Formal Methods for C

Seminar – Summer Semester 2014

Daniel Dietsch, Sergio Feo Arenis, Marius Greitschus, Bernd Westphal
Formal Methods
Once Upon a Time...
Provably Correct vs. Testing: Pocket Calculator

```
12345678 + 27
```

```
7 8 9 0 4 5 6 +
1 2 3 =
```
Requirement:
If $x$, $+$, and $y$ are displayed then after pressing $=$, the sum of $x$ and $y$ is displayed if $x + y$ has at most 8 digits, otherwise “-E-” is displayed.
Requirement: If $x$, $+$, and $y$ are displayed then after pressing $=\;\;$, the sum of $x$ and $y$ is displayed if $x + y$ has at most 8 digits, otherwise “-E-” is displayed.
• Requirement:
  If \( x \), \( + \), and \( y \) are displayed then after pressing \( = \),
  the sum of \( x \) and \( y \) is displayed if \( x + y \) has at most 8 digits,
  otherwise “-E-” is displayed.
Requirement:
If $x$, $+$, and $y$ are displayed then after pressing $=$, the sum of $x$ and $y$ is displayed if $x + y$ has at most 8 digits, otherwise “-E-” is displayed.
Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,
- $n + m$, $n$ small, $m$ small (for non error),
- $n + m$, $n$ big, $m$ big (for non error),
- $n + m$, $n$ huge, $m$ small (for error),
- ...
Testing the Pocket Calculator

Test some representatives of “equivalence classes”:

- \( n + 1, \ n \) small,
- \( n + m, \ n \) small, \( m \) small (for non error),
- \( n + m, \ n \) big, \( m \) big (for non error),
- \( n + m, \ n \) huge, \( m \) small (for error),
- ...

\( e.g. \ 27 + 1 \)
\( e.g. \ 13 + 27 \)
\( e.g. \ 12345 + 678 \)
\( e.g. \ 99999999 + 1 \)
Testing the Pocket Calculator

Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,
- $n + m$, $n$ small, $m$ small (for non error),
- $n + m$, $n$ big, $m$ big (for non error),
- $n + m$, $n$ huge, $m$ small (for error),
- ...

- e.g. $27 + 1$
- e.g. $13 + 27$
- e.g. $12345 + 678$
- e.g. $99999999 + 1$
Test some representatives of “equivalence classes”:

- \( n + 1 \), \( n \) small, e.g. \( 27 + 1 \)
- \( n + m \), \( n \) small, \( m \) small (for non error), e.g. \( 13 + 27 \)
- \( n + m \), \( n \) big, \( m \) big (for non error), e.g. \( 12345 + 678 \)
- \( n + m \), \( n \) huge, \( m \) small (for error), e.g. \( 99999999 + 1 \)
- ...

Test some representatives of “equivalence classes”:

- \( n + 1 \), \( n \) small, e.g. \( 27 + 1 \)
- \( n + m \), \( n \) small, \( m \) small (for non error), e.g. \( 13 + 27 \)
- \( n + m \), \( n \) big, \( m \) big (for non error), e.g. \( 12345 + 678 \)
- \( n + m \), \( n \) huge, \( m \) small (for error), e.g. \( 99999999 + 1 \)
- ...
Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,  
  - e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error),  
  - e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error),  
  - e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error),  
  - e.g. $99999999 + 1$
- ...

![Calculator Image]
Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small, e.g. $27 + 1$
- $n + m$, $n$ small, $m$ small (for non error), e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...
Test some representatives of “equivalence classes”:

- \( n + 1 \), \( n \) small,
- \( n + m \), \( n \) small, \( m \) small (for non error),
- \( n + m \), \( n \) big, \( m \) big (for non error),
- \( n + m \), \( n \) huge, \( m \) small (for error),
- ...

\[ 7 \quad 8 \quad 9 \quad 0 \]
\[ 4 \quad 5 \quad 6 \quad + \]
\[ 1 \quad 2 \quad 3 \quad = \]

13023

e.g. 27 + 1
e.g. 13 + 27
e.g. 12345 + 678
e.g. 99999999 + 1
Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,
- $n + m$, $n$ small, $m$ small (for non error),
- $n + m$, $n$ big, $m$ big (for non error),
- $n + m$, $n$ huge, $m$ small (for error),
- ...

e.g. $27 + 1$

e.g. $13 + 27$

e.g. $12345 + 678$

e.g. $99999999 + 1$
Testing the Pocket Calculator

Test some representatives of “equivalence classes”:

- $n + 1$, $n$ small,
- $n + m$, $n$ small, $m$ small (for non error), e.g. $13 + 27$
- $n + m$, $n$ big, $m$ big (for non error), e.g. $12345 + 678$
- $n + m$, $n$ huge, $m$ small (for error), e.g. $99999999 + 1$
- ...
Testing the Pocket Calculator: One More Try

```
1
+ 99999999
7 8 9 0
4 5 6 +
1 2 3 =
```
• Oops...
Behind the Scenes: Test 99999999 + 1 Failed...

```c
int add(int x, int y)
{
    if (y == 1) // be fast
        return ++x;
    else
        return x+y;
}
```
Behind the Scenes: Test 99999999 + 1 Failed...

```c
int add( int x, int y )
{
    if ( y == 1 ) // be fast
        return ++x;
    else
        return x+y;
}
```

- **Tester**: “Hey, you’ve got to care for the 8-digit constraint in line 6!”
Behind the Scenes: Test $99999999 + 1$ Failed...

```c
int add(int x, int y)
{
    if (y == 1) // be fast
        return ++x;
    else
        return x+y;
}
```

- **Tester**: “Hey, you’ve got to care for the 8-digit constraint in line 6!”
- **Programmer**: “Eh, piece of cake. *tippeditipp* Here you are!”
Behind the Scenes: Test 99999999 + 1 Failed...

Tester: “Hey, you’ve got to care for the 8-digit constraint in line 6!”

Programmer: “Eh, piece of cake. *tippeditipp* Here you are!”

```c
int add(int x, int y) {
    if (y == 1) // be fast
        return ++x;
    else
        return x+y;
}
```
Behind the Scenes: Test 99999999 + 1 Failed...

```c
int add( int x, int y )
{
    if ( y == 1 ) // be fast
        return ++x;
    else
        return x+y;
}
```

- **Tester**: “Hey, you’ve got to care for the 8-digit constraint in line 6!”
- **Programmer**: “Eh, piece of cake. *tippeditipp* Here you are!”
- **Tester**: “Fine, all tests passed!”
The Tests Revisited

With our test cases

- $27 + 1$,
- $13 + 27$,
- $12345 + 678$,
- $99999999 + 1$

we have

```c
int add( int x, int y )
{
    if ( y == 1 ) // be fast
        return ++x;
    int r = x + y;
    if ( r > 99999999)
        r = -1;
    return r;
}
```
With our test cases

- $27 + 1$,
- $13 + 27$,
- $12345 + 678$,
- $99999999 + 1$

we have

- 100% statement coverage,

```c
int add( int x, int y )
{
    if (y == 1) // be fast
        return ++x;

    int r = x + y;

    if (r > 99999999)
        r = -1;

    return r;
}
```
With our test cases

- $27 + 1$,
- $13 + 27$,
- $12345 + 678$,
- $99999999 + 1$

we have

- 100% statement coverage,
- 100% branch coverage,
The Tests Revisited

With our test cases

- $27 + 1$,
- $13 + 27$,
- $12345 + 678$,
- $99999999 + 1$

we have

- 100% statement coverage,
- 100% branch coverage,
- 100% condition coverage,

```c
int add( int x, int y )
{
    if ( y == 1 ) // be fast
        return ++x;
    int r = x + y;
    if ( r > 99999999 )
        r = -1;
    return r;
}
```
With our test cases

- $27 + 1$,
- $13 + 27$,
- $12345 + 678$,
- $99999999 + 1$

we have

- 100% statement coverage,
- 100% branch coverage,
- 100% condition coverage,
- ...

and still didn’t spot the bug.

To be sure, we’d need to test all (how many?) combinations – **impractical**!
#define DIGIT_8_MAX 99999999

```c
int add( int x, int y )
{
    int r;

    if ( y == 1 ) // be fast
        r = ++x;
    else {
        r = x + y;

        if ( r > DIGIT_8_MAX )
            r = -1;
    }

    return r;
}
```
(i) A precise (formal) specification:

- $x$ and $y$ are non-negative 8-digit numbers:
  $0 \leq x < 10^8$
  $0 \leq y < 10^8$

- all non-negative returned numbers are 8-digit:
  $r < 10^8$

```c
#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    int r;

    if (y == 1) // be fast
        r = ++x;
    else {
        r = x + y;

        if (r > DIGIT_8_MAX)
            r = -1;
    }

    return r;
}
```
(i) A precise (formal) specification:

- $x$ and $y$ are non-negative 8-digit numbers:
  $0 \leq x < 10^8$
  $0 \leq y < 10^8$

- all non-negative returned numbers are 8-digit:
  $r < 10^8$

(ii) A representation of the specification understood by a verification tool.

```c
#include <stdio.h>

#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    int r;

    if (y == 1) // be fast
        r = ++x;
    else {
        r = x + y;

        if (r > DIGIT_8_MAX)
            r = -1;
    }

    return r;
}
```
What If We Need to Be Sure?

(i) A precise (formal) specification:

- $x$ and $y$ are non-negative 8-digit numbers:
  $0 \leq x < 10^8$
  $0 \leq y < 10^8$
- all non-negative returned numbers are 8-digit:
  $r < 10^8$

(ii) A representation of the specification understood by a verification tool.

```c
#define DIGIT_8_MAX 99999999

int add( int x, int y )
{
    assert(x >= 0);
    assert(x <= DIGIT_8_MAX);
    assert(y >= 0);
    assert(y <= DIGIT_8_MAX);

    int r;

    if ( y == 1 ) // be fast
        r = ++x;
    else {
        r = x + y;
        if ( r > DIGIT_8_MAX )
            r = -1;
    }

    assert(r <= DIGIT_8_MAX);
    return r;
}
```
(i) A precise (formal) specification:
   • \( x \) and \( y \) are non-negative 8-digit numbers:
     \( 0 \leq x < 10^8 \)
     \( 0 \leq y < 10^8 \)
   • all non-negative returned numbers are 8-digit:
     \( r < 10^8 \)

(ii) A representation of the specification understood by a verification tool.

(iii) A verification tool:

```c
#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    assert(x >= 0);
    assert(x <= DIGIT_8_MAX);
    assert(y >= 0);
    assert(y <= DIGIT_8_MAX);

    int r;

    if (y == 1) // be fast
        r = ++x;
    else {
        r = x + y;

        if (r > DIGIT_8_MAX)
            r = -1;
    }

    assert(r <= DIGIT_8_MAX);
    return r;
}
```
% check add.c
line 19: assertion violated%
```
Fix and check the fixed version:

```c
#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    assert(x >= 0);
    assert(x <= DIGIT_8_MAX);
    assert(y >= 0);
    assert(y <= DIGIT_8_MAX);

    int r;

    if (y == 1) // be fast
        r = ++x;
    else
        r = x + y;

    if (r > DIGIT_8_MAX)
        r = -1;

    assert(r <= DIGIT_8_MAX);
    return r;
}
```

% check add.c verification succeeded

Fix and check the fixed version:

```c
#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    assert(x >= 0);
    assert(x <= DIGIT_8_MAX);
    assert(y >= 0);
    assert(y <= DIGIT_8_MAX);

    int r;

    if (y == 1) // be fast
        r = ++x;
    else
        r = x + y;

    if (r > DIGIT_8_MAX)
        r = -1;

    assert(r <= DIGIT_8_MAX);
    return r;
}
```

How is this possible?

Subject of the seminar!
What If We Need to Be Sure?

- Fix and check the fixed version:

```c
#define DIGIT_8_MAX 99999999

int add(int x, int y)
{
    assert(x >= 0);
    assert(x <= DIGIT_8_MAX);
    assert(y >= 0);
    assert(y <= DIGIT_8_MAX);

    int r;

    if (y == 1) // be fast
        r = ++x;
    else
        r = x + y;

    if (r > DIGIT_8_MAX)
        r = -1;

    assert(r <= DIGIT_8_MAX);
    return r;
}
```

- How is this possible?
- Subject of the seminar!

- Alternative outcome:

```c
% check add.c
verification succeeded
%
% check add.c
out of memory
%
```

- None the wiser...
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.
(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.
(iii) At best: an implementation of that algorithm.
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.
(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.
(iii) At best: an implementation of that algorithm.

Are we really sure then? – “There is no silver bullet” (surprise):
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.

(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.

(iii) At best: an implementation of that algorithm.

Are we really sure then? – “There is no silver bullet” (surprise):

- The requirements specification may upfront be wrong.
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.
(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.
(iii) At best: an implementation of that algorithm.

Are we really sure then? – “There is no silver bullet” (surprise):

- The requirements specification may upfront be wrong.
- The tool output may be interpreted in a wrong way.
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.
(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.
(iii) At best: an implementation of that algorithm.

Are we really sure then? – “There is no silver bullet” (surprise):

- The requirements specification may upfront be wrong.
- The tool output may be interpreted in a wrong way.
- The tool may have bugs or run on buggy hardware.
- ...

Bottom Line: Formal Methods for C
A working definition for “formal methods for C”:

(i) A precise, formal, mathematical requirements specification.
(ii) An algorithm which is able to prove or disprove for a given piece of C code whether it satisfies the specification.
(iii) At best: an implementation of that algorithm.

Are we really sure then? – “There is no silver bullet” (surprise):

- The requirements specification may upfront be wrong.
- The tool output may be interpreted in a wrong way.
- The tool may have bugs or run on buggy hardware.
- ...
- For production, the program may be compiled with a buggy compiler.
- ...
(Anticipated) Benefits

- Increased confidence.
- Sometimes reduced overall costs: “find errors early”, despite additional costs for formalisation.
Anticipated) Benefits

• Increased confidence.

• Sometimes reduced overall costs: “find errors early”, despite additional costs for formalisation.

Possible motivations:

• Loss of lives: aerospace, railway, automotive, fire alarm, ...

• Loss of health: medical devices, ...

• Loss of privacy: encryption protocols, ...

• Loss of money: satellites, factory automation, ...

• ...
(Anticipated) Benefits

- **Increased confidence.**
- Sometimes **reduced overall costs**: “find errors early”, despite **additional costs** for formalisation.

**Possible motivations:**

- **Loss of lives**: aerospace, railway, automotive, fire alarm, ...
- **Loss of health**: medical devices, ...
- **Loss of privacy**: encryption protocols, ...
- **Loss of money**: satellites, factory automation, ...
- ...

Errors sometimes already avoided by formalising requirements – e.g. “Heartbleed” could possibly have been avoided if RFC 6520 stated

\[
\text{A heartbeat protocol message is valid if and only if} \\
\ldots \land M.\text{payload\_length} = \text{length}(M.\text{payload}) \land \ldots
\]

*Not valid* messages **MUST** be discarded.
The Seminar
Seminar...?
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.

- **Choose** a verification tool from the list (or propose your own).
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.
- **Choose** a verification tool from the list (or propose your own).
- **Thread 1**: Literature research, what’s the theory behind the tool?
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.
- **Choose** a verification tool from the list (or propose your own).
- **Thread 1**: Literature research, what’s the theory behind the tool?
- **Thread 2**: Get your hands dirty.
  - get acquainted with the tool on the VM (“Hi tool, nice to meet you!”)
  - reproduce and understand the tool provider’s favourite example(s)
  - show one more property in that example, find one more bug in that (possibly reasonably modified) example
  - see how the tool does on these three examples:
    - `scan_ushort()`
    - low battery monitor – programming task
    - a big example
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.
- **Choose** a verification tool from the list (or propose your own).
- **Thread 1**: Literature research, what’s the theory behind the tool?
- **Thread 2**: Get your hands dirty.
The Task

- **Attend** the 2-3 introductory lectures on C and formal methods basics.
- **Choose** a verification tool from the list (or propose your own).
- **Thread 1**: Literature research, what’s the theory behind the tool?
- **Thread 2**: Get your hands dirty.
- **Present**: Block-Seminar, 30 min. (?) presentation with
  - tool name, brief history, etc.
  - what are the tool’s capabilities?
  - what’s the theory behind the tool?
  - how did the tool perform on the examples?
  - conclusion
  and participation in discussion after talk.


**Grade:** $r \cdot b \cdot (0.3 \cdot S + 0.7 \cdot T)$

- $r \in \{0, 1\}$: repeatability package* (RP) for favourite example
- $b \in \{0, 1\}$: low battery monitor, not obviously broken
- $S \in \{1.0, \ldots, 4.0, 6.0\}$: talk structure
- $T \in \{1.0, \ldots, 4.0, 6.0\}$: presentation (incl. RP for three examples)

**Deadlines:**

- 30.6.2014: “theory behind the tool” part of the talk
- 14.7.2014: talk structure
- **tba**: presentation

*: shell script, Makefile, etc. which produces the results reported on in the talk by running the chosen verification tool on the examples with necessary parameters etc.
Talk Structure

- Formal Methods for C Kickoff
  - Introduction, ca. 10 Slides
  - Formal Methods, ca. 3 Slides
  - Formalia, ca. 3 Slides
Talk Structure

- Formal Methods for C Kickoff
  - Introduction, ca. 10 Slides
  - Formal Methods, ca. 3 Slides
  - Formalia, ca. 3 Slides
Talk Structure Example

- **Formal Methods for C Kickoff**
  **Goal**: give sufficient information for semester planning regarding workload, i.e. sketch goals and content, fix requirements, discuss grading, agree on common language

- **Introduction (ca. 10 Slides)**
  **Goal**: point out difference between testing and verification
  - little story on pocket calculator: show a bug which happens to be missed by tests
  - give example for a proper formal requirement on pocket calculator, say how verification would be used given the C code

- **Formal Methods (ca. 3 Slides)**
  **Goal**: agree on common understanding of “formal methods”, give outlook on motivation for their use and their limitations
  - working definition: formal requirements, prove/disprove algorithm, tool
  - limitations: e.g. bugs in checking tool
  - benefits: increased confidence, maybe lower overall cost
  - motivation: safety critical domain (transport, health, ...)

- **Formalia (ca. 3 Slides)**
  **Goal**: agree on expected work, propose schedule and deadlines
  - firstly the C seminar, then choose a tool
  - then literature research and hands-on experience (two threads)
  - hands-on experience: tool’s favourite example and three given ones
  - finally, block seminar; sketch expected content of talk
  - clarify “structure” using bad/good example
Plan Proposal

- check the VM and the homepage for the offered tools/topics
- decide until next week: favourite (and second best) topic
- now: decide for meeting time(s) for introductory lecture
- next meeting: assign topics (and supervisor)