

Software Model Checking Using Bogor - a Modular and Extensible Model Checking Framework

3rd Estonian Summer School in
Computer and System Science (ESSCaSS'04)

Slide Set 07: Checking JML Specifications

<http://bogor.projects.cis.ksu.edu>
<http://www.cis.ksu.edu/~hatcliff/ESSCaSS04>

John Hatcliff

Matthew B. Dwyer

Robby

SANToS Laboratory, Kansas State University, USA

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Department of Defense
Advanced Research Projects Agency (DARPA)

Boeing
Honeywell Technology Center
IBM
Intel

Lockheed Martin
NASA Langley
Rockwell-Collins ATC
Sun Microsystems

Motivation and Acknowledgements

- All other model-checkers that we know of support only simple predicates on system states (e.g., the primitive propositions occurring in temporal logic formulas).
- Especially when modeling OO languages, states themselves can be quite complicated (they include the heap).
- Therefore we are interested in supporting specification predicates over Bogor states that are significantly stronger than those supported in other model-checking frameworks.
- Moreover, we are interested in supporting, as much as possible, rich specification languages that other verification tools using different technologies (e.g., theorem proving) also support.
- These slides are taken from our talk given at TACAS 2004 on "Checking Strong Specifications Using an Extensible Model-Checking Framework"
- A significant portion of this work was carried out by Edwin Rodriguez

Assertions for Software Verification

- Assertions have become a common practice among developers
 - 10 years ago assertions were not considered useful by developers
 - recent evidence of the effectiveness of assertions
 - David Rosenblum (1995)
 - now some programming languages have included assertions in their standard specifications
 - c.f. Java 1.4 assertions

Concurrent Queue based on Linked List (Doug Lea's util.concurrent package)

```
public class LinkedList {
    public Object value;
    public LinkedList next;
    public LinkedList(Object x) {
        value = x;
    }
    protected synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedList first = head.next;
            if (first != null) {
                x = first.value;
                first = first.next;
            }
        }
    }
    protected void insert(Object x) {
        synchronized (putLock) {
            LinkedList p = new LinkedList(x);
            synchronized (last) {
                last.next = p;
                last = p;
            }
            if (waitingForTake > 0) putLock.notify();
            return;
        }
    }
    public boolean isEmpty() {
        synchronized (head) {
            return head.next == null;
        }
    }
    public void put(Object x) {
        if (x == null)
            throw new IllegalArgumentException();
        insert(x);
    }
}
```

... allows concurrent access to put() and take()

An example

```
public class LinkedList {
    public Object value;
    public LinkedList next;
    public LinkedList(Object x) {
        value = x;
    }
    protected synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedList first = head.next;
            if (first != null) {
                x = first.value;
                first = first.next;
            }
        }
    }
    protected void insert(Object x) {
        synchronized (putLock) {
            LinkedList p = new LinkedList(x);
            synchronized (last) {
                last.next = p;
                last = p;
            }
            if (waitingForTake > 0) putLock.notify();
            return;
        }
    }
    public boolean isEmpty() {
        synchronized (head) {
            return head.next == null;
        }
    }
    public void put(Object x) {
        if (x == null)
            throw new IllegalArgumentException();
        insert(x);
    }
}
```

Specify that putLock is never null

An example

```
public class LinkedList {
    public Object value;
    public LinkedList next;
    public LinkedList(Object x) {
        value = x;
    }
    protected synchronized Object extract() {
        synchronized (head) {
            Object x = null;
            LinkedList first = head.next;
            if (first != null) {
                x = first.value;
                first = first.next;
            }
        }
    }
    protected void insert(Object x) {
        synchronized (putLock) {
            LinkedList p = new LinkedList(x);
            synchronized (last) {
                last.next = p;
                last = p;
            }
            if (waitingForTake > 0) putLock.notify();
            return;
        }
    }
    public boolean isEmpty() {
        synchronized (head) {
            return head.next == null;
        }
    }
    public void put(Object x) {
        if (x == null)
            throw new IllegalArgumentException();
        insert(x);
    }
}
```

Need more declarative formalisms

Specify that putLock is never null

Specification Languages

- We want specification languages that
 - have a rich set of primitives for observing program state
 - heap-allocated objects, concurrency, *etc.*
 - make it easy to write useful specifications
 - support lightweight and deep-semantic specifications
 - be checkable using a variety of analysis techniques
 - static analysis, theorem proving, *etc.*

Java Modeling Language (JML)

- Developed by G. Leavens and other colleagues at Iowa State University
 - very rich set of operators, especially for describing complex heap properties
 - `\reach(r)`, `\forallforall()`, `\old()`, *etc.*
 - support for specifications with varying degrees of complexity
 - lightweight vs. heavyweight specifications
 - has been checked with a variety of different techniques
 - so far, static analysis, theorem proving and runtime checking
- Emerging as a standard specification language for Java within the research community

Java Modeling Language (JML)

```

public class ListNode {
    public Object value;
    public ListNode next;

    public ListNode(Object x) {
        protected void insert(Object x) {
            assert(x != null);
            synchronized (putLock) {
                //@ requires x != null;
                protected void insert(Object x) {
                    synchronized (putLock) {
                        ListNode p = new ListNode(x);
                        synchronized (last) {
                            last.next = p;
                            last = p;
                        }
                        if (waitingForTake > 0) putLock.notify();
                    }
                }
            }
        }
    }
}

```

An example

```

public class ListNode {
    public Object value;
    public ListNode next;

    public ListNode(Object x) {
        value = x;
    }
}

protected synchronized Object extract() {
    assert(putLock != null);
    synchronized (head) {
        Object x = null;
        ListNode first = head.next;
        if (first != null) {
            x = first.value;
        }
    }
}

public class LinkedList {
    protected final /*@ non_null @*/ Object putLock;
    // ...
}

LinkedList l = new LinkedList();
l.put(1);
l.put(2);
l.put(3);
l.put(4);
l.put(5);
l.put(6);
l.put(7);
l.put(8);
l.put(9);
l.put(10);
l.put(11);
l.put(12);
l.put(13);
l.put(14);
l.put(15);
l.put(16);
l.put(17);
l.put(18);
l.put(19);
l.put(20);
l.put(21);
l.put(22);
l.put(23);
l.put(24);
l.put(25);
l.put(26);
l.put(27);
l.put(28);
l.put(29);
l.put(30);
l.put(31);
l.put(32);
l.put(33);
l.put(34);
l.put(35);
l.put(36);
l.put(37);
l.put(38);
l.put(39);
l.put(40);
l.put(41);
l.put(42);
l.put(43);
l.put(44);
l.put(45);
l.put(46);
l.put(47);
l.put(48);
l.put(49);
l.put(50);
l.put(51);
l.put(52);
l.put(53);
l.put(54);
l.put(55);
l.put(56);
l.put(57);
l.put(58);
l.put(59);
l.put(60);
l.put(61);
l.put(62);
l.put(63);
l.put(64);
l.put(65);
l.put(66);
l.put(67);
l.put(68);
l.put(69);
l.put(70);
l.put(71);
l.put(72);
l.put(73);
l.put(74);
l.put(75);
l.put(76);
l.put(77);
l.put(78);
l.put(79);
l.put(80);
l.put(81);
l.put(82);
l.put(83);
l.put(84);
l.put(85);
l.put(86);
l.put(87);
l.put(88);
l.put(89);
l.put(90);
l.put(91);
l.put(92);
l.put(93);
l.put(94);
l.put(95);
l.put(96);
l.put(97);
l.put(98);
l.put(99);
l.put(100);

```

Java Modeling Language (JML)

```

public class ListNode {
    public Object value;
    public ListNode next;

    public ListNode(Object x) {
        value = x;
    }
}

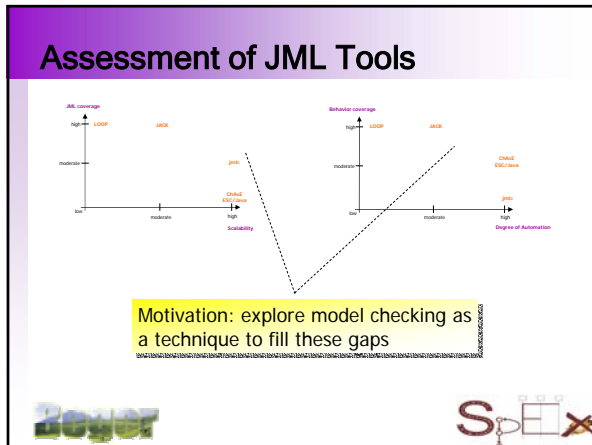
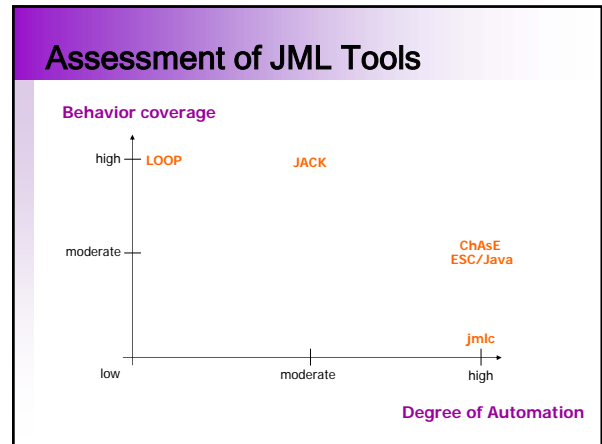
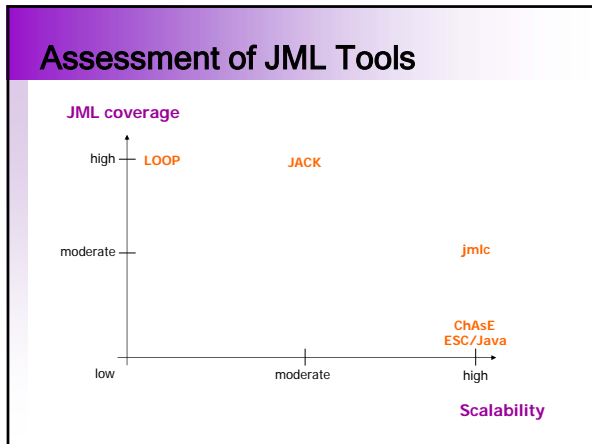
/*@ behavior
   @ assignable head, head.next.value;
   @ ensures \result == null || (\exists!t: ListNode n;
   @ \old(\reach(head)).has(n);
   @ \n.value == \result;
   @ && !(\reach(head).has(n));
   @*/
protected synchronized Object extract() {
    synchronized(head) {
        Object x = null;
        ListNode first = head.next;
        if (first != null) {
            x = first.value;
            first.value = null;
            head = first;
        }
        return x;
    }
}

protected void reinsert(ListNode n) {
    last.next = n;
    last = n;
}

```

Tool support for JML

- Many tools have been developed to support verification of JML
 - jmc (Leavens et al)
 - LOOP (Jacobs et al)
 - ESC/Java (Compaq SRC)
 - KeY (Ahrendt et al)
 - Calvin (Flanagan et al)
 - JACK (Burdy et al)
 - ChASE (N. Cataño)
 - Krakatoa (Marché et al)
 - Jive (Poetzsch-Heffter et al)
- Every tool provide different trade offs in terms of several factors
 - coverage of the JML language
 - coverage of Java
 - degree of automation
 - scalability



Bogor

Questions...

- What is it?
- Why is it useful?
- What makes it particularly good for checking JML?

Bogor's Heap Representation

Key Points...

- ...explicit heap representation
- ...after each transition, a topological sort gives heap objects a canonical order
- ...garbage is eliminated
- ...precise heap model
- ...precise alias information
- ...have access to all visited states (but, efficiently stored using collapse compression)

...transition may create new objects, garbage, etc.

...sort walks over heap, canonicalizes, and collects info

Bogor's Heap Representation – Enables JML Specs Check

Key Points...

- ... many JML features are easy to support in Bogor
- ...precise heap model (c.f., `reach`)
- ...precise alias information (c.f., `assignable`)
- ...can easily compare objects in methods pre/post-states (c.f., `old`)

...transition may create new objects, garbage, etc.

...sort walks over heap, canonicalizes, and collects info

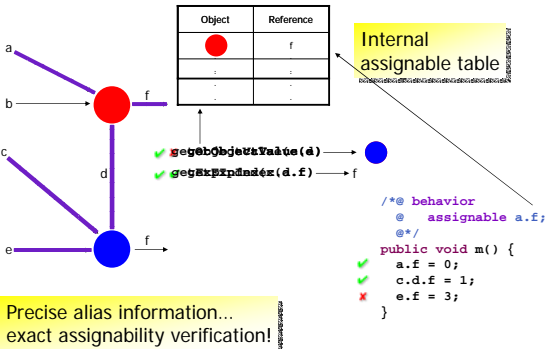
Checking JML Specs with Bogor

- Object operations
 - `assignable`, `\reach(x)`, `\lockset`, `\fresh(x1, ..., xn)`
 - Quantification over objects of a specified type
 - `\forall`, `\exists`
- Pre-/post-conditions, invariants
- Referencing Pre-states
- Methods in JML expressions (the purity issue)

Object operations - assignable

- `assignable` allows to specify frame conditions for a method
 - `assignable v1, v2, ..., vn`
 - v₁, v₂, ..., v_n can be modified by the method
 - modification to any other memory location is forbidden
- Traditionally hard to check
 - aliasing makes it hard to determine unambiguously which memory locations can actually be assigned to
 - verified conservatively in the best cases

Object operations - assignable



Object operations

- Maintaining a precise, dynamic heap model allows performing accurate object operations
 - Eliminated aliasing issues when checking `assignable`
 - Other object operations easily performed too
 - `\reach(x)` – simple DFS traversal from x
 - `\forall` – compute quantification set by DFS from root objects, then post-filtering by type

Checking JML Specs with Bogor

- Object operations
 - `assignable`, `\reach(x)`, `\lockset`, `\fresh(x1, ..., xn)`
 - Quantification over objects of a specified type
 - `\forall`, `\exists`
- Pre-/post-conditions, invariants
- Referencing Pre-states
- Methods in JML expressions (the purity issue)

LinkedList Example (JML)

```

public class LinkedList {
  public Object value;
  public LinkedList next;
  /*@ behavior
     @ requires value == null;
     @ ensures value == null;
     @ also behavior
     @ requires e == null;
     @ signals (Exception e) e instanceof IllegalArgumentException;
     @*/
  public void put(Object x) {
    ...
  }
}

public class LinkedList {
  protected final /*@ non_null @*/ Object putLock;
  protected /*@ non_null @*/ LinkedList head;
  protected /*@ non_null @*/ LinkedList last = head;
  protected int waitingForTake = 0;

  /*@ instance invariant waitingForTake >= 0;
   @ instance invariant \reach(head).has(last);
   @*/
  ...
}
    
```

LinkedList Example (JML)

```

public class LinkedList {
    public Object value;
    public LinkedList next;
}

/* behavior
 * requires x != null;
 * ensures last.value == x && \fresh(last);
 */
protected void insert(Object x) {
    synchronized (putLock) {
        LinkedList p = new LinkedList(x);
        synchronized (last) refactorInsert(p);
        if (waitingForTake > 0) putLock.notify();
        return;
    }
}

/* behavior
 * requires x != null;
 * assignable last, last.next;
 */
protected void refactorInsert(LinkedList n) {
    last.next = n;
}

/* behavior
 * requires x != null;
 * ensures true;
 * also behavior
 * requires n != null;
 * @signal (Exception e) instanceof IllegalStateException;
 */
synchronized (putLock) {
    LinkedList p = new LinkedList(x);
    synchronized (last) refactorInsert(p);
    if (waitingForTake > 0) putLock.notify();
    return;
}

```

Pre/Post-Conditions

```

T m(T1 x1, ..., Tn xn) {
    checkInv(S);
    checkPreCond(S(r1, ..., rn));
    T r;
    try {
        r = origM(r1, ..., rn);
        checkPostCond(S(r1, ..., rn, r));
        return r;
    }
    catch (JMLEntryPreconditionError r) {
        throw new JMLInternalPreconditionError(r);
    }
    catch (JMLAssertionError r) {
        throw r;
    }
    catch (Throwable r) {
        checkPostCond(S(r1, ..., rn, r));
    }
    finally {
        if (/* no postcondition violation? */) {
            checkInv(S);
            checkBC(S);
        }
    }
}

```

jmlc generates a wrapper method for each method in the class

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

```

T m(T1 x1, ..., Tn xn) {
    checkInv(S);
    checkPreCond(S(r1, ..., rn));
    T r;
    try {
        r = origM(r1, ..., rn);
        checkPostCond(S(r1, ..., rn, r));
        return r;
    }
    catch (JMLEntryPreconditionError r) {
        throw new JMLInternalPreconditionError(r);
    }
    catch (JMLAssertionError r) {
        throw r;
    }
    catch (Throwable r) {
        checkPostCond(S(r1, ..., rn, r));
    }
    finally {
        if (/* no postcondition violation? */) {
            checkInv(S);
            checkBC(S);
        }
    }
}

```

check invariants and method preconditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

```

T m(T1 x1, ..., Tn xn) {
    checkInv(S);
    checkPreCond(S(r1, ..., rn));
    T r;
    try {
        r = origM(r1, ..., rn);
        checkPostCond(S(r1, ..., rn, r));
        return r;
    }
    catch (JMLEntryPreconditionError r) {
        throw new JMLInternalPreconditionError(r);
    }
    catch (JMLAssertionError r) {
        throw r;
    }
    catch (Throwable r) {
        checkPostCond(S(r1, ..., rn, r));
    }
    finally {
        if (/* no postcondition violation? */) {
            checkInv(S);
            checkBC(S);
        }
    }
}

```

call original method

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

```

T m(T1 x1, ..., Tn xn) {
    checkInv(S);
    checkPreCond(S(r1, ..., rn));
    T r;
    try {
        r = origM(r1, ..., rn);
        checkPostCond(S(r1, ..., rn, r));
        return r;
    }
    catch (JMLEntryPreconditionError r) {
        throw new JMLInternalPreconditionError(r);
    }
    catch (JMLAssertionError r) {
        throw r;
    }
    catch (Throwable r) {
        checkPostCond(S(r1, ..., rn, r));
    }
    finally {
        if (/* no postcondition violation? */) {
            checkInv(S);
            checkBC(S);
        }
    }
}

```

check post-conditions

Figure 4.3, "A Runtime Assertion Checker for the Java Modeling Language", Y. Cheon

Pre/Post-Conditions

```

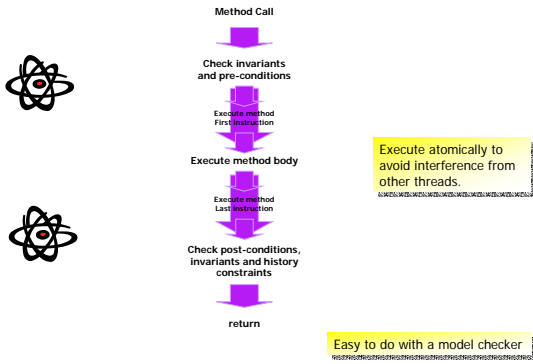
/* behavior
 * ensures \result <=> head.next == null;
 */
public boolean isEmpty() {
    synchronized (head) {
        return head.next == null;
    }
}

public boolean isEmpty() {
    --
    boolean r;
    --
    r = orig$isEmpty();
    checkPost$isEmpty$LinkedList(r);
    return r;
}

```

At this point a thread can interleave and insert an object in the LinkedList; so there actually exists an execution race where the post-condition is violated.

Pre/Post-Conditions



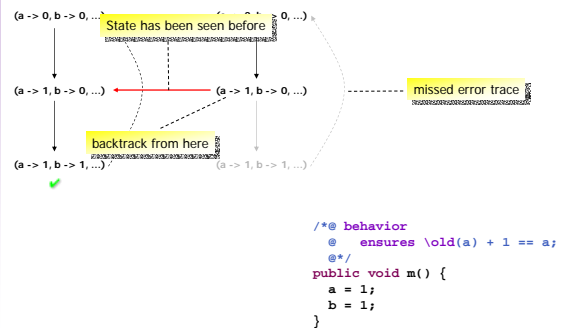
Checking JML Specs with Bogor

- Object operations
 - `assignable`, `\reach(x)`, `\lockset`, `\fresh(x1, ..., xn)`
 - Quantification over objects of a specified type
- Pre-/post-conditions, invariants
- Referencing Pre-states
- Methods in JML expressions (the purity issue)

JML's `\old()` clause

- The `\old()` clause provides a way to access pre-state values in post-state conditions
 - e.g. `ensures \old(a) + 1 == a;`
 - asserts that the current value of `a` has increased by one w.r.t. the value that it had at the beginning of this method
 - very useful for constraining the behavior of a method

Issues with `\old()` and model checking



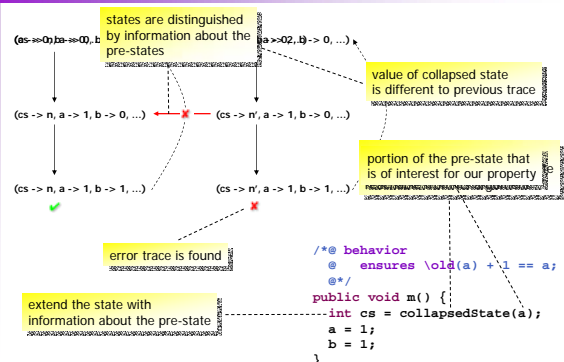
Issues with `\old()` and model checking

```

/*@ behavior
  @ ensures \old(a) + 1 == a;
  @*/
public void m() {
  a = 1;
  b = 1;
}

```

Issues with `\old()` and model checking



Issues with \old() and model checking

```

/*@ behavior
@ assignable head, head.next.value;
@ ensures result == null
@ || (\exists LinkedNode n;
@   \old(\reach(head)).has(n);
@   n.value == r)
@ && !(\reach(
@*/
protected Object extract() {
Object x = null;
LinkedNode first = head.next;
if (first != null) {
x = first.value;
first.value = null;
head = first;
}
return x;
}
    
```

use collapse compression for efficiency

more optimizations are possible

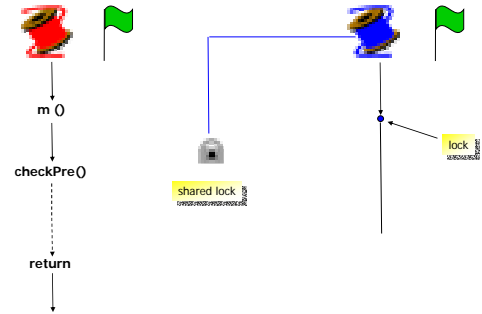
Checking JML Specs with Bogor

- Object operations
 - assignable, \reach(x), \lockset, \fresh(x₁, ..., x_n)
 - Quantification over objects of a specified type
Pre-/post-conditions, invariants
- Referencing Pre-states
- Methods in JML expressions (the purity issue)

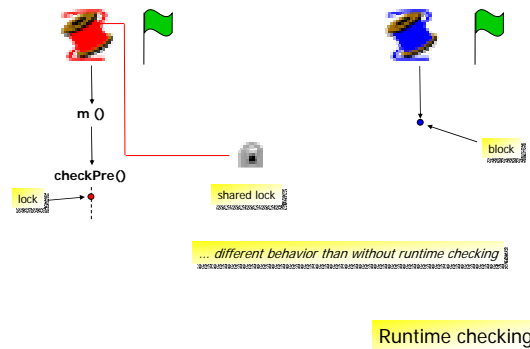
Methods in JML expressions

- All methods called from JML expressions must be **pure** :
 - A pure method must be guaranteed to have no side effects
 - ... must refine assignable \nothing;
 - JML considers the locking used in synchronization as a kind of side effect
 - synchronized methods cannot be declared pure

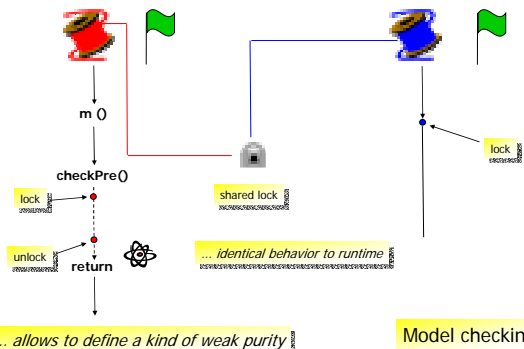
The purity issue



The purity issue



The purity issue



LinkedList Example (JML)

```
public class LinkedList {
    public Object value;
    public LinkedList next;
}

/* behavior
 * requires n != null;
 * ensures true;
 * also behavior
 * requires n == null;
 * signals (RuntimeException) e;
 */
public void set(Object n) {
    if (n == null)
        throw new RuntimeException();
    value = n;
}

/* behavior
 * ensures \result == null;
 */
public boolean isEmpty() {
    synchronized (head) {
        return head.next == null;
    }
}

/* behavior
 * requires n != null;
 * assignable last, last.next;
 */
protected void refactorInsert(LinkedList n) {
    last.next = n;
}

/* behavior
 * requires n != null;
 * assignable last, last.next;
 * postcondition (last) != null;
 */
protected void refactorInsert(LinkedList n) {
    last.next = n;
}
}
```

LinkedList Example

```
class LinkedList {
    public Object value;
}

/* behavior
 * requires n != null;
 * ensures \result == null || (\exists LinkedList n;
 * \void(\reach(head)).has(n);
 * n.value == \result && !(\reach(head).has(n)))
 */
protected Object refactorInsert(Object n) {
    Object x = null;
    LinkedList first = head.next;
    if (first != null) {
        x = first.value;
        first.value = null;
        head = first;
    }
    return x;
}

last = head = new LinkedList(null);
// ...

/* behavior
 * ensures \result == last.next == null;
 */
public boolean isEmpty() {
    return head.next == null;
}

/* behavior
 * requires n != null;
 * assignable last, last.next;
 * postcondition (last) != null;
 */
protected void refactorInsert(LinkedList n) {
    last.next = n;
}

/* behavior
 * requires n != null;
 * assignable last, last.next;
 * postcondition (last) != null;
 */
protected void refactorInsert(LinkedList n) {
    last.next = n;
}
}
```

LinkedList Example

```
class LinkedList {
    public Object value;
}

/* behavior
 * assignable head, head.next.value;
 * ensures \result == null || (\exists LinkedList n;
 * \void(\reach(head)).has(n);
 * n.value == \result && !(\reach(head).has(n)))
 */
protected fun specFun1(Object n,
    Object x,
    LinkedList first,
    LinkedList head) {
    if (first != null)
        x = first.value;
    head = first;
    if (!Set.contains(\reach(head).has(n)))
        return false;
    return true;
}

// ...

assert{[!r1] == null ||
    Set.exists2Context<(\reach(head).has(n)),
    Set.type<(\reach(head).has(n))>>()}}
specFun1,
State.preValSet.type<(\reach(head).has(n))>>(),
State.reachSet<(\reach(head).has(n))>>(),
State.getCurrentThreadId(),
[!r1],
State.reachSet<(\reach(head).has(n))>>(),
[!r1],
Set.type<(\reach(head).has(n))>>()}}
}
```

LinkedList Example

```
class LinkedList {
    public Object value;
}

/* behavior
 * assignable head, head.next.value;
 * ensures \result == null || (\exists LinkedList n;
 * \void(\reach(head)).has(n);
 * n.value == \result && !(\reach(head).has(n)))
 */
protected fun specFun1(Object n,
    Object x,
    LinkedList first,
    LinkedList head) {
    if (first != null)
        x = first.value;
    head = first;
    if (!Set.contains(\reach(head).has(n)))
        return false;
    return true;
}

// ...

assert{[!r1] == null ||
    Set.exists2Context<(\reach(head).has(n)),
    Set.type<(\reach(head).has(n))>>()}}
specFun1,
State.preValSet.type<(\reach(head).has(n))>>(),
State.reachSet<(\reach(head).has(n))>>(),
State.getCurrentThreadId(),
[!r1],
State.reachSet<(\reach(head).has(n))>>(),
[!r1],
Set.type<(\reach(head).has(n))>>()}}
}
```

LinkedList Example

```
class LinkedList {
    public Object value;
}

/* behavior
 * assignable head, head.next.value;
 * ensures \result == null || (\exists LinkedList n;
 * \void(\reach(head)).has(n);
 * n.value == \result && !(\reach(head).has(n)))
 */
protected fun specFun1(Object n,
    Object x,
    LinkedList first,
    LinkedList head) {
    if (first != null)
        x = first.value;
    head = first;
    if (!Set.contains(\reach(head).has(n)))
        return false;
    return true;
}

// ...

assert{[!r1] == null ||
    Set.exists2Context<(\reach(head).has(n)),
    Set.type<(\reach(head).has(n))>>()}}
specFun1,
State.preValSet.type<(\reach(head).has(n))>>(),
State.reachSet<(\reach(head).has(n))>>(),
State.getCurrentThreadId(),
[!r1],
State.reachSet<(\reach(head).has(n))>>(),
[!r1],
Set.type<(\reach(head).has(n))>>()}}
}
```

LinkedList Example

```
class LinkedList {
    public Object value;
}

/* behavior
 * assignable head, head.next.value;
 * ensures \result == null || (\exists LinkedList n;
 * \void(\reach(head)).has(n);
 * n.value == \result && !(\reach(head).has(n)))
 */
protected fun specFun1(Object n,
    Object x,
    LinkedList first,
    LinkedList head) {
    if (first != null)
        x = first.value;
    head = first;
    if (!Set.contains(\reach(head).has(n)))
        return false;
    return true;
}

// ...

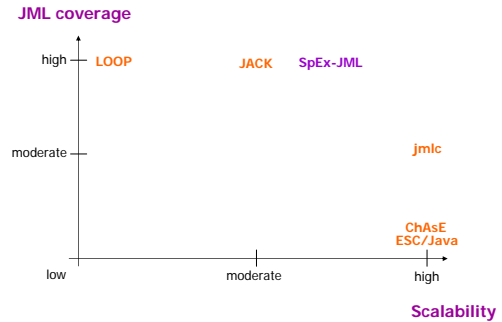
assert{[!r1] == null ||
    Set.exists2Context<(\reach(head).has(n)),
    Set.type<(\reach(head).has(n))>>()}}
specFun1,
State.preValSet.type<(\reach(head).has(n))>>(),
State.reachSet<(\reach(head).has(n))>>(),
State.getCurrentThreadId(),
[!r1],
State.reachSet<(\reach(head).has(n))>>(),
[!r1],
Set.type<(\reach(head).has(n))>>()}}
}
```


Bogor's Reduction Algorithms – Enables Checking JML Specs

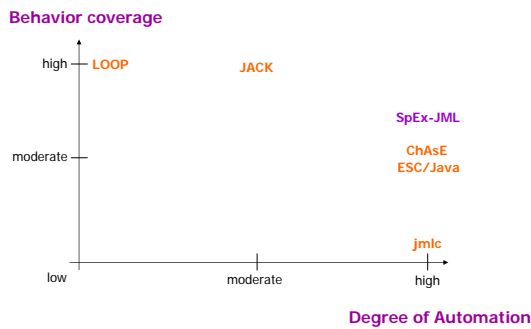
	w/ POR		w/o POR	
	w/ JML	w/o JML	w/ JML	w/o JML
LinkedQueue[12]	228 loc	1533 states	reach, \old, signals, \exists exists	17064 states
3 threads	2833 states	1533 states		11594 states
22 objects	10 sec/1.6 MB	5 sec/1.0 MB		38 sec/3.7 MB
5 threads	39050 states	12807 states		423538 states
32 objects	44 sec/5.9 MB	72 sec/2.5 MB		2415 sec/46.4 MB
RWVSN[12]	227 loc	183 states	\old	2255 states
4 threads	183 states	183 states		2621 states
5 objects	1 sec/1.0 MB	1 sec/0.8 MB		2 sec/1.5 MB
7 threads	18398 states	18398 states		4204332 states
9 objects	85 sec/6.8 MB	144 sec/3.0 MB		26153 sec/366.3 MB
ReplicatedWorkers[4]	543 loc	1751 states	fresh, \old, reach	322016 states
4 threads	1751 states	1751 states		269593 states
19 objects	14 sec/2.1 MB	13 sec/1.9 MB		897 sec/29.8 MB
6 threads	10154 states	10154 states		12347415 states
21 objects	99 sec/3.3 MB	92 sec/2.8 MB		10016554 states
				21734 sec/282.5 MB

Indicates little overhead compared with simply exploring the state-space

Assessment of JML Tools



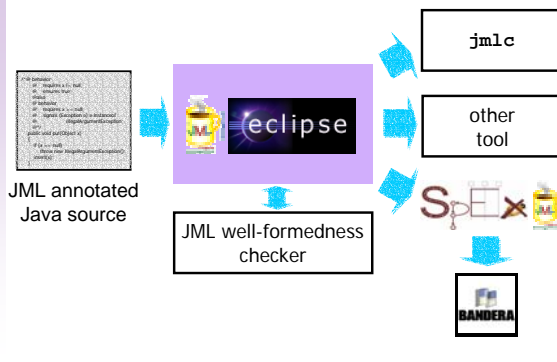
Assessment of JML Tools



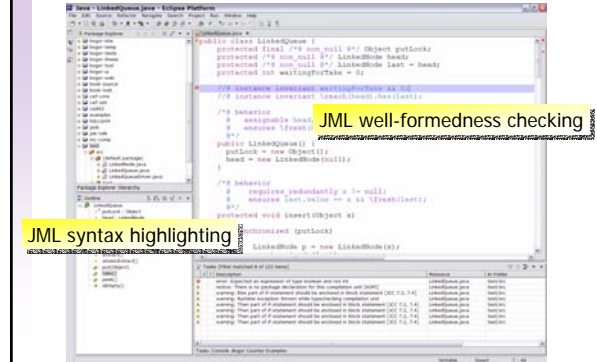
Assessment of SpEx-JML

- Model checking based JML verification technique
 - Bogor
- Excellent automation usability
 - similar to that of jmlc
- Very high JML coverage
- Moderate behavior coverage
 - determined by test harness
 - but sound with respect to the test harness
- Good scalability
 - unit level reasoning

JMLEclipse



JMLEclipse



Future Work

- Proposal for concurrency specifications in JML
 - thread-locality
 - regionized pre-/post-conditions
 - atomicity, etc.
- JMLEclipse as an open ended JML plugin for Eclipse
- Other specification formalisms

For More Information...



SAnToS Laboratory,
Kansas State University
<http://www.cis.ksu.edu/santos>



Bogor Project
<http://bogor.projects.cis.ksu.edu>



SpEx Project
<http://spex.projects.cis.ksu.edu>



JMLEclipse Project
<http://jmleclipse.projects.cis.ksu.edu>



Bandera Project
<http://bandera.projects.cis.ksu.edu>