

# 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

Main Challenge

First Impressions

VeriFast Tutorial

Towards a Process of Verification

Summary

## **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**

**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

```
34 void quicksort(int *a, int lo, int hi)
35 {
36     if (lo > hi) {
37         return;
38     } else {
39         int p = partition(a, lo, hi);
40         quicksort(a, lo, p-1);
41         quicksort(a, p+1, hi);
42     }
43 }
```

# Main Task when Working with VeriFast

## Holy Grail: Find the right annotations for the code

```
34 void quicksort(int *a, int lo, int hi)
35 {
36   if (lo > hi) {
37     return;
38   } else {
39     int p = partition(a, lo, hi);
40     quicksort(a, lo, p-1);
41     quicksort(a, p+1, hi);
42   }
43 }
```

```
265 void quicksort(int *a, int lo, int hi)
266   //@ requires a[lo..hi + 1] |-> ?vs;
267   //@ ensures a[lo..hi + 1] |-> ?vs2 &*R (count_eq)(vs2) == (count_eq)(vs) &*R is_sorted_between(none, vs2)
268 {
269   if (lo > hi) {
270     //@ switch (vs) { case nil: case cons(v0, vs0): }
271     return;
272   } else {
273     int p = partition(a, lo, hi);
274     //@ assert a[lo..p] |-> ?vslow0 &*R a[p] |-> ?pivot &*R a[p + 1..hi + 1] |-> ?vshigh0;
275     //@ assert (mplus)((count_eq)(vslow0), (count_eq)(cons(pivot, vshigh0))) == (count_eq)(vs);
276     //@ count_eq_append({pivot}, vshigh0);
277     quicksort(a, lo, p-1);
278     quicksort(a, p+1, hi);
279     //@ assert a[lo..p] |-> ?vslow &*R a[p] |-> pivot &*R a[p + 1..hi + 1] |-> ?vshigh;
280     //@ close ints(a + p, hi + 1 - p, _);
281     //@ ints_join(a + lo);
282     //@ assert a[lo..hi + 1] |-> ?vs2;
283     //@ assert vs2 == append(vslow, cons(pivot, vshigh));
284     //@ assert (count_eq)(vslow) == (count_eq)(vslow0);
285     //@ count_eq_append(vslow, cons(pivot, vshigh));
286     //@ count_eq_append({pivot}, vshigh);
287
288     //@ count_eq_forall(vslow0, vslow, (ge)(pivot));
289     //@ is_sorted_forall_ge(none, vslow, pivot);
290     //@ assert is_sorted_between(none, vslow, some(pivot)) == true;
291
292     //@ count_eq_forall(vshigh0, vshigh, (le)(pivot));
293     //@ is_sorted_forall_le(pivot, vshigh);
294     //@ is_sorted_append(none, vslow, pivot, cons(pivot, vshigh));
295   }
296 }
```

# Main Task when Working with VeriFast

## Holy Grail: Find the right annotations for the code

```
34 void quicksort(int *a, int lo, int hi)
35 {
36     if (lo > hi) {
37         return;
38     } else {
39         int p = partition(a, lo, hi);
40         quicksort(a, lo, p-1);
41         quicksort(a, p+1, hi);
42     }
43 }
```

```
265 void quicksort(int *a, int lo, int hi)
266     //@ requires a[lo..hi + 1] |-> ?vs;
267     //@ ensures a[lo..hi + 1] |-> ?vs2 &*R (count_eq)(vs2) == (count_eq)(vs) &*R is_sorted_between(none, vslow, vs)
268     {
269     if (lo > hi) {
270         //@ switch (vs) { case nil: case cons(v0, vs0): }
271         return;
272     } else {
273         int p = partition(a, lo, hi);
274         //@ assert a[lo..p] |-> ?vslow0 &*R a[p] |-> ?pivot &*R a[p + 1..hi + 1] |-> ?vshigh0;
275         //@ assert (mplus)((count_eq)(vslow0), (count_eq)(cons(pivot, vshigh0))) == (count_eq)(vs);
276         //@ count_eq_append({pivot}, vshigh0);
277         quicksort(a, lo, p-1);
278         quicksort(a, p+1, hi);
279         //@ assert a[lo..p] |-> ?vslow &*R a[p] |-> pivot &*R a[p + 1..hi + 1] |-> ?vshigh;
280         //@ close ints(a + p, hi + 1 - p, _);
281         //@ ints_join(a + lo);
282         //@ assert a[lo..hi + 1] |-> ?vs2;
283         //@ assert vs2 == append(vslow, cons(pivot, vshigh));
284         //@ assert (count_eq)(vslow) == (count_eq)(vslow0);
285         //@ count_eq_append(vslow, cons(pivot, vshigh));
286         //@ count_eq_append({pivot}, vshigh);
287
288         //@ count_eq_forall(vslow0, vslow, (ge)(pivot));
289         //@ is_sorted_forall_ge(none, vslow, pivot);
290         //@ assert is_sorted_between(none, vslow, some(pivot)) == true;
291
292         //@ count_eq_forall(vshigh0, vshigh, (le)(pivot));
293         //@ is_sorted_forall_le(pivot, vshigh);
294         //@ is_sorted_append(none, vslow, pivot, cons(pivot, vshigh));
295     }
296 }
```

Example **quicksort**: blows up from 43 LOC to 296 LOC

# First Impressions

---



# First Impressions From Tool

## Example createNode()

The screenshot shows the VeriFast IDE interface. The main editor displays the following C code for the `createNode` function:

```
1x s->cnt = 0;
1x return s;
}

struct node *createNode(int v)
  /*@ requires true;
   * ensures malloc_block_node(result) &* &
   * result->value |-> v &* &
   * result->next |-> 0;
   */
{
  1x struct node *n = malloc(sizeof(struct node));
  3x if (n == 0) { abort(); }
  1x n->value = v;
  1x n->next = 0;
  1x return n;
}

void push(struct stack *s, int v)
  /*@ requires s->head |-> ?h &* &
   * s->cnt |-> ?c &* & c < INT_MAX;
   */
```

The right sidebar shows a call graph for the function `Verifying function 'createNode'`. The graph consists of several nodes: a root node (black), two intermediate nodes (black), and three leaf nodes (green). A tooltip "Rechteckiges Ausschneiden" is visible over one of the green nodes.

The bottom of the IDE has three panels: "Steps", "Assumptions", and "Heap chunks", all of which are currently empty.

# First Impressions From Tool

## Example createNode()

The screenshot shows the C-Editor interface with the following components:

- Status Line:** A green bar at the top displaying "0 errors found (33 statements verified)".
- Code Editor:** The main window showing the C code for `createNode()` and `push()` with annotations. The code includes comments like `//@ requires true;`, `/*@ ensures malloc_block_node(result) &*6`, and `result->value |-> v &*6`.
- Proof Tree:** A tree diagram on the right side of the editor, titled "Verifying function 'createNode'", showing a hierarchy of nodes (black and green).
- Bottom Panels:** Three panels labeled "Steps", "Assumptions", and "Heap chunks" are visible at the bottom of the interface.

Status Line

Proof Tree

C-Editor

## Example createNode()

### Basic Data Structures

```
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;

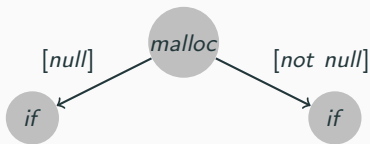
    return n;
}
```

# 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;
}
```

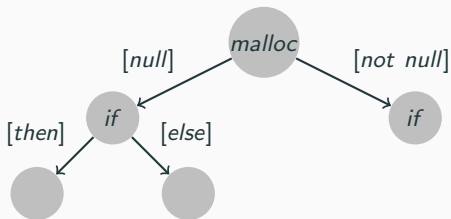


# Example Stack

## 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;
}
```

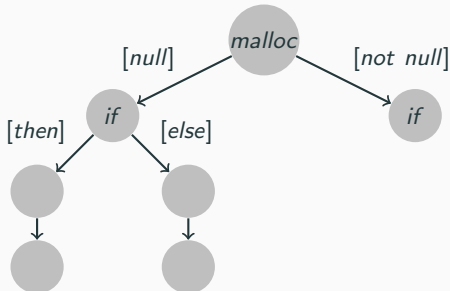


# Example Stack

## 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;
}
```

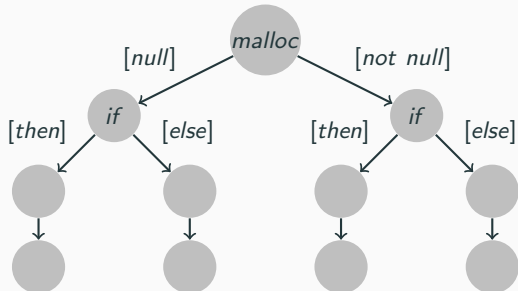


# Example Stack

## 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;
}
```



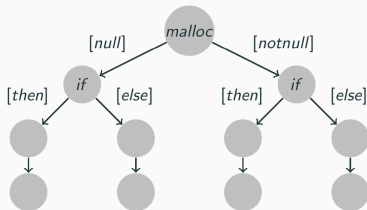


# Proof Tree

## 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;
}
```

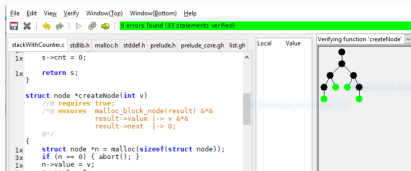
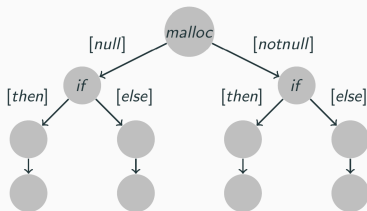


# Proof Tree

## 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;
}
```

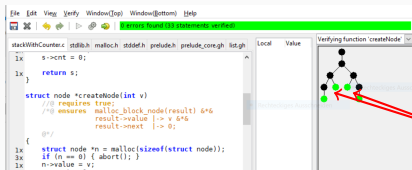
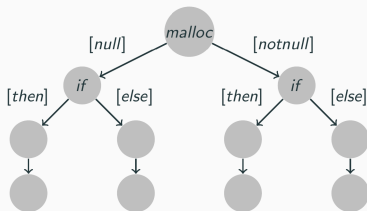


# Proof Tree

## 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;
}
```



VeriFast optimizes  
Proof Tree

# First Impressions From Tool

## Example createNode() with Bug

File Edit View Verify Window(Top) Window(Bottom) Help

No matching heap chunks: node\_value(n,\_) ?

stackWithCounter.c stdlib.h malloc.h stddef.h prelude.h prelude\_core.gh list.gh Local Value

```
if (s == 0) { abort(); }
s->head = 0;
s->cnt = 0;

return s;

struct node *createNode(int v)
  /*@ requires true;
   *@ ensures malloc_block_node(result) &* &
   result->value |-> v &* &
   result->next |-> 0;
  */
  @*/
  {
    struct node *n = malloc(sizeof(struct node));
    //if (n == 0) { abort(); }
    n->value = v;
    n->next = 0;

    return n;
  }

void push(struct stack *s, int v)
```

Verifying function 'createNode'

Local Value

n	n
v	v

Verifying function 'createNode'

Steps

- Verifying function 'createNode'
- Producing assertion
- Executing statement
- Executing statement
- Executing first branch

Assumptions

- true
- n = null\_pointer

Heap chunks

# First Impressions From Tool

## Example createNode() with Bug

File Edit View Verify Window(Top) Window(Bottom) Help

No matching heap chunks: node\_value(n,\_) ?

stackWithCounter.c stdlib.h malloc.h stddef.h prelude.h prelude\_core gh list.gh

```
if (s == 0) { abort(); }
s->head = 0;
s->cnt = 0;

return s;

struct node *createNode(int v)
  /*@ requires true;
   *@ ensures malloc_block_node(result) &&
   result->value |-> v &&
   result->next |-> 0;

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

  return n;
}

void push(struct stack *s, int v)
```

Local Value

n n  
v v

Verifying function 'createNode'

Steps

Verifying function 'createNode'

Producing assertion

Executing statement

Executing statement

Executing first branch

Assumptions

true

n = null\_pointer

Heap chunks

Var-Value Map

Restrictions on Values

# First Impressions From Tool

## Summary

The screenshot displays the VeriFast IDE interface. At the top, a red status bar indicates "No matching heap chunks: node\_value.(n, \_)". The main editor shows the following C code:

```
stackWithCounter.c  stdlib.h  malloc.h  stddef.h  prelude.h  prelude_core.gh  list.gh
if (s == 0) { abort(); }
s->head = 0;
s->cnt = 0;
return s;
}
struct node *createNode(int v)
/*@ requires true;
   *@ ensures malloc_block_node(result) &&
   *@          result->value == v &&
   *@          result->next == 0;
   */
{
  struct node *n = malloc(sizeof(struct node));
  //if (n == 0) { abort(); }
  n->value = v;
  n->next = 0;
  return n;
}
void push(struct stack *s, int v)
```

The "Local Value" pane on the right shows:

```
n  n
v  v
```

The "Verifying function 'createNode'" pane displays a verification diagram with nodes and edges, including a red node at the bottom.

The "Steps" pane at the bottom left shows the execution flow:

- Verifying function 'createNode'
- Producing assertion
- Executing statement
- Executing statement
- Executing first branch

The "Assumptions" pane shows:

```
true
n = null_pointer
```

The "Heap chunks" pane is currently empty.

# First Impressions From Tool

## Summary

The screenshot displays the VeriFast tool interface. At the top, a red error banner reads "No matching heap chunks: node\_value.(n,\_)". The main window shows the source code for a function named 'createNode'. The code includes a loop that aborts if 's == 0', and a 'struct node' definition with fields 'value' and 'next'. The function 'createNode' allocates a node and sets its fields. The 'Local Value' pane on the right shows variables 'n' and 'v'. The 'Verifying function 'createNode'' pane on the right shows a proof tree with nodes represented by black and red circles. The bottom panel shows the 'Steps' pane with 'Producing assertion', 'Executing statement', and 'Executing first branch'. The 'Assumptions' pane shows 'true' and 'n = null\_pointer'. The 'Heap chunks' pane is empty.

+ 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

The screenshot shows the VeriFast IDE interface. At the top, a red status bar displays the message "Assertion might not hold. (Cannot prove false.)". The main editor window shows the following C code:

```
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;
}
```

The line `@ assert false;` is highlighted in yellow. To the right of the code editor, the "Local Value" panel shows the current state of variables: `i` is `(x + 5)`, `res` is `((x + 5) + 5)`, and `x` is `x`. At the bottom of the IDE, there are three panels: "Steps" (showing "Verifying function 'mExactSpe"), "Assumptions" (showing "10 < x"), and "Heap chunks" (empty).

# VeriFast in Action

## assert - Annotation for Debugging

The screenshot shows the VeriFast IDE interface. At the top, a red status bar displays the message "Assertion might not hold. (Cannot prove false.)". The main editor window shows the source code for the function `mExactSpecWithAsserts`. The code includes preconditions `requires x > 10;` and `ensures result == x + 10;`. Inside the function, it declares `int i = x + 5;`, `int res = i + 5;`, and contains three assertions: `assert i == x + 5;`, `assert i > 15;`, and `assert false;`. The `assert false;` line is highlighted in yellow. To the right of the editor, a "Local Value" panel shows the current state of variables: `i (x + 5)`, `res ((x + 5) + 5)`, and `x x`. At the bottom, a "Steps" panel shows "Verifying function 'mExactSpe", an "Assumptions" panel shows "10 < x", and a "Heap chunks" panel is empty.

```
File Edit View Verify Window(Top) Window(Bottom) Help
Assertion might not hold. (Cannot prove false.)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
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;
  }
Local Value
i (x + 5)
res ((x + 5) + 5)
x x
Steps
Verifying function 'mExactSpe
Assumptions
10 < x
Heap chunks
```

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

# VeriFast in Action

## assert - Annotation for Debugging

```
File Edit View Verify Window(Top) Window(Bottom) Help
Assertion might not hold. (Cannot prove false.)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
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;
  }
Local Value
i (x + 5)
res ((x + 5) + 5)
x x
Steps
Verifying function 'mExactSpe
Assumptions
10 < x
Heap chunks
```

Assumption normalizes original expression

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

# VeriFast in Action

## assert - Annotation for Debugging

The screenshot shows the VeriFast IDE interface. At the top, a red status bar displays the message "Assertion might not hold. (Cannot prove false.)". The main editor window shows the source code for the function `mExactSpecWithAsserts`. The code includes preconditions, postconditions, and several assertions. The last assertion, `//@ assert false;`, is highlighted in yellow. To the right of the code editor is a "Local Value" panel showing the current state of variables: `i` is `(x + 5)`, `res` is `((x + 5) + 5)`, and `x` is `x`. At the bottom, there are three panels: "Steps" (Verifying function 'mExactSpe...'), "Assumptions" (10 < x), and "Heap chunks".

```
File Edit View Verify Window(Top) Window(Bottom) Help
Assertion might not hold. (Cannot prove false.)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
Local Value
i (x + 5)
res ((x + 5) + 5)
x x

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;
  }

Steps
Verifying function 'mExactSpe

Assumptions
10 < x

Heap chunks
```

+ 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

The screenshot displays the VeriFast IDE interface. At the top, a red status bar indicates a verification failure: "Cannot prove condition. (Cannot prove 20 < (x + 5).)".

The main editor shows the following C code:

```
int mIf(int x)
//@ requires x > 10;
//@ ensures result > 20;
{
    int i = x + 5;
    if (i < 20) {
        return i+5;
    } else {
        //return i+1; // success
        return i; // failure
    }
}
```

Below the code editor, there are three panels:

- Steps:** Shows the current step as "Producing assertion".
- Assumptions:** Lists the assumptions  $10 < x$  and  $\neg((x + 5) < 20)$ .
- Heap chunks:** Currently empty.

On the right side, a window titled "Verifying function 'mIf'" displays a proof tree. The tree has a root node (black), two children (black), and three leaf nodes (green, black, red). A button labeled "Rechteckiges Auswählen" is visible below the tree.

Local	Value
result	(x + 5)
x	x



# VeriFast in Action

## if - Statement Splits Proof-Tree

The screenshot shows the VeriFast IDE interface. At the top, a red error banner reads "Cannot prove condition. (Cannot prove 20 < (x + 5).)". The main editor displays the following C code:

```
int mIf(int x)
  //@ requires x > 10;
  //@ ensures result == 20;
  {
    int i = x + 5;
    if (i < 20) {
      return i+5;
    } else {
      //return i+1; // success
      return i; // failure
    }
  }
}
```

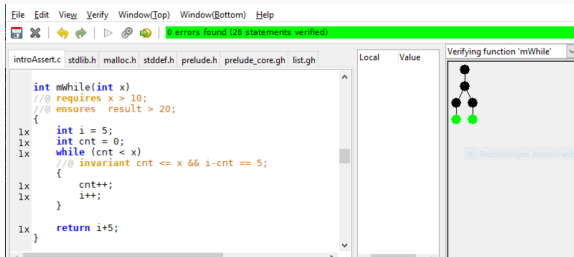
Below the code, the "Steps" panel shows "Verifying function 'mIf'" and "Producing assertion". The "Assumptions" panel shows "10 < x" and "!(x + 5) < 20". The "Heap chunks" panel is empty.

On the right, the "Verifying function 'mIf'" panel displays a proof tree diagram. The root node is black. It has two children: a green node on the left and a red node on the right. A red arrow points from the text "Split for 'then' and 'else'" to the red node.

Split for 'then'  
and 'else'

# VeriFast in Action

## while - Statement Requires Invariant



The screenshot shows the VeriFast IDE interface. At the top, a green status bar indicates "0 errors found (28 statements verified)". The main editor displays the following C code:

```
int mWhile(int x)
  //@ requires x > 10;
  //@ ensures result > 20;
{
  1x int i = 5;
  1x int cnt = 0;
  1x while (cnt < x)
    //@ invariant cnt <= x && i-cnt == 5;
    {
      1x cnt++;
      1x i++;
    }
  1x return i+5;
}
```

On the right side, a window titled "Verifying function 'mWhile'" shows a control flow graph (CFG) with five nodes. The top node is black, and the two nodes at the bottom are green, indicating they have been successfully verified. Below the graph, there is a button labeled "Rechteckiges Ausschneiden".

# VeriFast in Action

## while - Statement Requires Invariant

The screenshot shows the VeriFast IDE interface. The main window displays the source code for a function named `mWhile`. The code includes preconditions, an invariant, and a loop body. The IDE status bar indicates that 28 statements were verified. On the right side, a 'Verifying function 'mWhile'' panel shows a verification tree with nodes representing different parts of the code. A red arrow points to a specific node in the tree.

```
File Edit View Verify Window(Top) Window(Bottom) Help
0 errors found (28 statements verified)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh Local Value Verifying function 'mWhile'
int mWhile(int x)
  //@ requires x > 10;
  //@ ensures result > 20;
{
  1x int i = 5;
  1x int cnt = 0;
  1x while (cnt < x)
    //@ invariant cnt <= x && i - cnt == 5;
    {
      cnt++;
      i++;
    }
  1x return i+5;
}
```

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

# VeriFast in Action

## 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;
}
```

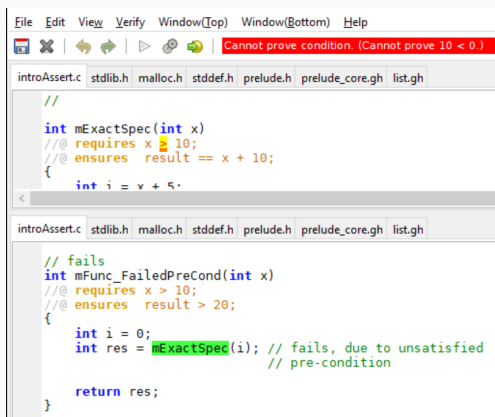
```
int mAbstractSpec(int x)
//@ requires x > 10;
//@ ensures result > 20;
{
    int i = x + 5;
    int res = i + 5;

    return res;
}
```



# VeriFast in Action

## Function Call - Precondition is Checked



The screenshot shows the VeriFast IDE interface. At the top, a red error banner reads "Cannot prove condition. (Cannot prove 10 < 0.)". Below this, the code editor displays two functions. The first function, `mExactSpec`, has a precondition `requires x > 10;` and an assertion `ensures result == x + 10;`. The second function, `mFunc_FailedPreCond`, has a precondition `requires x > 10;` and an assertion `ensures result > 20;`. Inside `mFunc_FailedPreCond`, the variable `i` is set to 0, and `mExactSpec(i)` is called. A green highlight is under `mExactSpec(i)`, and a comment indicates it fails due to an unsatisfied precondition. The IDE tabs at the top include `introAssert.c`, `stdlib.h`, `malloc.h`, `stddef.h`, `prelude.h`, `prelude_core.gh`, and `list.gh`.

```
File Edit View Verify Window(Top) Window(Bottom) Help
Cannot prove condition. (Cannot prove 10 < 0.)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
//
int mExactSpec(int x)
/*@ requires x > 10;
   @ ensures result == x + 10;
 */
{
    int i = x + 5;
}
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
// fails
int mFunc_FailedPreCond(int x)
/*@ requires x > 10;
   @ ensures result > 20;
 */
{
    int i = 0;
    int res = mExactSpec(i); // fails, due to unsatisfied
                           // pre-condition
    return res;
}
```

# VeriFast in Action

## Function Call - Precondition is Checked

```
File Edit View Verify Window(Top) Window(Bottom) Help
Cannot prove condition. (Cannot prove 10 < 0.)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
//
int mExactSpec(int x)
/*@ requires x > 10;
    @ ensures result == x + 10;
 */
{
    int i = x + 5;
}

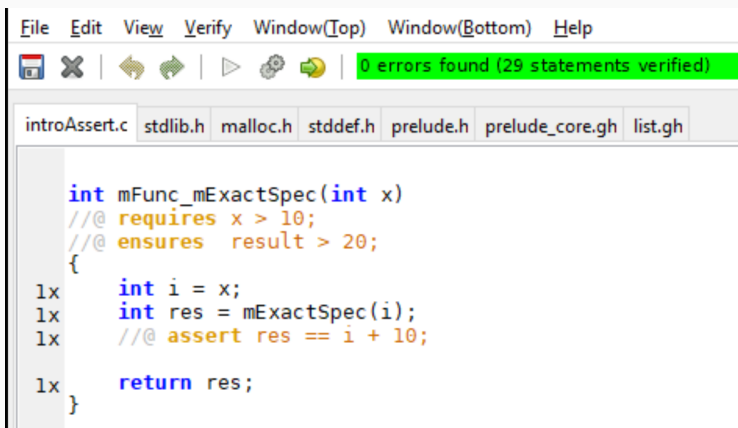
// fails
int mFunc_FailedPreCond(int x)
/*@ requires x > 10;
    @ ensures result > 20;
 */
{
    int i = 0;
    int res = mExactSpec(i); // fails, due to unsatisfied
                           // pre-condition
    return res;
}
```

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



The screenshot shows the VeriFast IDE interface. At the top, a menu bar includes File, Edit, View, Verify, Window(Top), Window(Bottom), and Help. Below the menu is a toolbar with icons for file operations, navigation, and a status bar that reads "0 errors found (29 statements verified)". The main editor window shows the file "introAssert.c" with several tabs for other files: stdlib.h, malloc.h, stddef.h, prelude.h, prelude\_core.gh, and list.gh. The code in the editor is as follows:

```
int mFunc_mExactSpec(int x)
//@ requires x > 10;
//@ ensures  result > 20;
{
1x   int i = x;
1x   int res = mExactSpec(i);
1x   //@ assert res == i + 10;

1x   return res;
}
```

# VeriFast in Action

## Function Call - Postcondition is Only Knowledge

```
File Edit View Verify Window(Top) Window(Bottom) Help
Assertion might not hold. (Cannot prove res = (x + 10).)
introAssert.c stdlib.h malloc.h stddef.h prelude.h prelude_core.gh list.gh
// fails
int mFunc_mAbstractSpec(int x)
//@ requires x > 10;
//@ ensures result > 20;
{
    int i = x;
    int res = mAbstractSpec(i);
    // @ assert res == i + 10; // fails
    return res;
}
Verifying function 'mFunc_mAbstractSpec'
Producing assertion
Executing statement
Assumptions
10 < x
20 < res
Heap
```

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

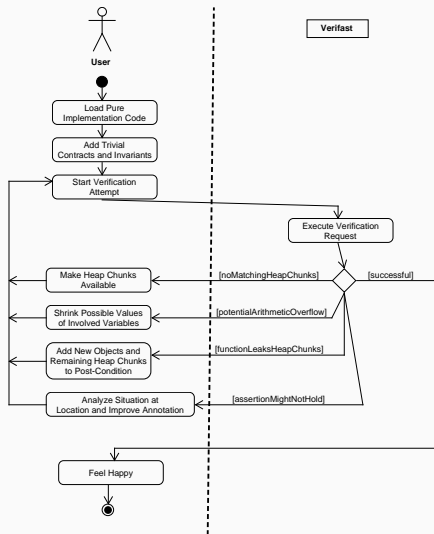
    return res;
}
```

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.**