A Path-based Approach to the Detection of Infinite Looping*

Jian Zhang
Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100080, China
Email: zj@ios.ac.cn

Abstract

Infinite looping is a common type of program error. This paper studies the detection of infinite loops in imperative programs. A sufficient condition is given, which is based on program paths. For a simple loop, if we can establish the infeasibility of certain paths within the loop body, we can conclude that the loop does not terminate. Several examples are given to show the effectiveness of the approach. Its main benefits are that it is accurate and it can be supported by automated tools.

1 Introduction

Program verification aims at proving the correctness of programs. Since the late 1960's, many approaches (e.g., Hoare's logic) have been proposed in this area. Systems and tools have also been developed to support the verification process. However, it is still very difficult to verify non-trivial programs. There are various reasons for this. For example, many methods and tools are not so easy to use, because they require that the user has a strong background in mathematics, and they provide little mechanical support.

Another reason is that most approaches assume that the program is correct. But in fact, many programs are not. This may cause difficulties to verification systems. To quote the developers of the PVS system [7] (p.171):

> "One question ... is what to do when an automated proof attempt fails. This can happen for two reasons: the theorem may be true but the automated procedures are inadequate to prove it, or the theorem may be false. In our experience, it requires some skill to distinguish between these cases, and it may not be easy to develop that skill when relying on automated proof procedures."

Instead of proving the correctness of programs, we assume that the program has errors, and try to find as many errors as possible. We are interested in *automatic* methods and tools for showing the *incorrectness* of programs.

There are many kinds of errors in programs. A common one is *infinite looping*. A loop may not terminate in some cases. As a result, the program hangs. It is desirable if we can find all such loops. However, determining the termination of programs is an undecidable problem. It is impossible to detect all kinds of infinite looping fully automatically.

Our goal is to develop effective methods that can be helpful to programmers for finding many non-terminating loops. In this paper, we describe a method for showing the *non-termination* of *imperative* programs, which is based on paths in the program's flow graph. Infinite looping can be detected if we can show that, once the loop is entered, none of the paths in the loop body leads to the violation of the loop condition.

The paper is organized as follows. In the next section, we recall some concepts and briefly describe some notations and an analysis tool which will be used later. In Section 3, we give a sufficient condition for the existence of infinite looping, and discuss automatic tool support for checking the condition. The approach is illustrated by several examples. Finally we mention some related work as well as limitations of our method, and outline our future research directions.

2 Basic Concepts and Notations

In this paper, we study sequential programs without procedures and functions. We write programs and (pseudo) algorithms in the syntax of C. Thus an array `a` of size `N` consists of the elements: `a[0]`, `a[1]`, ..., `a[N-1]`. The symbol '`=`' means assignment, while '`==`' means equality.

*This work was supported by the National Natural Science Foundation of China (No. 69703014) and the National "973" project (No. G1998030600).

Proceedings of the Second Asia-Pacific Conference On Quality Software (APAQS'01)
0-7695-1287-9/01 $17.00 © 2001 IEEE

A comment begins with '`/*`' and ends with '`*/`'. The logical operators NOT, OR, AND are denoted by '`!`', '`||`', '`&&`', respectively. However, when writing formulas, we still follow the convention that '=' means equality, and the connectives NOT, OR, AND are denoted by '$\neg$', '$\vee$', '$\wedge$', respectively.

A loop consists of a *loop condition* and a *loop body*, which looks like the following:

```
while (loop_cond) {
   loop_body
}
```

There can be `break` and `return` statements within the `loop_body`. However, for simplicity, we assume that there are no `goto` statements. (It is quite rare that the programmer wants to jump into the middle of a loop.) Thus a loop has only one entry, but it can have multiple exits.

The structure of a program can be described graphically as a flow graph which is a directed graph. Each node corresponds to a simple executable statement (e.g., assignment, input/output). The edges show the potential flow of control between the statements. Each edge is an ordered pair of nodes, $\langle n_i, n_j \rangle$, which means there can be a transfer of control from n_i to n_j. An edge may be annotated with a condition (i.e., a set of logical expressions). The transfer of control takes place only when the condition is satisfied.

A path in the flow graph represents a possible way of executing the program. It can be represented as a sequence of logical expressions and simple statements. A path is *executable* or *feasible* if we can assign suitable values to the input variables such that the program executes along that path. Otherwise, it is called *non-executable* or *infeasible*.

For a simple example, let us examine the following code fragment:

```
            int i, j;
/* S1 */    i = 3;
/* S2 */    i = i-1;
/* S3 */    if (i < 0)
/* S4 */      j = 5-i;
/* S5 */    else j = 4+i;
/* S6 */    output(j);
```

The corresponding flow graph is given in Fig. 1. The predicate `P3` refers to the condition `i < 0`.

Here we have two paths:

(1) S1; S2; P3; S4; S6.

(2) S1; S2; ! P3; S5; S6.

After executing statement `S2`, the value of i is 2. Thus the program is executed along the second path. The first path is infeasible. However, if we delete statement `S1`, then both paths are executable. The actual execution trace of the program depends on the initial value of the variable i.

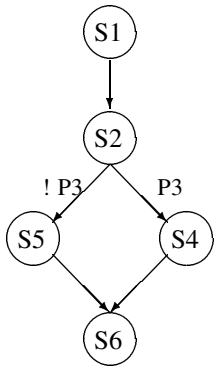

Figure 1. Flow Graph

To decide the feasibility of a program path, we first obtain a set of constraints on the input variables, called the *path condition*, and then checks its satisfiability. The path condition is taken as the conjunction of the constraints. It is satisfiable when the variables can be assigned suitable values such that every constraint becomes true. One method for obtaining the path condition is through *symbolic execution*. The program is "executed" using symbolic values (rather than concrete values) of the variables. As a result, a set of constraints are generated. For more details about the approach, see for example, [5, 6]. A path is executable if and only if the corresponding path condition is satisfiable.

We have developed a tool for deciding path feasibility, called PAT [10]. It is based on a constraint solver called BoNuS [9]. For simplicity, we do not consider output statements, because usually they do not affect the program's flow of control. An input statement is similar to an assignment.

To use PAT, we should specify a program path (i.e., a sequence of assignments and logical expressions), preceded by the necessary variable declarations. And the output of PAT is a set of constraints which will be solved by BoNuS. For the first path in the previous example, we may give the following as input to PAT:

```
  int i, j;
{
  i = 3;
  i = i-1;
@ i < 0;
  j = 5-i;
}
```

Here the symbol '`@`' denotes a logical expression (rather than an assignment statement). PAT will tell us that the path is infeasible.

However, if the first assignment statement `S1` is deleted, PAT will generate the path condition: $i - 1 < 0$. The generation of this path condition is as follows. If we want the program to be executed along the first path, the value of i must be less than 0 before the control flow reaches the label `S3`. So the value of $(i - 1)$ should be negative before the statement `S2`. Thus the path condition consists of the single constraint $(i - 1) < 0$, or equivalently, $i < 1$.

In general, PAT works backward, processing the path from the last element to the first element. Each element may be a logical expression or an assignment. In the first case, the expression is added to the path condition, as a new constraint. (Initially the path condition is empty.) In the second case, suppose the assignment is `x = e`. Then we replace every occurrence of `x` in the path condition by `e`. During the processing, the constraints can be simplified. It is possible that some new constraint contradicts with existing ones. For more details about PAT, see [10].

3 A Sufficient Condition for Infinite Looping

In this paper, we study simple loops (rather than nested loops). The body of a simple loop does not contain loops. It may have conditional statements and assignments.

A simple loop does not terminate if we can show that, once the loop condition is satisfied, no exit can be reached (e.g., the negation of the condition is guaranteed to be unsatisfiable after the loop body is executed).

We define a *complete path* to be a non-looping path which begins with the entry and ends with an exit. It corresponds to one iteration of the loop body. From the flow graph of a simple loop, we can identify a finite set of complete paths.

For each complete path P, we define the *extended complete path* P' as follows. Add the loop condition C before P (as a precondition in P'). If P ends with a `break` or `return` statement, delete this statement; otherwise, P ends with the "last statement" of the loop body, and we add the negation of C to the end of P (as a postcondition in P').

For a simple loop, there should be a *finite* number of (extended) complete paths. We give the following

Sufficient Condition for Infinite Looping: Every extended complete path is infeasible (i.e., unexecutable).

Showing the feasibility of a path is equivalent to proving the theorem $\exists V\,(PC)$, where PC is the path condition and V is the set of variables in PC. Thus, in order to check the above sufficient condition, we need to prove the theorem $\neg\exists V_1\,(PC_1) \wedge \neg\exists V_2\,(PC_2) \wedge \ldots$ or equivalently, the theorem $\forall V_1\,\neg(PC_1) \wedge \forall V_2\,\neg(PC_2) \wedge \ldots$ In practice, such theorems can be proved more easily using constraint solvers rather than conventional deductive theorem provers.

Now let us consider a few examples.

Example 1. A common error with loops is illustrated by the following code fragment:

```
sum = 0;
i = 4;
while (i >= 0) {
    sum = sum + a[i];
    i = i + 1;
}
```

The programmer increases the integer i in the loop body, but actually it should be decreased.

The only one complete path consists of the statements:

```
sum = sum + a[i];
i = i + 1;
```

The corresponding extended complete path is as follows.

```
@ i >= 0;
  sum = sum + a[i];
  i = i+1;
@ i < 0;
```

It is not difficult to see that this path is not executable. Because when i is non-negative initially, it is impossible to make i negative by executing the two assignment statements.

We may also use automatic tools like PAT. For the above extended path, PAT generates the following path condition:

$$i \geq 0;\ \ i + 1 < 0.$$

It is clearly unsatisfiable. Thus the extended complete path is infeasible, and infinite looping is detected.

Example 2. The following program does not terminate:

```
int i;
i = 0;
while (i < 5) {
  if (i == 4)
      i = 0;
  i = i + 1;
}
```

The body of the loop has two paths (denoted by P_1 and P_2), corresponding to the two branches of the `if` statement. In path P_1, the condition `(i==4)` is satisfied; while in path P_2, the condition is not satisfied.

To demonstrate the infinite looping, we show that for each path P_j $(j = 1, 2)$, the extended path "C; P_j; ! C" is infeasible. (Here C denotes the loop condition: `i < 5`.) This can be done with the help of the tool PAT.

For path P_1, we give the following as the input to PAT:

```
    int i;
{
  @ i < 5;
  @ i == 4;
    i = 0;
    i = i+1;
  @ i >= 5;
}
```

Contradiction is found while PAT is generating the path condition.

For path P_2, we give the following as the input to PAT:

```
    int i;
{
  @ i < 5;
  @ i != 4;
    i = i+1;
  @ i >= 5;
}
```

And the generated path condition is the following set of constraints:

$$i - 4 \geq 0;\ \ i \neq 4;\ \ i - 5 < 0.$$

Obviously no integer i satisfies all the constraints.

In both cases, the infeasibility are easily determined. Thus the non-termination of the loop is proved.

The Effect of Constants

The previous sufficient condition says that, for each extended complete path, the path condition is not satisfiable. This may be too strong. It means that the loop can be running infinitely, even if it is isolated. The non-termination is independent of its environment.

In general, we consider code fragments of the form

S1; S2; ...; Sn; simple `while`-loop (*)

Here S1, S2, ..., Sn are assignments or conditional statements.

We can further simplify the problem, considering "code fragments" of the form

SS1; SS2; ...; SSn; simple `while`-loop (**)

where SS1, SS2, ..., SSn are logical expressions or simple statements like assignments. They may also be regarded as "hyperpaths", differing from ordinary paths in that the last element is a `while`-loop. A code fragment of form (*) may correspond to several hyperpaths.

For a simple example, let us consider the code fragment:

```
if (i < 3)
  i = 3;
else i = i+1;
while (i > 0)
  i = i-1;
```

From this we can generate two hyperpaths:

```
HP1:
   @ i < 3;
     i = 3;
     while (i > 0)
       i = i-1;
```

and

```
HP2:
   @ i >= 3;
     i = i+1;
     while (i > 0)
       i = i-1;
```

In some cases, the path condition involves variables that are not modified in the loop. Such variables can be regarded as "constants", with respect to the loop.

Let PC denote the path condition, C and V denote the set of "constants" and the set of other variables, respectively. ($C \cup V$ is the set of all variables involved in PC.) What we need to do is to prove the theorem $\exists C(\forall V_1 \neg PC_1 \wedge \forall V_2 \neg PC_2 \wedge \ldots)$, rather than the original version $\forall C(\forall V_1 \neg PC_1 \wedge \forall V_2 \neg PC_2 \wedge \ldots)$.

Example 3. Let us modify the program in Example 2 slightly:

```
int i, j;
input(j);
i = 0;
while (i < 5) {
  if (i == j)
      i = 0;
  i = i + 1;
}
```

We can analyze it similarly. The first extended complete path (corresponding to the case `i==j`) is as follows.

```
@ i < 5;
@ i == j;
  i = 0;
  i = i + 1;
@ i >= 5;
```

The path condition is unsatisfiable, because it contains the inequality $0 + 1 \geq 5$. The unsatisfiability is independent of the variable j.

The second extended complete path is:

```
@ i < 5;
@ i != j;
  i = i + 1;
@ i >= 5;
```

The path condition consists of the following constraints:

$$i < 5,\ i \neq j,\ i + 1 \geq 5;$$

or equivalently,

$$i = 4,\ i \neq j.$$

To satisfy the sufficient condition for infinite looping, we have to find a value for j such that for any i, $(i \neq 4 \vee i = j)$. This can be achieved when $j = 4$.

Thus when the initial value of j is 4, the loop does not terminate. Note that other values of j (e.g., 3 or 2) may also cause infinite looping.

In the general case, there may be assignments and logical expressions preceding the loop. For such a code fragment, we may *symbolically execute* those assignments and expressions, before checking our sufficient condition. The result of symbolic execution is a set of expressions in terms of the variables' input values. Then we try to prove the theorem $\exists C \forall V \neg(PC^*)$, where PC^* is obtained from the original path condition PC by replacing each constant with its symbolic expression. For example, suppose that there is an assignment `j = j+3` after the input statement. Then we have to prove that $\exists j \forall i\, (i \neq 4 \ \vee \ i = j + 3)$.

Example 4. The following code fragment is adapted from an example in [1] (p.28). It tries to find a target `t` in the array `a[N]`, using binary search. It returns the index `m` which satisfies `a[m] == t`. If no element of the array is equal to the target, (-1) is returned. All the variables are integers.

```
{
 /*S1*/  l = 0;
 /*S2*/  u = N-1;
 /*S3*/  while (l <= u) {
 /*S4*/    m = (l+u)/2;
 /*S5*/    if (a[m] < t)
 /*S6*/       l = m;
 /*S7*/    else if (a[m] > t)
 /*S8*/       u = m;
 /*S9*/    else return m;
         }
 /*S10*/ return (-1);
}
```

There are errors in the above code. Statement S6 should be `l = m+1;` statement S8 should be `u = m-1;` and the program does not terminate for some input data. To determine this, we analyze all the paths within the body of the `while`-loop. There are three (syntactic) paths that can leave the loop:

```
S4; S5; S6.
S4; S7; S8.
S4; S9.
```

Infinite looping occurs when all the corresponding extended paths are infeasible. We first consider Path 1:

```
{
  @ l <= u ;
    m = (l+u)/2;
  @ a[m] < t ;
    l = m;
  @ l > u;
}
```

The path condition contains the following two inequalities:

$$l \leq u;\quad (l + u)/2 > u$$

Obviously they are contradictory. Thus Path 1 is infeasible.

Now we examine Path 2:

```
{
  @ l <= u ;
    m = (l+u)/2;
  @ a[m] > t ;
    u = m;
  @ l > u;
}
```

Again, the path condition is unsatisfiable, because it contains the following two inequalities:

$$l \leq u;\quad l > (l + u)/2$$

So Path 2 is not feasible, either.

Finally, let us examine Path 3:

```
{
  @ l <= u ;
    m = (l+u)/2;
  @ a[m] == t ;
}
```

For this path, it is feasible only when the following constraints are satisfied

$$l \leq u;\quad a[(l + u)/2] = t$$

Since the array `a[]` is not modified in the loop body, the infeasibility of Path 3 is ensured if the array does not contain the element `t` before the control flow reaches the loop.

In summary, if the input data are such that the target `t` does not appear in the array `a[]`, the program will loop forever.

Note that the above is a sufficient condition for nontermination, but not a necessary condition. Other input data may also cause the infinite looping. In fact, the counterexample given in [1] is as follows:

```
a[5] = {10, 20, 30, 40, 50};
t = 50;
```

Given such data as the input, the program does not terminate, either.

On Automatic Tool Support

As mentioned earlier, we try to find potential infinite looping with code fragments of the form (**). To analyze such a hyperpath, we need to solve decision problems which are more complicated than those involved in the feasibility analysis of conventional paths. Specifically, the problem is to decide the theoremhood of formulas like $\exists j \forall i\,(i \neq 4 \vee i = j)$. Such formulas are also called first-order constraints.

There are already some tools available. For example, REDLOG [4] (REDUCE *logic system*) is an extension to the computer algebra system REDUCE. It implements symbolic algorithms (e.g. quantifier elimination) on first-order formulas w.r.t. temporarily fixed first-order languages and theories.

Let us get back to Example 3. To evaluate the formula $\exists j \forall i\,(i \neq 4 \vee i = j)$, we can give the following as input to REDLOG:

```
f := ex(j, all(ii, ii=j or ii<>4));
```

and then evaluate the formula `f`. It is TRUE, of course.

Quantifier elimination methods have been well studied. However, the current version of REDLOG does not support integers yet. The supported domains include real numbers and complex numbers.

Another possibility is to use a decision procedure for Presburger arithmetic, which are available in such tools as PVS [7]. (Presburger arithmetic is a subset of first-order formulas on the integer domain, which does not involve nonlinear operations like multiplication or division.)

Besides the above tools, path analysis tools like PAT [10] also help to get the simplified sufficient conditions, as demonstrated in Example 4. After the simplification, it becomes easier for the user to decide whether it is possible for the loop to be executed infinitely.

4 Related Work

Since the late 1960's, researchers have proposed various methods for proving the termination of programs. One of the main ideas is to associate a *measure function* (also called *bound function* or *convergence function*) with a loop (or a recursive procedure), such that the measure is decreased with each iteration of the loop (or each procedure call). The challenge is to find such a function. This requires some creativity and good understanding of the program.

There has been much research work on proving the termination of functional and logic programs automatically. However, little has been done on the automatic termination analysis of conventional programs. To quote Walther ([8], p.144), "... proving termination is much more complicated for algorithms defined in an imperative language like ALGOL 60 with assignments, goto- and while- statements than for algorithms given in a pure functional notation". Brauburger and Giesl also stated that " a direct termination proof for imperative programs is hard to perform automatically" ([3], p.114), and, "up to now there have been very few attempts to automate termination analysis for imperative programs" ([3], p.130).

Recently, Brauburger [2] proposed to transform imperative programs into functional programs, and then prove that the latter terminate. The basic idea of the transformation is to introduce a recursive procedure for each loop. A termination predicate is computed for a procedure. This predicate is true only for those input data which make the procedure terminate. And an induction theorem prover is used to prove that the predicate is true for every input. The approach was shown to be feasible on some programs. But it is not known what happens when the program does not terminate.

5 Concluding Remarks

Infinite looping is a common type of error in programs. In this paper, a method is proposed for detecting infinite loops, based on program paths. The method is appealing in that it is accurate, because it uses the semantics of statements and expressions; yet it can be supported to a large extent by automated tools. Basically what we need to do is to check the feasibility of several paths, which can be done automatically.

Of course, our current work has some limitations. We have not considered nested loops. The method is sound, but not complete. What is given is only a sufficient condition. Some infinite loops cannot be detected. Moreover, when checking the condition, we might need to prove some theorems involving quantifiers, arrays and integer arithmetic. It is hard to prove them fully automatically.

In the future, we are going to extend and combine existing tools (e.g., PAT and decision procedure for Presburger arithmetic) to provide better assistance for detecting infinite loops. We shall also obtain more experimental results with our method.

Acknowledgements

The author would like to thank the anonymous reviewers for their helpful comments and suggestions. Dr. Wenhui Zhang provided some references to incorrectness and nontermination of programs. Prof. Volker Weispfenning, Dr. Thomas Sturm and Dr. Dongming Wang provided some information on quantifier elimination tools for solving linear constraints.

References

[1] J. Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, Reading, MA., 1988.

[2] J. Brauburger. *Automatic Termination Analysis for Functional and Imperative Programs*. Dissertationen zur Künstlichen Intelligenz (DISKI 207). Infix, Sankt Augustin, 1999.

[3] J. Brauburger and J. Giesl. Approximating the domains of functional and imperative programs. *Science of Computer Programming*, 35(2):113–136, 1999.

[4] A. Dolzmann and T. Sturm. Redlog user manual, edition 2.0. Technical Report MIP-9905, Department of Mathematics and Informatics, University of Passau, Germany, 1999.

[5] L.A. Clarke and D.J. Richardson. Symbolic evaluation methods – implementations and applications. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 65–102. North-Holland, 1981.

[6] P.D. Coward. Symbolic execution and testing. *Information and Software Technology*, 33(1):53–64, 1991.

[7] J. Rushby and M. Srivas. Using PVS to prove some theorems of David Parnas. In J.J. Joyce and C.-J.H. Seger, editors, *Proc. of the Sixth Int'l Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*, pages 163–173. Springer, 1994.

[8] C. Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[9] J. Zhang. Specification analysis and test data generation by solving boolean combinations of numeric constraints. In T.H. Tse and T.Y. Chen, editors, *Proc. of the First Asia-Pacific Conference on Quality Software*, pages 267–274. IEEE Computer Society, 2000.

[10] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. *Int'l J. of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.