MoveOn

Group #9

Jourdan Scott

Mike Roy

About Us

Project: VTL+

Verifier: Java Path Finder (JPF)

Paper: Predicate abstraction for software verification

Cormac Flanagan and Shaz Qadeer

Goal

Simplify the process of verification.

Background

Pre and postconditions are useful for verification.

Not suitable for statically analysing loops.

Need a way to specify what a loop should do.

Need **loop invariants.**

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length */
/*@ ensures \result == a.length ||
              [\result] */
int find(int[] a, boolean[] b) {
  int spot = a.length;
  for (int i = 0; i < a.length; i++){
      if (spot == a.length && a[i] != 0)
                spot = i;
      b[i] = (a[i] != 0);
  }
  return spot;
}
```

Loop Invariants

A loop invariant is a property that holds before (and after) each iteration.

For example:

```
/*@ loop_invariant spot == a.length
    || (b[spot] && spot < i) */
```

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length */
/*@ ensures \result == a.length ||
            b[\result] */
int find(int[] a, boolean[] b) {
  int spot = a.length;
  for (int i = 0; i < a.length; i++){
      if (spot == a.length && a[i] != 0)
              spot = i;
      b[i] = (a[i] != 0);
  }
  return spot;
}
```

Using Invariants

Verifying this code requires a loop invariant like

```
/*@ loop_invariant spot == a.length
     || (b[spot] && spot < i) */
```

With a invariant set we could apply a static checker like ESC/Java.

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length */
/*@ ensures \result == a.length ||
            b[\result] */
int find(int[] a, boolean[] b) {
  int spot = a.length;
  for (int i = 0; i < a.length; i++){
      if (spot == a.length && a[i] != 0)
              spot = i;
      b[i] = (a[i] != 0);
  }
  return spot;
}
```

The Problems

The pre and postconditions serve as good documentation.

Loop invariants are laborious to write and don't serve as great documentation.

Automatic generation would be awesome.

But how do we generate?

```
/*@ requires a != null && b != null */
/*@ requires a.length == b.length */
/*@ ensures \result == a.length ||
            b[\result] */
int find(int[] a, boolean[] b) {
  int spot = a.length;
  for (int i = 0; i < a.length; i++){
      if (spot == a.length && a[i] != 0)
                spot = i;
      b[i] = (a[i] != 0);
  }
  return spot;
}
```

Generating loop Invariants

Loop Predicates

Skolem Constants

Loop Desugaring

Loop Predicates

Loop Invariants are made up of predicates

```
/*@ loop_invariant spot==a.length || (b[spot] && spot < i)
*/
```

Predicates: `spot==a.length, b[spot], spot<i`

Generated from pre and post conditions.

Usually take a long time.

A subset of the possible loop predicates can be combined to create a single loop invariant.

Skolem Constants

When a property needs to be universally quantified (forall),
it be broken down into useful sub predicates.

```
\forall int j; j >= 0 && j < current_max_pos -> list[j] < current_max
```

Because `j` is universally quantified the only predicate is the entire `\forall` statement.

Introducing a skolem constant lets us remove the `\forall` and use the sub predicates to construct a loop invariant.

```
\skolem_constant int j;
loop_predicate j >=0 , j < current_max_pos, list[j] < current_max
```

Loop desugaring

```
for(int i = 0; i < 2; i++)
        print(i);
```

```
int i = 0;
while(i < 2){
        print(i);
        i++;
}
```

```
int i = 0;
if(i < 2){
        print(i);
        i++;
        if(i < 2){
                print(i);
                i++;
                if(i < 2){
                        print(i);
                        i++;
                }
        }
}
```

Generation of loop invariants

The algorithm presented in the paper:

1. Derive a set of loop predicates from the pre and post conditions.

2. Look at the set of states that are reachable through the loop

3. Join predicates to form a loop invariant.
    a. When steady state of reachable states is formed stop.

Example

```
/*@ requires a.length > 0 */
/*@ ensures (\forall int j; 0 <= j && j < a.length ==> m >= a[j]) */
int max(int n,const int a[n]) {
    int m = a[0];
    int i = 1;
    while (i != n) {
        if (m < a[i]){
            m = a[i];
        }
        ++i;
    }
    return m;
}
```

Example

```
/*@ requires a.length > 0 */
/*@ ensures (\forall int j; 0 <= j && j < a.length ==> m >= a[j]) */
```

Universal quantifier, therefore introduce skolem constant

```
/*@ skolem_constant int j;*/
```

Predicates:

```
a.length > 0, 0 <= j, j < a.length, m >= a[j]
```

Process combinations of predicates to find fix point of reachable states.

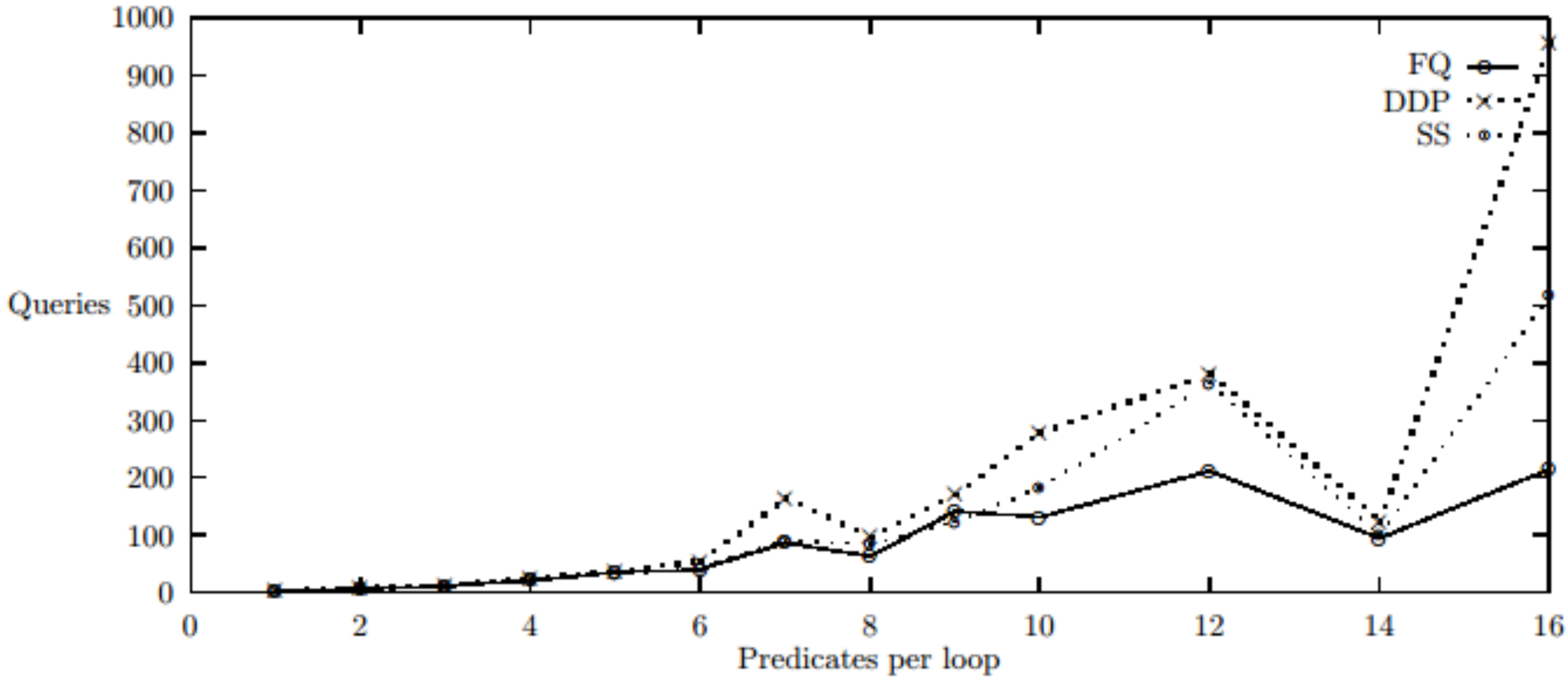
This is the loop invariant

```
/*@ loop_invariant (\forall int j; 0 < j && j < i ==> m >= a[j]) */
```

Example

```
int max(int n,const int a[n]) {
    int m = a[0];
    int i = 1;
    /*@ loop_invariant (\forall int j; 0 < j && j < i ==> m >= a[j]) */
    while (i != n) {
        if (m < a[i]){
            m = a[i];
        }
        ++i;
    }
    return m;
}
```

Experiments



With respect to our system

Unfortunately we were unable to apply these techniques to our system because we had only one loop

With respect to our system

Unfortunately we were unable to apply these techniques to our system because we had only one loop

And it was trivial

Questions

- **4 sections**
  - **The Problem**
  - **The Solution**
  - **The Language**
  - **The Algorithm**

references:

http://www.slideshare.net/icsm2010/ponsini-automatic-slides **- example slides**

THE GAY SLIDE