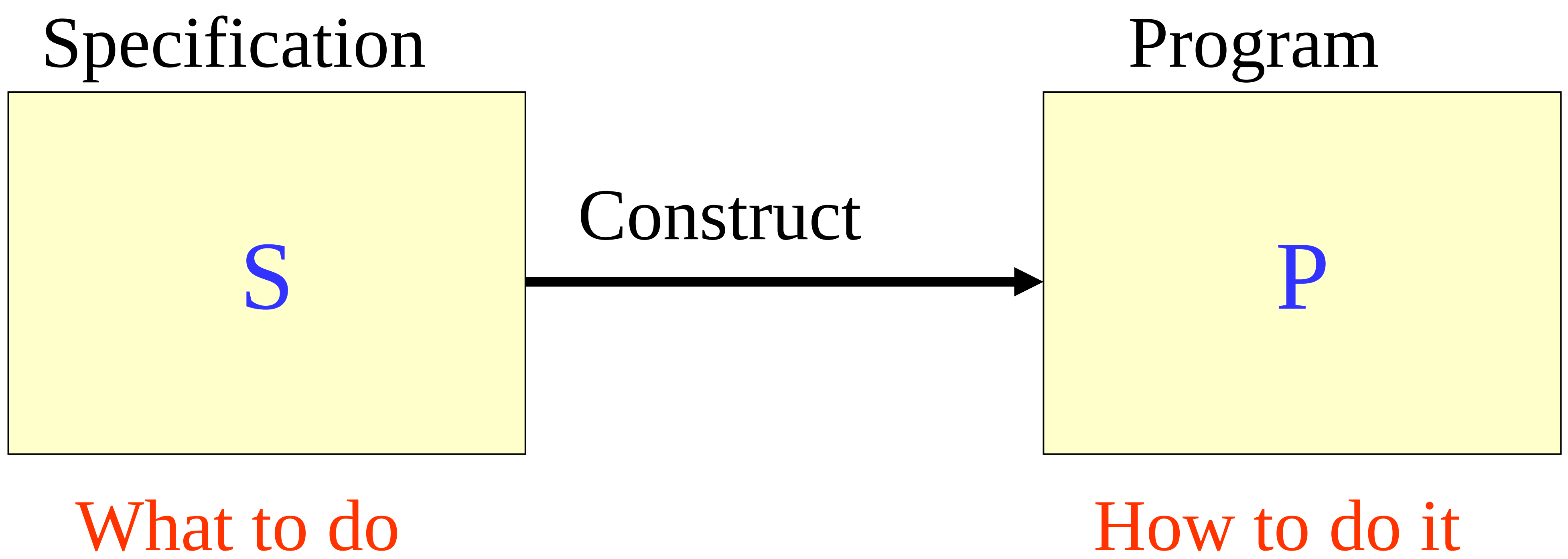
SENG3320/6320: Software Verification and Validation

School of Electrical Engineering and Computing

Semester 1, 2020

Formal Methods

Problems in software development



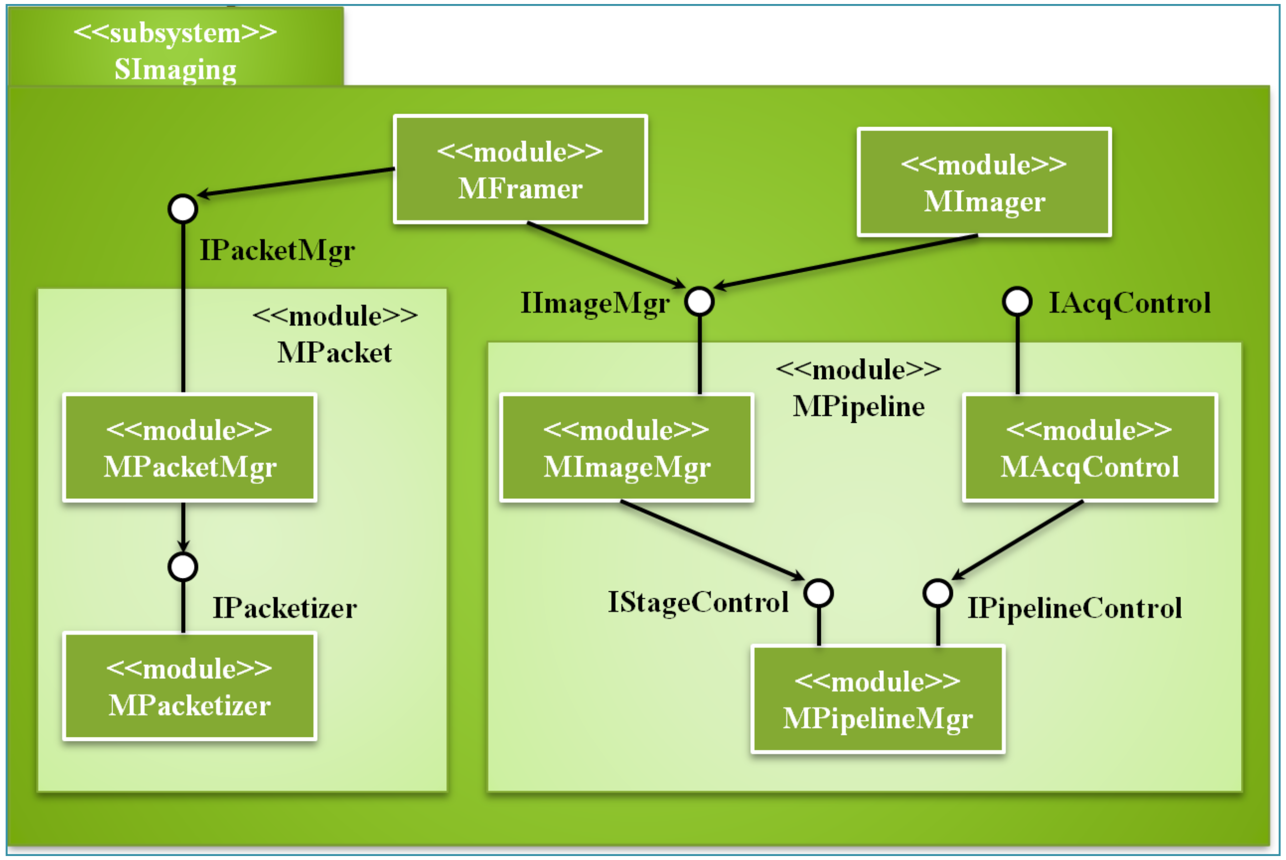
- How to ensure that S is not ambiguous so that it can be correctly understood by all the people involved?
- How can S be effectively used for inspecting and testing P?
- How can software tools effectively support the analysis of S, transformation from S to P, and verification of P against S?

Example of informal specifications

Natural language:

“A software system for an Automated Teller Machine (ATM) needs to provide services on various accounts. The services include …. The operations on a current or savings account include deposit, withdraw, show balance, and print out transaction records.”

Graphical:



The major problems with informal specifications:

- Informal specifications are likely to be ambiguous, which is likely to cause misinterpretations.
- Informal specifications are difficult to be used for inspection and testing of programs because of the big gap between the functional descriptions in the specifications and the program structures.
- Informal specifications are difficult to be analyzed for their consistency and validity.

A possible solution to these problems:

Formal Methods!

Formal Methods

- **Formal methods include**
  - Formal specification
  - Formal verification

These are all based on mathematical representation and analysis of software

- Set theory, (temporal/first-order/higher-order) logics, automata, etc.
- Unambiguous syntax & sound semantics

Formal Specifications

- Formal specification is concerned with producing an unambiguous set of product specifications so that customer requirements, as well as environmental constraints and design intentions are correctly reflected, thus reducing the chances of accidental fault injections.
- Formal specifications typically focus on the functional aspect or the correctness of expected program behaviour.
- With formal specifications, the desirable properties for software specifications can be more easily and sometimes formally analyzed and assured.

Formal Specifications

- Many formal specifications are descriptive.
- *Descriptive specifications* focus on the properties or conditions associated with software products and their components.
    - Model (or state-based) specifications focus on states and models (e.g Z and VDM languages).
    - Algebraic specifications focus on functional computation carried out by a program or program-segment and related properties. (e.g., Larch, Common Algebraic Specification Language)
- Foundations: set theory and logic

Formal Verification

- Formal verification techniques attempt to show the absence of faults.
    - Software testing shows the presence of defects, not their absence.
- The basic idea is to verify the correctness, or absence of faults, of some program code or design, against its formal specifications.
    - The presence of a formal specification is a prerequisite for formal verifications.

Formal Languages and Tools

- Many languages & Tools
  - Languages: Alloy, B, CSP, Event-B, Petri Nets, VDM, Z, Object-Z, LTL, CTL, etc.
  - Tools: Alloy Analyzer, Isabelle/HOL, ProB, Rodin, SPIN, SMV, Z/EVES, etc.
- Used for proof of correctness
  - Unlike testing: demonstration of presence of defects

An Example of Formal Specification in Z

$BirthdayBook$ Gives the name of the schema

$known : \mathbb{P}\, NAME$ Variable declarations go above the dividing line

$birthday : NAME \rightarrowtail DATE$

$known = \mathrm{dom}\, birthday$ Conditions go beneath the line

$AddBirthday$ Name of the Schema

$known : \mathbb{P}\, NAME$ Explicit inclusion of all declarations

$known' : \mathbb{P}\, NAME$

$birthday : NAME \rightarrowtail DATE$

$birthday' : NAME \rightarrowtail DATE$

$name? : NAME$

$date? : DATE$

$known = \mathrm{dom}\, birthday$ Invariants included by hand

$known' = \mathrm{dom}\, birthday'$

$name? \notin known$ Predondition

$birthday' = birthday \cup \{name? \mapsto date?\}$ Postcondition

Tools: SPIN Model Checker

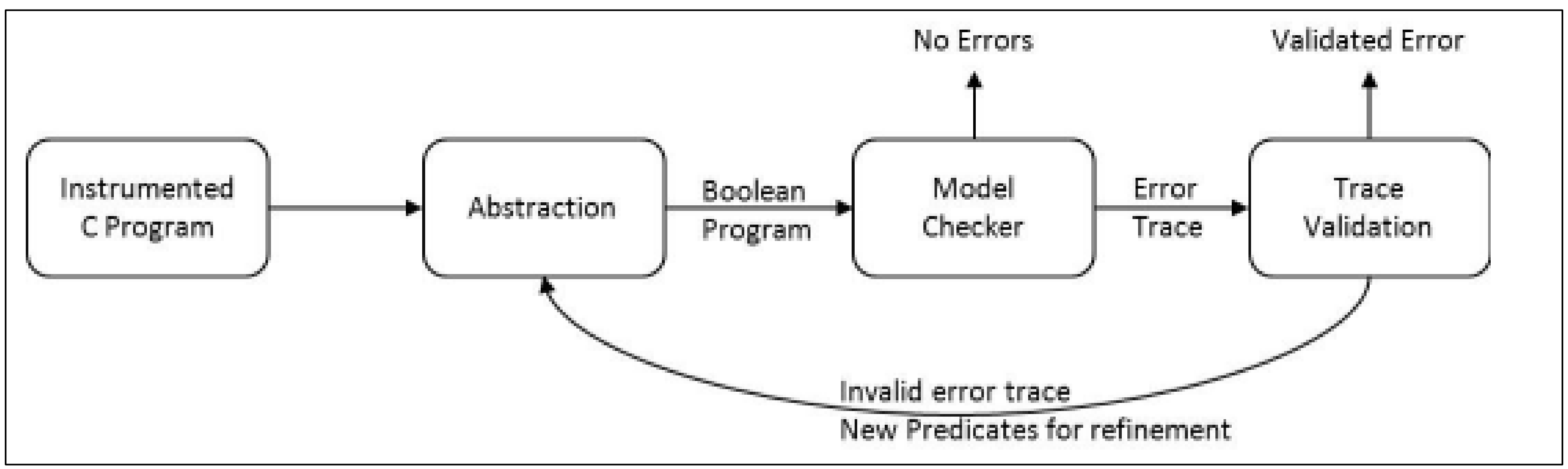
- **S**imple **P**romela **IN**terpreter
- Well established and freely available model checker
  - Model processes
  - Verify linear temporal logic properties

http://spinroot.com/

2015 (c) C. Le Goues

Tools: Microsoft SLAM

- Statically check Microsoft Windows drivers
  - Drivers are difficult to test
- Uses Counter *Example-Guided Abstraction Refinement* (CEGAR)
  - Based on model checking
  - Refines over-approximations to minimize false positives



CEGEAR process implemented in SLAM

http://research.microsoft.com/en-us/projects/slam/

2015 (c) C. Le Goues

Tools: ESC/Java

- Extended Static Checking for Java
  - Uses the Simplify theorem prover to evaluate each routine in a program
  - Embedded assertions/annotations
    - Checker readable comments
    - Based on the Java Modeling Language (JML)

```
/*@ requires i > 0 */
public void div(int i, int j) {
  return j/i;
}
```

C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. *Extended static checking for Java*

2015 (c) C. Le Goues

Use of Formal Methods

- **The principal benefits of formal methods are in reducing the number of errors in systems** so their main area of applicability is **critical systems:**
  - **Air traffic control information systems,**
  - **Railway signalling systems**
  - **Spacecraft systems**
  - **Medical control systems**
- In this area, the **use of formal methods is most likely to be cost-effective**

Formal Methods – Benefits / limitations

+ Requirements and specifications are unambiguous
+ Errors due to misunderstandings are reduced
+ Eases implementation
+ Correctness proofs can be carried out
+ Validation of requirements specifications is possible

- Formal specifications are difficult to read
- Can't model all aspects of real world
- Correctness proofs are resource intensive
- Development cost increases

Z Language

Sets

- Are collections of elements
- Are represented by standard notation
- Are manipulated by standard elementary operations
- Are entities which can interact with each other

Examples of Sets

Notation:{ }

- Set of colours: {green,blue,yellow}
- Set of sports: {tennis,football, equestrian}
- Set of courses: {SENG3130, SENG6140}
- Empty set: { } or ∅

Set Construction

$$\{set : range | condition \cdot operation\}$$

- In notational form (aka comprehensive specification):

{Signature | Predicate • Term}

$$\{x : X \,|\, P(x) \cdot E(x)\}$$

Alternate even numbers:

$$\{ x : N \,|\, x \bmod 2 = 0 \cdot 2 * x \} = \{0,4,8,12,16,20,\ldots\}$$

Tens:

$$\{ x : N \,|\, x \cdot 10 * x \} = \{0,10,20,30,40,\ldots\}$$

Set Operators

Sets:

$S : \mathbb{P}\,X$	S is declared as a set of X's.
$x \in S$	x is a member of S.
$x \notin S$	x is not a member of S.
$S \subseteq T$	S is a subset of T: Every member of S is also in T.
$S \cup T$	The union of S and T: It contains every member of S or T or both.
$S \cap T$	The intersection of S and T: It contains every member of both S and T.
$S \setminus T$	The difference of S and T: It contains every member of S except those also in T.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just x.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\,X$	S is declared as a finite set of X's.
$\max(S)$	The maximum of the nonempty set of numbers S.

Functions:

$f : X \rightarrowtail\!\!\!\!\!\!+\; Y$	f is declared as a partial injection from X to Y
$\mathrm{dom}\ f$	The domain of f: the set of values x for which $f(x)$ is defined.
$\mathrm{ran}\ f$	The range of f: the set of values taken by $f(x)$ as x varies over the domain of f.
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y.
$\{x\} \mathbin{\unlhd\!\!\!-} f$	A function like f, except that x is removed from its domain.

Logic:

$P \wedge Q$	P and Q: It is true if both P and Q are true.
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false.
$\theta S' = \theta S$	No components of schema S change in an operation.

Set Operators

- Examples:
  - Newcastle $\in$ {NSW cities}
  - Melbourne $\notin$ {NSW cities}
  - {Newcastle, Sydney} $\subseteq$ {NSW cities}
  - #{Newcastle, Sydney} = 2

  - {Tom, Jim, James} $\cup$ { James, Kathy} = {Tom, Jim, James, Kathy}
  - {Tom, Jim, James} $\cap$ { James, Kathy} = { James}
  - {Tom, Jim, James} \ { James, Kathy} = {Tom, Jim}

The Z Language

- A well-known language formal specification
- Based on set theory
- Equally used to model (specify) state as well as operations on states
- First developed in1977–1990 at the University of Oxford with industrial partners (IBM, Inmos)

An Example

Standard Example from the Z reference manual:
The Birthday Book

- Stores a list of names with associated birthdays
- Operations for:
  - Add new people
  - Query for a given persons birthday
  - Query for all birthdays on a given date
  - …

For the birthday book, we have to deal with two basic types:

- Names of people
- Birthday dates (day and month)

To introduce the two types, we just specify them: *[NAME, DATE]*.

An Example

$BirthdayBook$ Gives the name of the schema

$known : \mathbb{P}\, NAME$ Variable declarations go above the dividing line

$birthday : NAME \mathrel{+\!\!\!\!\rightarrow} DATE$

$known = \mathrm{dom}\, birthday$ Conditions go beneath the line

- One possible state of the system has three people in the set known, with their birthdays recorded by the function birthday:

  known = { John; Mike; Susan }

  birthday = { John 25-Mar,
  Mike 20-Dec,
  Susan 20-Dec
  }

$AddBirthday$ Name of the Schema

$known : \mathbb{P}\, NAME$ Explicit inclusion of all declarations

$known' : \mathbb{P}\, NAME$

$birthday : NAME \nrightarrow DATE$

$birthday' : NAME \nrightarrow DATE$

$name? : NAME$

$date? : DATE$

$known = \mathrm{dom}\; birthday$ Invariants included by hand

$known' = \mathrm{dom}\; birthday'$

$name? \notin known$ Precondition

$birthday' = birthday \cup \{name? \mapsto date?\}$ Postcondition

- name is the before state
- name' is the after state
- name? is an input variable
- name! is an output variable

The simplified version:

$$
\begin{array}{l}
\textit{AddBirthday} \\
\hline
\Delta \textit{BirthdayBook} \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\
\hline
\end{array}
$$

Δ is implicitly used to:

- Include SchemaName and SchemaName'
- Signal to the reader that this modifies the state

A modifying operation describes the relationship of two states, one before and one after the operation

Pre- and Postconditions

Precondition:

- known = dom birthday
- name? $\notin$ known

Postcondition:

- birthday' = birthday $\cup$ {name? $\rightarrow$ date?}

Formal verification can be performed based on Precondition and Postcondition

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindBirthday}} \\
\hline
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?) \\
\hline
\end{array}
$$

$\Xi SchemaName$ is used for operations that do not change the state

Some operations do not change the state of a system, but just query it. Examples in our application:

- Given a person, find the birthday
- Given a date, find all persons

Java Modeling Language

What is JML? (www.jmlspecs.org)

- History
  - Emerged in early 2000s out of ESC/Java2
- Goals
  - Integration of formal methods throughout the software process
  - Formal specification accessible to programmers
  - Direct support for design by contract
  - Integration with a real language (Java)
- JML allows us to mix specifications directly with the Java code
  - Preconditions
  - Postconditions
  - Loop invariants
  - Class invariants
- Tool: http://www.openjml.org/

JML Annotations

- Not Java annotations (starting with @)
- JML annotation comments
  - Line starting with //@
  - Between /*@ and @*/, ignoring @'s starting lines
- Properties are specified as Java boolean expressions, extended with a few operators (*\old*, *\forall*, *\result*, . . .).
- Using a few keywords (*requires*, *ensures*, *signals*, *assignable*, *pure*, *invariant*, *non null*, . . .)

33

JML Basics

JML specifications are *special comments* in a Java program:

`//@` for one-liners

`/*@ …. @*/` for multiple-liners

Pre-conditions/post-conditions:

$\{P\}\ s_1; s_2; \ldots; s_n\ \{Q\}$ ⇨

is written in JML/Java as

(*P* and *Q* are written as Java boolean expressions, and use parameters, local, and class variables as arguments.)

```
/*@ requires P ;
      ensures Q ;
@*/
type method (parameters) {
   local variables
   s1; s2; …; sn
}
```

Some JML Keywords

JML Expression	Meaning
`requires p ;`	`p` is a precondition for the call
`ensures p ;`	`p` is a postcondition for the call
`signals (E e) p;`	When exception type `E` is raised by the call, then `p` is a postcondition
`loop_invariant p;`	`p` is a loop invariant
`invariant p ;`	`p` is a class invariant (see next section)
`\result == e`	`e` is the result returned by the call
`(\product int x ; p(x); e(x))`	$\prod_{x \in \mathtt{p(x)}} \mathtt{e(x)}$; i.e., the product of `e(x)`

JML – An Example

```
public class MaybeAdd {
    //@ requires a > 0;
    //@ requires b > 0;
    //@ ensures \result == a+b;
    public static int add(int a, int b){
        return a-b;
    }

    public static void main(String args[]){
        System.out.println(add(2,3));
    }
}
```

OpenJML (ESC) @ rise4fu ×
www.rise4fun.com/OpenJMLESC/MaybeAdd

openjml (esc)

Formal Verification using JML
http://www.openjml.org/

Does this program do what it is supposed to do?

```
1  // Can you spot the two errors
2  // in this program?
3
4  public class MaybeAdd {
5      //@ requires a > 0;
6      //@ requires b > 0;
7      //@ ensures \result == a+b;
8      public static int add(int a, int b){
9          return a-b;
10     }
11
12     public static void main(String args[]){
13         System.out.println(add(2,3));
14     }
15 }
16
```

DISCLAIMER: OpenJML (ESC) is a 3rd party tool offered by OpenJML. By clicking '▸', you instruct rise4fun to send the source to OpenJML's OpenJML (ESC) to be analyzed. Please refer to the terms of use and privacy policy of OpenJML (ESC). Contact support for details.

home permalink
'▸' shortcut: Alt+B

	Description	Line	Column
1	The prover cannot establish an assertion (Postcondition: /tmp/tmpujhJ23/MaybeAdd.java:7:) in method add	0	0

```
/tmp/tmpujhJ23/MaybeAdd.java:9: warning: The prover cannot establish an assertion (Postcondition: /tmp/tmpujhJ23/MaybeAdd.java:7: ) in method add
        return a-b;
        ^
/tmp/tmpujhJ23/MaybeAdd.java:7: warning: Associated declaration: /tmp/tmpujhJ23/MaybeAdd.java:9:
    //@ ensures \result == a+b;
        ^
2 warnings
```

This API call handled by the Verily Web Framework: http://goverily.org

Further Reading

- Jean François Monin and Michael G. Hinchey, *Understanding formal methods*, Springer, 2003.
- J.M. Spivey. “An introduction to Z and formal specifications”. Software Engineering Journal 4(1):40-50, 1989.

  http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=28089&tag=1

- Jim Woodcock and Jim Davies, “Using Z: Specification, Refinement, and Proof’’, Prentice Hall, 1996.
- Hoare, An axiomatic basis for computer programming, Communications of the ACM 12(10):576-580.
- JML tool and Reference Manual: http://www.openjml.org/

Thanks!


Testing
Software
development
design
articles
management
test
quality
code
risk
good
usability
People
Project
information
web
use
Agile
article
system
check
testers