturally comes to our mind that it would be nice if we can use machines to automatically generate those loop invariants.

3 Automatic Generation of Loop Invariants using SMT Solvers

3.1 Introduction

Based on SAT solver, SMT solver expand the ability of SAT solver to handle first-order logic. SMT solver can handle the satisfiability of logical formulas with quantifiers, which is beyond the ability of SAT solver. It can also handle the satisfiability of logical formulas with some specific theories, such as the theory of linear integer arithmetic, the theory of real arithmetic, the theory of arrays, etc.

3.2 An Example

Let's consider the following program:

```
int i = 0;
while(i < 10) {
  i = i + 1;
}
assert(i == 10);
```

An obvious loop invariant is $0 \leq i \leq 10$.

We want to automatically generate the loop invariant, so now we denote it as $Inv(i)$. then we can get the following logical formula:

$$\frac{\{Inv(i) \land i < 10\}\ i = i + 1\ \{Inv(i)\}}{\{Inv(i)\}\ \texttt{while}\ (i < 10)\ i = i + 1\ \{\neg(i < 10) \land Inv(i)\}}$$

This cannot be understanded by SMT solvers, because it contains the while loop, which is not a boolean expression.

We need to encode the while loop into boolean expressions:

$\forall i.i = 0 => Inv(i)$

$\forall i.Inv(i) \land i < 10 => Inv(i + 1)$

$\forall i.Inv(i) \land not(i < 10) => i = 10$

Then we type in these formula to z3, a famous SMT solver, then get the following result:

```
sat
  (model
    (define-fun Inv ((x!0 Int)) Bool
      (or (not (<= 1 x!0))
          (and (<= 1 x!0)
               (<= 2 x!0)
               ...
               (<= 10 x!0)
               (not (<= 11 x!0)))
               ...
  )
```

The simplified result is $x \leq 10$. Acually we can find that it is a weaker version of the loop invariant we have found.

3.3 The solving strategy of SMT solver

Although we can just regard SMT solver as a black box, it is still necessary to know some basic knowledge about how SMT solver works.

So far, most SMT solvers are using lazy approaches. We can regard them as the integration of SAT solvers and one or more theory solvers. They will first use SAT solvers to check the satisfiability of the logical formulas without considering the theory. If the logical formulas are satisfiable, then they will use the theory solvers to check whether the logical formulas are satisfiable.

Although SAT problems are NP-complete, the SAT solvers are pretty efficient. SAT solvers are using some heuristic methods to accelerate the process, such as the conflict-driven clause learning (CDCL) algorithm.

In the above example, the key uninterpreted function *Inv* is solved by the uninterpreted function theory solver. The essence of the solution method is to construct a series of equivalence classes by means of a concurrent lookup set algorithm, and from these equivalence classes the enumerated form of the function is constructed.

For this example, the uninterpreted function Inv(x) is constructed as two enumerated equivalence classes, $[-\infty, 10]$ and $[11, +\infty]$.

3.4 Drawback and Improvement

It is often difficult for the SMT solver to tackle uninterpreted functions.

Luckily, there are several ways to accelerate the process. Affine inequality (a template of loop invariant with undetermined coefficients), Guess and Check (heuristically guessing loop invariants and check them), Farkas' Lemma (eliminate the universal quantifier), etc.

Typically, Guess and check is a simple but pretty effective method.

For example, in the previous example, we can find some special value $0, 1, 10$and special relation $i = 0, i \leq 10, i = 0$. Then we can generate some loop invariant candidates according to some variation or combination of the special thing above, and check them using SMT solvers.

Moreover, we can use some heuristic methods, such as the one proposed by Gulwani et al. [3] or Carlo A. Furia and Bertrand Meyer [2]. Most of their ideas are straightforward, and we are able to get those ideas naturally as we practice the invariant construction ourselves

4 Conclusion

In this survey, we have explaind basic information about loop invariant, SMT solver. We have also introduced how to use SMT solver to automatically generate loop invariants. After that some methods are mentioned to accelerate the process.

References

[1] Edsger Wybe Dijkstra. 1976. *A discipline of programming*. Prentice-Hall.

[2] Carlo A Furia and Bertrand Meyer. 2009. Loop invariants: analysis, classification and examples. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 1731–1738. https://doi.org/10.1145/1529282.1529747

[3] Sumit Gulwani, Ashish Tiwari, Ramarathnam Venkatesan, and Sriram K Rajamani. 2008. Synthesis of loop-free programs. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 62–73. https://doi.org/10.1145/1328438.1328446

[4] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_24