Real-time Java API Specifications for High Coverage Test Generation

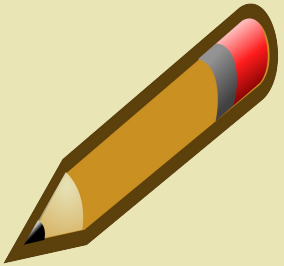
Wolfgang Ahrendt
Chalmers Technical University

Wojciech Mostowski
University of Twente

Gabriele Paganelli
Chalmers Technical University

Contributions

JML Formalisation of Real-Time Specification for Java (RTSJ)



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

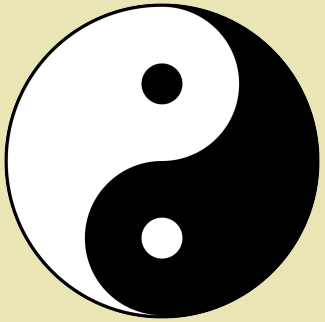
Test industrial code using KeYTestGen and formal specification

KeYTestGen

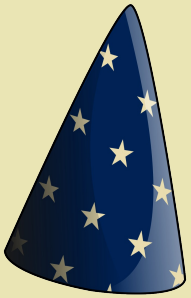
- A **Theorem Proving** Based Test Case Generator
- **Input**: source code and specifications
- An **eclipse** plugin
- Aiming to be a **push-button** technology

KeYTestGen

Symbolic Execution



Constraint solving



Test **code** generation

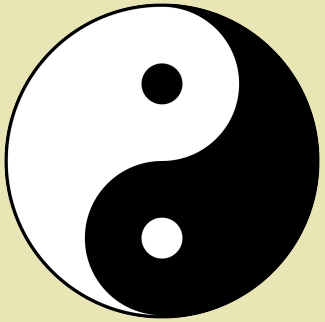
Java+JML

Runnable Test Suite

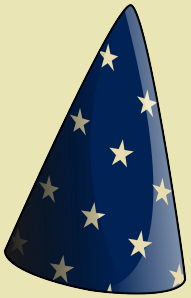
KeYTestGen

Set of constraints
Describing paths
inside the code

Symbolic
Execution



Constraint
solving



Test **code**
generation

Java+JML

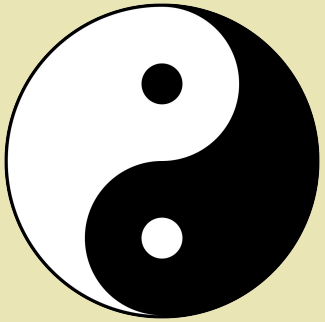
Runnable
Test
Suite

KeYTestGen

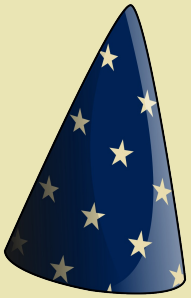
Set of constraints
Describing paths
inside the code

Concrete values:
Test inputs

Symbolic
Execution



Constraint
solving



Test **code**
generation

Java+JML

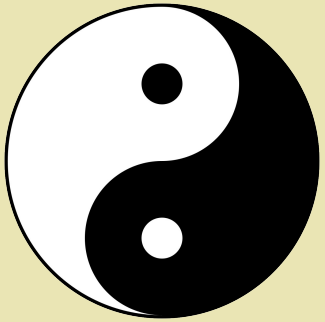
Runnable
Test
Suite

KeYTestGen

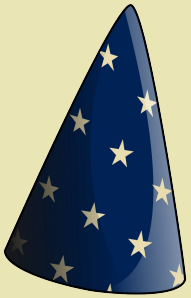
Set of constraints
Describing paths
inside the code

Concrete values:
Test inputs

Symbolic
Execution



Constraint
solving



Test **code**
generation

Java+JML

Postcondition: decides test pass/fail

Runnable
Test
Suite

Symbolic execution

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

Symbolic execution

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

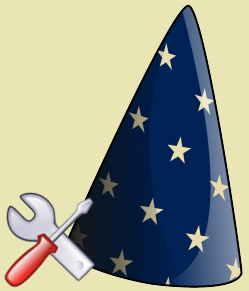
Symbolic execution

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

Symbolic execution

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

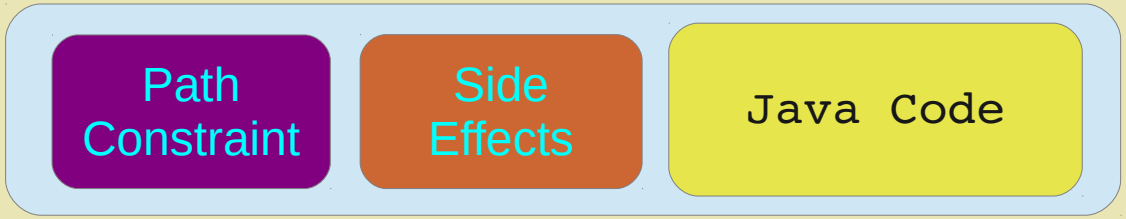
Constraint solvers

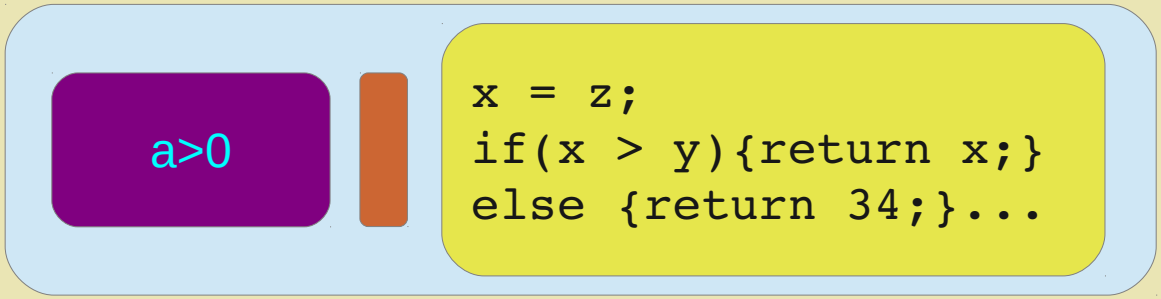


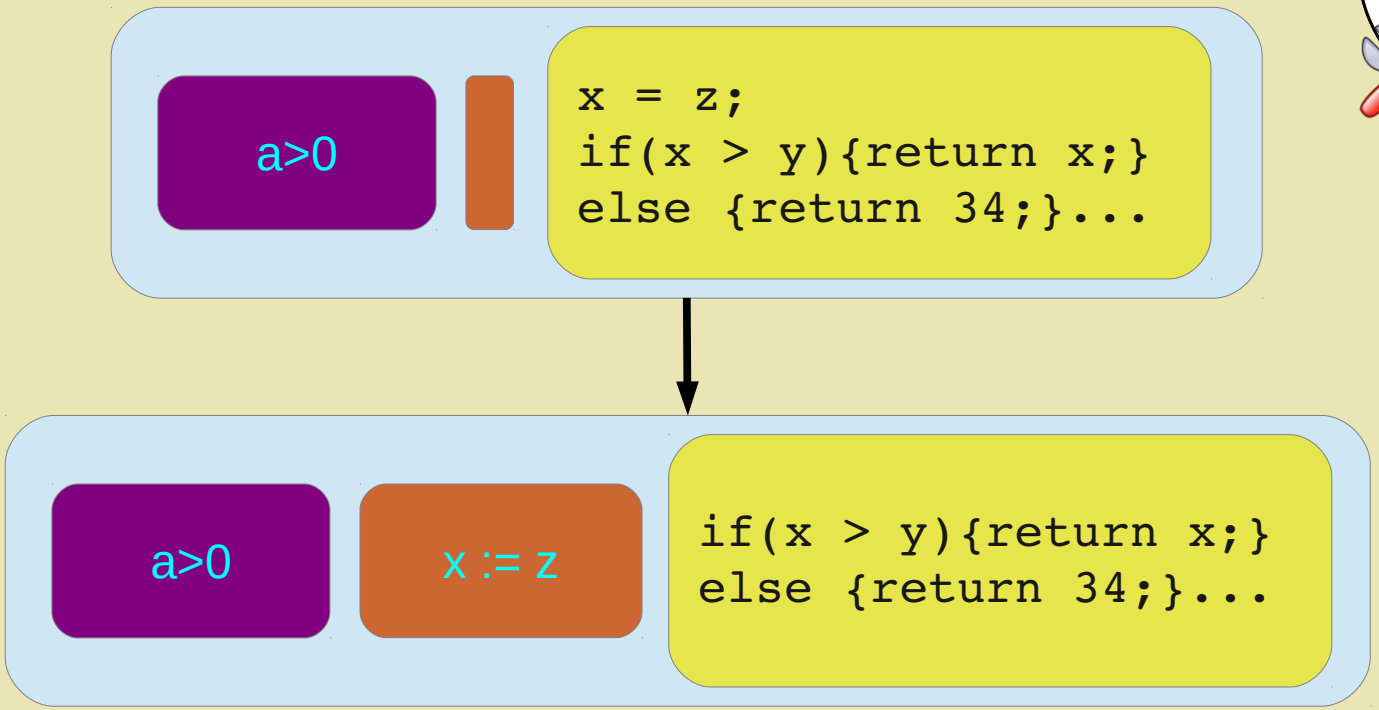
- Input:
  - Logical formula F containing variables V
    - Numerical
    - Boolean
    - ...
- Output:
  - An assignment to V making F *true*, if it exists
  - Unknown/unsatisfiable otherwise

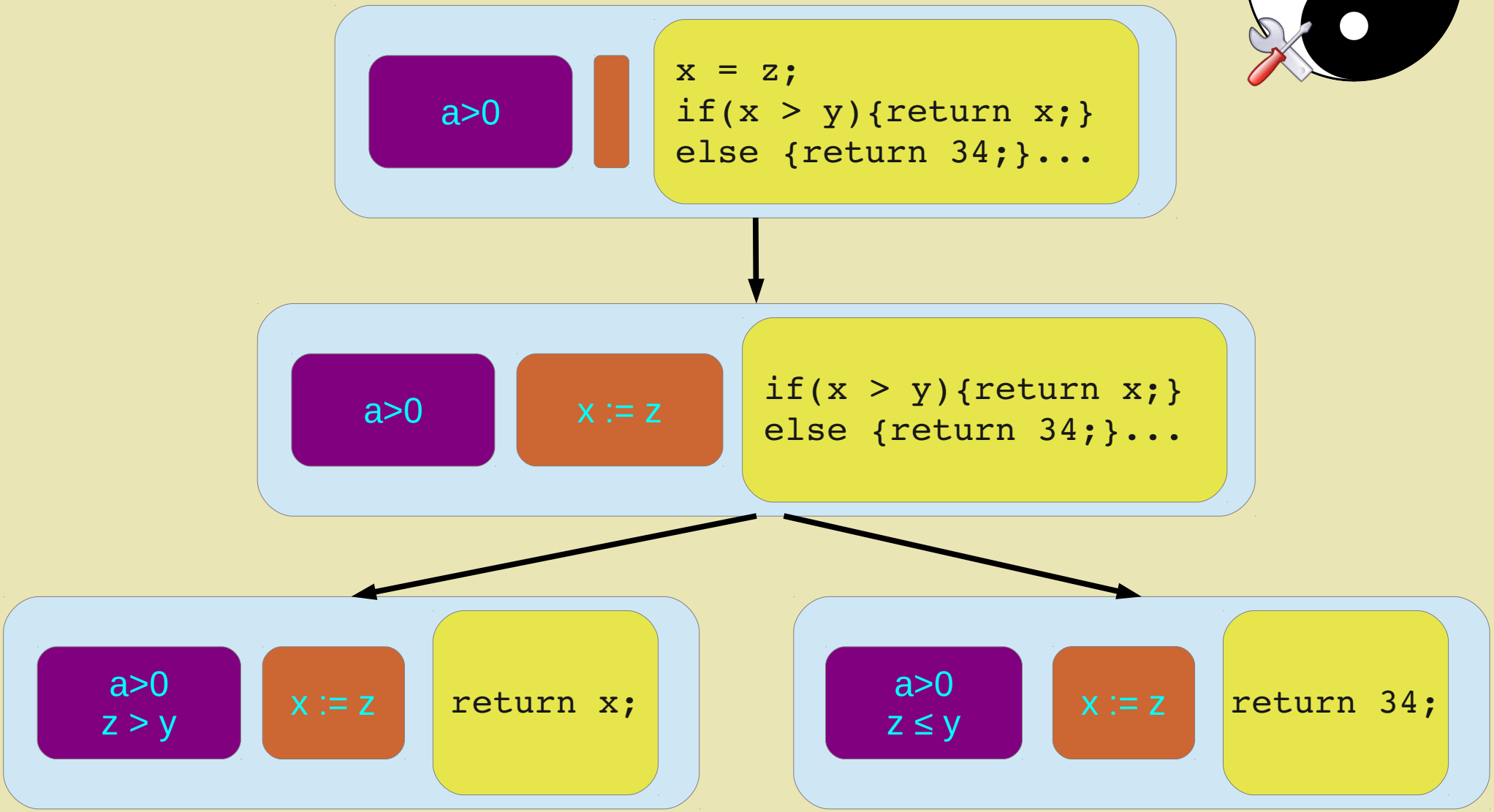
KeYTestGen

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

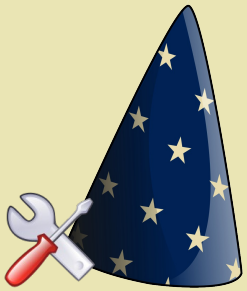



a>0
x = z;
if(x > y){return x;}
else {return 34;}...


a>0
x = z;
if(x > y){return x;}
else {return 34;}...
a>0
x := z
if(x > y){return x;}
else {return 34;}...


a>0
x = z;
if(x > y){return x;}
else {return 34;}...
a>0
x := z
if(x > y){return x;}
else {return 34;}...
a>0
z > y
x := z
return x;
a>0
z ≤ y
x := z
return 34;

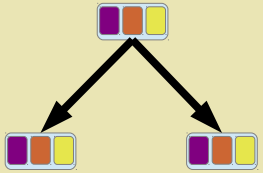
Path constraints



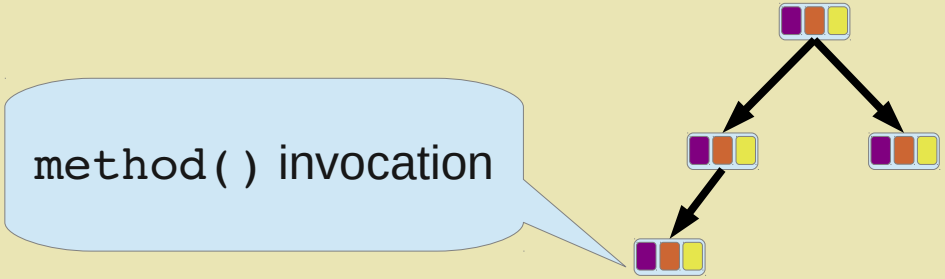
- Finding a satisfying assignment for the path constraint gives a test input
- Constraints are solved with external solvers
  - Simplify
  - Microsoft Z3

Specification matters: why?

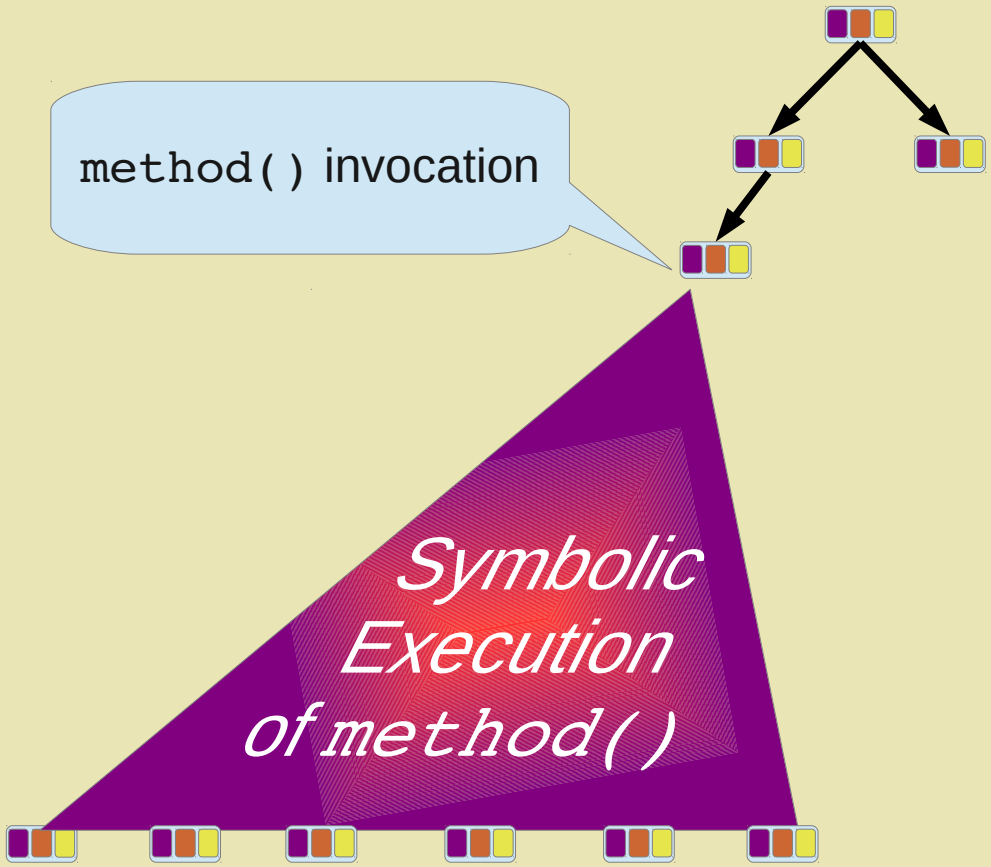
Specification matters: why?



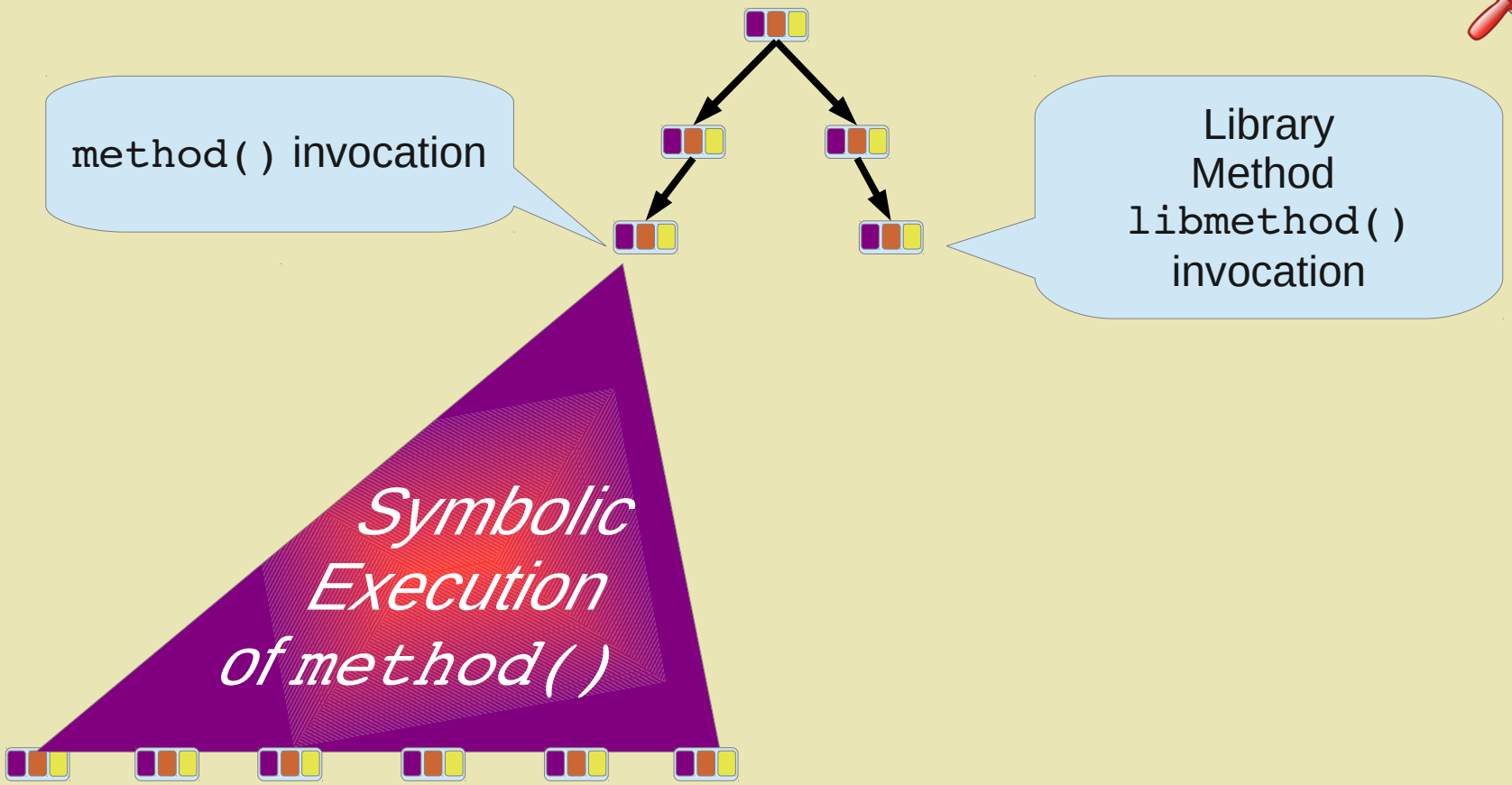
Specification matters: why?



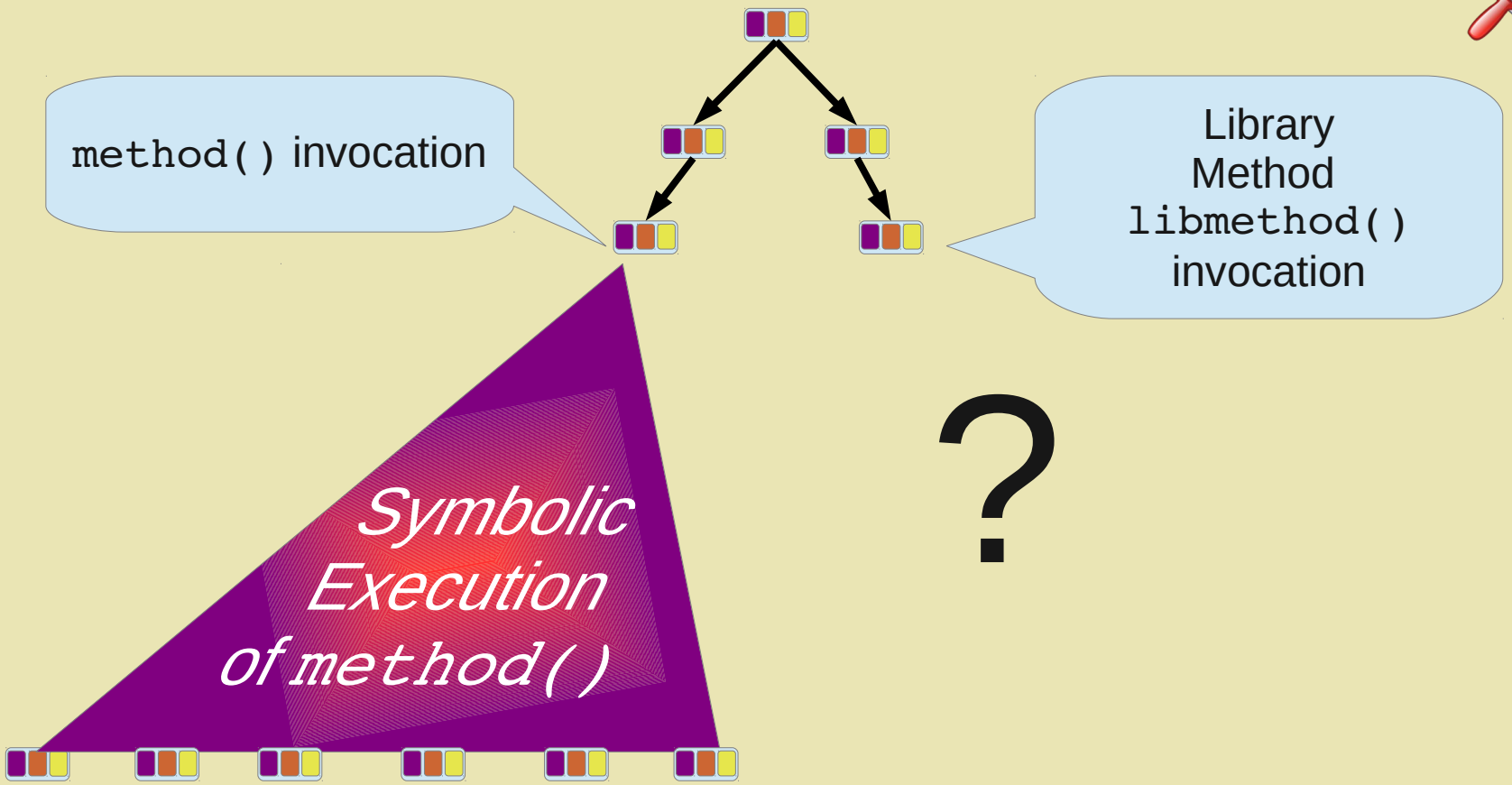
Specification matters: why?



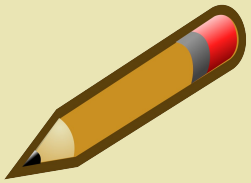
Specification matters: why?



Specification matters: why?

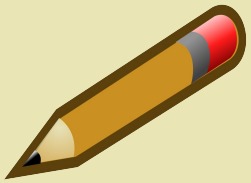


Specification in Theorem Proving based test case generation



```
public void underTest(){
        OtherMethod();
        LibraryMethod();
}
```

Specification in Theorem Proving based test case generation



Conjectural use

Assume the precondition holds. Is the postcondition satisfied?

```
public void underTest(){
        OtherMethod();
        LibraryMethod();
}
```

- Encodes the tests

Specification in Theorem Proving based test case generation

Conjectural use

Assume the precondition holds. Is the postcondition satisfied?

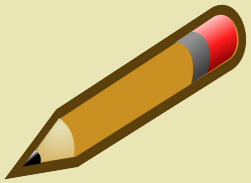
```
public void underTest(){
        OtherMethod();
        LibraryMethod();
}
```

- Encodes the tests

Axiomatic use

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

- Replaces code
- Keeps the method feasible



Formalization Of RTSJ

Limitations of KTG+Solvers:

- Quantifiers

Formalization
Of
RTSJ

Limitations of KTG+Solvers:

- Quantifiers

Modularity requirements:

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

Formalization Of RTSJ

Limitations of KTG+Solvers:

- Quantifiers

Modularity requirements:

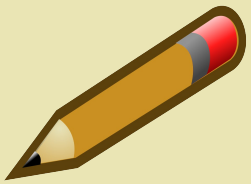
- Do not refer to implementation details (use specification-only fields)

Formalization Of RTSJ

Testing Requirements:

- Contracts preconditions have to cover all input space of methods

AsyncEventHandler specification



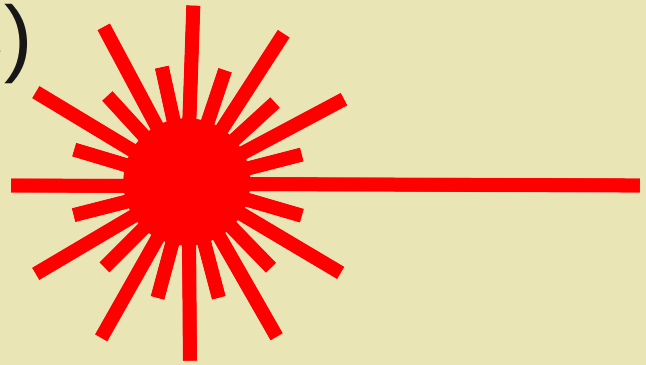
```
/*@ normal behavior
requires true;
ensures handler == null ||
                !handledBy(handler);
assignable this._handlers[*]; @*/
public void removeHandler(
        /*@ nullable @*/ AsyncEventHandler handler);
```

Evaluation

- Formal RTSJ used for evaluating:
  - **Verification**
    - Collision detector benchmark (CDx)
  - **Functional Testing of**
    - **client code** (Lightgun driver)
    - **API implementation** (JamaicaVM)

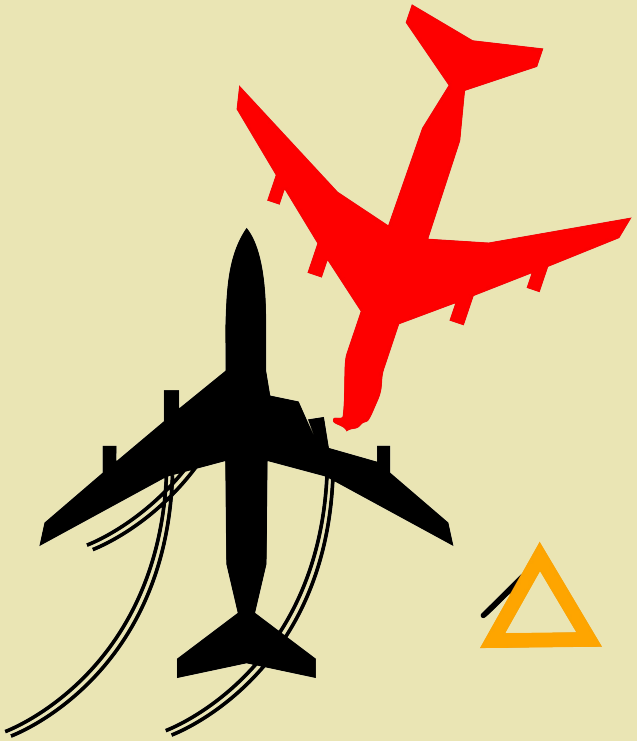
Evaluation: Testing Lightgun driver

- A small application ~ 700 loc
  - Driver for a CRT-compatible lightgun
  - Realtime: syncing with the screen refresh
- Tested (semiautomatically) easily with KeYTestGen
- Coverage: MC/DC (DO178B/C)
  - Achieved thanks to **Symbolic Execution**



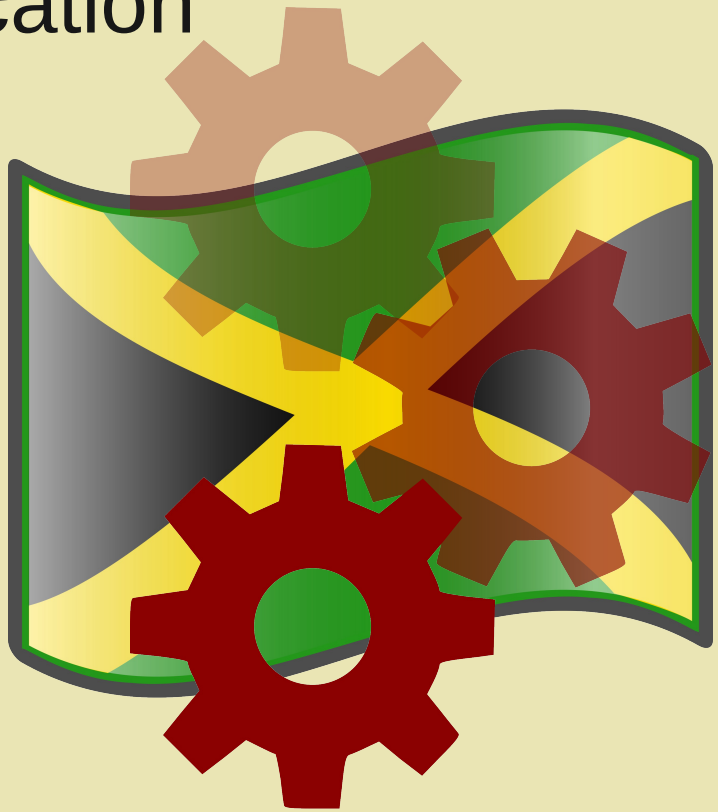
Evaluation:
Verifying correctness of RTSJ code

- CDx Real-Time Java Benchmark
- A collision detector for aerial traffic
- Proofs (with KeY) can be hard
  - Some automatic
  - Others require user input
    - Which is the norm



Evaluation: Testing of API implementation

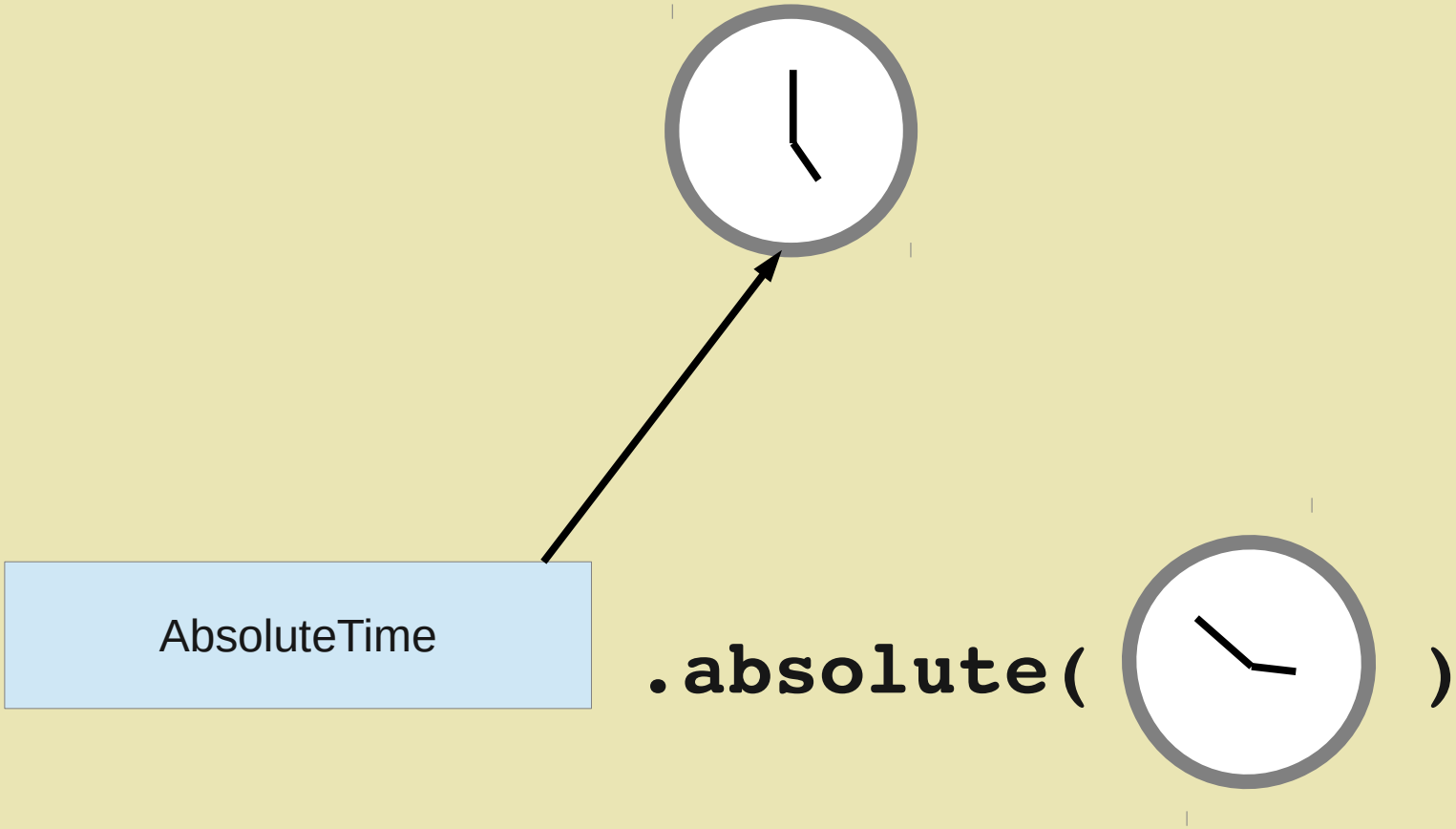
- JamaicaVM implementation
- Tested against the API specification
- Our method found a problem automatically



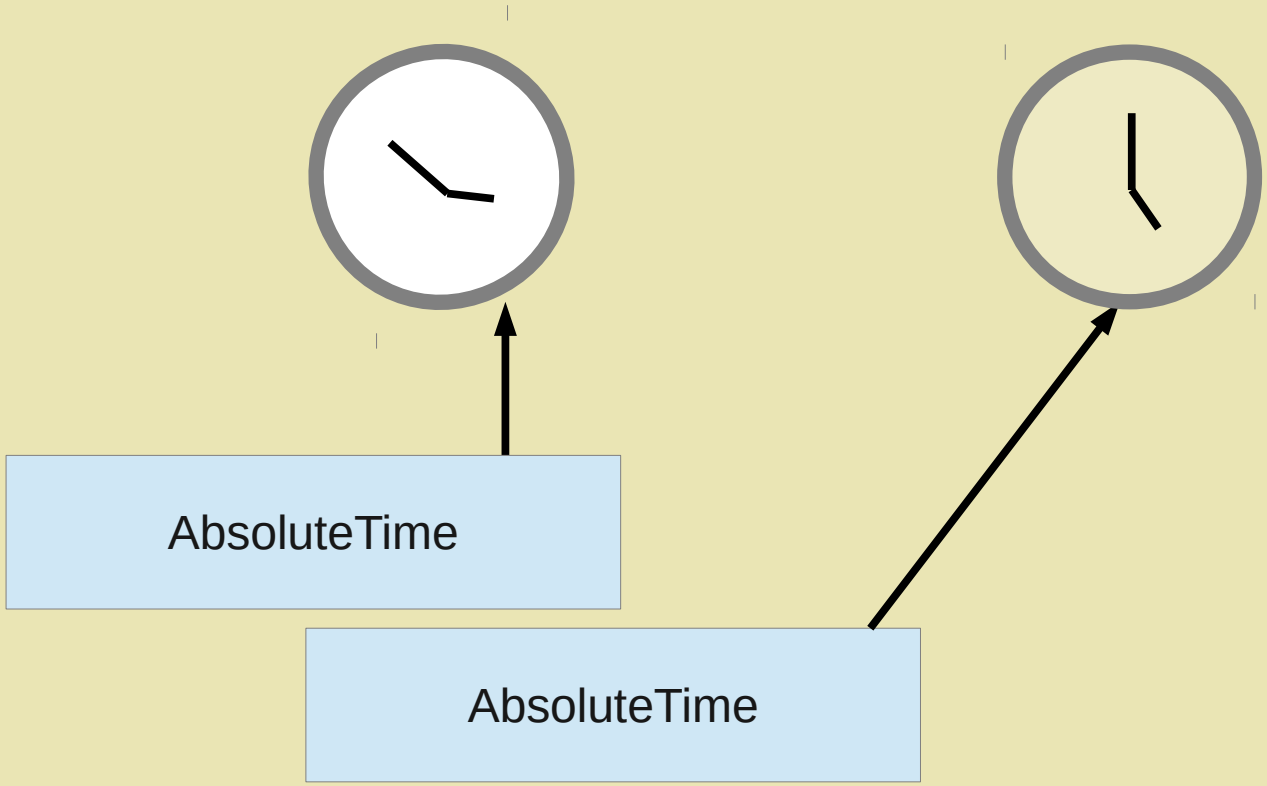
`absolute()` method

- `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. This method is the implementation of the abstract method of the HighResolutionTime base class. No conversion into AbsoluteTime is needed in this case. 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() 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);
```

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; RTSJ does not tell how to do this

Challenges and future+related work

- Quantifiers
  - Gladisch proposes an algorithm to handle them
- Concrete instantiation of reference type
  - Specification & solutions to constraints tells just what the result is, but not how to build it
  - Practice:
    - Quickcheck
    - JMLUnitNG (caching of constructors)

Questions

KeYTestGen

Formal Specification

Evaluation

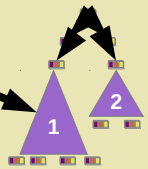
Symbolic execution

Constraint solving

Code generation

Code generation from specification

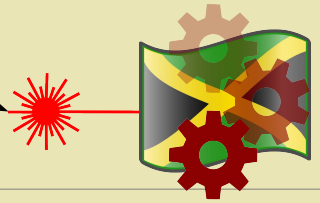
Dual usage of specification



Replacement of missing/unknown code

Feasibility of the approach

Verification: collision detector



Test case generation:
JamaicaVM, Ligthgun Driver

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/
- QuickCheck
  - http://java.net/projects/quickcheck/pages/Home
- JML formalized RTSJ API
  - http://wwwhome.ewi.utwente.nl/~mostowskiwi/