Testing

Often the only feasible option

Required for software certification

Testing

Cannot show correctness.

Cheap

What is a test?

- It is like an experiment:
  - Initial state
  - Test logic
  - Expected state vs. real state

How good are my test cases?

- If they find bugs they are good for sure
- Otherwise:
  - Need to find a measure of the likelihood of finding bugs
  - E.g. “All statement are exercised at least once”
    - Statement coverage
  - E.g. “All branches are exercised both when their guard is true and when it is false”
    - Branch coverage

Theorem proving

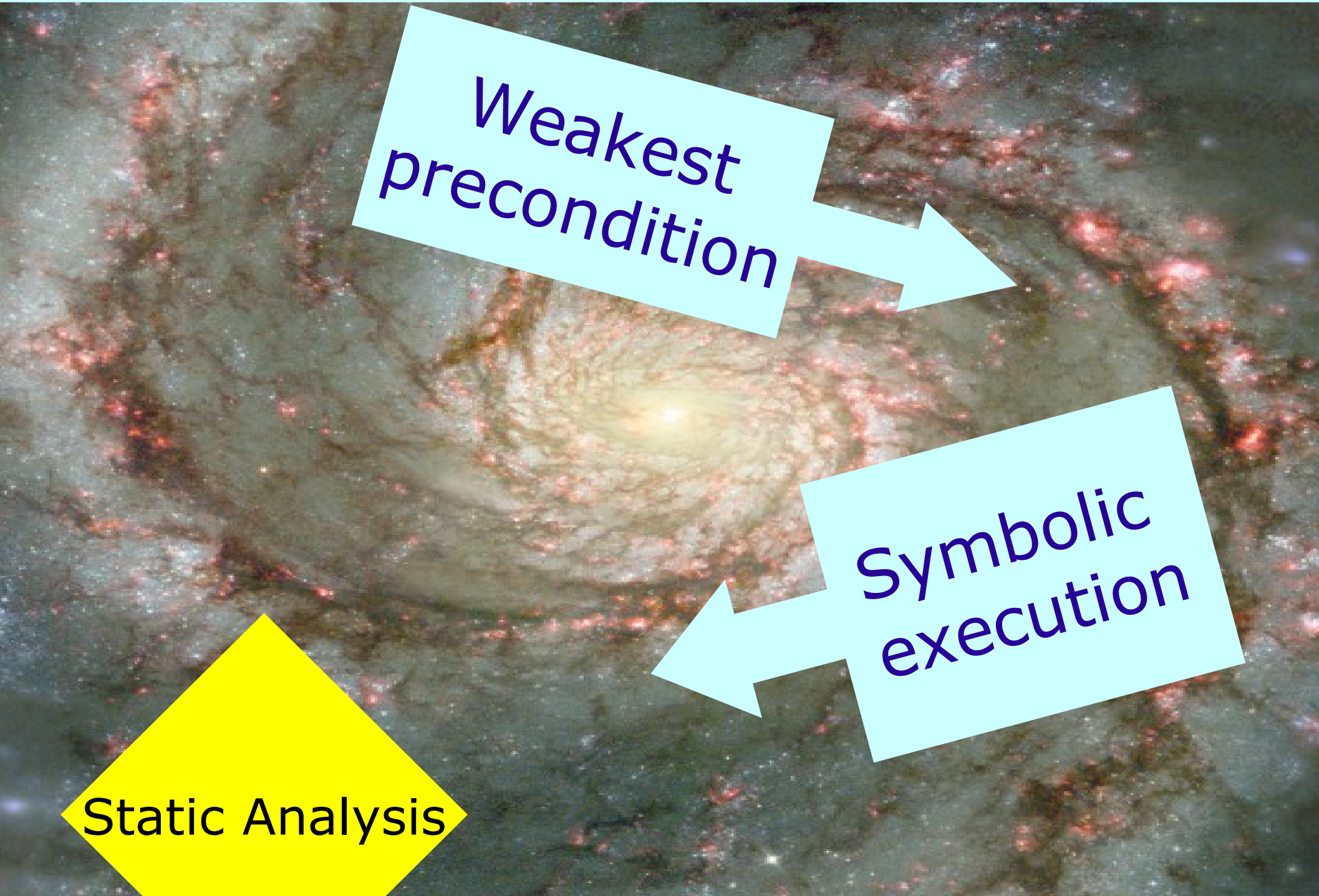
FPhile

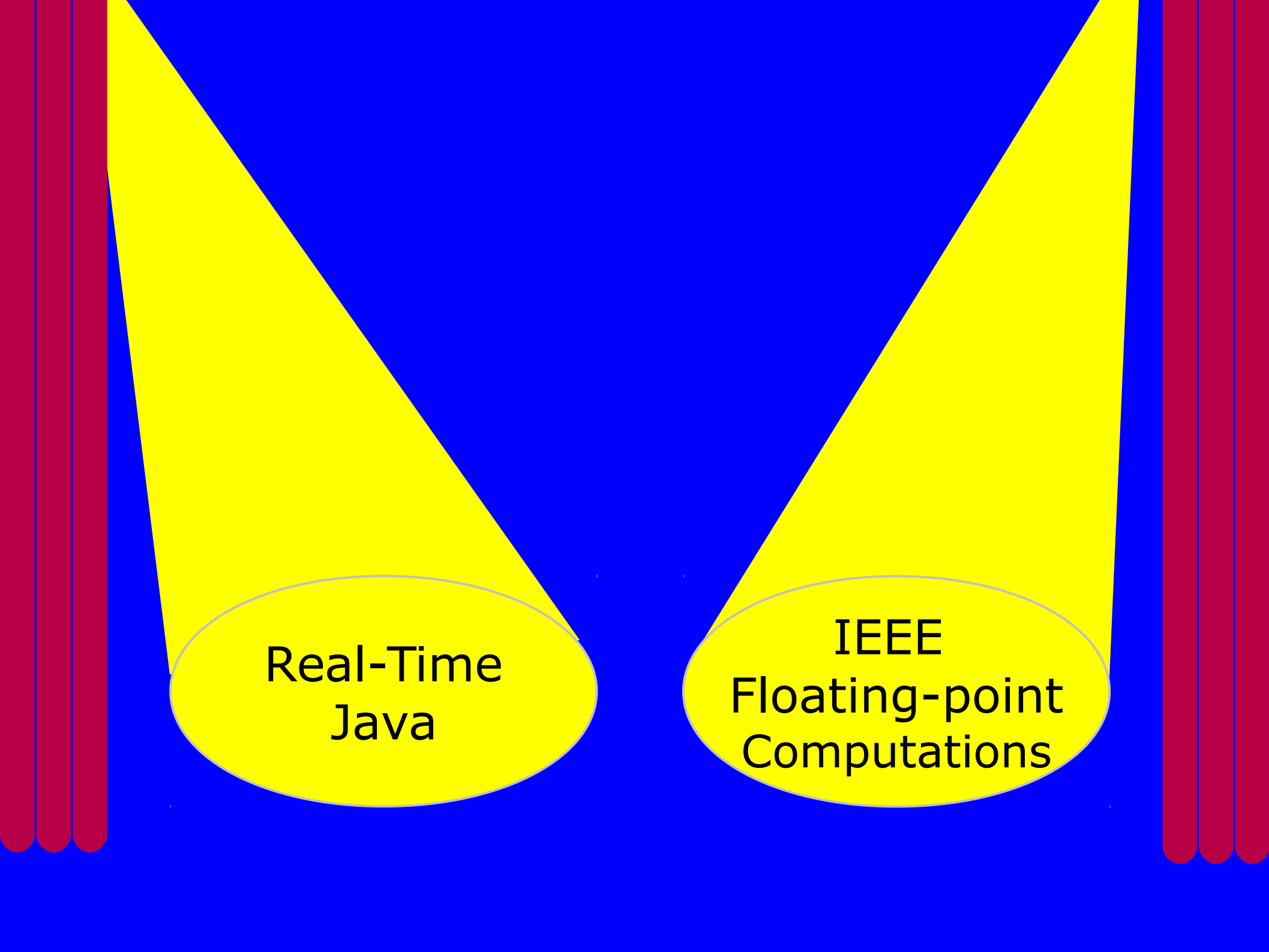
Formal specification

JML

Java Modelling Language

SW Formal Verification Techniques Galaxy



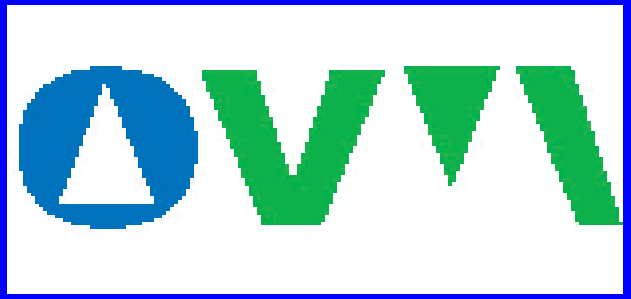

Real-Time
Java
IEEE
Floating-point
Computations


Java


Java

Real-Time Java Virtual Machines that implement the Real-Time Java Specification

Real-Time Java Virtual Machines



Floating point numbers

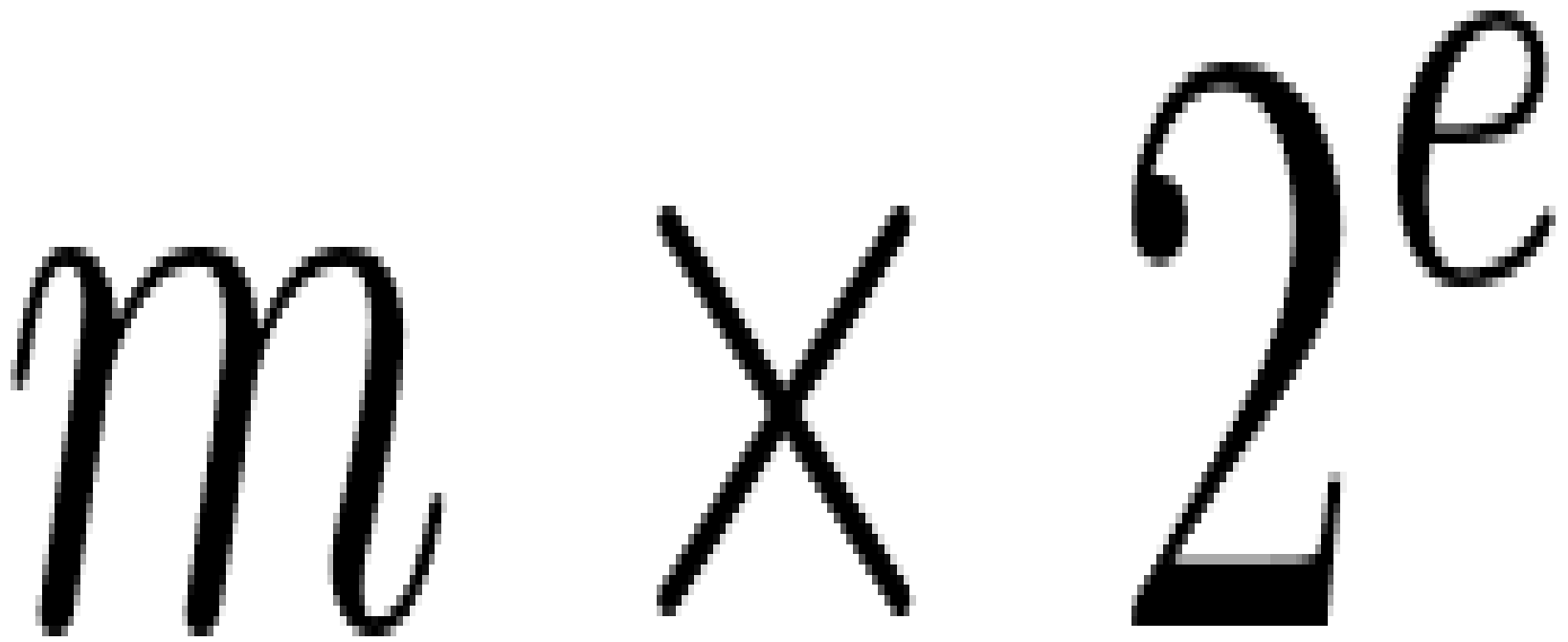
Approximation of reals
in scientific notation

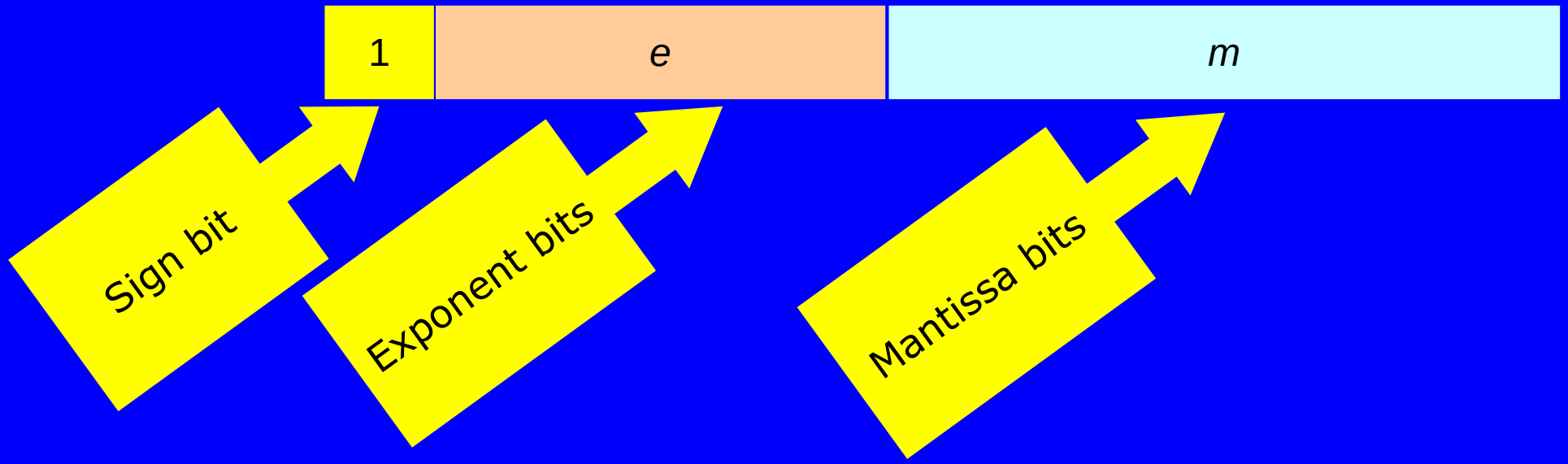
$$m \times \beta^e$$

m antissa (decimal)

β ase (integers)

e xponent (integers)





Number of bits

=

Width

=

Precision

Real-time Java Specifications for High Coverage Test Generation

Wolfgang Ahrendt Wojciech Mostowski Gabriele Paganelli

In: *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and Embedded Systems*,
JTRES 2012

ACM 2012

Contributions

Formalisation of Real-Time Specification for Java (RTSJ)

A test-case generator (KeYTestGen) using formal specification and source code

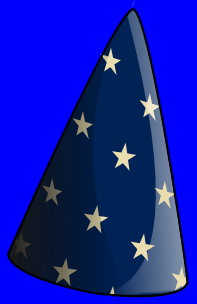
Test industrial code using KeYTestGen and formal specification

KeYTestGen

Symbolic Execution

Constraint solving

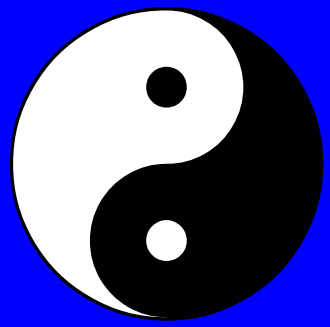
Test **code** generation



Java+Specification

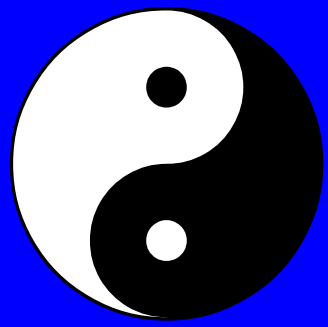
Runnable Test Suite

Symbolic execution



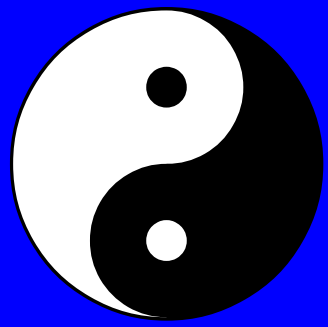
- Execution of a program with symbolic values
- all executions (runs) can be expressed

Symbolic execution



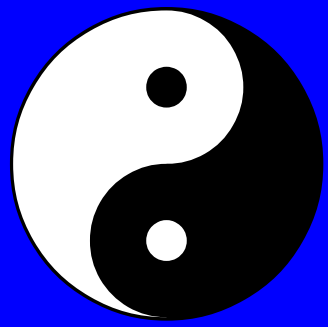
- It is similar to developing an algebraic expression with literals
  - *a*(b+c)* →

Symbolic execution



- It is similar to developing an algebraic expression with literals
  - $a*(b+c) \rightarrow a*b + a*c$

Symbolic execution



- It is similar to developing an algebraic expression with literals
  - $a*(b+c) \rightarrow a*b + a*c$
- One can substitute *a,b,c* with any value (e.g. in integers)
  - The result will still be correct

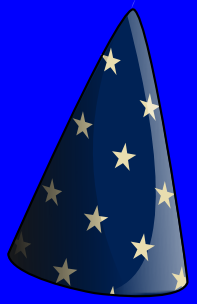
KeYTestGen

Sets of constraints
Describing paths
inside the code

Symbolic
Execution

Constraint
solving

Test **code**
generation



Java+Specification

**Runnable
Test
Suite**

KeYTestGen

- Based on KeY, a theorem prover for **dynamic logic (DL)**
  - A DL formula is built from specification+code

Pre-condition

```
/*@
  requires t > 0;
  ensures x+y > t   ==> \result == t;
  ensures x+y <= t ==> \result == x+y;
@*/
```

Post-condition

```
public int saturation(int x, int y, int t){
   x = x+y;
   if(x > t){return t;}
   else      {return x;}
}
```

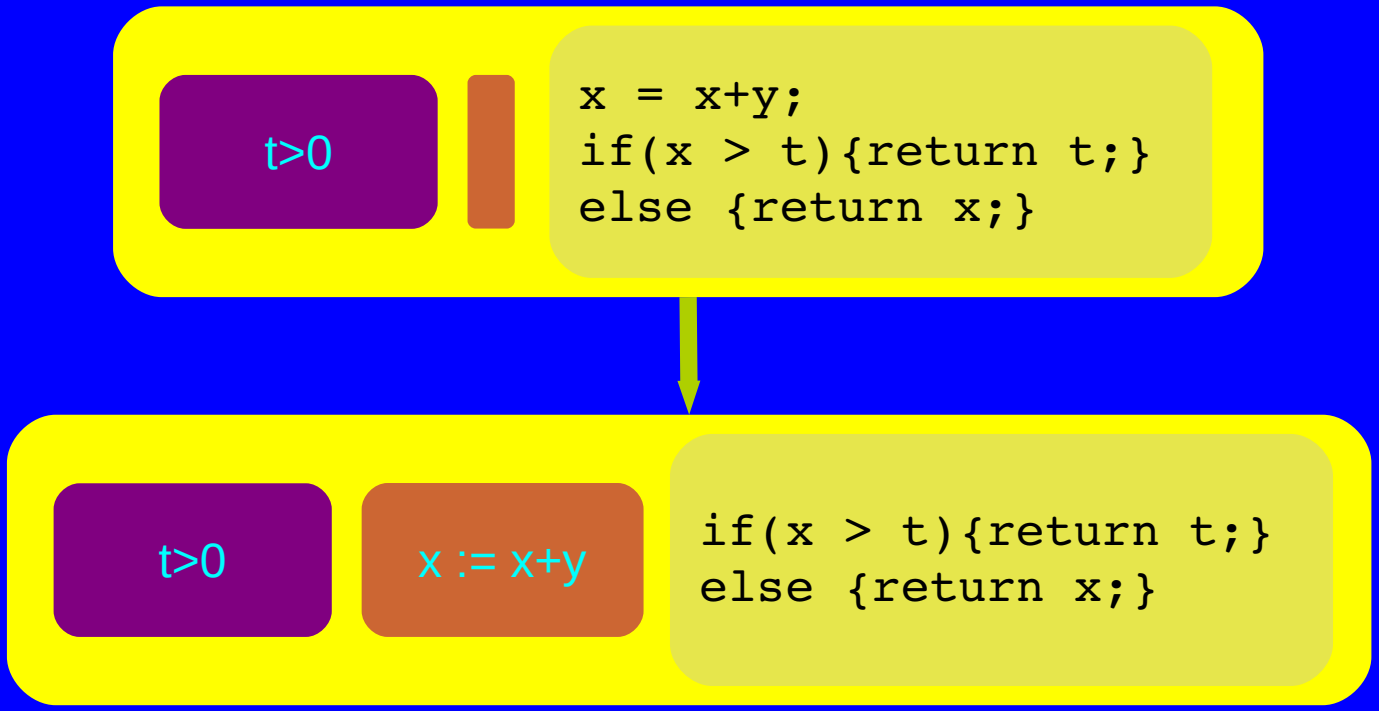
```
requires t > 0;
```

t>0

```
x = x+y;
if(x > t){return t;}
else {return x;}
```

t>0

```
x = x+y;
if(x > t){return t;}
else {return x;}
```


t>0
x = x+y;
if(x > t){return t;}
else {return x;}
t>0
x := x+y
if(x > t){return t;}
else {return x;}

t>0

```
x = x+y;
if(x > t){return t;}
else {return x;}
```

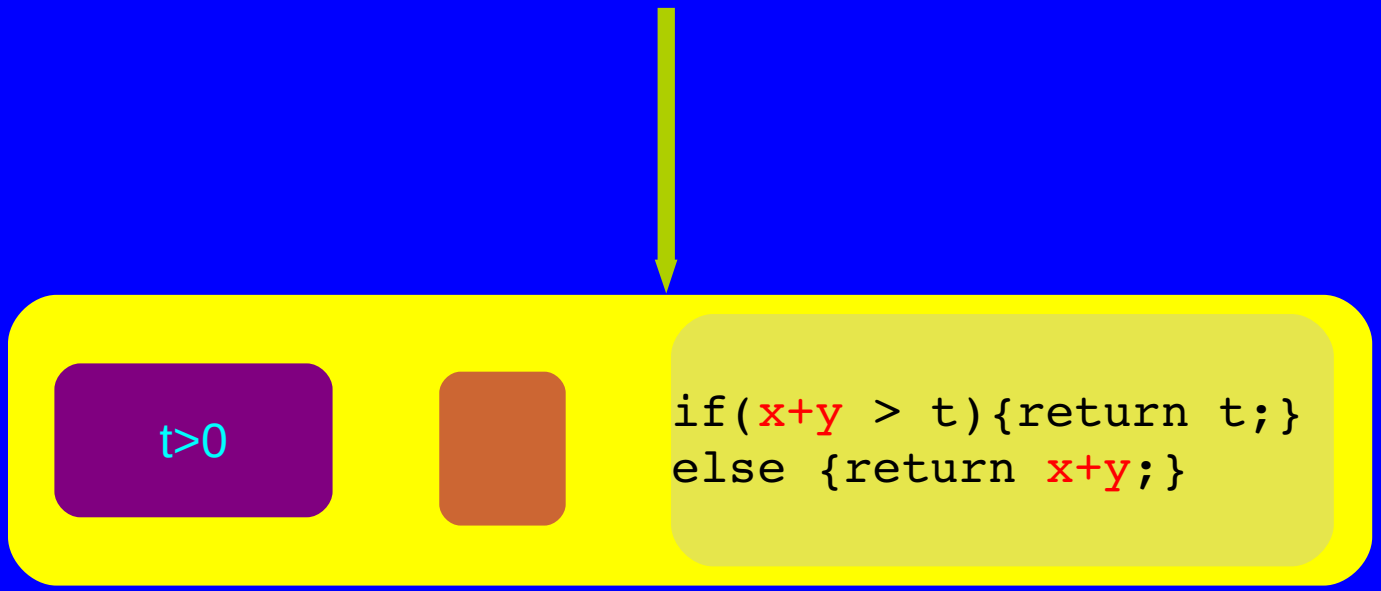
t>0

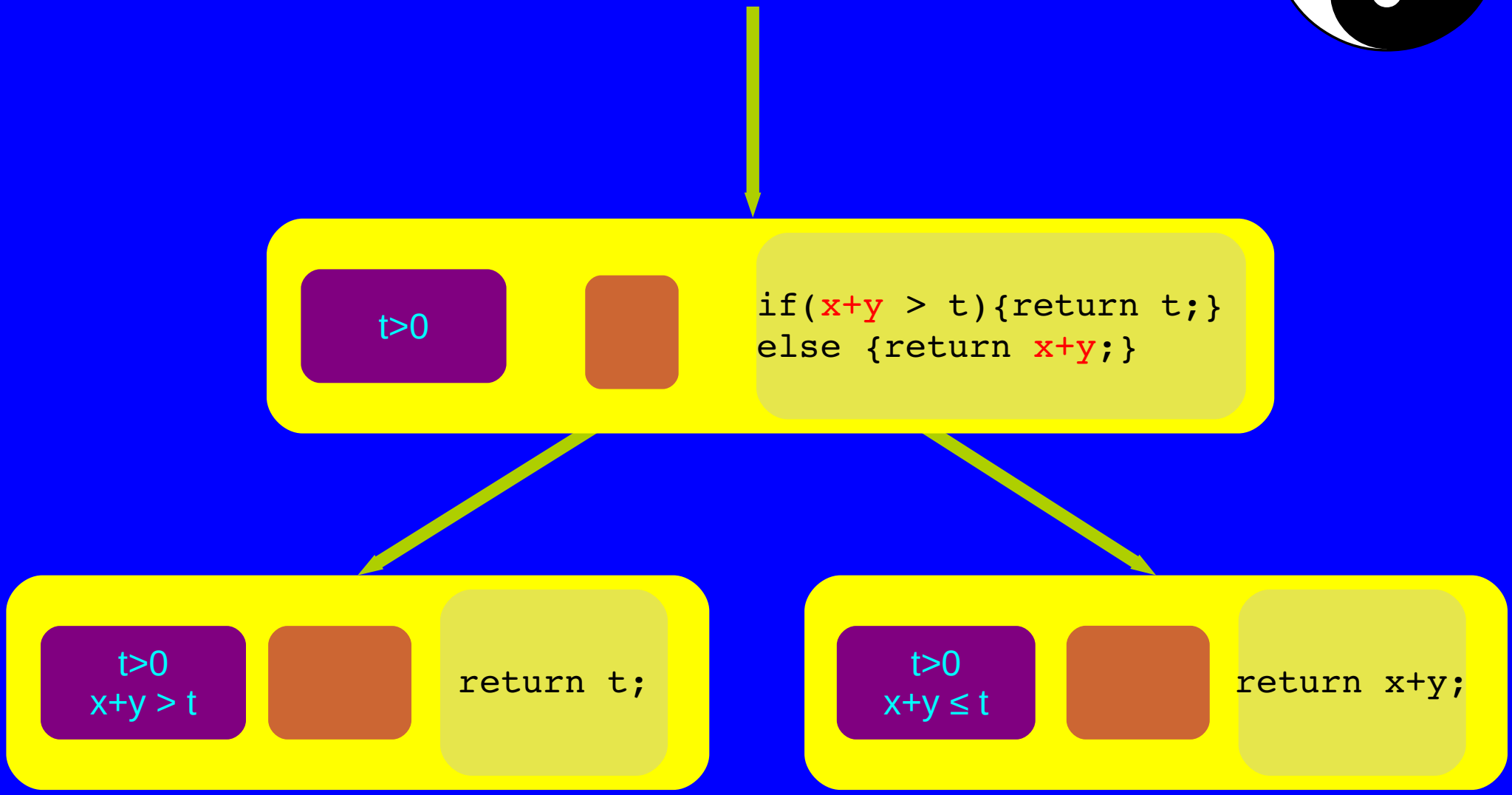
x := x+y

```
if(x > t){return t;}
else {return x;}
```

t>0

```
if(x+y > t){return t;}
else {return x+y;}
```


t>0
if(x+y > t){return t;}
else {return x+y;}


t>0
if(x+y > t){return t;}
else {return x+y;}
t>0
x+y > t
return t;
t>0
x+y ≤ t
return x+y;

KeYTestGen

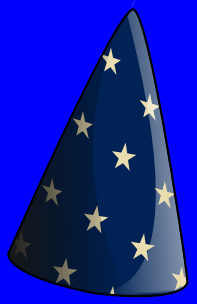
Sets of constraints Describing paths inside the code

Concrete values: Test inputs

Symbolic Execution

Constraint solving

Test **code** generation



Java+Specification

Runnable Test Suite

KeYTestGen

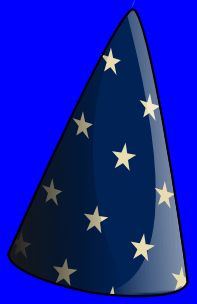
Sets of constraints Describing paths inside the code

Concrete values: Test inputs

Symbolic Execution

Constraint solving

Test **code** generation

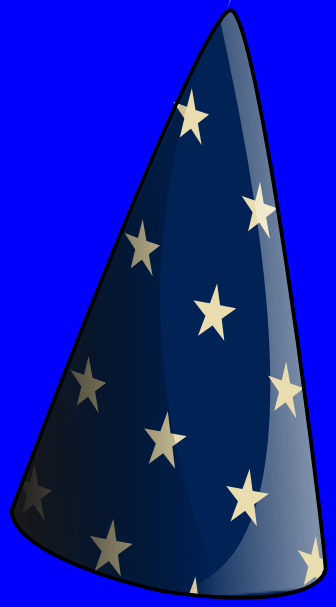


Java+Specification

Postcondition:

decides test pass/fail

Runnable Test Suite



Java+Specification

Postcondition:

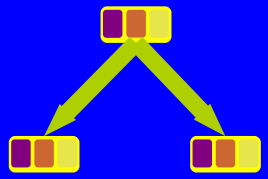
decides test pass/fail

Runnable Test Suite

```
ensures x+y > t  ==> \result == t;
ensures x+y <= t ==> \result == x+y;
```

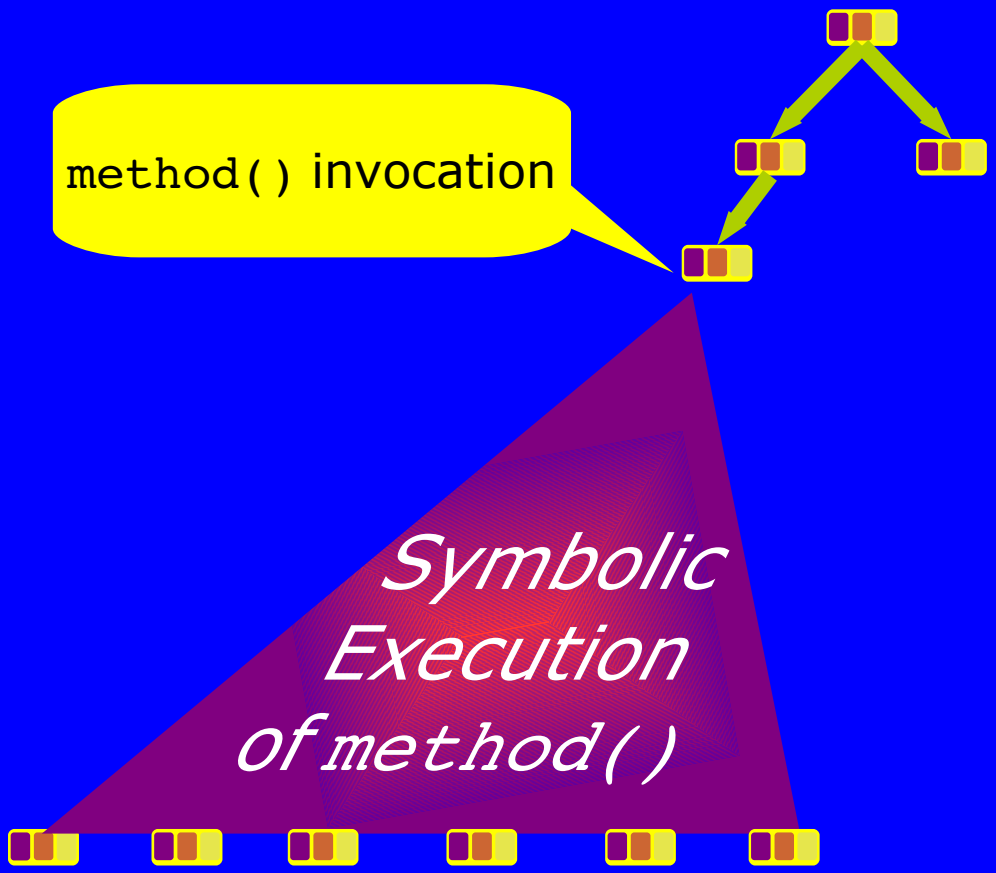
Specification matters

Specification matters

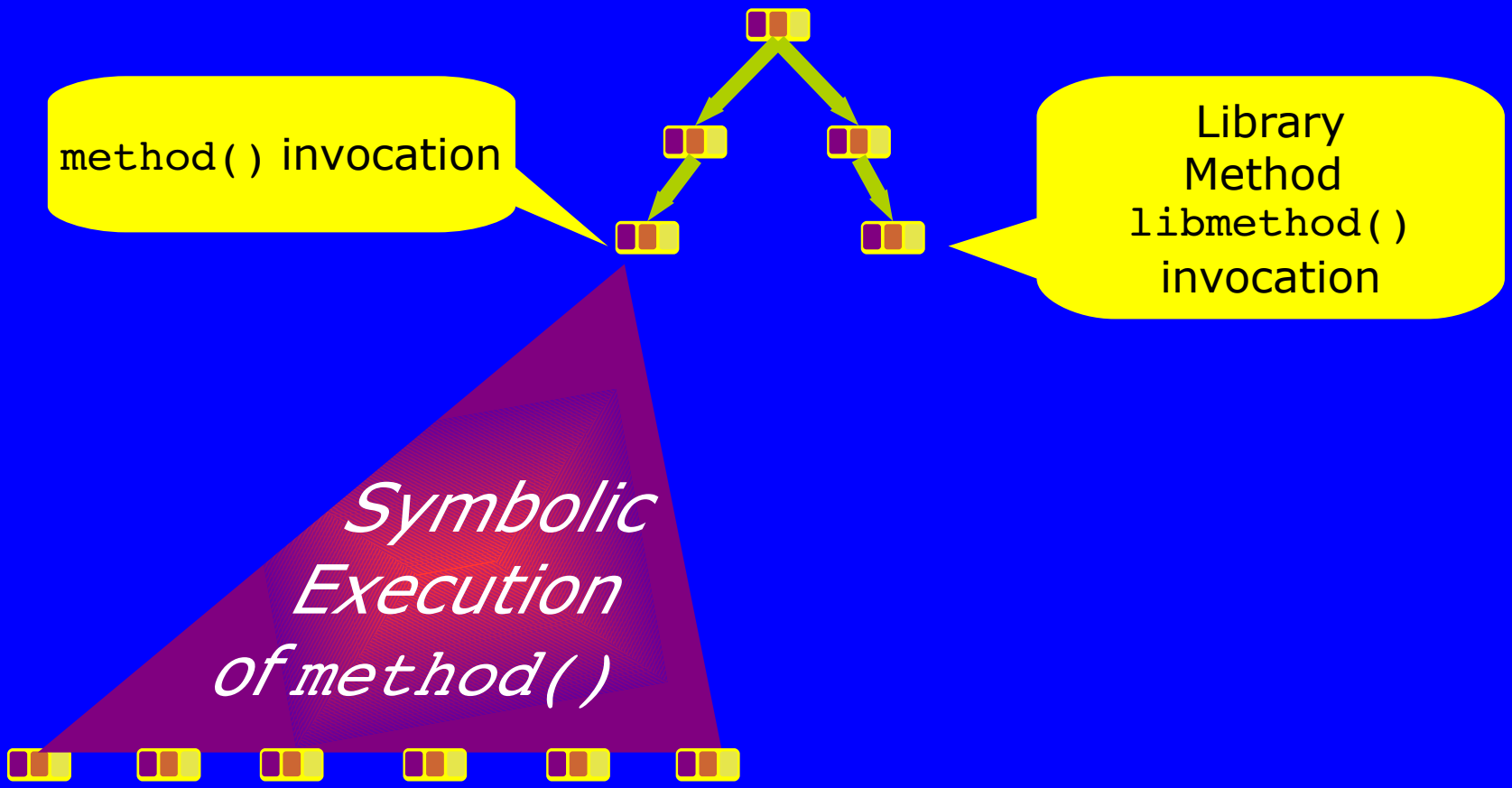


Specification matters

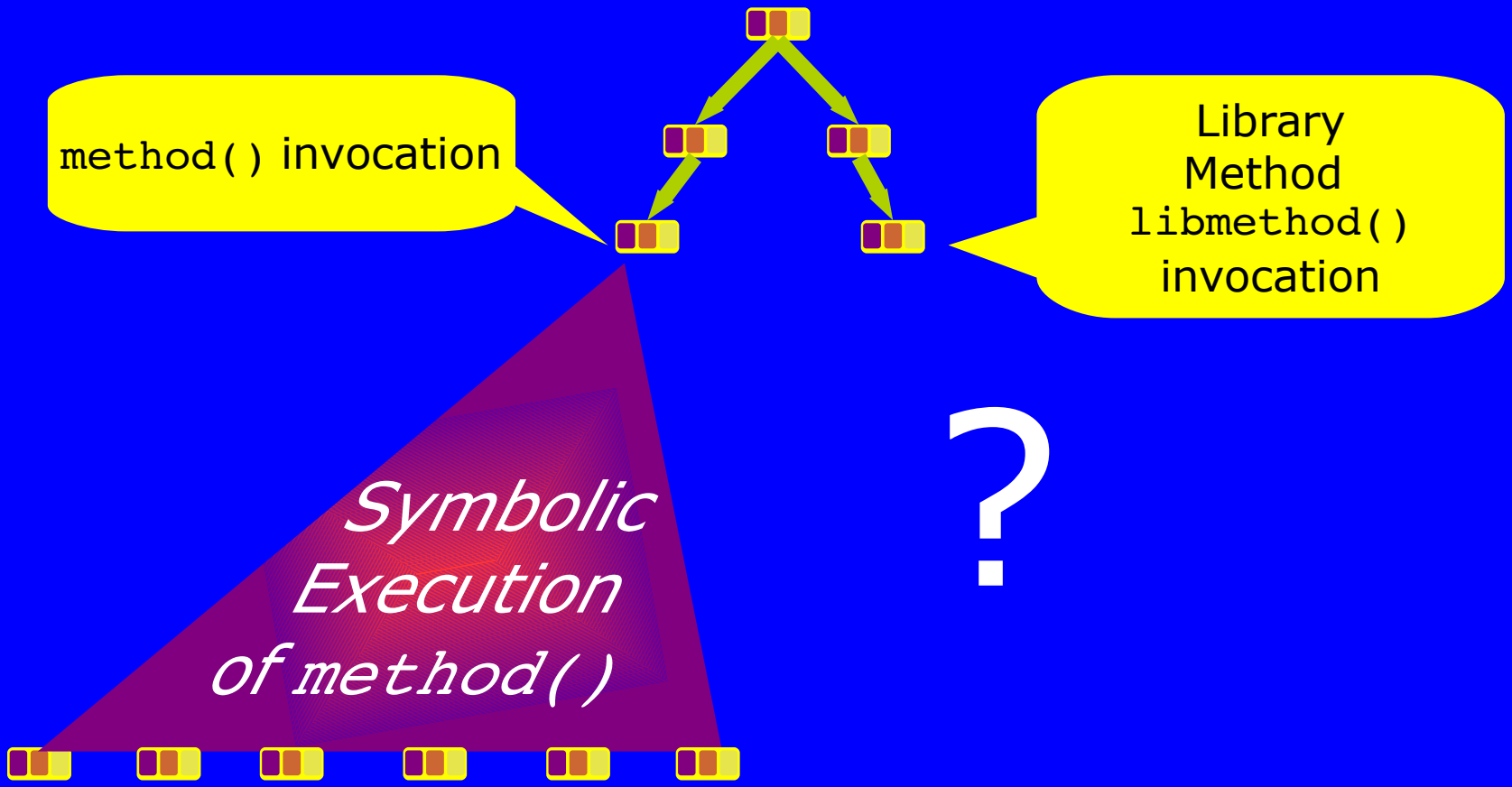
Specification matters



Specification matters



Specification matters



Specification in Theorem Proving based test case generation

```
public void underTest(){
      ...
      otherMethod();
      ...
}
```

Specification in Theorem Proving based test case generation

`underTest`'s specification

Assume the precondition holds. Is the postcondition satisfied?

```
public void underTest(){
        ...
        otherMethod();
        ...
}
```

```
requires t > 0;
```

- Encodes the tests

t>0

```
x = x+y;
if(x > t){return t;}
else {return x;}
```

Conjectural use

Specification in Theorem Proving based test case generation

`underTest`'s specification

Assume the precondition holds. Is the postcondition satisfied?

```
public void underTest(){
    ...
    otherMethod();
    ...
}
```

`otherMethod`'s specification

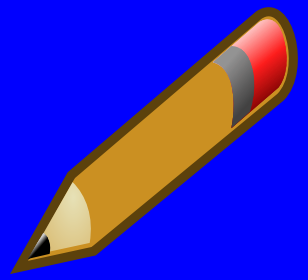
Does the precondition hold at this point? Then assume the postcondition to hold.

- **Encodes the tests**

Conjectural use

- **Replaces code**
- **Keeps the method feasible**

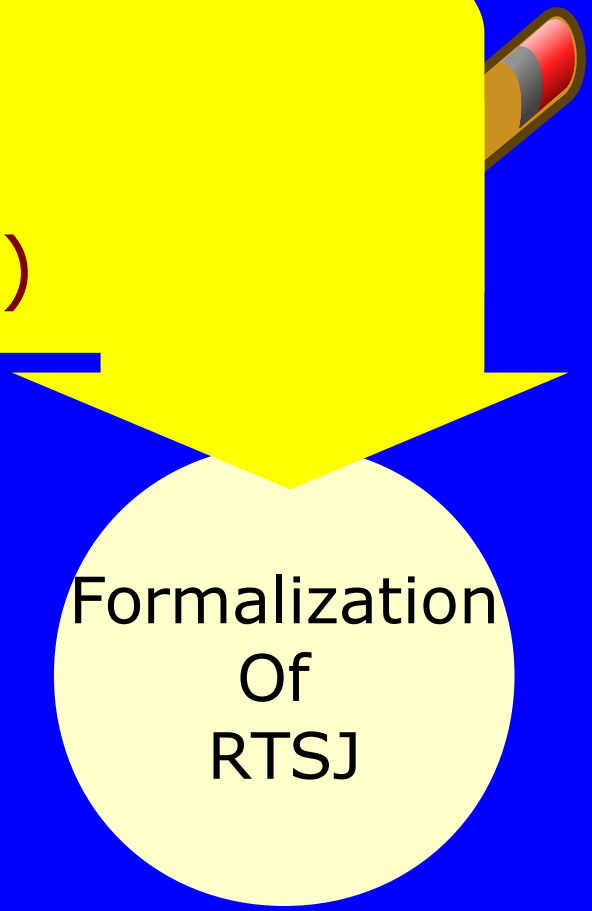
Axiomatic use



Formalization
Of
RTSJ

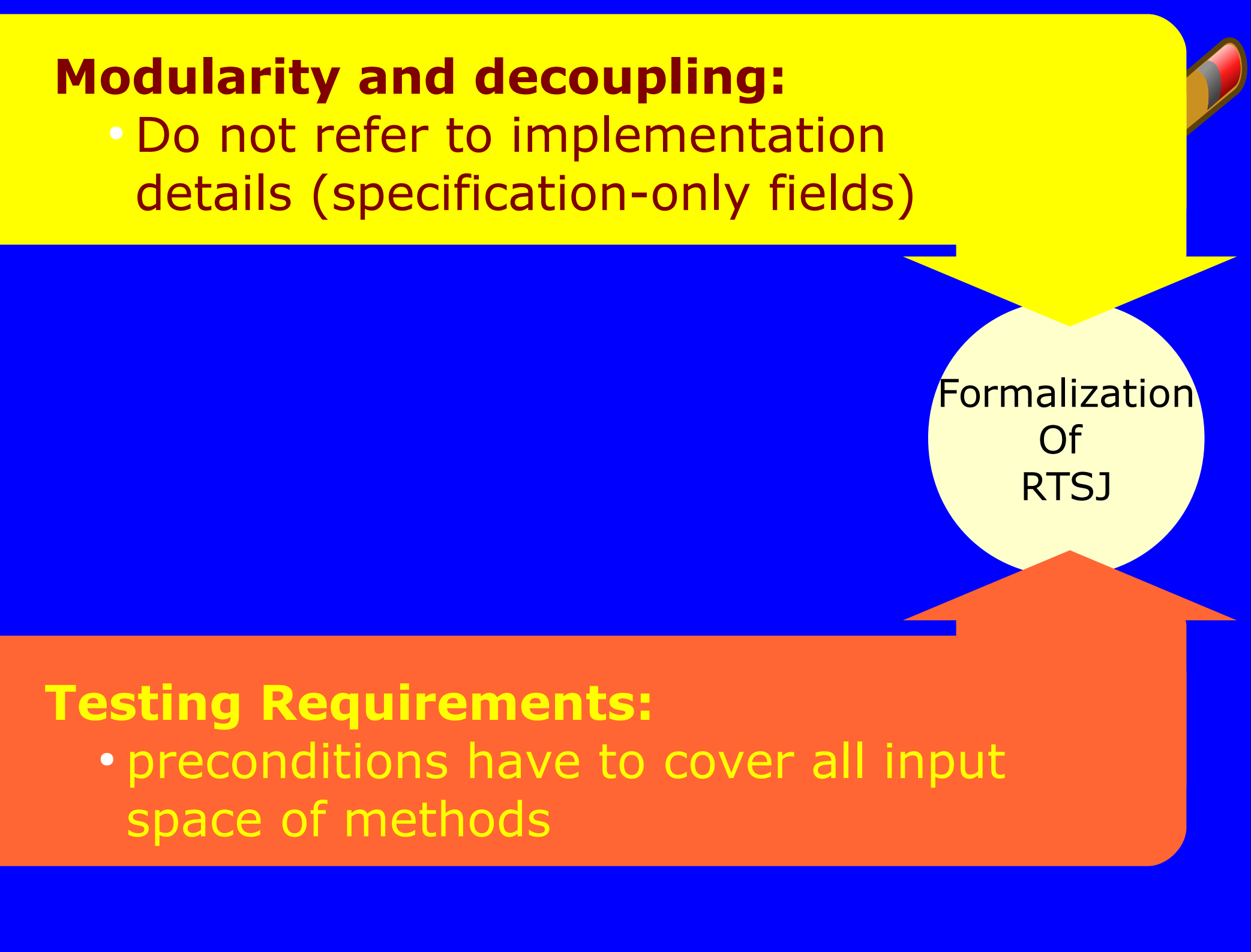
Modularity and decoupling:

- Do not refer to implementation details (specification-only fields)



Modularity and decoupling:

- Do not refer to implementation details (specification-only fields)

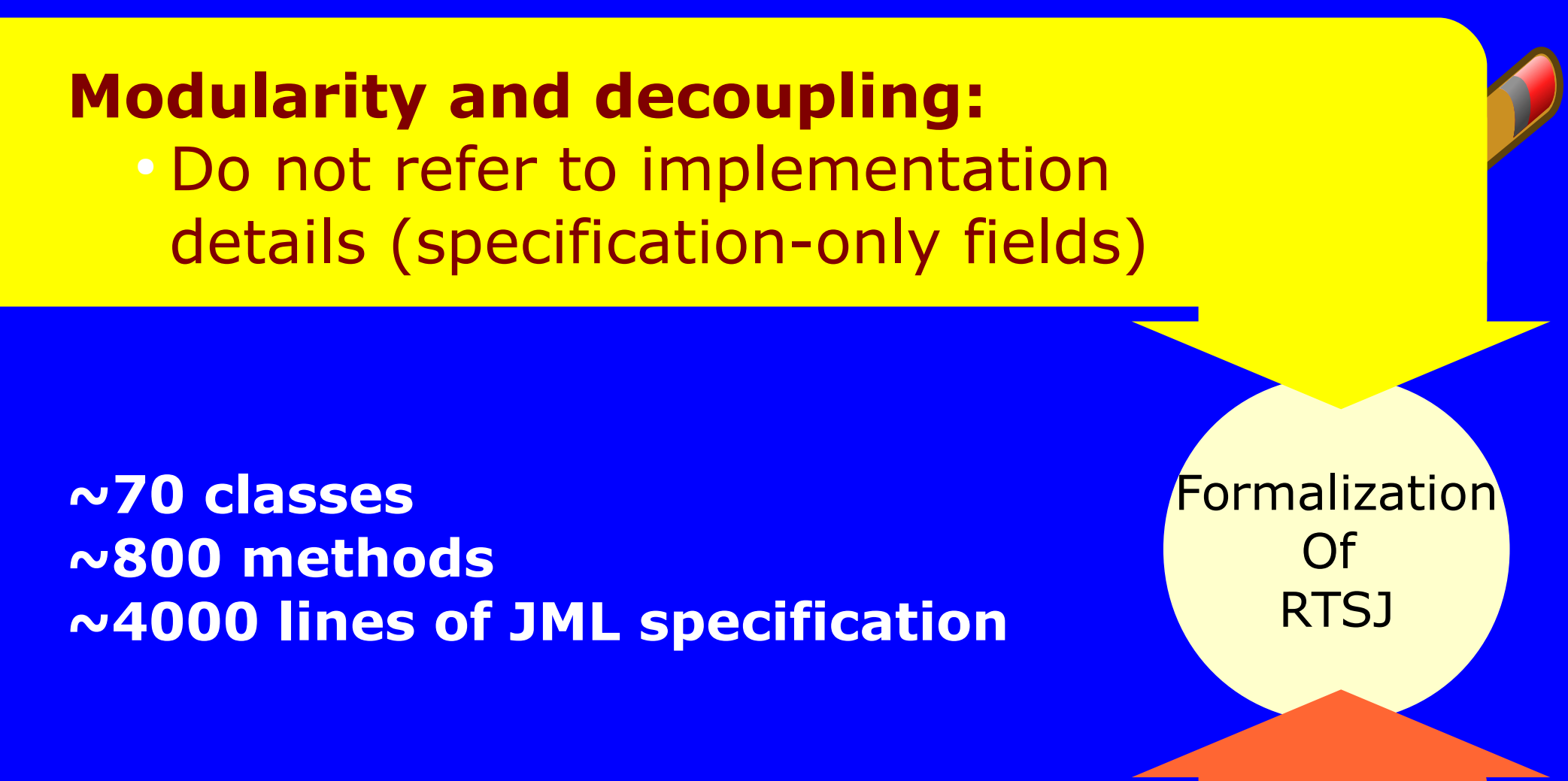


Testing Requirements:

- preconditions have to cover all input space of methods

Modularity and decoupling:

- Do not refer to implementation details (specification-only fields)



Testing Requirements:

- preconditions have to cover all input space of methods

Evaluation: Testing Lightgun driver

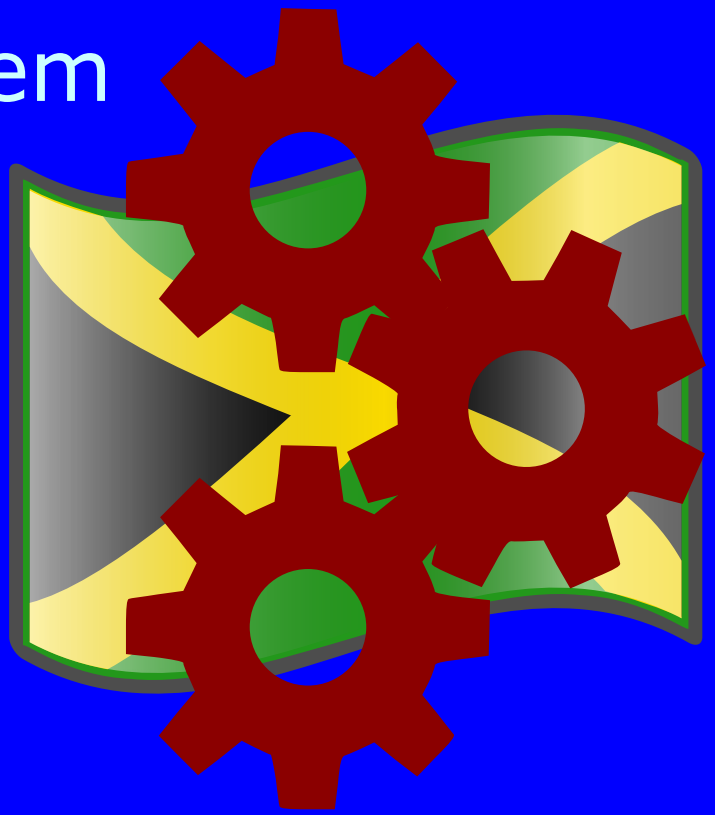
- A small application ~ 700 loc
  - Driver for a CRT-compatible lightgun
  - Realtime: syncing with the screen refresh
- Coverage: MC/DC

Evaluation:
Verifying correctness of RTSJ code with KeY

- CDx Real-Time Java Benchmark
- A collision detector for aerial traffic
- Proofs can be hard
  - Some automatic
  - Others require user input

Evaluation: Testing of API implementation

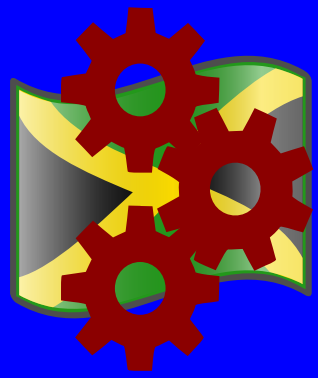
- JamaicaVM implementation
- Tested against our specification
- Our method found a problem automatically



Time in RTSJ

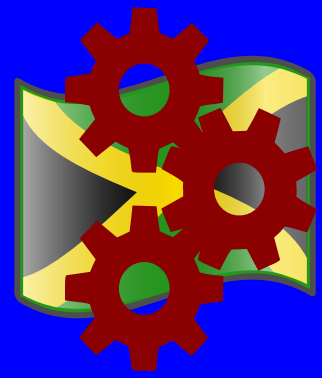
- `Clock`:
  - Entity that measures time. The default one is called *Real-time clock*.
- `AbsoluteTime`:
  - Elapsed time of a specific Clock

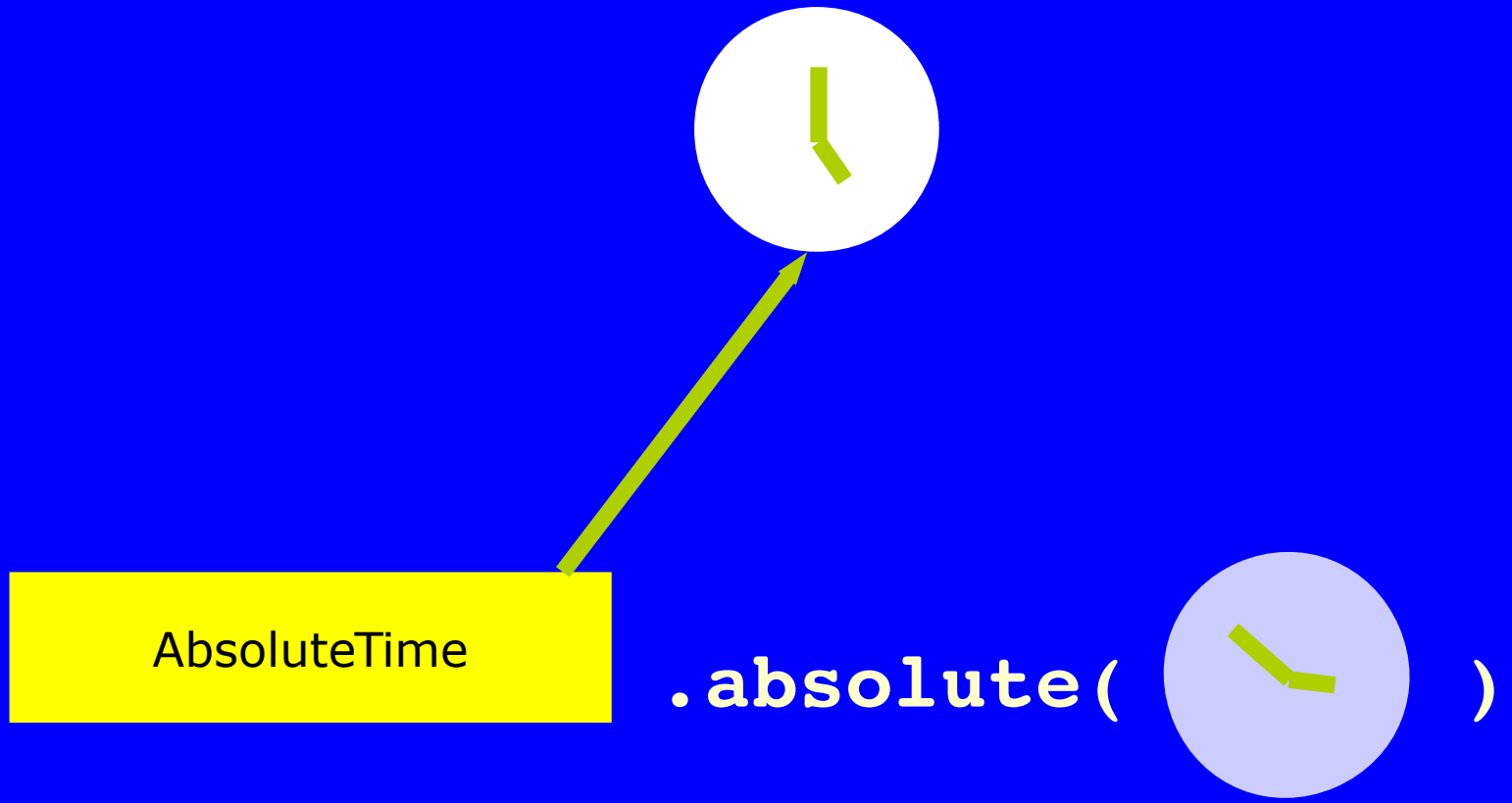
`absolute()` method in `AbsoluteTime` class



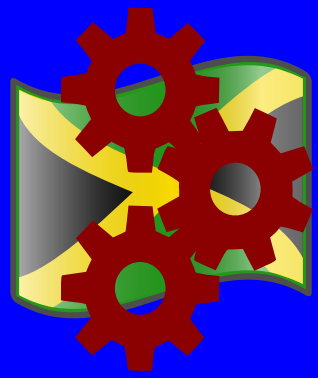
- **`public`** `AbsoluteTime absolute(Clock clock)`
  - Return a copy of `this` modified if necessary to have the specified clock association.
  - A new object is allocated for the result. […]
  - The clock association of the result is with the clock passed as a parameter.
  - If clock is null the association is made with the real-time clock.

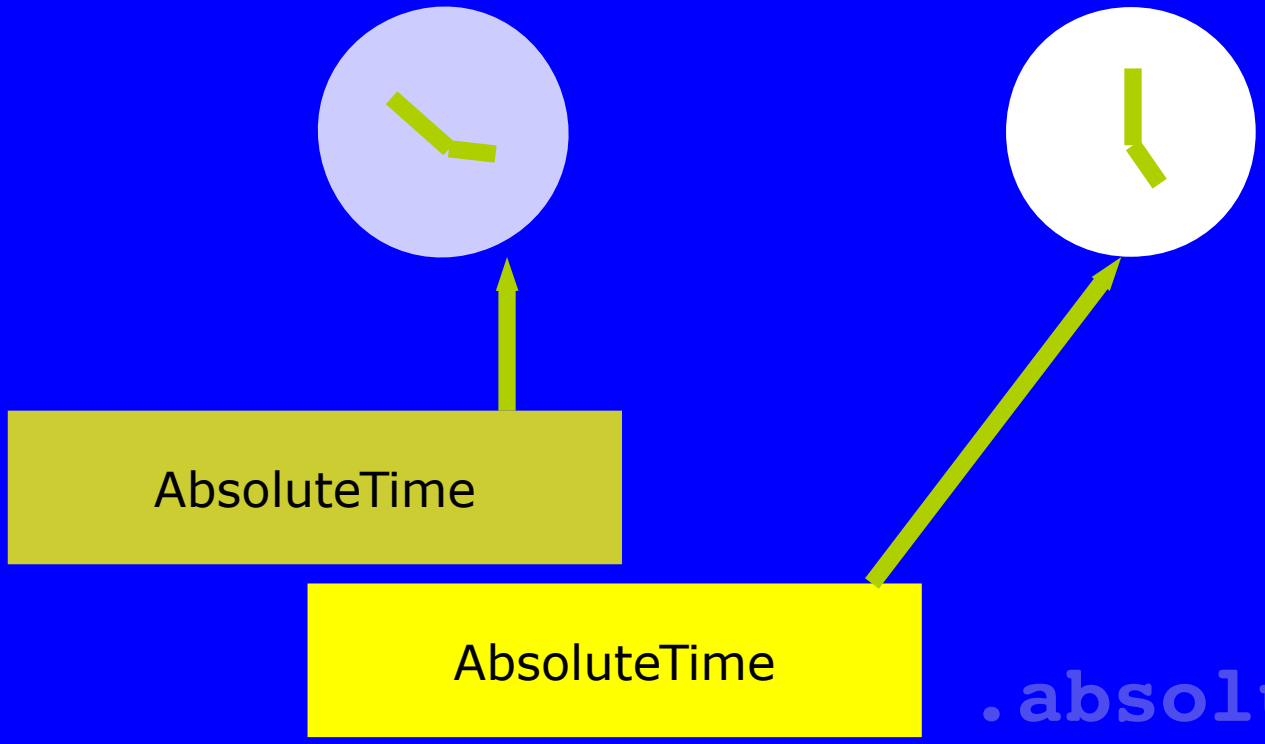
absolute() method





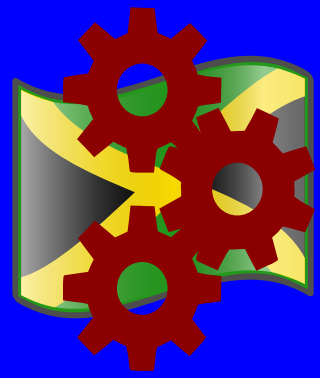
absolute() method





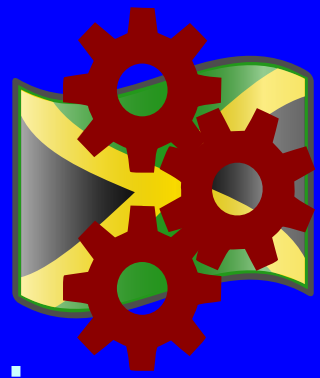
.absolute( )

absolute() method



```
/*@
ensures clock != null ==>
          \result.getClock() == clock;

ensures clock == null ==>
          \result.getClock() ==
                    Clock.getRealtimeClock();
*/
public AbsoluteTime absolute(Clock clock);
```

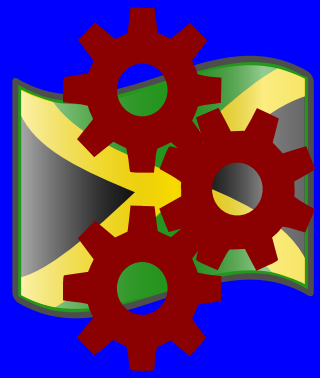
The inconsistency

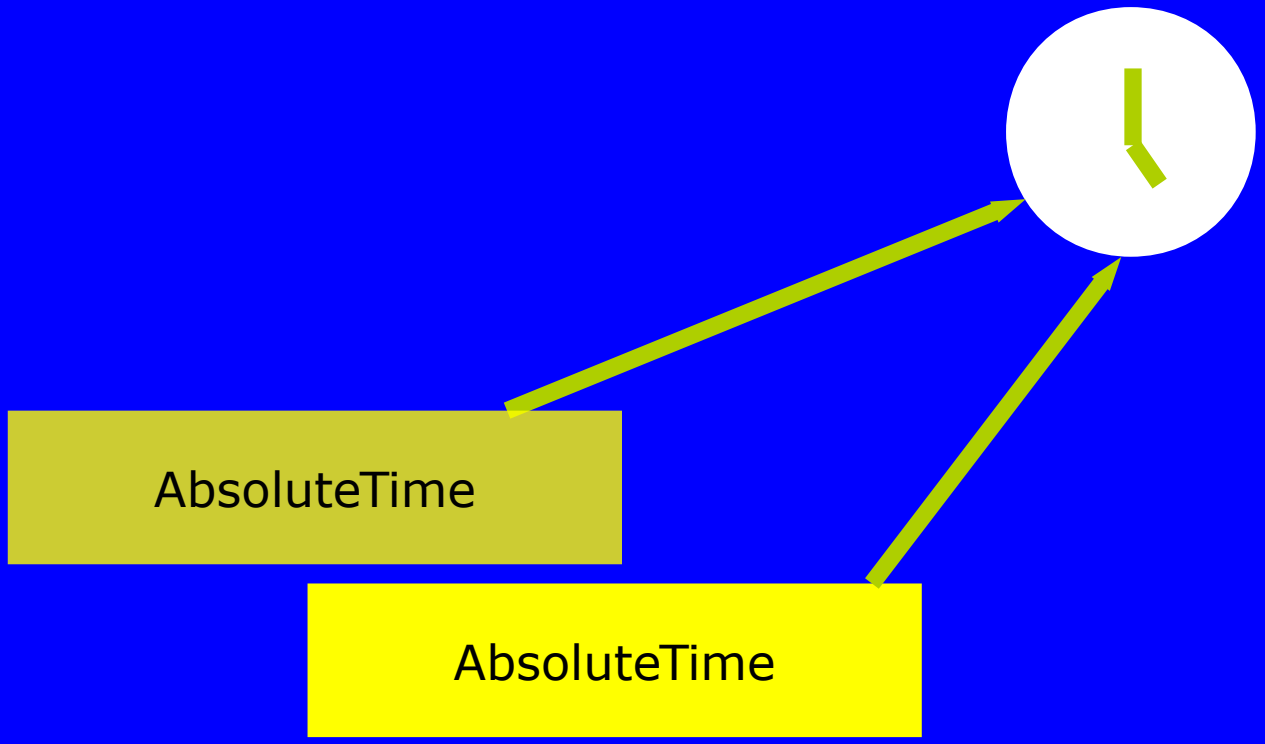


- KeYTestGen showed (*automatically*) that:
- If a clock is passed as argument, the reference to it is not set

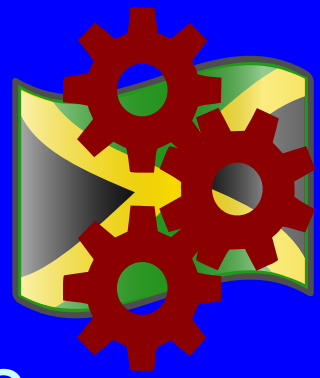
```
/*@
ensures clock != null ==>
          \result.getClock() == clock;

ensures clock == null ==>
          \result.getClock() ==
                    Clock.getRealtimeClock();
*/
public AbsoluteTime absolute(Clock clock);
```

absolute() method





The inconsistency



- If a clock is passed as argument, the reference to it is not set
- This was intentional
- There is no way to *add* a clock in RTSJ

Challenges and related work

- Better handling of quantifiers
  - Christoph Gladisch.
    "Test Data Generation for Programs with Quantified First-Order Logic Specifications"
- Concrete instantiation of reference type
  - Specification & solutions to constraints tells just what the result is, but not how to build it
- Other Java+JML approaches:
  - JMLUnitNG
    - Test the constructor, and then cache the created objects for future tests

Verifying (in-)stability in floating-point programs

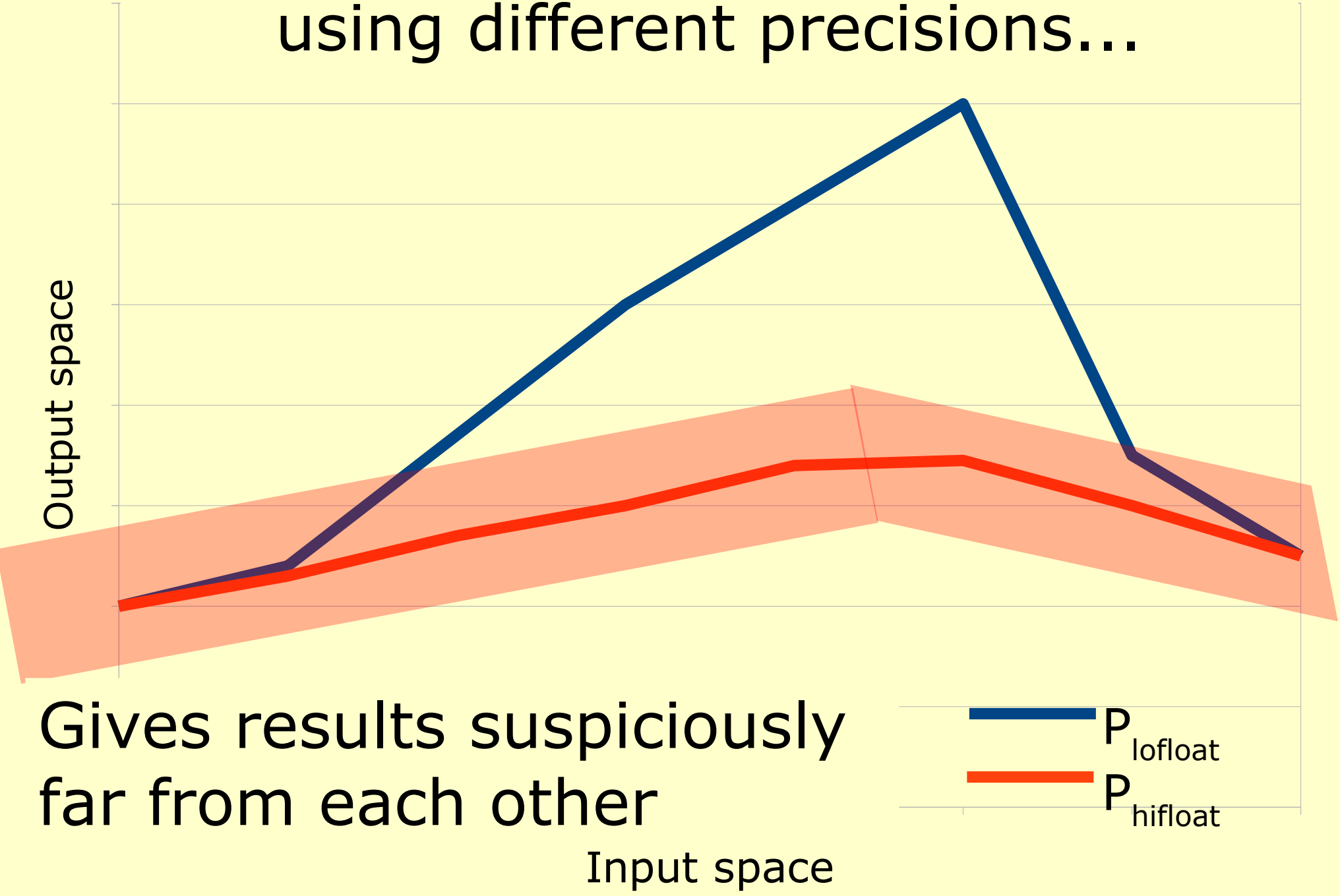
by increasing precision using SMT solving

Gabriele Paganelli

Wolfgang Ahrendt

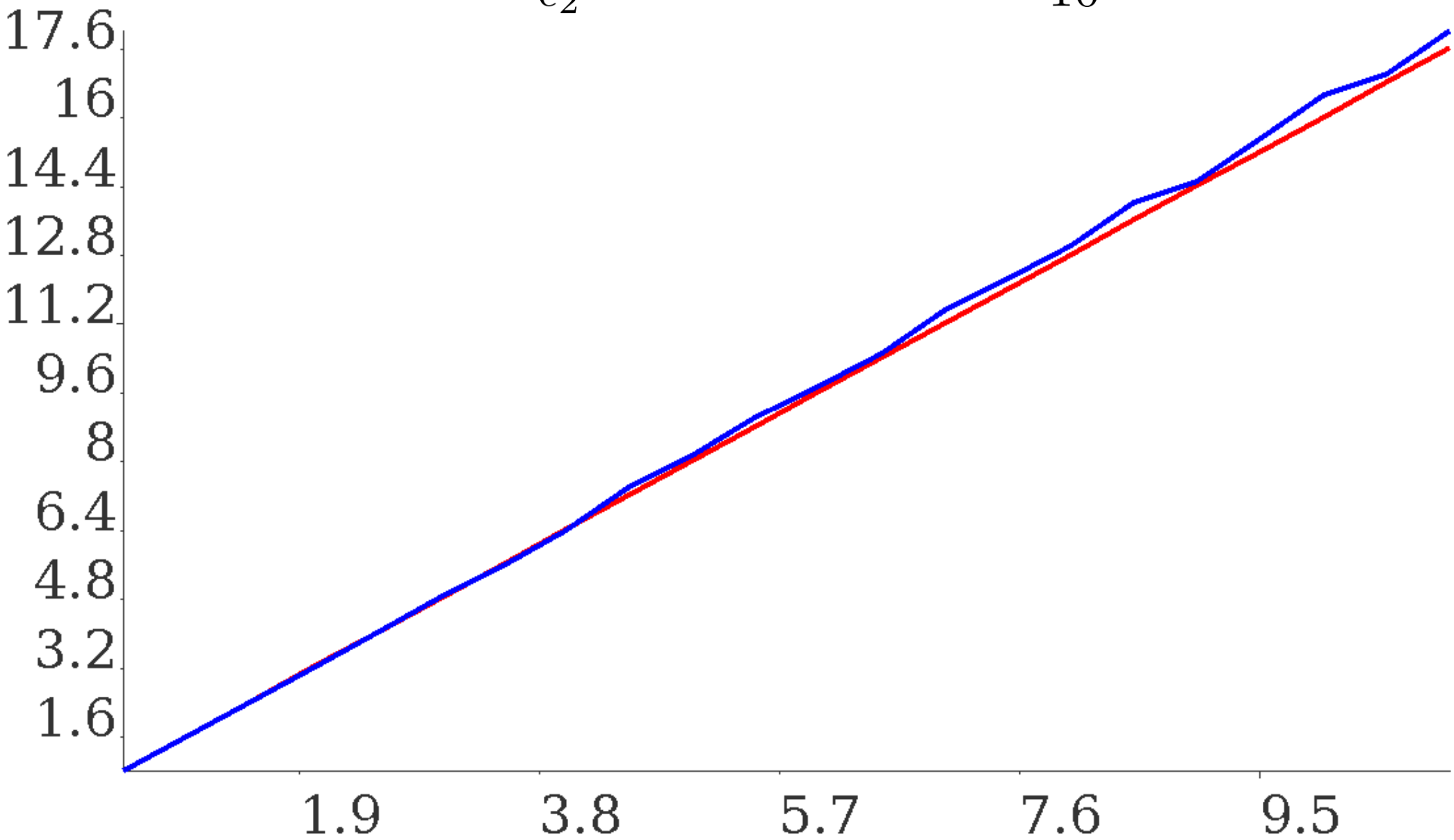
To appear in: *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing,*
SYNASC 2013
IEEE Computer Society, 2013

Instability: running program **P** using different precisions...

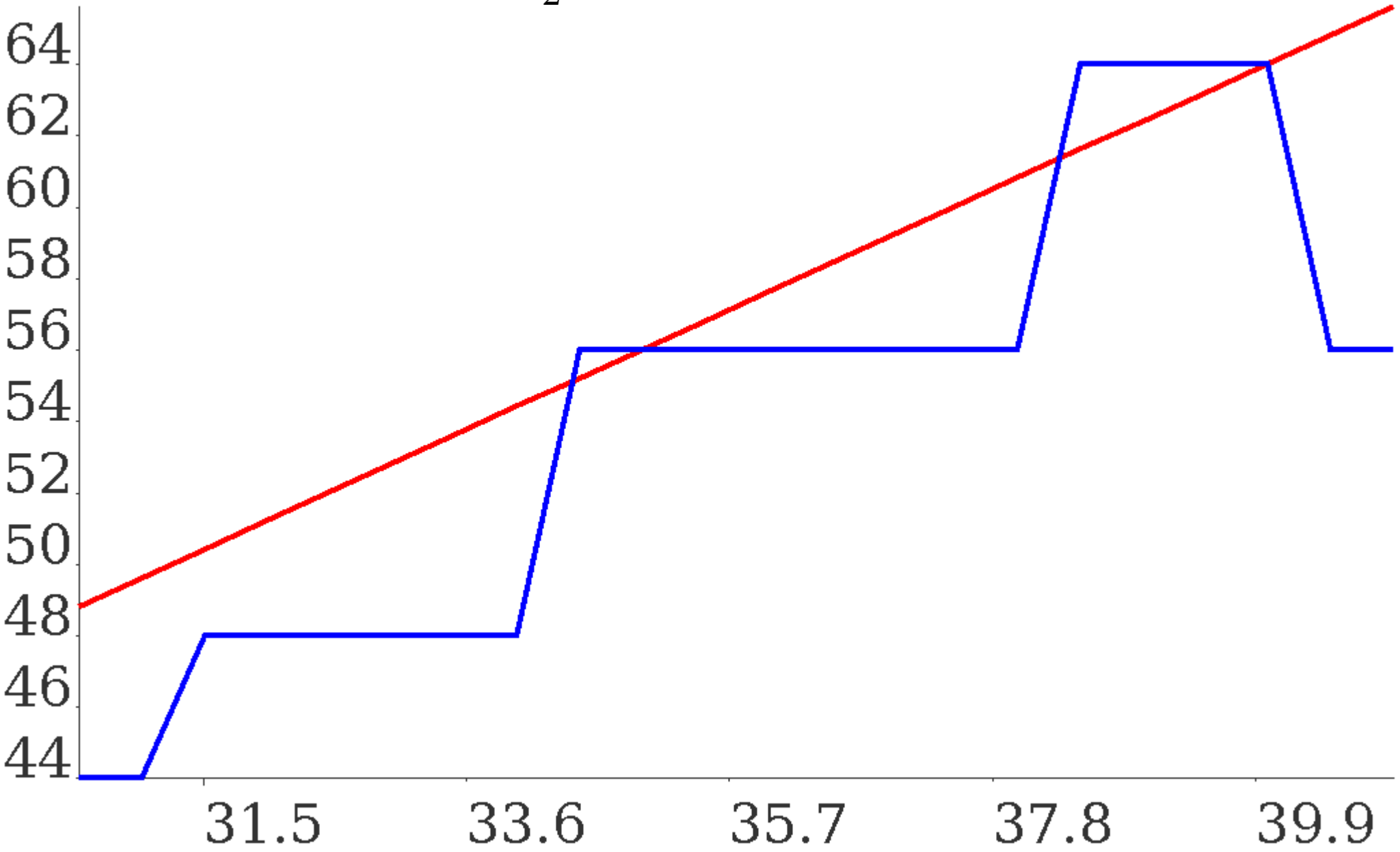


Gives results suspiciously far from each other

$$f(x) = \frac{(x-\epsilon_1)^2 - (x-\epsilon_2)^2}{\epsilon_2} \qquad \epsilon_1 = \frac{2^{-15}}{10} \quad \epsilon_2 = 2^{-16}$$



$$f(x) = \frac{(x-\epsilon_1)^2 - (x-\epsilon_2)^2}{\epsilon_2} \qquad \epsilon_1 = \frac{2^{-15}}{10} \qquad \epsilon_2 = 2^{-16}$$



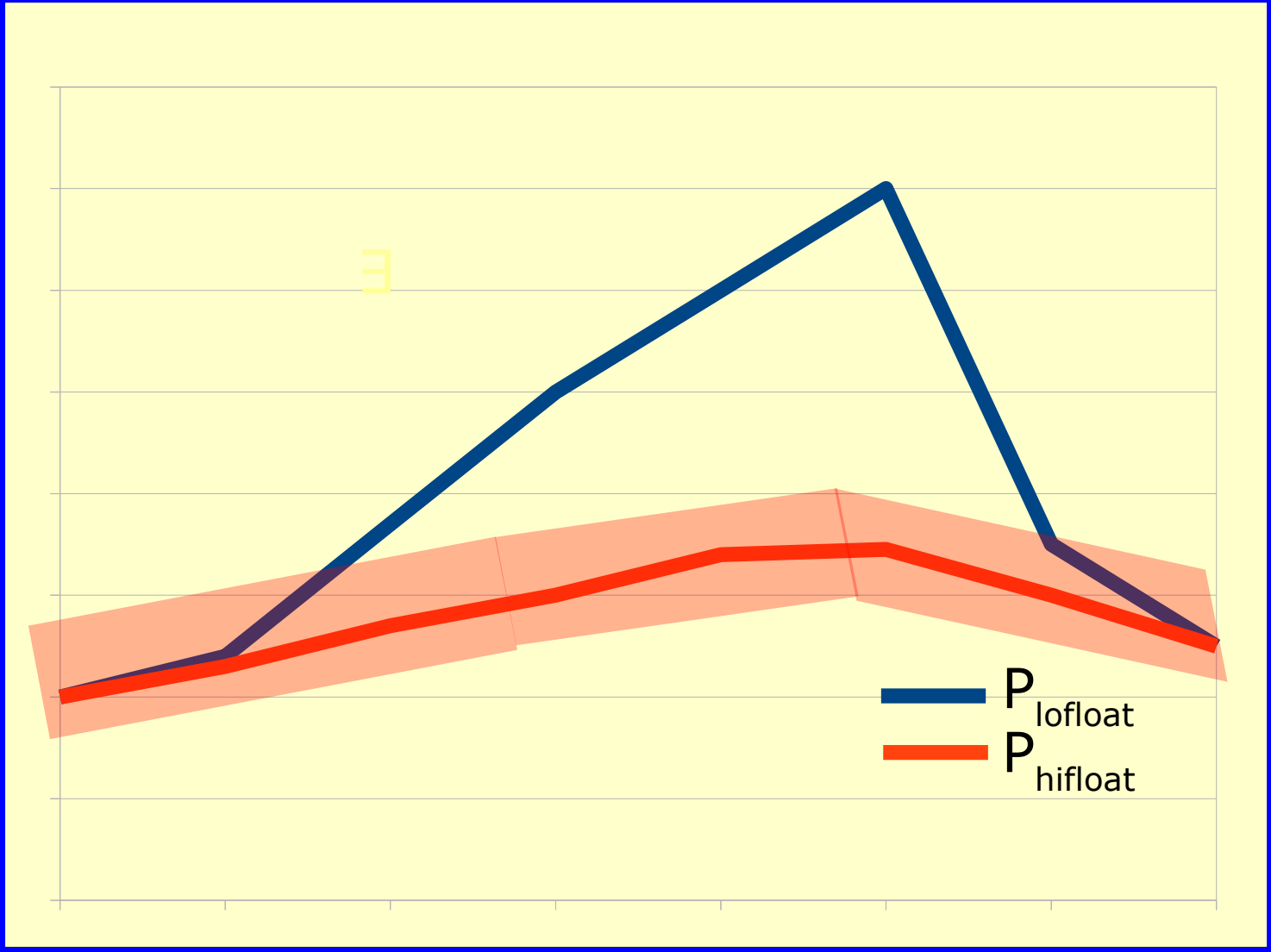
Idea:
Find witnesses of instability for **P**

such that it does not require user numerical expertise to do it
(no proofs)


Deterministic
alternative to
random testing

- Is there any input value v for P_{lofloat} and P_{hifloat}

such that the relative error between them is bigger than a certain specified value?



Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Implemented with the

Prototypical language

FPhile

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Implemented with

Program syntactical transformation

and

weakest precondition

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Implemented with

Satisfiability Modulo Theory (SMT) Solver

Floating-Point While

Java-like IEEE support

Two floating-point types:
lofloat and **hifloat**

FPhile: types

- The **lofloat** and **hifloat** types are “abstract”:
- User-defined precision(in bits)
  - Exponent
  - Mantissa

FPhile annotation statements

assume (bool b)

assert (bool b)

Precision comparing predicate

stable(e @ r)

- Meaning:
  - At the point of the program where this predicate occurs
  - the evaluation of $e_{lofloat}$ differs relatively from $e_{hifloat}$ by at most $r_{hifloat}$

Specify the admissible error

Method

Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Implemented with

Program syntactical transformation

and

weakest precondition

```
lofloat f,g;
g = 100.0;
assume stable(f@0.0)
if(f>0.0){
  f = f+(g*f);
  ...
}
assert stable(f@2^-24)
```

```
lofloat f,g;
g = 1
assum
if(f>
  f =
  ...
}
assert
```

```
hifloat hf,hg;
g = 100.0;
hg = 100.0;
assume abs( (hf – f) / hf ) <= 0.0
if(f>0.0){
  f = f+(g*f);
  ...
}
if(hf>0.0){
  hf = hf+(hg*hf);
  ...
}
assert abs( (hf – f) / hf ) <= 2-24
```

Compare $P_{lofloat}$ with P_{hi}

Act 1

```
lofloat f,g;
hifloat hf,hg;
g = 100.0;
hg = 100.0;
assume abs( (hf – f) / hf ) <= 0.0
if(f>0.0){
  f = f+(g*f);
  ...
}
if(hf>0.0){
  hf = hf+(hg*hf);
  ...
}
assert abs( (hf – f) / hf ) <= 2^-24
```

Weakest precondition of program **P**

- First-order logic formula
- It encodes the least constraining input that
  - satisfies **P**'s assertions

Compare $P_{lofloat}$ with P_{hi}

Act 2

Method

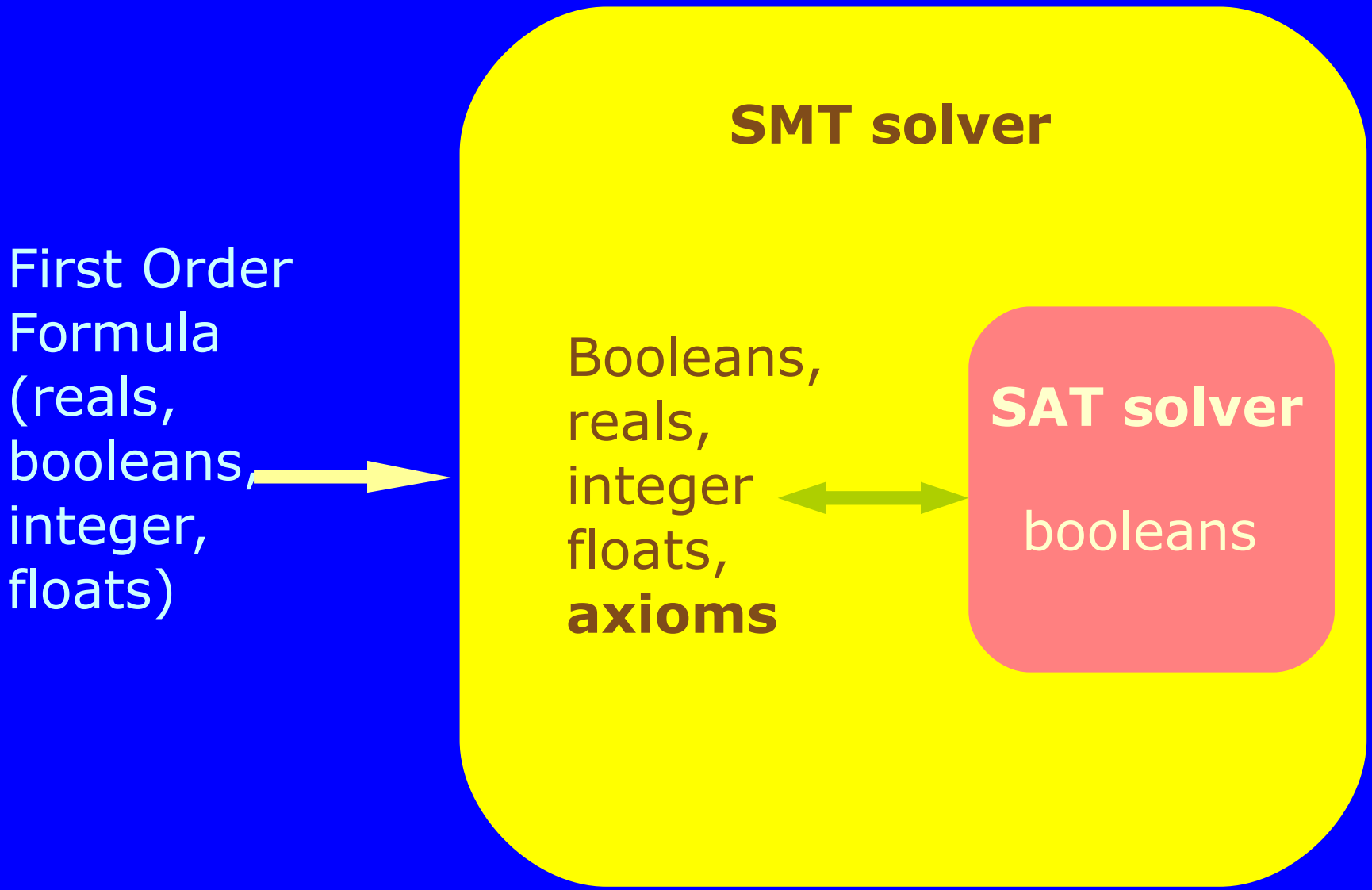
Given program $P_{lofloat}$ and the admissible error

1) Compare $P_{lofloat}$ with $P_{hifloat}$
2) Find a witness of instability

Implemented with

Satisfiability Modulo Theory (SMT) Solver

Satisfiability Modulo Theory solvers



Find a witness of instability

- Instability witness found by
  - Finding satisfiable assignment of the negated weakest precondition
- Witness:
  - The input that makes **P** fail to satisfy its assertions.

Implementation details

- **Microsoft Z3** supports floating-point arithmetic
- Weakest precondition to SMT input translates directly
  - No axiomatization of a IEEE floating-point unit in e.g. reals

Performance

- We compared this approach with random testing
- Stability between 32 and 64 bits IEEE computations
- Machine was a normal desktop (specs on paper)

Heron's triangle area formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

$$s = \frac{a+b+c}{2}$$

- This formula is numerically bad
- Instability (bound = 2^{-22}) is found earlier by testing:
  - Testing:
    - 10^4 tests, 3000 failures, 1.6 minutes
  - Fphile:
    - Counterexample in 2.68 minutes

Heron's formula, improved

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}$$

- This is a rearrangement suggested by W.Kahan
- Numerically better, but unstable (bound = 2^{-22})
  - Testing:
    - 10^7 tests, no failure, 27 hours
  - Fphile:
    - Counterexample in 6.5 minutes

Parameters affecting performance in **Z3**

- Precision(s) used
- Operations. Most expensive are:
  - *** RoundToIntegral
  - ** Square root
  - ** Multiplication/division
  - * Addition/subtraction
- Number of variables
- Nature of formula
  - unsatisfiable, “hardly” satisfiable...

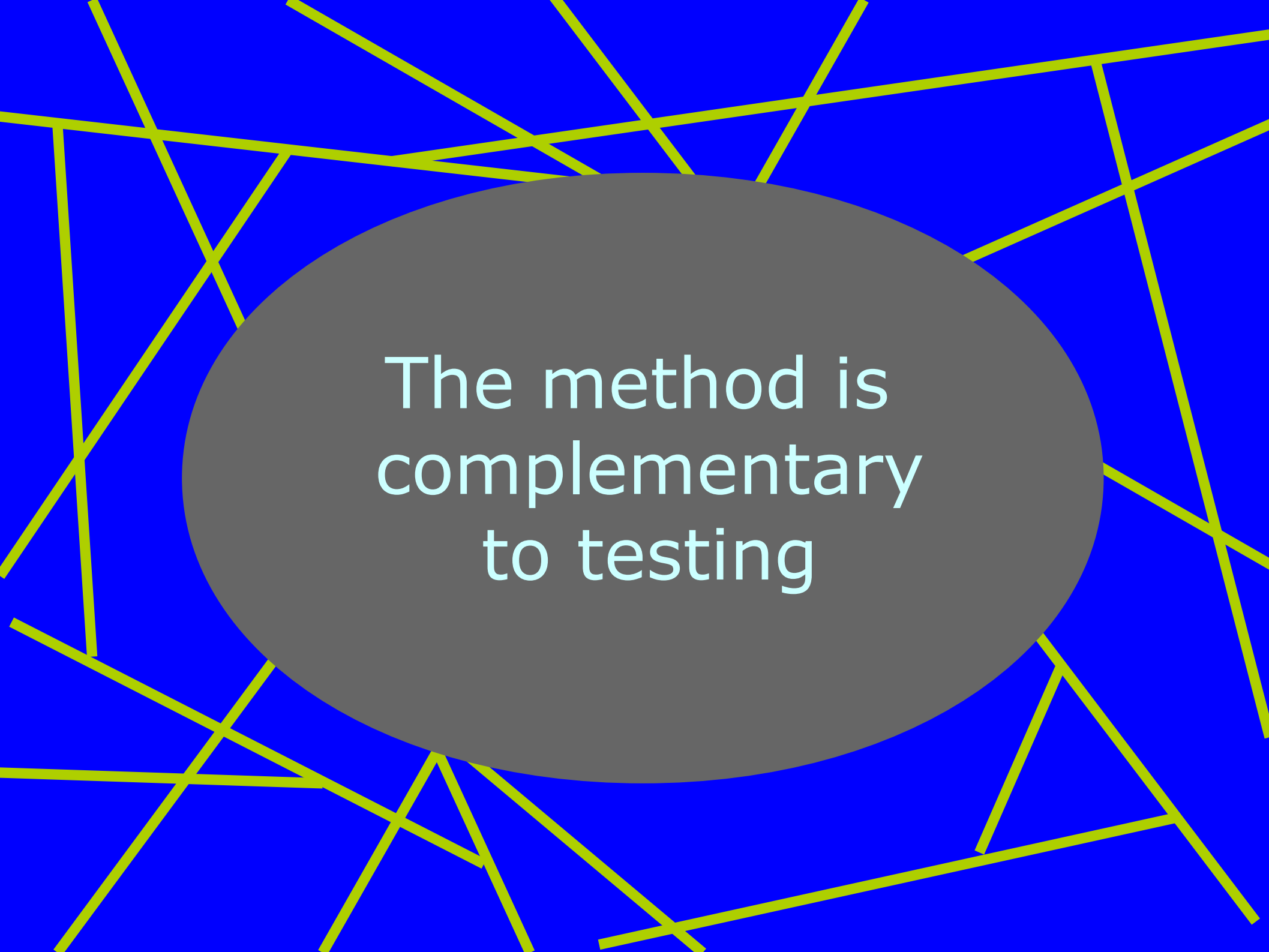
Conclusion

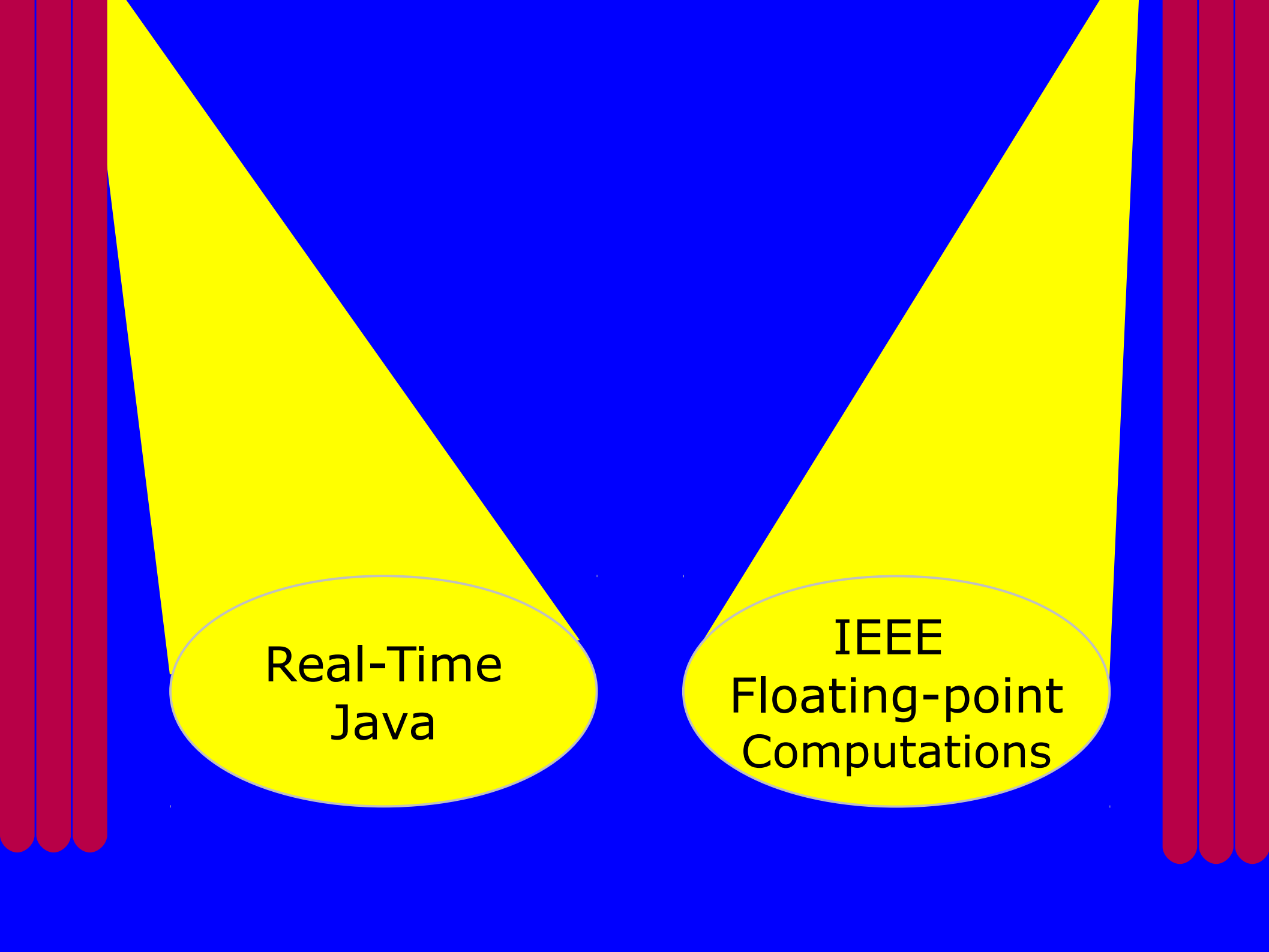
- An automated analysis to detect instability.
- The witnesses can be used for further analysis.
- It is inspired by W.Kahan's manuscripts on floating-point debuggers
  - that do not require [illegible] floating-[illegible]

P is not executed!

Conclusion – SMT solving

- We found some bugs in **Z3**
  - promptly fixed by the **Z3**ers
- About the SMT floating-point theory:
  - Among its first applications
    - to our knowledge
  - We contribued with some refinements
    - Subnormality predicate, casting, literal representation


The method is
complementary
to testing


Real-Time
Java
IEEE
Floating-point
Computations

CHARTER

Critical and High Assurance Requirements Transformed through Engineering Rigour

http://charterproject.ning.com/

Real-Time Java Specifications for High Coverage Test Generation

KeYTestGen

Formal Specification

Evaluation

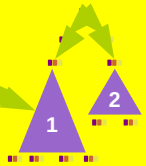
Symbolic execution

Constraint solving

Code generation

Code generation from specification

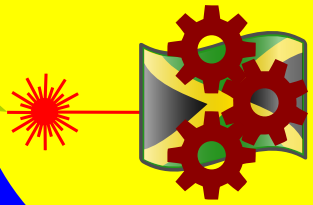
Dual usage of specification



Replacement of missing/unknown code

Feasibility of the approach

Verification: collision detector



Test case generation: JamaicaVM, Ligthgun Driver

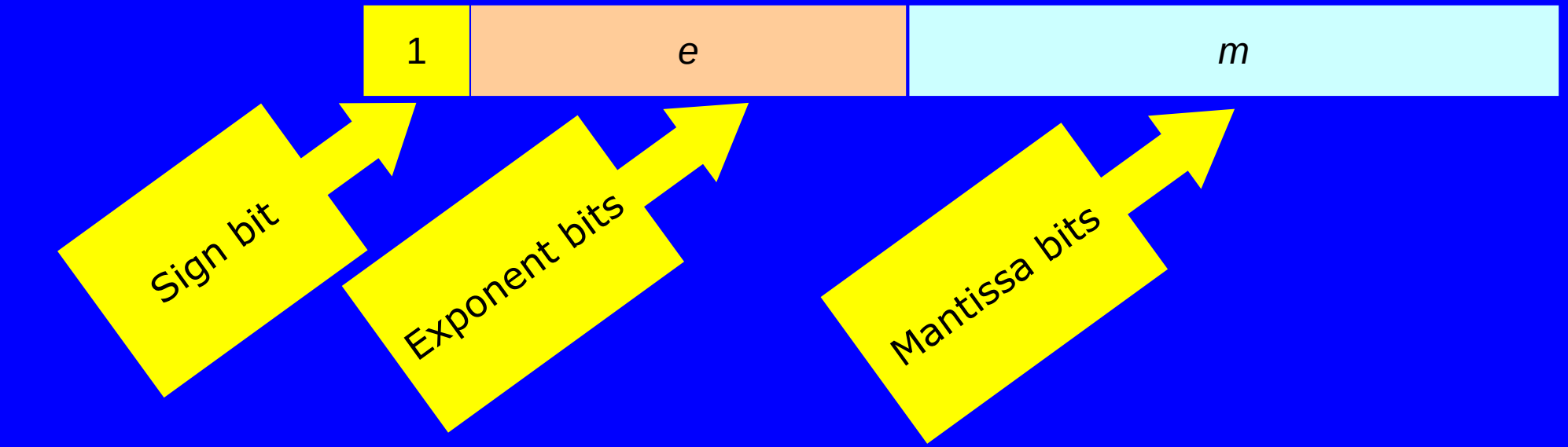
Verifying (in-)stability in floating-point programs by increasing precision using SMT solving

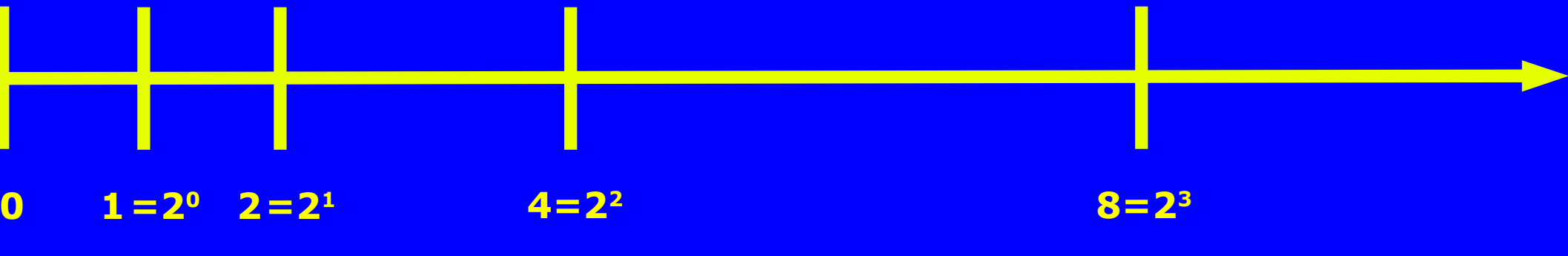
- Instability
- Specification
- Program duplication
- Weakest precondition
- SMT solving and floating-point
- Performances

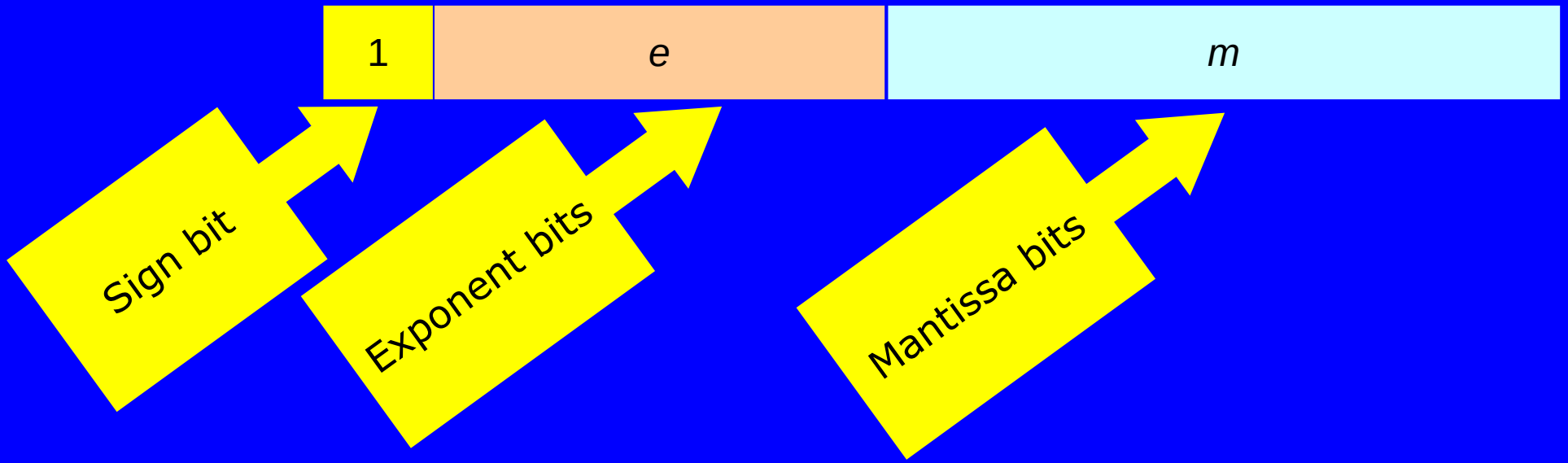
www.cse.chalmers.se/~gabpag/devel/fphile/

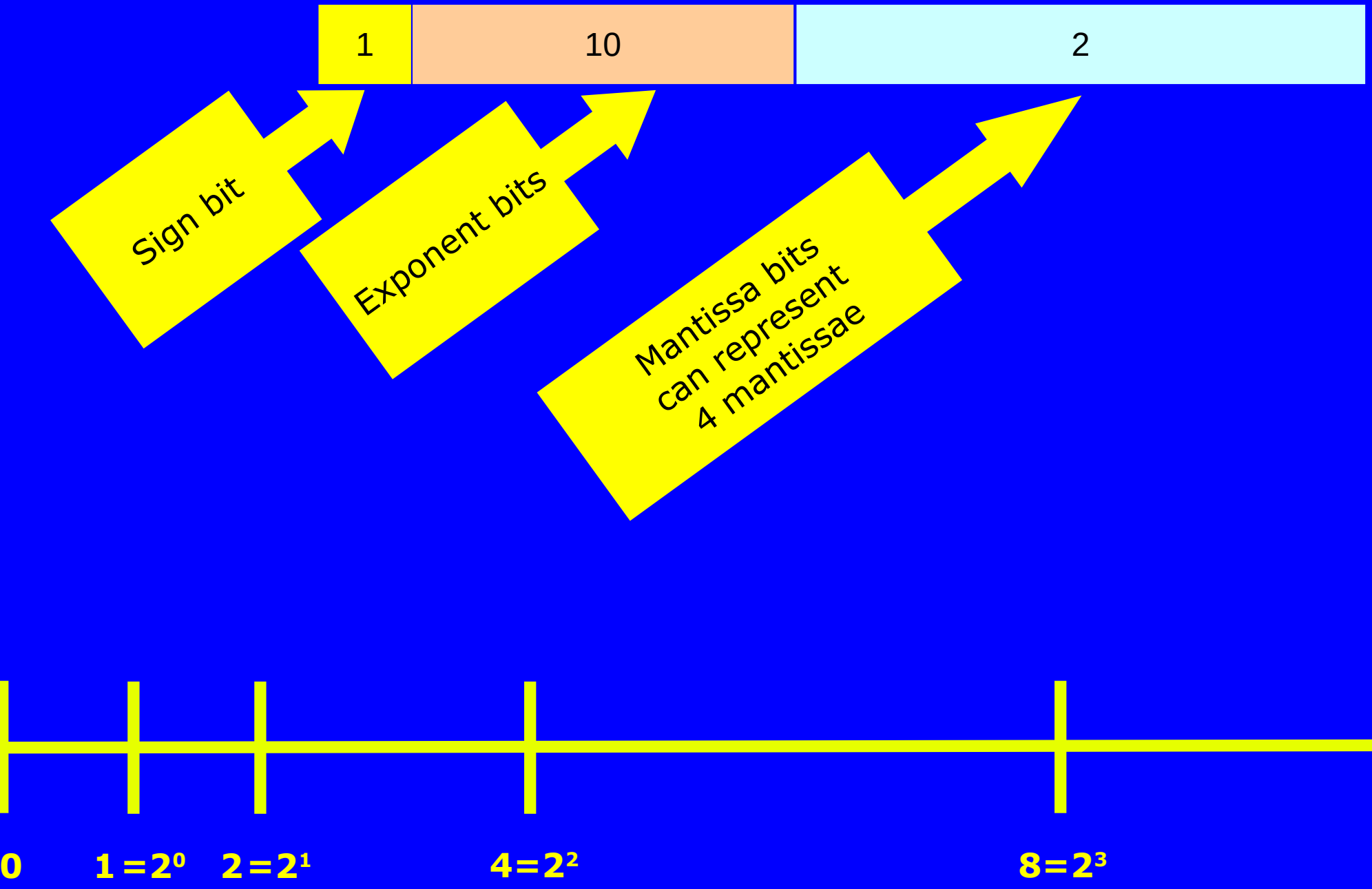
Links

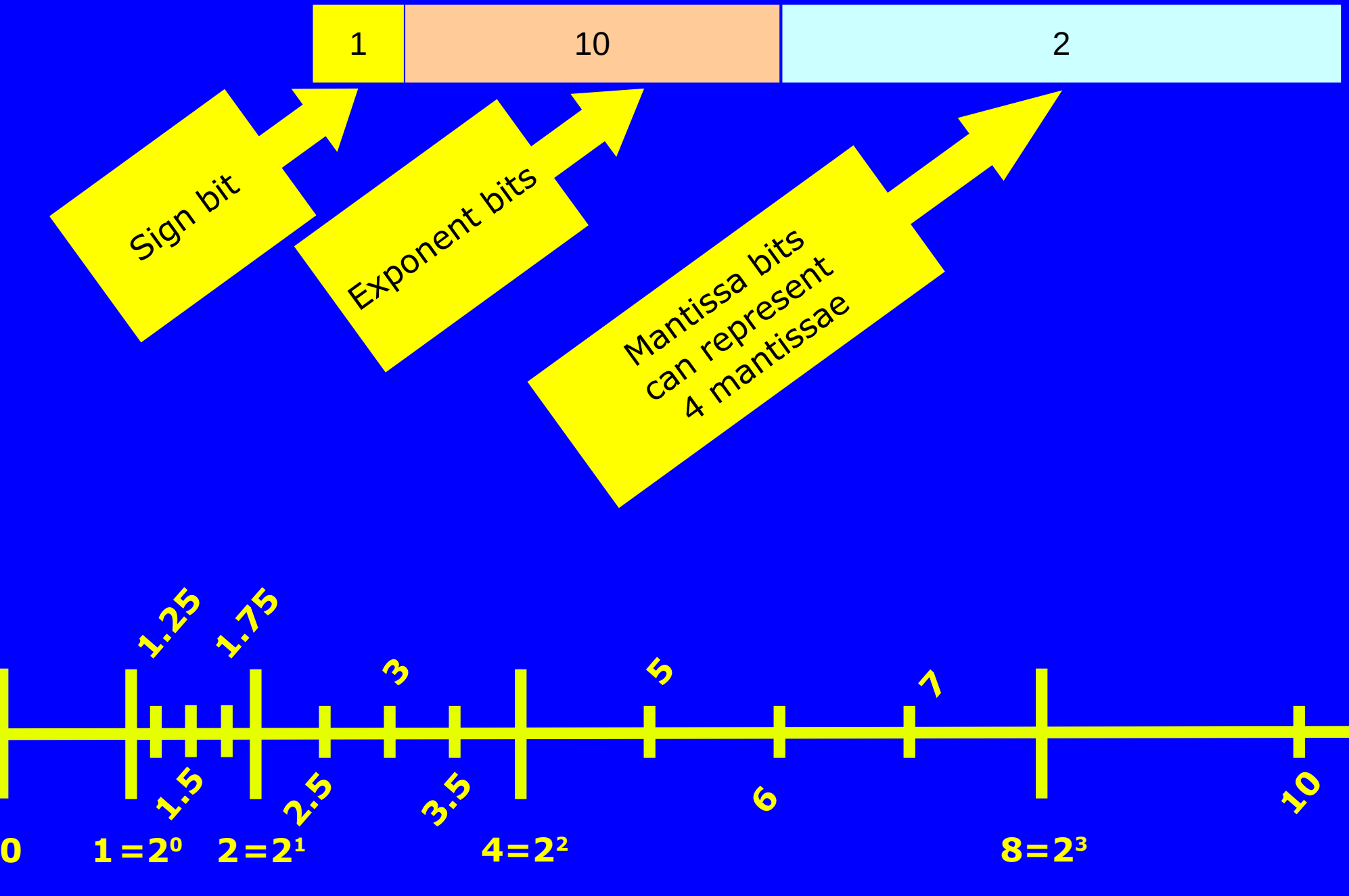
- CDx benchmark
  - http://sss.cs.purdue.edu/projects/cdx/
- KeY
  - http://www.key-project.org
- KeYTestGen eclipse update site
  - http://www.cse.chalmers.se/~gabpag
- JMLUnitNG
  - http://formalmethods.insttech.washington.edu/software/jmlunitng/
- JML formalized RTSJ API
  - http://wwwhome.ewi.utwente.nl/~mostowskiwi/


1
e
m
Sign bit
Exponent bits
Mantissa bits


0
1 = 2⁰
2 = 2¹
4 = 2²
8 = 2³


1
e
m
Sign bit
Exponent bits
Mantissa bits


1
10
2
Sign bit
Exponent bits
Mantissa bits
can represent
4 mantissae
0
1 =2⁰
2 =2¹
4=2²
8=2³


1
10
2
Sign bit
Exponent bits
Mantissa bits
can represent
4 mantissae
0
1 =2⁰
2 =2¹
4=2²
8=2³
1.25
1.75
3
5
7
1.5
2.5
3.5
6
10

MC/DC

- Modified Condition/Decision Criterion
- For all boolean expressions *d* in program under test:
  - A swap in the value of boolean literal *c* in *d*
  - Swaps the value of *d*
  - Maintaining fixed other conditions *c'* in *d*
- Shown with a pair of tests for each *c*
- Enables safety-critical software certifications (DO178C)