The Use of JML in Embedded Real-Time Systems

Joseph Kiniry
Technical University of Denmark

JTRES 2012
24 October 2012

Acknowledgements

- Some content based on an OOPSLA tutorial by: Gary T. Leavens, Curtis Clifton, Hridesh Rajan, and Robby
- which in turn was based on a CAV tutorial by: Gary T. Leavens, Joseph R. Kiniry, and Erik Poll
- which in turn was based on ECOOP, ETAPS, FM, FMCO, and TOOLS tutorials by some of the above and: David Cok, Fintan Fairmichael, and Dan Zimmerman

This Talk

- a bit of a Java Modeling Language tutorial
  - (to help all of you who are using JML in your research and talks not have to re-introduce JML in each talk and to proselytize a bit about the language)
- details about constructs relevant to specifying and reasoning about RT Java
  - (some advanced facets of the language)
- identification of research opportunities
  - (try to be visionary and inspirational)

The Java Modeling Language (JML)

- Today:
  - formal
  - sequential
  - functional behavior
  - mathematical models
  - Java 1.4, JavaCard, Personal Java, etc.
- Ongoing:
  - mechanized semantics
  - multithreading
  - temporal logic
  - resources
  - Java 1.5 and later

JML’s Goals

- usable by and useful for “normal” Java programmers
- JML syntax is an extension of Java’s syntax
- practical and effective for detailed model-based designs
- useful for specifying existing code or performing design-by-contract
- support a wide range of tools

Detailed Design Specification

- JML handles:
  - inter-module interfaces
  - classes and interfaces
  - fields (data)
  - methods (behavior)
- JML does not handle:
  - user interface
  - architecture
  - dataflow
  - design patterns

Basic Approach

- Floyd/Hoare-style specifications (contracts)
- method pre- and postconditions
  - preconditions are client obligations
  - postconditions are supplier obligations
- class and object invariants
  - invariants must hold during quiescence
- ...and then add a load of features necessary to specify programs in an OO language as rich (and messy, and complex) as Java

A First JML Specification Example

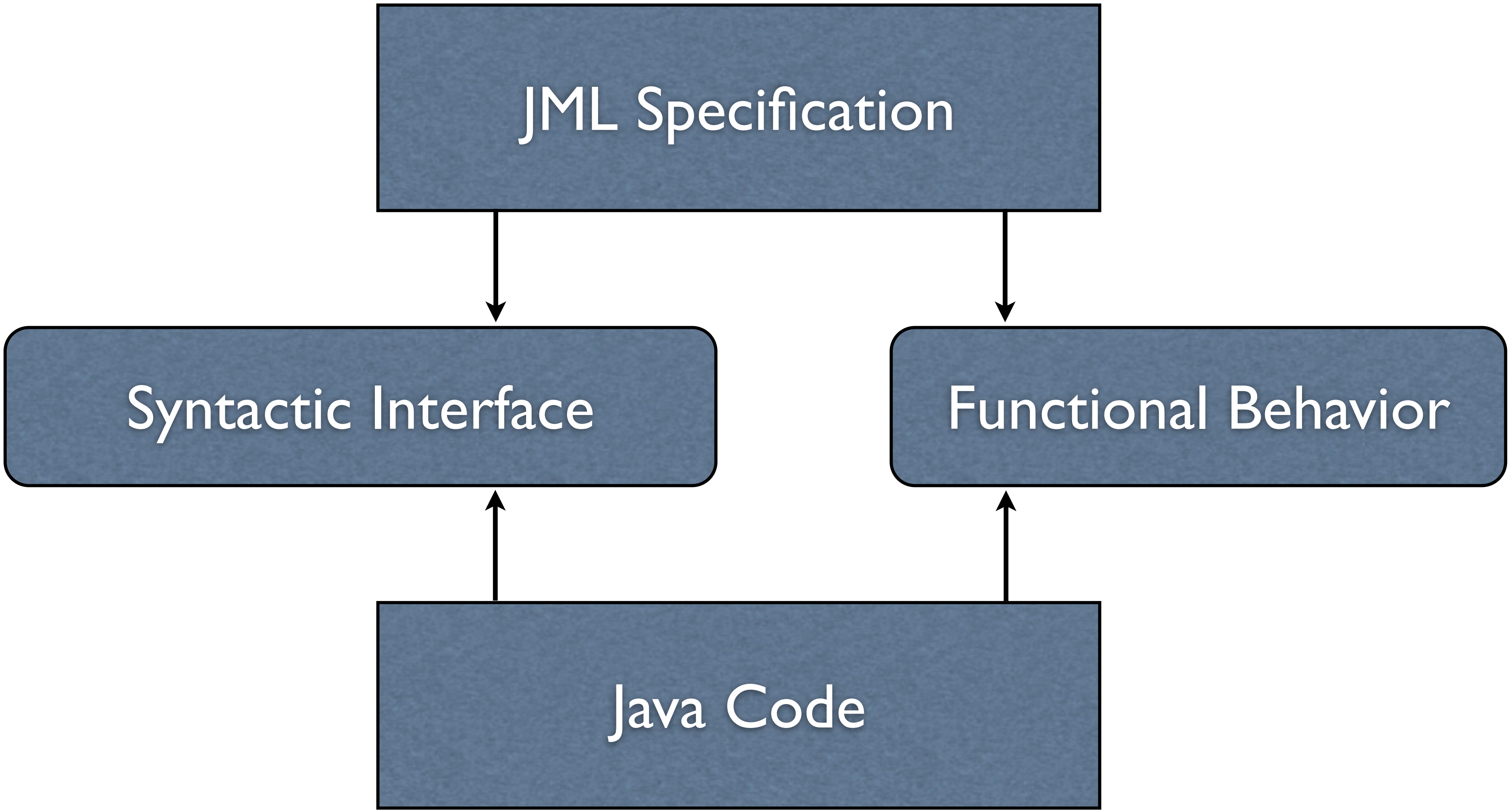
```
public class ArrayOps {
  private /*@ spec_public @*/ Object[] a;
  //@ public invariant 0 < a.length;
  /*@ requires 0 < arr.length
    @ ensures this.a == arr; @*/
  public void init(Object[] arr) {
    this.a = arr;
  }
}
```

field specification

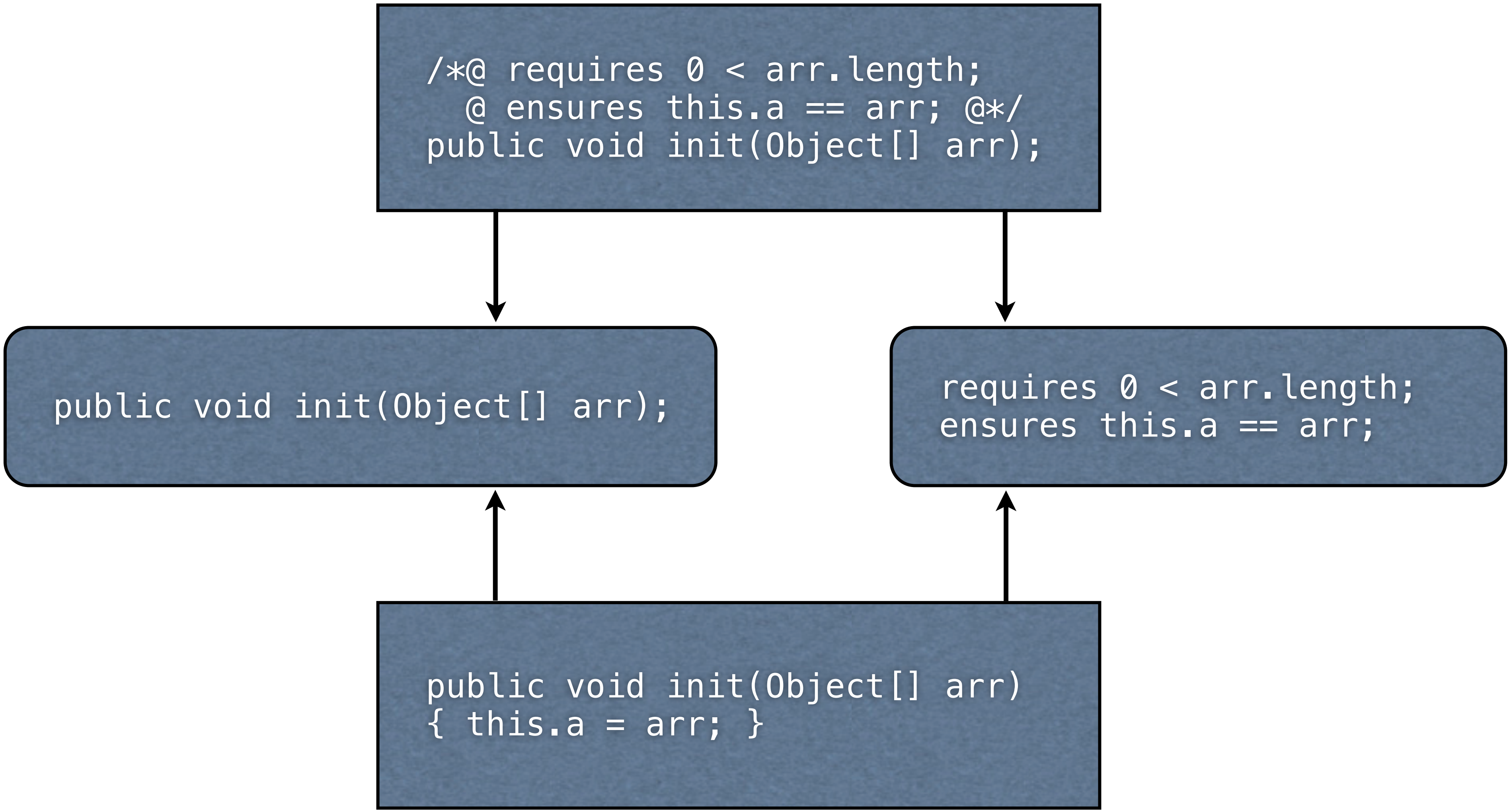
object invariant

method specification

Interface Specification



Interface Specification



Advanced Features

- specifications that include just pre- and postconditions and invariants are just the tip of the iceberg
- a variety of convenience annotations are available for common specification patterns
  - non-null default semantics, non-null elements in collections, strong validity of expressions, specification lifting for fields; initial state and history constraints; redundant specifications; exceptional termination; informal specifications; freshness; purity; examples; set comprehension; concurrency patterns
- a multitude of concepts that support rich specifications also exist
  - lightweight vs. heavyweight specs; privacy modifiers and visibility; instance vs. static specs; alias control via the universe type system; data refinement; datagroups; heap access and reachability; first-order quantifiers and boolean logic operators; generalized quantifiers; type operators; loop annotations; assumptions and assertions; axioms; several models of arithmetic; non-termination; frame axioms

Advanced Example(s)

```
// The classic Bag of integers example

class Bag {
  int[] a = new int [0];
  int n;

  Bag(int[] i) {
      n = i.length;
      a = new int[n];
      System.arraycopy(i, 0,
                       a, 0, n);
  }

  int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    if (a != null) {
        for (int i = 1; i <= n; i++) {
            if (a[i] < m) {
                mindex = i;
                m = a[i];
            }
        }
        n--;
        a[mindex] = a[n];
        return m;
    } else {
        return 0;
    }
  }
}
```

Lightweight Specs

```
class Bag {
  int[] a;
  int n;
  //@ invariant 0 <= n && n <= a.length;
  //@ public ghost boolean empty;
  //@ invariant empty == (n == 0);

  //@ modifies a, n;
  //@ ensures this.empty == (input.length == 0);
  public /*@ pure */ Bag(int[] input) {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
    //@ set empty = n == 0;
  }

  //@ ensures \result == empty;
  public /*@ pure @*/ boolean isEmpty() {
    return n == 0;
  }

  //@ requires !empty;
  //@ modifies empty;
  //@ modifies n, a[*];
  public int extractMin() {
    int m = Integer.MAX_VALUE;
    int mindex = 0;
    for (int i = 0; i < n; i++) {
      if (a[i] < m) {
        mindex = i;
        m = a[i];
      }
    }
    n--;
    //@ set empty = n == 0;
    //@ assert empty == (n == 0);
    a[mindex] = a[n];
    return m;
  }
```

full, basic lightweight specification

notice the default non-null semantics

abstraction of “empty-ness”

introduce purity

in-line assertions for validation and verification

new methods to support specification abstraction

frame axioms for non-pure methods

Document It!

add Javadocs for humans

tighten specs on formal parameters

```
/**
 * A bag of integers.
 *
 * @author The DEC SRC ESC/Java research teams
 * @author Joe Kiniry (kiniry@acm.org)
 * @version JTRES-23102012
 */
class Bag {
  /** A representation of the elements of
      this bag of integers. */
  int[] my_contents;
  /** This size of this bag. */
  int my_bag_size;
  /*@ invariant 0 <= my_bag_size &&
        my_bag_size <= my_contents.length; */
  //@ public ghost boolean empty;
  //@ invariant empty == (my_bag_size == 0);

  /**
   * Build a new bag, copying
   * <code>input</code> as its initial
   * contents.
   * @param the_input the initial contents
   * of the new bag. */
  //@ assignable my_contents, my_bag_size;
  /*@ ensures empty ==
        (the_input.length == 0); */
  public /*@ pure @*/ Bag(final int[]
                          the_input) { ... }

  /** @return if this bag is empty. */
  //@ ensures \result == empty;
  public boolean isEmpty() { ... }

  /** @return the minimum value in this bag
     and remove it from the bag. */
  //@ requires !empty;
  //@ modifies empty;
  //@ modifies my_bag_size, my_contents[*];
  public int extractMin() { ... }
}
```

hide unnecessary methods and method bodies henceforth

Lift Abstraction

introduce model variables

tighten visibility

hide Javadocs henceforth

specify exceptional behavior

use heavyweight specs

```
class Bag {
  private /*@ spec_public */ int[] my_contents;

  private /*@ spec_public */ int my_bag_size;
  /*@ invariant 0 <= my_bag_size &&
        my_bag_size <= my_contents.length; */

  //@ public ghost boolean empty;
  //@ invariant empty == (my_bag_size == 0);

  //@ public behavior
  //@   assignable my_bag_size, my_contents, empty;
  //@   ensures empty == (the_input.length == 0);
  //@   signals (Exception) false;
  public /*@ pure @*/ Bag(final int[] the_input)
  { ... }

  //@ public behavior
  //@   ensures \result == empty;
  //@   signals (Exception) false;
  public /*@ pure */ boolean isEmpty() { ... }

  //@ public behavior
  //@   requires !empty;
  //@   assignable empty, my_contents[*], my_bag_size;
  //@   signals (Exception) false;
  public int extractMin() { ... }
```

introduce datagroups

Data Abstraction

```
class Bag {
  private /*@ spec_public */ int[] my_contents;
    //@ in objectState;
  //@ maps my_contents[*] \into objectState;

  private /*@ spec_public */ int my_bag_size;
    //@ in objectState;
  /*@ invariant 0 <= my_bag_size &&
        my_bag_size <= my_contents.length; */

  //@ public ghost boolean empty; in objectState;
  //@ invariant empty == (my_bag_size == 0);

  //@ public behavior
  //@   assignable objectState;
  //@   ensures empty == (the_input.length == 0);
  //@   signals (Exception) false;
  public /*@ pure */ Bag(final int[] the_input)
    { ... }

  //@ public behavior
  //@   requires !empty;
  //@   assignable objectState;
  //@   signals (Exception) false;
  public int extractMin() { ... }
```

add data refinement

now supports specification evolution

Control Aliasing

use universe type system

```
class Bag {
  private /*@ \rep */ int[] my_contents;
    //@ in objectState;
  //@ maps my_contents[*] \into objectState;

  private /*@ \rep */ int my_bag_size;
    //@ in objectState;
  /*@ private invariant 0 <= my_bag_size &&
        my_bag_size <= my_contents.length; */

  //@ public model boolean empty; in objectState;
  //@ represents empty <- isEmpty();
  //@ public invariant empty <==> (my_bag_size == 0);

  //@ public behavior
  //@   assignable objectState;
  //@   ensures isEmpty() <==> (the_input.length == 0);
  //@   signals (Exception) false;
  public /*@ pure */ Bag(final int[] the_input) {
    my_bag_size = the_input.length;
    my_contents = new /*@ rep */ int[my_bag_size];
    System.arraycopy(the_input, 0,
      my_contents, 0, my_bag_size);
  }
```

refine specification visibility

use logical operators

Specs for Reasoning

```
class Bag {
  private /*@ \rep */ int[] my_contents;
    //@ in objectState;
  //@ maps my_contents[*] \into objectState;

  private /*@ \rep */ int my_bag_size;
    //@ in objectState;
  /*@ private invariant 0 <= my_bag_size &&
        my_bag_size <= my_contents.length; */

  //@ public model boolean empty; in objectState;
  //@ represents empty <- isEmpty();
  //@ public invariant empty <==> (my_bag_size == 0);

  //@ public behavior
  //@   assignable objectState;
  //@   ensures isEmpty() <==> (the_input.length == 0);
  //@   ensures my_contents.equal(the_input);
  //@   ensures my_bag_size == the_input.length;
  //@   signals (Exception) false;
  public /*@ pure */ Bag(final int[] the_input) { ... }

  //@ public behavior
  //@   requires !empty;
  //@   assignable objectState;
  //@   ensures my_bag_size == \old(my_bag_size - 1);
  //@   ensures (* one smallest element is removed *);
  /*@   ensures (\exists SortedSet set, int smallest,
                         List<int> list;
         list = Arrays.asList(my_contents) ==>
         set = new TreeSet(list) ==>
         smallest = s.first();
         Collections.frequency(list, smallest) ==
         \old(Collections.frequency(list,
              smallest) - 1)); */

  //@   signals (Exception) false;
  public int extractMin() { ... }
}
```

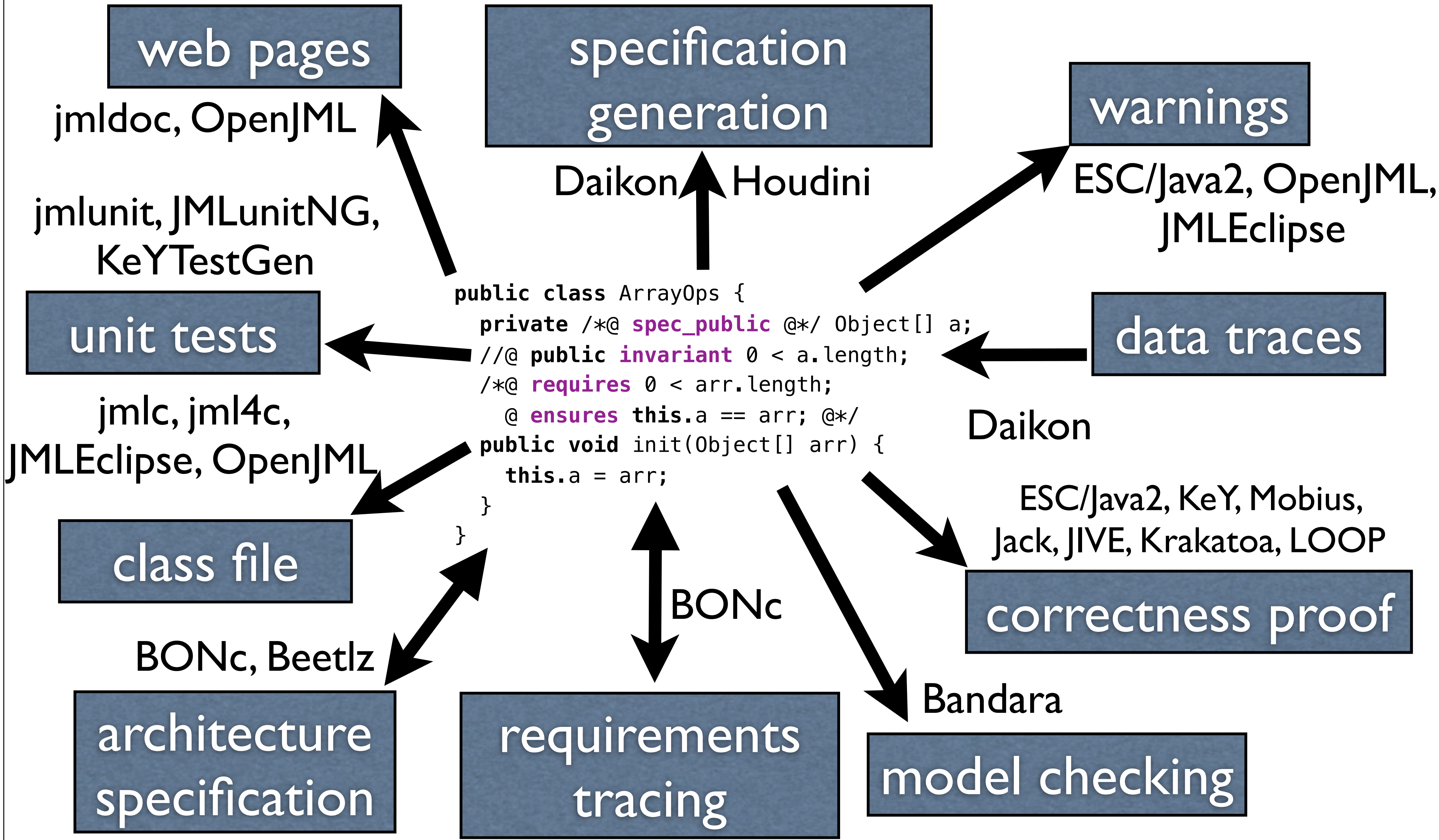
fully specify interface behavior

Internal Specs for Reasoning

```
public int extractMin() {
  int m = Integer.MAX_VALUE;
  int mindex = 0;
  /*@ maintaining m != Integer.MAX_VALUE ==>
      (\forall int j; 0 <= j & j < i & j != mindex;
       my_contents[j] < m & my_contents[mindex] == m);
   */
  //@ decreasing my_bag_size - i;
  for (int i = 0; i < my_bag_size; i++) {
    if (my_contents[i] < m) {
      mindex = i;
      m = my_contents[i];
    }
  }
  my_bag_size--;
  my_contents[mindex] = my_contents[my_bag_size];
  return m;
}
}
```

add loop specifications

Many Tools, One Language



Complementary Tools

- different strengths
  - runtime checking exhibits real errors
  - static checking ensures better coverage
  - verification provides strong guarantees

Typical Methodology

1. runtime checker (program and tests)
2. extended static checking
3. verification

Rigorous Methodology

1. perform formal analysis and high-level design (e.g., with UML or BON)
2. generate or hand-write detailed design in JML (Beetlz)
3. check soundness and measure quality of specifications using static checkers (Metrics, ESC/Java2)
4. generate unit tests (jmlunit, JMLunitNG, KeYTestGen)
5. use runtime checker during validation and execution
6. perform syntactic and semantic static analysis (CheckStyle, PMD, FindBugs, Metrics, ESC/Java2, Beetlz, AutoGrader)
7. perform verification (Jack, JIVE, Krakatoa, Mobius PVE, KeY, CHARGE!)

Interest in JML

- dozens of tools
- state-of-the-art specification language
- large and open research community
  - nearly 30 research groups worldwide
  - over 200 research papers published
  - dozens of PhD dissertations

See jmlspecs.org

Advantages to JML

- reuse language design
- ease communication with other researchers
- share customers for science and engineering

Join us!

More at www.jmlspecs.org

- documents
  - “Design by Contract with JML”
  - “An overview of JML tools and applications”
  - “Preliminary Design of JML”
  - “JML’s Rich, Inherited Specifications for Behavioral Subtypes”
  - “JML Reference Manual”
- Also:
  - Examples, teaching material.
  - Downloads, SourceForge project.
  - Links to papers, etc.

JML's Relevance to RT Java

- existing API specifications
- specification-only constructs
  - ghost fields
  - model fields, methods, classes, and programs
  - native models
- memory-related specification constructs
- resource specifications

Existing API Specs

- existing API specs for the JDK are poor, but for JavaCard and RT Java are quite good
- API specifications are written lazily and in bursts during JML "Specathons" run by myself and Zimmerman
  - a novel spec-writing process and tool support has been published in TAP'12
- moderately complete specification exist for few core JDK packages (java.[io, lang, util])
- poor specs exist for other core JDK packages (java.[awt, math, net, security, sql])
- complete specs exist for javacard.framework and javax.realtime thanks to Nijmegen researchers et al.

Ghosts

- **ghost** fields and variables are useful for explicitly modeling *explicit* specification-only data
- they are used inside of assertions like contracts and invariants
- their value is explicitly updated using the set statement
- recall:

```
//@ public model boolean empty; in objectState;
//@ represents empty <- isEmpty();
//@ public invariant empty <==> (my_bag_size == 0);
```

and inside of extractMin()

```
//@ set empty = n == 0;
//@ assert empty == (n == 0);
```

Models

- model fields, methods, classes, and programs are extremely useful for modeling platform constructs and algorithms
  - model programs are used to specify abstract algorithms and a concrete method's execution must refines its model program
  - model classes and methods are useful for abstracting domain concepts into a specification
    - e.g., novel memory models like in RT Java

Native Models

- native models permit one to define the semantics of a JML model in another formalism/tool
  - some JML model classes (pure, functional, executable, ADT-based sets, lists, bags, etc.) have native models expressed in Coq, Isabelle, or PVS
  - some JDK concurrency constructs have native models expressed in LTL or PVS
  - the Java memory model has native models expressed in rich heap models in various HOLs and SMT

Memory-related Specs

- **reach** expressions permit one to specify and reason about the set of objects reachable from a reference within a heap

```
//@ public invariant
//@   (\forall Object o, p, MemoryArea a, b;
//@     a = MemoryArea.getMemoryArea(o) &
//@     b = MemoryArea.getMemoryArea(p) & a != b;
//@     (a instanceof ImmortalMemory) &
//@     (b instanceof HeapMemory) ==>
//@     reach(b).intersection(reach(a)).isEmpty());
```

Resource Specs: Stack Depth

- **measured_by** permits one to specify the measure of recursion to reason about termination, a la PVS's measure construct, except limits to the integer type

```
factorial(x: nat): RECURSIVE nat =
    IF x = 0 THEN 1 ELSE x * factorial(x - 1) ENDIF
    MEASURE (LAMBDA (x: nat): x)

//@ measured_by x;
int factorial(int x) {
  if (x == 0) return 1;
  else return x * factorial(x-1);
}
```

Primitive Space Complexity

- **working_space** is used to specify the maximum amount of heap space, in bytes, used by a method call or constructor

```
//@ public behavior
//@   assignable objectState;
//@   ensures isEmpty() <==> (the_input.length == 0);
//@   signals (Exception) false;
//@   working_space 4 * the_input.length;
//@   working_space_redundantly
//@     \working_space(\type(int)) * the_input.length;
public Bag(final int[] the_input)
```

Space for an Object

- a **space** specification describes the amount of space consumed by an object (much like sizeof in the C family of languages)

```
//@ public behavior
//@   assignable objectState;
//@   ensures isEmpty() <==> (the_input.length == 0);
//@   ensures space(my_contents) == space(the_input);
//@   signals (Exception) false;
//@   working_space 4 * the_input.length;
public Bag(final int[] the_input)
```

Primitive Time Complexity

- the **duration** clause is used to specify the maximum number of virtual machine cycles a method (not counting garbage collection time)
- unfortunately, general-purpose VM cycle time for instructions has never been specified in the Java VM specification
- duration clause parameter is of type long, not an algebraic expression (not big-O notation)

Research Opportunities

- tool development and maintenance
- extensible tool architecture
- integration with modern IDEs
- unification of tools
- integration with Java annotations
- domain-specific language extensions
  - via new models and language extensions

JML Models and Extensions for RT Java

- RT Java deserves rich native model-based specifications for:
  - memory-related classes using a rich abstracted heap model
  - threads, scheduling, and synchronization
  - time, clocks, and timers
  - asynchrony

Java Level X Extensions for RT Java

- this community should propose and experiment with new JML annotations for:
  - time complexity that understands big-O (and related) notations
  - memory types
  - timers and asynchronous events
  - ACET and WCET scheduling

The State of JML

- many experimental compilers are available for "modern" Java
    - AJML2 (aspect-based), JAJML (JastAdd-based), JIR (DOM-like model of specified code), JML3 (Eclipse JDT-based), JMLEclipse (JDT-based also), OpenJML (OpenJDK-based), JML4 (JDT-based), JML6 (Java-annotation + JDT-based)
- OpenJML and JavaContract are the cleanest foundation for research tools

The Future of JML

- The future of JML is up to the community, which can easily include you.
- The language evolves due to community need and research opportunity.
- Tools get written and maintained because they are necessary for research, experimentation, and teaching.
- Personally, my group will continue to work on maintaining ESC/Java2, ADLs for Java (BON), refinement to/from JML (Beetlz), releasing a new Mobius PVE, finishing OpenJML, new specification and reasoning constructs for OO systems, lots of case studies, and writing "The JML Book" and "Dependable Software Engineering" with colleagues.