Verification using VeriFast – Useful Feedbacks and other Supporting Tools

Thomas Baar

Hochschule für Technik und Wirtschaft (HTW) Berlin
Department of Engineering I

Talk @ KeY Symposium 2023
Høgskulen på Vestlandet (HVL), Bergen, Norway
August 8 - 10, 2023

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Who is Thomas Baar?

Who is Thomas Baar?

Short Bio

- Joined KeY-Team in 1999 as PhD student
- PhD thesis on *Semantics of UML/OCL* in 2002
- Post-Doc 2003 - 2007 at EPFL, Switzerland
- 2007 - 2011: Software developer in a small company
- Since 2011: Professor at HTW Berlin

Who is Thomas Baar?

Short Bio

- Joined KeY-Team in 1999 as PhD student
- PhD thesis on *Semantics of UML/OCL* in 2002
- Post-Doc 2003 - 2007 at EPFL, Switzerland
- 2007 - 2011: Software developer in a small company
- Since 2011: Professor at HTW Berlin

Interests

- Better teaching of **Formal Methods**:
  - find small and convincing examples
  - find right metaphor
  - get hands-on experience with (large) case studies

Who is Thomas Baar?

Short Bio

- Joined KeY-Team in 1999 as PhD student
- PhD thesis on *Semantics of UML/OCL* in 2002
- Post-Doc 2003 - 2007 at EPFL, Switzerland
- 2007 - 2011: Software developer in a small company
- Since 2011: Professor at HTW Berlin

Interests

- Better teaching of **Formal Methods**:
  - find small and convincing examples
  - find right metaphor
  - get hands-on experience with (large) case studies
- **Domain-Specific Languages** as a versatile tool in engineering

Outline

Main Challenge

VeriFast in a Nutshell

- Mainly developed by Bart Jacobs (starting around 2008)
- Program verifier for C (Pointers!) and other languages (Java, C++)
- Based on Separation Logic

VeriFast in a Nutshell

- Mainly developed by Bart Jacobs (starting around 2008)
- Program verifier for C (Pointers!) and other languages (Java, C++)
- Based on Separation Logic

Verification Objectives

- **Functional Correctness** wrt. partial correctness
  - also total correctness can be configured
- **Memory Safety**
  - no arithmetic under-/overflow
  - safe access via pointers (*heap chunks*)

Verification using VeriFast

User's Steps: Annotate source code for **automatic verification**

Verification using VeriFast

User's Steps: Annotate source code for **automatic verification**

Annotation language

- Boolean expressions attached as
  - pre-/postcondition (requires/ensures)
  - invariant (of loop)
  - assertion
- Nice integration with code variables
- Auxiliary constructs (predicates, fixpoints, lemmata, . . .)

Main Task when Working with VeriFast

Holy Grail: Find the right annotations for the code

Main Task when Working with VeriFast

Holy Grail: Find the right annotations for the code

```
void quicksort(int *a, int lo, int hi)
{
  if (lo > hi) {
    return;
  } else {
    int p = partition(a, lo, hi);
    quicksort(a, lo, p-1);
    quicksort(a, p+1, hi);
  }
}
```

Main Task when Working with VeriFast

Holy Grail: Find the right annotations for the code

```
void quicksort(int *a, int lo, int hi)
{
  if (lo > hi) {
    return;
  } else {
    int p = partition(a, lo, hi);
    quicksort(a, lo, p-1);
    quicksort(a, p+1, hi);
  }
}
```

```
void quicksort(int *a, int lo, int hi)
  //@ requires a[lo..hi + 1] |-> ?vs;
  //@ ensures a[lo..hi + 1] |-> ?vs2 &*& (count_eq)(vs2) == (count_eq)(vs) &*& is_sorted_between(no
{
  if (lo > hi) {
    //@ switch (vs) { case nil: case cons(v0, vs0): }
    return;
  } else {
    int p = partition(a, lo, hi);
    //@ assert a[lo..p] |-> ?vslow0 &*& a[p] |-> ?pivot &*& a[p + 1..hi + 1] |-> ?vshigh0;
    //@ assert (mplus)((count_eq)(vslow0), (count_eq)(cons(pivot, vshigh0))) == (count_eq)(vs);
    //@ count_eq_append({pivot}, vshigh0);
    quicksort(a, lo, p-1);
    quicksort(a, p+1, hi);
    //@ assert a[lo..p] |-> ?vslow &*& a[p] |-> pivot &*& a[p + 1..hi + 1] |-> ?vshigh;
    //@ close ints(a + p, hi + 1 - p, _);
    //@ ints_join(a + lo);
    //@ assert a[lo..hi + 1] |-> ?vs2;
    //@ assert vs2 == append(vslow, cons(pivot, vshigh));
    //@ assert (count_eq)(vslow) == (count_eq)(vslow0);
    //@ count_eq_append(vslow, cons(pivot, vshigh));
    //@ count_eq_append({pivot}, vshigh);

    //@ count_eq_forall(vslow0, vslow, (ge)(pivot));
    //@ is_sorted_forall_ge(none, vslow, pivot);
    //@ assert is_sorted_between(none, vslow, some(pivot)) == true;

    //@ count_eq_forall(vshigh0, vshigh, (le)(pivot));
    //@ is_sorted_forall_le(pivot, vshigh);
    //@ is_sorted_append(none, vslow, pivot, cons(pivot, vshigh));
  }
}
```

Main Task when Working with VeriFast

Holy Grail: Find the right annotations for the code

```c
void quicksort(int *a, int lo, int hi)
{
  if (lo > hi) {
    return;
  } else {
    int p = partition(a, lo, hi);
    quicksort(a, lo, p-1);
    quicksort(a, p+1, hi);
  }
}
```

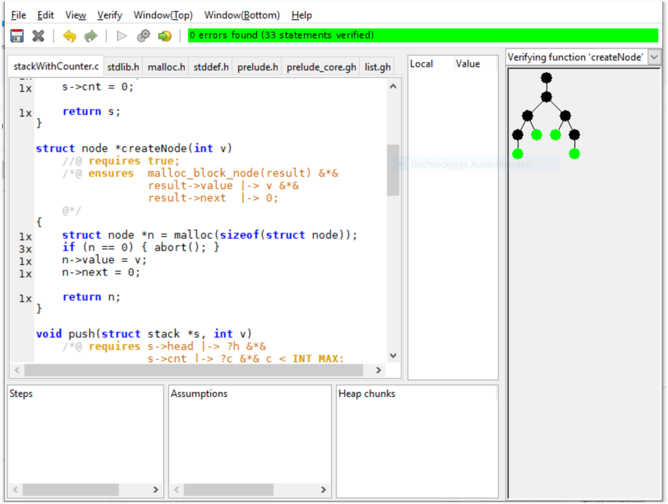
```c
void quicksort(int *a, int lo, int hi)
  //@ requires a[lo..hi + 1] |-> ?vs;
  //@ ensures a[lo..hi + 1] |-> ?vs2 &*& (count_eq)(vs2) == (count_eq)(vs) &*& is_sorted_between(no
{
  if (lo > hi) {
    //@ switch (vs) { case nil: case cons(v0, vs0): }
    return;
  } else {
    int p = partition(a, lo, hi);
    //@ assert a[lo..p] |-> ?vslow0 &*& a[p] |-> ?pivot &*& a[p + 1..hi + 1] |-> ?vshigh0;
    //@ assert (mplus)((count_eq)(vslow0), (count_eq)(cons(pivot, vshigh0))) == (count_eq)(vs);
    //@ count_eq_append({pivot}, vshigh0);
    quicksort(a, lo, p-1);
    quicksort(a, p+1, hi);
    //@ assert a[lo..p] |-> ?vslow &*& a[p] |-> pivot &*& a[p + 1..hi + 1] |-> ?vshigh;
    //@ close ints(a + p, hi + 1 - p, _);
    //@ ints_join(a + lo);
    //@ assert a[lo..hi + 1] |-> ?vs2;
    //@ assert vs2 == append(vslow, cons(pivot, vshigh));
    //@ assert (count_eq)(vslow) == (count_eq)(vslow0);
    //@ count_eq_append(vslow, cons(pivot, vshigh));
    //@ count_eq_append({pivot}, vshigh);

    //@ count_eq_forall(vslow0, vslow, (ge)(pivot));
    //@ is_sorted_forall_ge(none, vslow, pivot);
    //@ assert is_sorted_between(none, vslow, some(pivot)) == true;

    //@ count_eq_forall(vshigh0, vshigh, (le)(pivot));
    //@ is_sorted_forall_le(pivot, vshigh);
    //@ is_sorted_append(none, vslow, pivot, cons(pivot, vshigh));
  }
}
```

Example quicksort: blows up from 43 LOC to 296 LOC

First Impressions

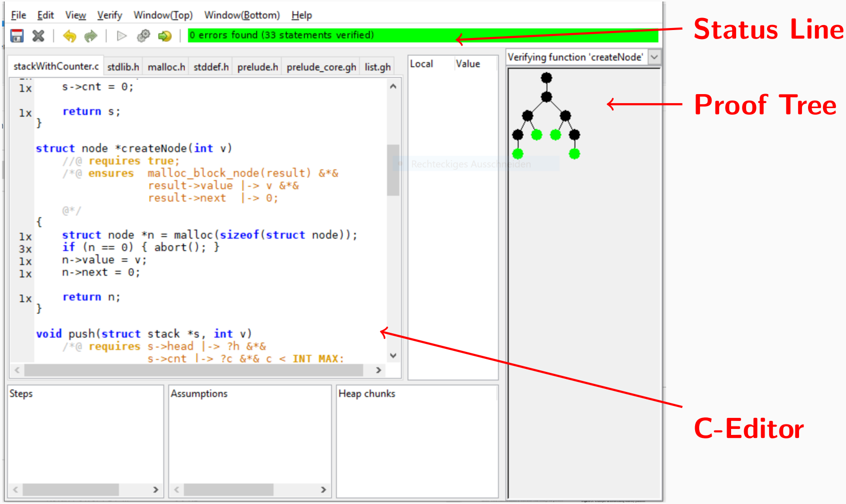
First Impressions From Tool

Example `createNode()`



First Impressions From Tool

Example `createNode()`



```c
struct node {
    int value;
    struct node *next;
};

struct stack {
    struct node *head;
    int cnt;
};
```

Example Stack

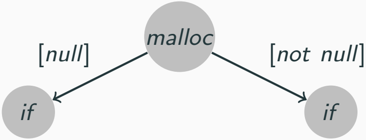
Symbolic Execution of `createNode()`

```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```

Example Stack

Symbolic Execution of `createNode()`

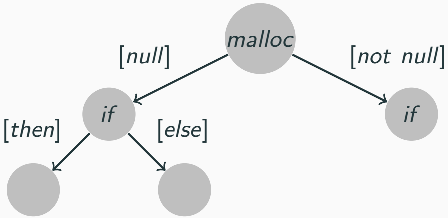
```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```



Example Stack

Symbolic Execution of `createNode()`

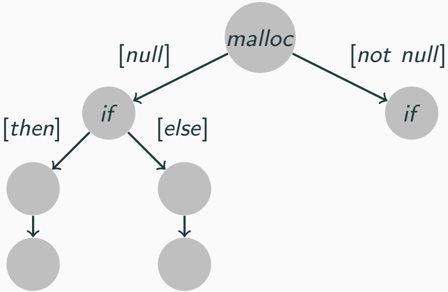
```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```



Example Stack

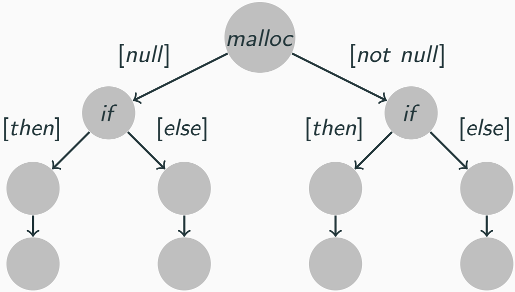
Symbolic Execution of `createNode()`

```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```



Example Stack
Symbolic Execution of `createNode()`

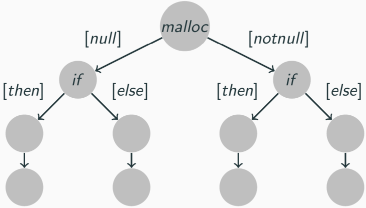
```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```



Proof Tree

Represents Symbolic Code Execution

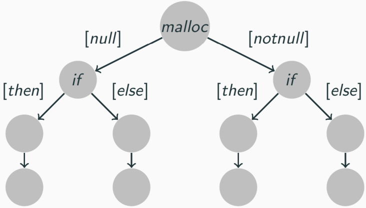
```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```

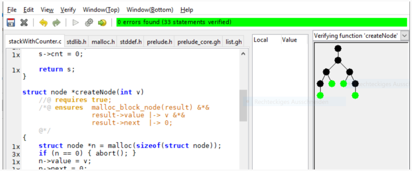


Proof Tree

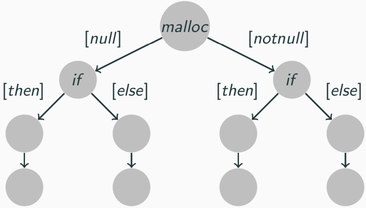
Represents Symbolic Code Execution

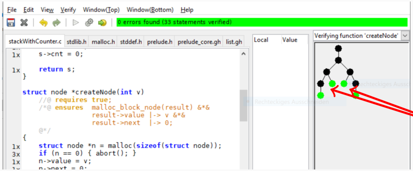
```
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```





```c
struct node *createNode(int v)
{
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
}
```

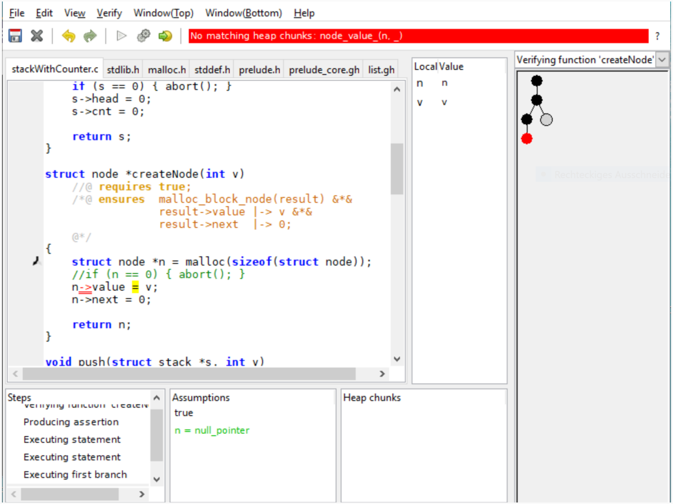




VeriFast optimizes Proof Tree

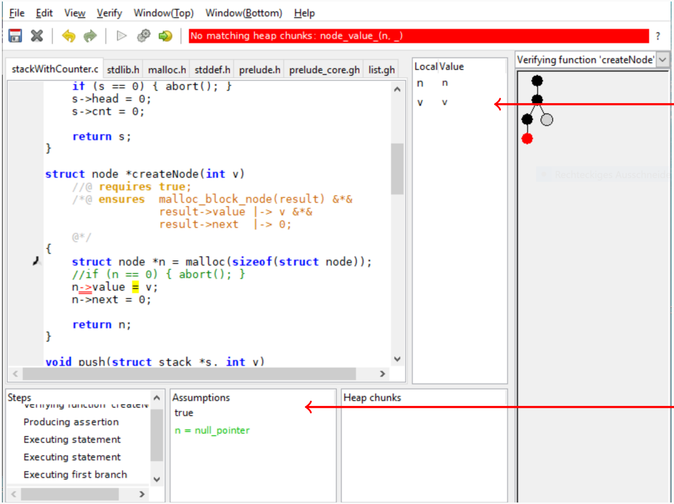
First Impressions From Tool

Example `createNode()` with Bug



First Impressions From Tool

Example `createNode()` with Bug

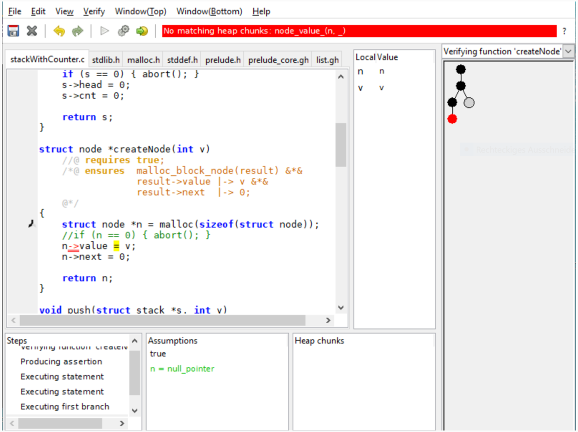


Var-Value Map

Restrictions on Values

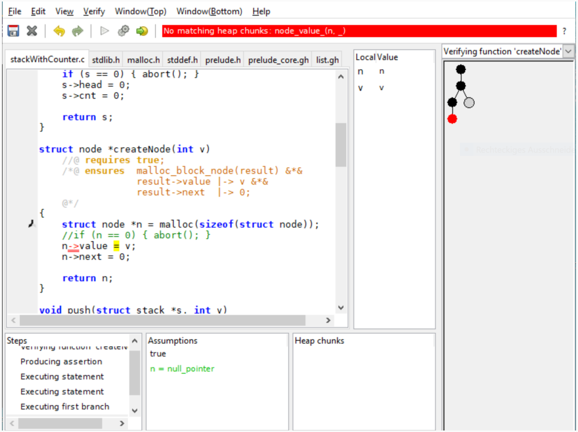
First Impressions From Tool

Summary



First Impressions From Tool

Summary



+ Code Oriented
+ Proof Tree
+ PT Node Inspection

- PT Optimization

VeriFast Tutorial

VeriFast in Action

assert - Annotation Placed within Code for Debugging

```
int mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

VeriFast in Action

assert - Annotation Placed within Code for Debugging

```
int mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

VeriFast in Action

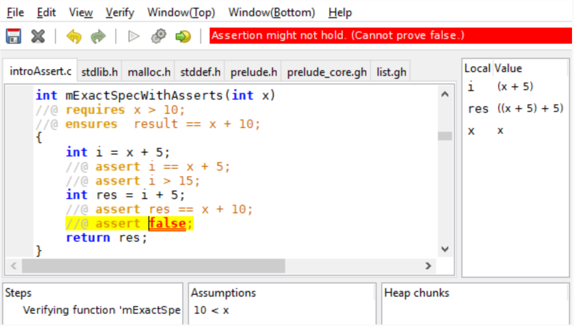
`assert` - Annotation Placed within Code for Debugging

```
int mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

```
int mExactSpecWithAsserts(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    //@ assert i == x + 5;
    //@ assert i > 15;
    int res = i + 5;
    //@ assert res == x + 10;
    // assert false;
    return res;
}
```

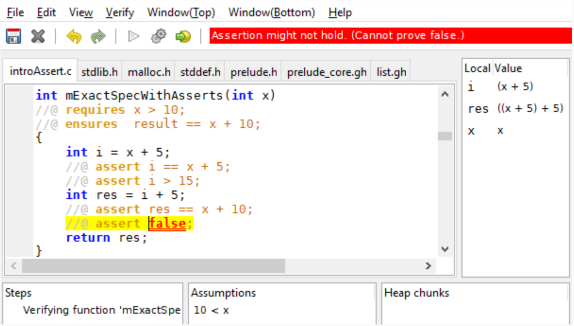
VeriFast in Action

`assert` - Annotation for Debugging



VeriFast in Action

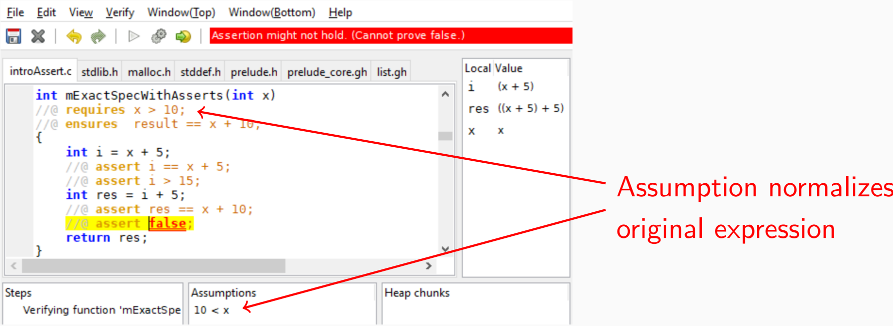
`assert` - Annotation for Debugging



`//@ assert false;` allows to stop symbolic execution and to inspect current situation!

VeriFast in Action

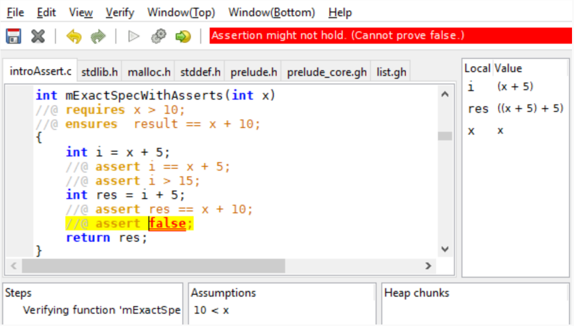
`assert` - Annotation for Debugging



`//@ assert false;` allows to stop symbolic execution and to inspect current situation!

VeriFast in Action

`assert` - Annotation for Debugging



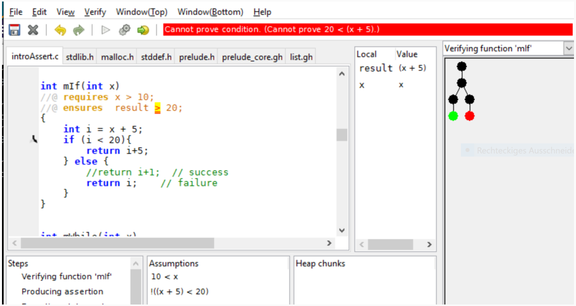
\+ Verification can be stopped

\- Normalization of assumptions

`//@ assert false;` allows to stop symbolic execution and to inspect current situation!

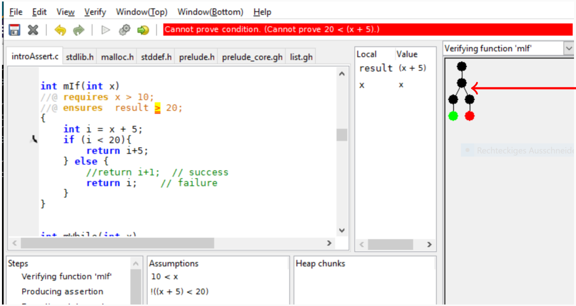
VeriFast in Action

`if` - Statement Splits Proof-Tree



VeriFast in Action

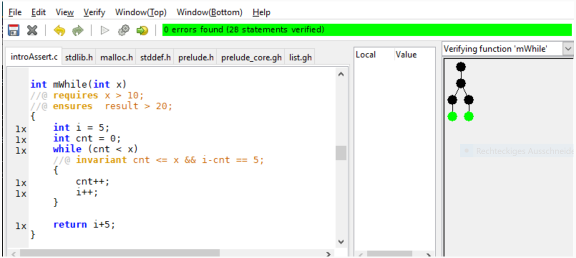
`if` - Statement Splits Proof-Tree



Split for 'then' and 'else'

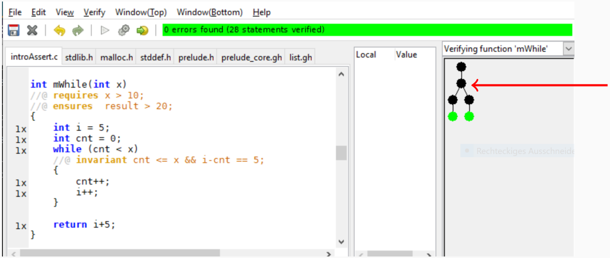
VeriFast in Action

`while` - Statement Requires Invariant



VeriFast in Action

`while` - Statement Requires Invariant



Split for proving invariant and skipping loop body

VeriFast in Action

while - Statement Requires Invariant

```
int mWhile2(int x)
//@ requires x > 10;
//@ ensures  result > 20;
{
    int i = 5;
    int cnt = 0;
    while (cnt < x)
    //@ invariant cnt <= x && i-cnt == 5;
    {
        cnt++;
        i++;
    }
    //@ assert cnt==x && i == x + 5;

    return i+5;
}
```

VeriFast in Action

while - Statement Requires Invariant

```
int mWhile2(int x)
//@ requires x > 10;
//@ ensures  result > 20;
{
    int i = 5;
    int cnt = 0;
    while (cnt < x)
    //@ invariant cnt <= x && i-cnt == 5;
    {
        cnt++;
        i++;
    }
    //@ assert cnt==x && i == x + 5;

    return i+5;
}
```

annotation for debugging

Function Call

VeriFast in Action

Function Call

Two functions with same implementation but different post-conditions:

```
int mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

VeriFast in Action

Function Call

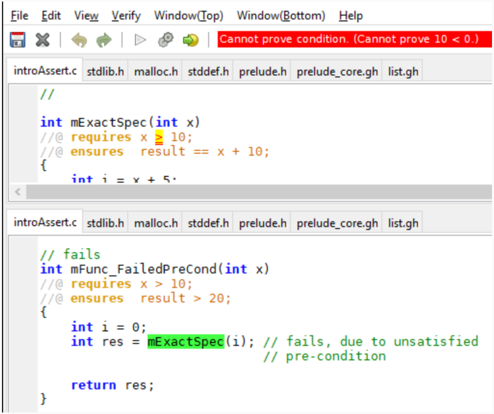
Two functions with same implementation but different post-conditions:

```
int mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result == x + 10;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

```
int mAbstractSpec(int x)
//@ requires x > 10;
//@ ensures  result > 20;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

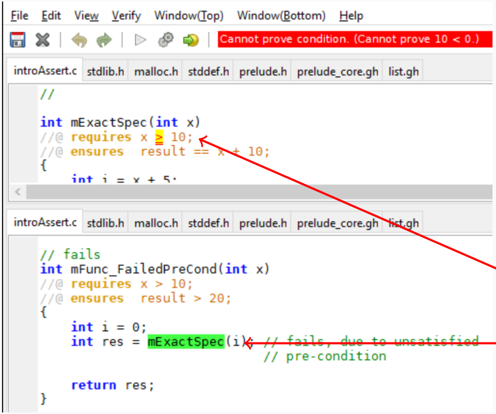
VeriFast in Action

Function Call - Precondition is Checked



VeriFast in Action

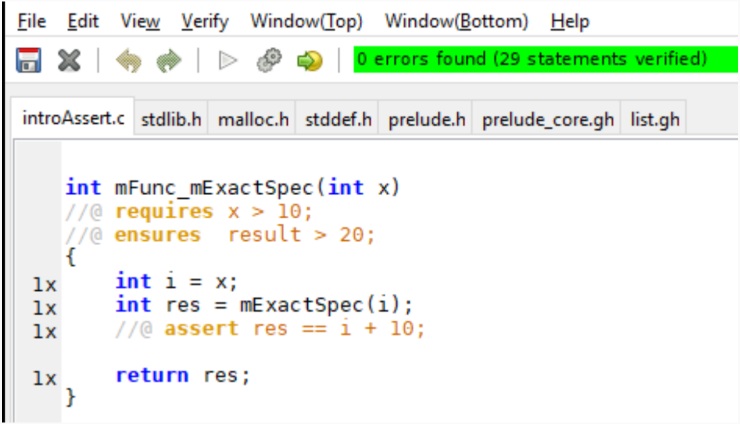
Function Call - Precondition is Checked



Error shown with multiple culprits

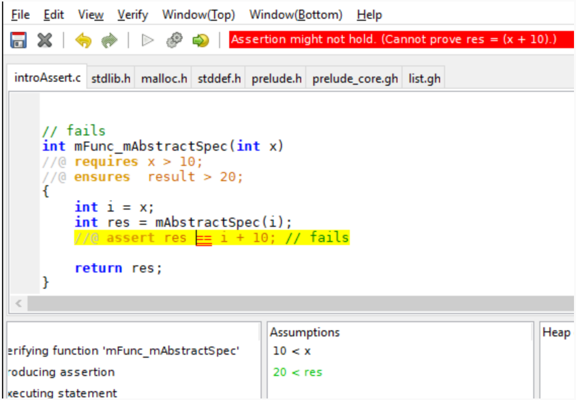
For each function call, the validity of the pre-condition in the current situation is checked.

VeriFast in Action
Function Call - Postcondition is Only Knowledge



VeriFast in Action

Function Call - Postcondition is Only Knowledge

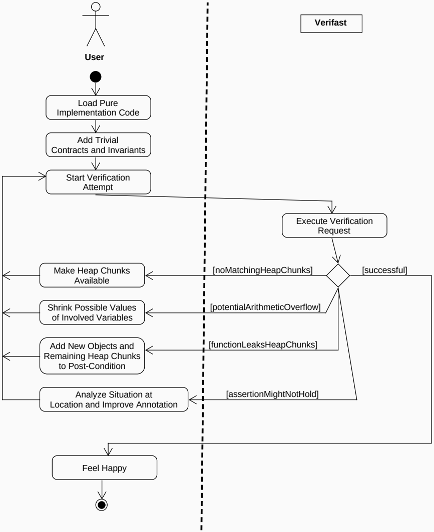



```
int mAbstractSpec(int x)
//@ requires x > 10;
//@ ensures  result > 20;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```

For each function call, the implementation of the called function is irrelevant. Only the post-condition counts!

Towards a Process of Verification

Process of Verification Steps



Demo

Summary

Summary of VeriFast

Advantages

- **User interacts with Code!!!**
  - Error feedback: Current node in Proof Tree can be inspected
  - Debugging: Verification can be stopped at any code location
  - Annotation language is aligned to C (e.g. declaration of ghost variables)
  - Assertion statement possible at every code location

Summary of VeriFast
Advantages

- **User interacts with Code!!!**
  - Error feedback: Current node in Proof Tree can be inspected
  - Debugging: Verification can be stopped at any code location
  - Annotation language is aligned to C (e.g. declaration of ghost variables)
  - Assertion statement possible at every code location
- Modular verification
- Proof-Carrying-Code (PCC) philosophy

Summary of VeriFast
Advantages

- **User interacts with Code!!!**
  - Error feedback: Current node in Proof Tree can be inspected
  - Debugging: Verification can be stopped at any code location
  - Annotation language is aligned to C (e.g. declaration of ghost variables)
  - Assertion statement possible at every code location
- Modular verification
- Proof-Carrying-Code (PCC) philosophy

PCC: Once the proof is done, it's really done . . .

Summary of VeriFast
Dis-Advantages

- Implementation code is polluted with VeriFast-annotations

Summary of VeriFast

Dis-Advantages

- Implementation code is polluted with VeriFast-annotations
- Incomplete coverage of C language (array, multi-dim arrays)
- Lack of tutorials
- Examples hard to follow (without docu)
- Confusing identifiers for predicates/lemmata/ (very short IDs)

Summary of VeriFast
Dis-Advantages

- Implementation code is polluted with VeriFast-annotations

- Incomplete coverage of C language (array, multi-dim arrays)
- Lack of tutorials
- Examples hard to follow (without docu)
- Confusing identifiers for predicates/lemmata/ (very short IDs)

- Optimization of proof tree
- Normalization in assumptions ($<$ instead of $>$)

Questions to the Audience

Questions to the Audience

Can AI-based systems help to find the right VeriFast annotation?

Questions to the Audience

Can AI-based systems help to find the right VeriFast annotation?

Is it worthwhile to allow multiple contracts per function?

Thank you.