Verifying C programs in VCC

Sumesh Divakaran

Department of Computer Science and Engineering
Government Engineering College, Idukki.

24th Nov 2016, IISc
Outline

1. Introduction
2. Verifying Simple Programs in VCC
3. Function Contracts
4. Loop Invariants
5. Data Abstraction
Verifier for Concurrent C

- VCC is a tool for Verifying Concurrent C programs
- It takes a program annotated with property specifications
  - Function Contracts (relationship between pre- and post-states)
  - State assertions
  - Invariants (object or loop)
- It tries to prove mathematically that the program meets these properties
- It applies a sound approach for program verification
How VCC proves correctness?

- It applies a deductive approach for verification
- It generates a mathematical statement called Verification Conditions (VCs) from an annotated program
- Weakest precondition analysis is used to generate VCs
- VCs are proved using an automatic theorem prover
- Failures if any will be reported in terms of the program
Verifying C programs in VCC

Annotation for state property (assertions)

```c
int test()
{
    int x,y,z;
    if (x <= y)
        z = x;
    else
        z = y;
    _(assert z <= x )
    return z;
}
```
Verifying C programs in VCC

Annotation for state property (**assertions**)

```c
int test()
{
    int x,y,z;
    if (x <= y)
        z = x;
    else
        z = y;
    _(assert z <= x )
    return z;
}
```

What is the strongest property which can be asserted?
What is the strongest property which can be asserted?

\(( z \leq x ) \&\& ( z \leq y )\)
Verifying C programs in VCC

Annotation for state property (assertions)

```c
int test()
{
    int x, y, z;
    if (x <= y)
        z = x;
    else
        z = y;
    _(assert z <= x )
    return z;
}
```

What is the strongest property which can be asserted?

\(( z <= x ) \&\& ( z <= y ) \&\& ( ( z == x ) || ( z == y ) )\)
Assume annotation

```c
void foo(int x, int y)
{
    int r;
    %(assume y != 0)
    r = x/y;
}
```
1. In the following program fragment, which assertion will fail?

```c
int x, y;
_(assert x > 10)
_(assert x > 5)
```

2. Is there any difference between

- `(assume p)`
- `(assume q)`

and

- `(assume q)`
- `(assume p)`

What if assertions are used in place of assumes?
VCC cannot verify this assertion

To verify this assertion we need to add function contract to the function $\text{foo}$

Function contract typically specifies what a function does.
**Function Contracts**

- Specification of a function is called a function contract, since it gives obligations on both the function and its callers.
- A requirement say \( X \) on the callers (*precondition*) can be specified using a `requires` annotation `\( \text{requires } X \)`.
  
  A caller should ensure that \( X \) is true when it calls the function.
- A requirement \( Y \) on the function (*postcondition*) can be specified using an `ensures` annotation `\( \text{ensures } Y \)`.
  
  The function should ensure that \( Y \) holds when it terminates, by assuming that preconditions hold when it begins.
- A function contract may typically contain a `writes` annotation also to specify the side effects if any.
Modular verification in VCC

Modular verification using function contract

```c
int foo(int a, int b)
  _(requires true)
  _(ensures
    (\result <= a) &&
    (\result <= b) &&
    ((\result == a) ||
     (\result == b))
{
  if (a <= b)
    return a;
  else
    return b;
}
```

```c
void bar()
{
  int x, y, z;
  z = foo(x, y);
  _(assert z <= x)
    { //statement to print z
      //statement to print z
  }
}
```
What VCC will do for a function contract

- When it sees a function call $y = \text{foo}(x)$ inside a function `bar`
  
  It replaces the statement $y = \text{foo}(x)$ in `bar` with the following annotations
  
  _{(assert \ preconditions\text{-for-foo})}
  _{(assume \ postconditions\text{-of-foo})}

- To prove a function contract
  
  It assumes the preconditions and
  Tries to prove the postconditions
Verifying C programs in VCC

Verify that the return value is the median

```c
int middle(int x, int y, int z)
{
    int m;
    m = z;
    if( y < z )
    {
        if( x < y )
            m = y;
        else if( x < z )
            m = x;
    }
    else
    {
        if( z > y )
            m = y;
        else if( x > z )
            m = x;
    }
    return m;
}
```
Loop invariants

- An assertion after a loop cannot be verified, unless
  - The assertion is implied by the invariant clause of the loop, or
  - The assertion is in terms of variables that are not modified in the loop.

- The strongest possible loop invariant of a loop
  - In general, just by looking at the loop, we cannot define a notion of the strongest possible loop invariant.

- Need to give a suitable loop invariant such that
  \[ \neg \text{loop condition} \land \text{loop invariant} \implies POST \]
Finding **suitable** loop invariant

**Suitable loop invariant**

```c
int multiply(int x, int y)
{
    int i;
    for (i = 0; i != y; ++i)
    {
        ret += x;
    }
    return ret;
}
```

**POST :** $ret = x \times y$

- *true* is always a loop invariant. Is it suitable?
# Finding suitable loop invariant

## Suitable loop invariant

```c
int multiply(int x, int y) {
    int i;
    for (i = 0; i != y; ++i) {
        ret += x;
    }
    return ret;
}
```

**POST**: \( ret = x \times y \)

- **true**: is always a loop invariant. Is it suitable?
- **Suitable loop invariant**: \( ret = x \times i \)
  \[
  (i == y) \&\& (ret = x \times i) \implies ret = x \times y
  \]
Data abstraction in VCC

- Data abstraction in VCC can be used to reduce specification clutter for verifying properties about a complex program.
- For example, maps may be used to abstract a data object implemented using self-referential data structures.
- Then function contract may be specified in terms of the abstract representation (maps).
- Data structure invariants should capture the relationship (abstraction relation) between the abstract and concrete representations of data.
- A function updating a concrete object should also update the abstract representation so that the abstraction relation (invariant) is satisfied by the post state of the function.
Data abstraction - example 1

Representation of a double-precision unsigned number

```c
#define ONE (\natural 1)
#define RADIX (UINT_MAX + ONE)
#define DBL_MAX (UINT_MAX + UINT_MAX * RADIX)

typedef struct Double {
    // abstract value
    _(ghost \natural val)

    // implementation
    unsigned low;
    unsigned high;
    //coupling invariant
    _(invariant val == low + high * RADIX)
} Double;
```
Representation of a double-precision unsigned number

Initialise

```c
void dblNew(Double *d)
  _(writes \extent(d))
  _(ensures \wrapped(d) && d->val == 0)
{
  d->low = 0;
  d->high = 0;
  _(ghost d->val = 0)
  _(wrap d)
}
```
Increment

void dblInc(Double *d)
  _(maintains \wrapped(d))
  _(writes d)
  _(requires d->val + 1 < DBL_MAX)
  _(ensures d->val == \old(d->val) + 1)
{
  _(unwrapping d) {
    if (d->low == UINT_MAX) {
      d->high++;
      d->low = 0;
    } else {
      d->low++;
    }
  }
  _(ghost d->val = d->val + 1)
}
Suppose we want to verify the correctness of a C program which implements a set data structure using linear linked list:

- We can use a map to abstract the concrete representation.
- Map should represent the set of items stored in the linked list.
- Abstraction relation should say that the set contains exactly the elements stored in the linked list.
- Now contract for functions updating the linked list can be specified in terms of the set.
Data abstraction using maps

Node definition for a linked list implementation of a set

```c
struct Node {
    struct Node *next;
    int data;
};
```

How can one represent a linked list implementation of a set using maps?

- Define a map from `int` to `bool`
- Specify an invariant saying that the set contains exactly the elements stored in the linked list
Function contract for adding an element to a set

```c
int add(struct List *l, int k)
    _(requires \wrapped(l))
    _(ensures \wrapped(l))
    _(ensures (\result == -1) =>
        l->membership == \old(l->membership))
    _(ensures (\result != -1) =>
        \forall int p; l->membership[p] ==
        \old(l->membership)[p] || p == k))
    _(writes l)
{
...
```

- Contract is specified in terms of the abstract set
- `add` is expected to return `-1` if it fails to allocate memory for a new node
Function with a set abstraction for a linked list

Function for adding an element to a set

```c
int add(struct List *l, int k)
{
    struct Node *n = malloc(sizeof(*n));
    if (n == NULL) return -1;
    unchecked(n) {
        n->next = l->head; n->data = k;
        unchecked(n); l->head = n;
        unchecked(ghost {
            l->owns += n;
            l->membership = (\lambda int z; z == k || l->membership[z]);
        })
    }
    return 0;
}
```
How can we make the abstraction sound

- Add an auxiliary object to each node $x$, which represents the set of integers which is represented by the list starting at $x$
- A map form $(\text{Node } \times \text{int})$ to \text{bool} can model this
- Now the set associated with the root node gives exactly the set represented by the linked list
Map representing a set implemented using a linked list

_(dynamic_owns) struct List {
  _(ghost bool membership[int];)
  struct Node *head;
  _(ghost bool followers[struct Node *][int])
  _(invariant membership == followers[head])
  _(invariant head != NULL ==> \mine(head))
  _(invariant \forall struct Node *n;
      \mine(n) ==> n->next == NULL || \mine(n->next))
  _(invariant followers[NULL] == \lambda int k;\false)
  _(invariant \forall struct Node *n;
      \mine(n) ==> \forall int e;
      followers[n][e] <===> followers[n->next][e] || e == n->data)
};
Function to check membership

```c
int member(struct List *l, int k)
  _(requires \wrapped(l))
  _(ensures \result != 0 <=> l->membership[k])
{
  struct Node *n;
  for (n = l->head; n; n = n->next)
    _(invariant n != NULL ==> n \in l->\owns)
    _(invariant l->membership[k] <=>
        l->followers[n][k])
    {
      if (n->data == k)
        return 1;
    }
  return 0;
}
```
Thank You