CIS 842: Specification and Verification of Reactive Systems

Lecture 8: Overview of BSL --- The Bandera Temporal Property Specification Language

BSL: Bandera Specification Language

- Propositions stated in terms of source code features
- Based on an extensible system of temporal specification patterns
- Heap objects are named via object quantification
Assertion Forms

• Pre-conditions
  ```
  @assert PRE <name> <exp>;
  /**
   * @assert
   *   PRE foo: (I > 5);
   *   POST bar: (I < 10);
   *   LOCATI ON[here] checka:
   *     (m.q.a == 4);
   */
  public mymethod(int I) {
    ....
    here:
    ....
  }
  ```

• Post-conditions
  ```
  @assert POST <name> <exp>;
  ```

• Arbitrary Locations
  ```
  @assert LOCATION[<label>] <name> <exp>;
  ```

Predicate Forms

• Static/Instance Data Constraints
  ```
  @observable EXP <name>(p1, ..., pn) <exp>;
  ```

• Invocation
  ```
  @observable INVOC E <name>(p1, ..., pn) <exp>;
  ```

• Return
  ```
  @observable RETURN <name>(p1, ..., pn) <exp>;
  ```

• Arbitrary Locations
  ```
  @observable LOCATI ON[<label>] <name>(p1, ..., pn) <exp>;
  ```
Classification of Predicates

- **Static vs. Instance**
  - *static* predicates are applied to static fields and static methods
  - *instance* predicates are applied to instance fields and virtual methods
  - These are distinguished syntactically: instance predicates have ‘\texttt{this}’ as first parameter

- **Location sensitive vs. Location insensitive**
  - The ‘\texttt{EXP}’ predicate is location insensitive --- it is simply a predicate on the program’s data space. The rest of the predicates are location sensitive
  - For location sensitive predicates to be true, control must be at a position indicated by the predicate form

Semantic Issues

**Relationship between assertion PRE and predicate INVOKE**

\texttt{@assert PRE \langle name \rangle \langle exp \rangle;}

**IMPLICATION:** holds if \(\langle \text{exp} \rangle\) is true \(WHEN\) control is at entry of method (false otherwise)

\texttt{@observable INVOKE \langle name \rangle(\ p1, \ldots, \ p_n) \langle exp \rangle;}

**CONJUNCTION:** holds if \(\langle \text{exp} \rangle\) is true \(AND\) control is at entry of method
Semantic Issues

Dealing with possible exceptional conditions in predicates

\[
(o1 \neq \text{null}) \&\& \\
(o1.\text{next} \neq \text{null}) \&\& \\
(o1.\text{next}.\text{value} == 0)
\]

- Guards are added implicitly to each predicate dereference expression
  - Ensures that no exceptional condition can arise during predicate evaluation (relying on short-circuit semantics of operators)

Methodology:

Property Specification

*Given a software requirement...*

- **Identify observables** (propositions) in requirement
- **Define propositions** in source Java-doc comments
- Use GUI to **select appropriate temporal pattern** parameterized by declared observables
- **Add quantification** if property contains instance propositions.
Bounded Buffer

class BoundedBuffer {
Object[] buffer;
int head; /* next available slot */
int tail; /* last available slot */
int bound; /* max # of elements */

public BoundedBuffer(int b) {
    bound = b;
    buffer = new Object[bound];
    head = 0;
    tail = bound - 1;
}

public synchronized boolean isEmpty() {
    return head == ((tail + 1) % bound);
}

public synchronized void add(Object o) {
    // Implementation
}

public synchronized Object take() {
    // Implementation
}
}
Bounded Buffer

```java
public synchronized void add(Object o) {
    while (tail == head)
        try { wait(); } catch (InterruptedException ex) {}
    buffer_[head] = o;
    head = (head + 1) % bound;
    notifyAll();
}

public synchronized Object take() {
    while (head == ((tail + 1) % bound))
        try { wait(); } catch (InterruptedException ex) {}
    tail = (tail + 1) % bound;
    notifyAll();
    return buffer_[tail];
}
```

Bounded Buffer Properties

- Full buffers eventually become non-full
- Indices always stay in range
- Empty buffers must be added to before being taken from
- Buffers are constructed with positive bounds
- Elements are always added in correct position
Property Specification

/**
 * @observable
 * EXP Full(this): (head == tail);
 */

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public synchronized
        void add(Object o)
        {..}

    public synchronized
        Object take ()
        {..}
}

Requirement 1:
If a buffer becomes \textbf{full} it will eventually become \textbf{non-full}.

Bandera Specification:
Ful tToNonFul l:
forall[b:BoundedBuffer].
{!Full(b)} responds to
{Full(b)} globally;

Property Specification

/**
 * @observable
 * EXP HeadRange(this):
 *  head >= 0 && head < bound;
 * EXP TailRange(this):
 *  tail >= 0 && tail < bound;
 */

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public synchronized
        void add(Object o)
        {..}

    public synchronized
        Object take ()
        {..}
}

Requirement 2:
Indices always stay in range.

Bandera Specification:
IndexRangeInvariant:
forall[b:BoundedBuffer].
{HeadRange(b) &&
TailRange(b)}
is universal globally;
**Property Specification**

```java
/**
 * @observable
 * @require Empty(this):
 * head == ((tail+1) % bound);
 */
class BoundedBuffer {
    int head, tail, bound;
    /**
     * @observable INVOKE Call(this);
     */
    public synchronized void add(Object o) {
        // Implementation
    }
    /**
     * @observable RETURN Return(this);
     */
    public synchronized Object take() {
        // Implementation
    }
}
```

**Requirement 3:**

Empty buffers must be added before being taken from.

**Bandera Specification:**

NoTakeWhileEmpty:

forall [b: BoundedBuffer].
{take. Return(b)} is absent
after {Empty(b)}
until {add. Call(b)};

**Property Specification**

```java
/**
 * @assert
 * @pre PositiveBound:
 * (b > 0);
 */
public BoundedBuffer(int b) {
    bound = b;
    buffer = new Object[bound];
    head = 0;
    tail = bound - 1;
}
```

**Requirement 4:**

Buffers are constructed with positive bounds.

**Bandera Specification:**

PositiveBound:

enable assertions
{PositiveBound};
Implementation Issues

A quick overview of implementation of...

- Assertions
- Predicates
- Object quantification

Implementing Assertions

```java
class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    /**
     * @assertion
     *    PRE PosBound1: (n > 0);
     */
    public BoundedBuffer(int n) {
        Bandera.assert(n > 0);
        ...
    }

    public synchronized void add(Object o) {
        ...
    }
}
```

Specification:
PositiveBound: enable assertions {PosBound};
Implementing Predicates

- (needs to be filled in)

Quantification

\[\forall b: \text{BoundedBuffer}. P(b)\]

- Quantified set is not fixed
  - varies within executions
  - varies across executions
- Solution
  - by adding a state variable (for b) that will eventually be bound non-deterministically to each instance
  - by enabling checking of the formula only when variable is bound to an instance
Quantification (Cont’d)

(!selected U (selected && P(b))) || []!selected

[1] new BoundedBuffer (n)
[2] new BoundedBuffer (n)
[3] new BoundedBuffer (k)

new BoundedBuffer (n)
new BoundedBuffer (n)
new BoundedBuffer (k)

[1] selected
[2] selected
[3] selected

Quantification (Cont’d)

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public BoundedBuffer(int n) {
        ...}
}

class heap {
    public static BoundedBuffer b;
}

class BoundedBuffer {
    Object [] buffer;
    int head, tail, bound;

    public BoundedBuffer(int n) {
        if (heap.b == null &&
            Bandera.choose()) {
            heap.b = this;
        } else {
            ...}
    }
}
Quantification (Cont’d)

forall[b:BoundedBuffer].
  {Full(b)} leads to {!Full(b)} globally;

( heap. b = null U
  ( heap. b != null &&
    ( [] (heap. b. head = heap. b. tail) ->
      (heap. b. head != heap. b. tail)) )))
  || [] (heap. b = null)

Test Result & Assessments

<table>
<thead>
<tr>
<th>Property</th>
<th>Sliced</th>
<th>Never-claim States</th>
<th>Stored States</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferAssertions</td>
<td>Yes</td>
<td>-</td>
<td>17797</td>
</tr>
<tr>
<td>IndexRangeInv</td>
<td>Yes</td>
<td>14</td>
<td>45215</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufferAssertions</td>
<td>Yes</td>
<td>14</td>
<td>115387</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FullToNonFull</td>
<td>Yes</td>
<td>25</td>
<td>64687</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FullToNonFull</td>
<td>Yes</td>
<td>25</td>
<td>154842</td>
</tr>
</tbody>
</table>
Example: Observer Pattern

Intent of the Observer pattern

```
...define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically'' [Gamma, Helm, Johnson, Vlissides]
```

- Many different implementations
- We will use an implementation from Java Development Kit
  - `java.util.Observer`
  - `java.util.Observable`

```
Observable addObserver(this, ...
Tells the observable object who is observing – i.e., who should be notified when a change occurs.
```
Example: Observer Pattern

- **Observable**:
  - addObserver(this, …)

- **Observers**:
  - State change causes all registered observers to be notified
  - update(this, …)

Tells the observer who has changed (note: an observer may be observing multiple objects)
Example: Observer Pattern

Observable

Observers

deleteObserver(this, …)

Tells the observer who should be removed from its list of observers.

Example: Observer Pattern

Observable

Observers

addObserver(this, …)
Example: Observer Pattern

Observer interface

```java
interface Observer {
    public void update(Observable o, Object arg);
}
```

- `update` is call-back method that is passed the object `o` whose state has changed along with some supplementary developer-specified data `arg`.

Example: Observer Pattern

Observable class code (part 1)

```java
class Observable {
    protected boolean changed = false; // 'true' if state changed
    protected Vector obs; // Table of registered observers
    public Observable() { obs = new Vector(); }
    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) { obs.addElement(o); }
    }
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }
    public synchronized void setChanged() {
        changed = true;
    }
    // ...
}
```

Example: Observer Pattern

Observable class code (part 2)

```java
public void notifyObservers() {
    notifyObservers(null);
}

public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed) return;
        arrLocal = obs.toArray();
        changed = false;
    }
    window:
    for (int i = arrLocal.length - 1; i >= 0; i --)
        ((Observer) arrLocal[i]).update(this, arg);
}
```

Specifications:
Black-box vs. White-box

- Black-box specifications
  - interface specifications
  - only refer to states or actions of a system or sub-system that are observable at its public interface
  - These specifications can be used as the basis of design contracts and to verify an implementation's conformance to design specifications.

- White-box specifications
  - details of an implementation are explicitly referenced in the statement of the correctness property

Bandera supports both black- and white-box specifications
Requirements

1. Only registered observers are notified of updates
2. All registered observers are notified of updates

We will express these using black- and white-box specifications

Revisiting the Code

public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed) return;
        arrLocal = obs.toArray();
        changed = false;
    }
    for (int i=arrLocal.length-1; i>=0; i--)
        ((Observer) arrLocal[i]).update(this, arg);
}

Label window indicates the point after which the collection of registered Observer's has been constructed but before the call-backs have been made.

- The code is implemented this way so that the Observable instance's lock is not held while the call-back methods execute (since they may take an arbitrary amount of time).
White-box Specification

```java
/** @assert * LOCATION[window] obsCountsEqual:
 * arrLocal.length == obs.size(); */

public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed) return;
        arrLocal = obs.toArray();
        changed = false;
    }

    window:
    for (int i=arrLocal.length-1; i>=0; i--)
        ((Observer) arrLocal[i]).update(this, arg);
}
```

ObsAssertions: enable assertions { obsCountEqual };
Toward a Black-box Specification

- Since Observable instances do not make their table of registered observers public, a black box specification must “record” registration information by observing sequences of `addObserver()` and `deleteObserver()` calls.

Requirement One

Break the requirement down into two cases:

- An observer must not be updated for a change in an observable before it registers for the first time.
- An observer must not be updated for a change in an observable between the time that it unregisters and the time that it reregisters for that observable.
Observer Pattern

/** @observable
   *   INVOKE begin (this, Observable subject):
   *           subject == o;
   */
interface Observer {
   public void update(Observable o, Object arg);
}

Example: Observer Pattern

/** @observable
   *   RETURN end (this, Observer watcher):
   *           watcher == o;
   */
public synchronized void addObserver(Observer o) {
   if (!obs.contains(o)) { obs.addElement(o); }
}

/** @observable
   *   INVOKE begin (this, Observer watcher):
   *           watcher == o;
   */
public synchronized void deleteObserver(Observer o) {
   obs.removeElement(o);
}
Requirement One

// No notification of updates prior to initial registration
InitiallyNotNotified:
forall [o: Observer]. forall [s: Observable].
{update.begin(o, s)} is absent
before {addObserver.end(s, o)};

// No notification of updates while unregistered
NotRegisteredNotNotified:
forall [o: Observer]. forall [s: Observable].
{update.begin(o, s)} is absent
after {deleteObserver.begin(s, o)}
until {addObserver.end(s, o)};

Assessment

- Note that when we represent delimiting events by declaring an INVOKE or RETURN predicate, we typically what to choose between INVOKE/RETURN so that the interval being marked is as large as possible. This yields a stronger specification.
- Therefore, for the interval beginning with a deleteObserver and ending with an addObserver, we observe the beginning of an deleteObserver and the end of an addObserver.
- Note the after/until scope does not require the second condition mentioned in the scope (i.e., addObserver.end) to occur (it's related to “weak until”).
- Does part I of the requirement hold?
- Does part II of the requirement hold?
Requirement Two

Intuition:

- All register observers are notified of updates
- If an Observer registers for an Observable and does not unregister up to the point where a call to Observable.notifyObservers is made, then Observer.update() will be called before Observable.notifyObservers returns.

// All registered Observers should get notifications

RegisteredNotified:

forall [o: Observer]. forall [s: Observable].

{notifyObservers.begin(s)}

responds to {addObserver.end(s, o)}

without {deleteObserver.begin(s, o)}

leads to

{update.begin(o, s)} precedes {notifyObservers.end(s)};
Additional Pattern

pattern {
    name = "Constrained-Response, Precedence Chain"
    scope = "Global"
    parameters = {A, B, C, D, E}
    format = "( {B} responds to {A} without {C} )
             leads to ( {D} precedes {E} )"
    ltl = "[](({A} -> (!{C} U {B})) ->
            <>({B} && (!{E} U ({D} || []!{E}))))"
}

Summary

- Motivation:
  
  Makes it easier for non-expert to write temporal specifications for Java programs

- How?
  - Source level
  - Pattern based approach
  - Tool supports:
    - automatic translation
    - guides for writing specification (GUI)
    - HTML documentation generation