Using Analysis Patterns to Uncover Specification Errors

William Heaven    Alessandra Russo

Dept. of Computing
Imperial College London
Motivating Example

Method *insert* in *Queue*:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
@*/
boolean insert(Entry e) { .. }
```

<table>
<thead>
<tr>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert: boolean</td>
</tr>
</tbody>
</table>
Motivating Example

Method *insert* in *Queue*:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
/*@*/
boolean insert(Entry e) { .. }
```

Method *add* in *ModelQueue*:

```java
/*@ ensures ..
   (result.size() >=\old(size()));
/*@*/
ModelQueue add(Entry e) { .. }
```
Motivating Example

Method *insert* in *Queue*:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e))); */

boolean insert(Entry e) { .. }
```

Overriding method *insert* in *BoundedQueue*:

```java
/*@ ensures size() == entries.size(); */
/*@ pure */
int size() { .. }

/*@ also */
ensures size() > \old(entries.size()) && size() <= MAX;

boolean insert(Entry e) { .. }
```
Motivating Example

Method `insert` in `Queue`:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
/*@*/
boolean insert(Entry e) { .. }
```

Overriding method `insert` in `BoundedQueue`:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size(){ .. }

/*@ also
   ensures size() > \old(entries.size()) && size() <= MAX;
/*@*/
boolean insert(Entry e { .. }
```

Postcondition here implicitly includes spec from overridden method
Motivating Example

Method `insert` in `Queue`:

```java
/*@ ensures (result ==> contains(e))
   & (entries == \old(entries.add(e)));
@
boolean insert(Entry e) { .. }
```

Overriding method `insert` in `BoundedQueue`:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size() { .. }

/*@ also
   ensures size() > \old(entries.size()) & size() <= MAX;
@
boolean insert(Entry e { .. }
```

To check:

```
BoundedQueue::insert\_POST
⇒ Queue::insert\_POST
```
Motivating Example

To check:

\[ \text{BoundedQueue}::\text{insert}_\text{POST} \Rightarrow \text{Queue}::\text{insert}_\text{POST} \]

- Checking implication automatically (e.g. with Alloy Analyzer, via JML encoding) yields *positive* result
Motivating Example

To check:

\[ \text{BoundedQueue}::\text{insert}_{\text{POST}} \Rightarrow \text{Queue}::\text{insert}_{\text{POST}} \]

- Your tool of choice may additionally tell you that it’s in fact valid
  - in the sense of not possibly false
Motivating Example

Method `insert` in `Queue`:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
@*/
boolean insert(Entry e) { .. }
```

Overriding method `insert` in `BoundedQueue`:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size() { .. }

/*@ also ensures size() > \old(entries.size()) && size() <= MAX;
@*/
boolean insert(Entry e { .. }
```

What if we’d made a typo?

To check:

```
BoundedQueue::insert_{\text{POST}} \\
\Rightarrow \text{Queue::insert}_{\text{POST}}
```
Motivating Example

Method *insert* in *Queue*:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
*/

boolean insert(Entry e) { .. }
```

Overriding method *insert* in *BoundedQueue*:

```java
/*@ ensures size() == entries.size(); */
/*@ pure */
int size() { .. }

/*@ also */
ensures size() < \old(entries.size()) && size() <= MAX;

boolean insert(Entry e) { .. }
```

To check:

**BoundedQueue::insert**_{POST}  
⇒ **Queue::insert**_{POST}
Motivating Example

To check:

\[ \textit{BoundedQueue::insert}_{\text{POST}} \Rightarrow \textit{Queue::insert}_{\text{POST}} \]

• With the typo we get the same positive result ..
Motivating Example

To check:

\[ \text{BoundedQueue}::\text{insert}_{\text{POST}} \rightarrow \text{Queue}::\text{insert}_{\text{POST}} \]

• With the typo we get the same positive result..
• This is perfectly correct. Indeed, the implication’s still valid.
Motivating Example

To check:

\[ \text{BoundedQueue::insert}_{\text{POST}} \Rightarrow \text{Queue::insert}_{\text{POST}} \]

\[ \Rightarrow \text{VACUOUS} \]

- \[ \text{BoundedQueue::insert}_{\text{POST}} \Rightarrow \text{Queue::insert}_{\text{POST}} \]

Unsatisfiable!
Motivating Example

Method *insert* in *Queue*:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
@*/
boolean insert(Entry e) { .. }
```

Overriding method *insert* in *BoundedQueue*:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/
int size(){ .. }

/*@ also
   ensures size() < \old(entries.size()) && size() <= MAX;
@*/
boolean insert(Entry e { .. }
```

Method *add* in *ModelQueue*:

```java
/*@ ensures ..
   (result.size() >=\old(size()));
@*/
ModelQueue add(Entry e) { .. }
```

These formulae are inconsistent
Motivating Example

Method \textit{insert} in \textit{Queue}:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
@*/

boolean insert(Entry e) { .. }
```

Overriding method \textit{insert} in \textit{BoundedQueue}:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size(){ .. }

/*@ also
   ensures size() < \old(entries.size()) && size() <= MAX;
@*/

boolean insert(Entry e) { .. }
```

Method \textit{add} in \textit{ModelQueue}:

```java
/*@ ensures ..
   (result.size() >=\old(size()));
@*/

ModelQueue add(Entry e) { .. }
```

These formulae are inconsistent

- \( size() < \old(entries.size()) \)
- \( entries.size() < \old(entries.size()) \)
Motivating Example

Method \textit{insert} in \textit{Queue}:

```java
/*@ ensures (result ==> contains(e))
   && (entries == \old(entries.add(e)));
@*/
boolean insert(Entry e) { .. }
```

Overriding method \textit{insert} in \textit{BoundedQueue}:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size() { .. }

/*@ also ensures size() < \old(entries.size()) && size() <= MAX;
@*/
boolean insert(Entry e) { .. }
```

Method \textit{add} in \textit{ModelQueue}:

```java
/*@ ensures ..
   (result.size() >= \old(size()));
@*/
ModelQueue add(Entry e) { .. }
```

These formulae are inconsistent:

- size() < \old(entries.size())
- entries.size() < \old(entries.size())
- result.size() >= \old(entries.size())
- entries.size() >= \old(entries.size())
Motivating Example

Method insert in Queue:

```java
/*@ ensures (result ==> contains(e))
    && (entries == \old(entries.add(e)));
@*/
boolean insert(Entry e) { .. }
```

Method add in ModelQueue:

```java
/*@ ensures ..
    (result.size() >=\old(size()));
@*/
ModelQueue add(Entry e) { .. }
```

Overriding method insert in BoundedQueue:

```java
/*@ ensures size() == entries.size(); @*/
/*@ pure */ int size(){ .. }

/*@ also
    ensures size() < \old(entries.size()) && size() <= MAX;
@*/
boolean insert(Entry e { .. }
```

These formulae are inconsistent

- `size() < \old(entries.size())`
- `entries.size() < \old(entries.size())`
- `result.size() >= \old(entries.size())`
- `entries.size() >= \old(entries.size())`
The Problem

• Errors are easily introduced by hand and go unnoticed
• Compounded by functional abstraction
  – which is great
  – but keeps large parts of a specification hidden
• “Superficial” feedback from automated analyses gives specifiers a false sense of security
• We want an analysis that explores deeply enough to give richer feedback
• We advocate systematic exploration of the satisfiability of the constituent subformulae of each formula under analysis
SAT Oracle

• We assume a sound and complete decision procedure for SAT:

\[ SAT: \Phi \rightarrow \{s, u\} \]

\[ SAT(\phi) = s \text{ iff } \phi \text{ is satisfiable } \quad [\text{written } s(\phi)] \]
\[ SAT(\phi) = u \text{ otherwise } \quad [\text{written } u(\phi)] \]

• And define further *satisfiability values* in terms of \( s \) and \( u \)..

\[ v(\phi) \text{ iff } u(\neg \phi) \]

\[ c(\phi) \text{ iff } s(\phi) \land s(\neg \phi) \]

\[ \overline{v}(\phi) \text{ iff } s(\neg \phi) \]
Ordering of $\varphi, \neg \varphi$ pairs
Ordering of satisfiability values

\[ \begin{align*}
&v > s, \quad c > s, \quad c > \overline{v}, \quad u > \overline{v}, \quad s > \sim, \quad \overline{v} > \sim
\end{align*} \]
Obtaining Values

• In addition to obtaining values through the oracle $SAT$, values may also be inferred: e.g. knowing $s(\varphi)$ and $s(\neg \varphi)$ gives $c(\varphi)$ and $c(\neg \varphi)$

• To permit inference, we store obtained values in a lookup table:

$$SAT_{Table}: \Phi \rightarrow \{v, c, u, s, \overline{v}, \neg\}$$

• We now have two ways of querying a value: oracle and lookup table

• For any given query, we simply want least upper bound of these two:

$$GetVal(\varphi) = \text{LUB} [SAT(\varphi), SAT_{Table}(\varphi)]$$

• $? (\varphi)$ denotes a satisfiability query
Implication Pattern

(result ==> contains(e))
&& (entries == \old(entries.add(e)))
&& ensures size() < \old(entries.size())
&& size() <= MAX

⇒

(result ==> contains(e))
&& (entries == \old(entries.add(e)))
Implication Pattern

(result ==> contains(e))
&& (entries == \old(entries.add(e)))
&& ensures size() < \old(entries.size())
&& size() <= MAX

⇒

(result ==> contains(e))
&& (entries == \old(entries.add(e)))
Implication Pattern

(result ==> contains(e))
&& (entries == \old(entries.add(e)))
&& ensures size() < \old(entries.size())
&& size() <= MAX
Implication Pattern

\( \text{implication} \quad \text{pattern} \)

\[
\begin{align*}
\text{(result} & \implies \text{contains(e))} \\
& \land (\text{entries} = \text{old(entries.add(e)))} \\
& \land \text{ensures size()} < \text{old(entries.size())} \\
& \land \text{size()} \leq \text{MAX}
\end{align*}
\]

We can continue by applying the pattern for this subformula in order to find the problem.
Conjunction Pattern

(result ==> contains(e))
  && (entries == \old(entries.add(e)))
  && ensures size() < \old(entries.size())
  && size() <= MAX
Conjunction Pattern

\[
\text{(result ==> contains(e))} \\
\&\& \text{(entries == \old(entries.add(e)))} \\
\&\& \text{ensures size() < \old(entries.size())} \\
\&\& \text{size() <= MAX}
\]
Conjunction Pattern

\[(\text{result} \implies \text{contains}(e)) \land (\text{entries} == \text{old(entries}.\text{add}(e))) \land \text{ensures size}() < \text{old(entries}.\text{size}()) \land \text{size}() \leq \text{MAX}\]

Try combinations of conjunct until you find the inconsistent set.
Conjunction Pattern

\[(\text{result} \implies \text{contains(e)}) \land (\text{entries} = \text{old(entries.add(e)))}) \land \text{ensures size}() < \text{old(entries.size())} \land \text{size()} \leq \text{MAX}\]

This includes exploring the specifications of \textit{contains}, \textit{add} and both \textit{sizes}, with pure method specs treated as special kinds of conjunction.
Conjunction Pattern

\( (\text{result} \implies \text{contains(e)}) \)  
\&\& (\text{entries} == \text{old(entries.add(e)))}  
\&\& \text{ensures size() < \text{old(entries.size()}}}  
\&\& \text{size() <= MAX}

Terminates with inconsistent pair:

**BoundedQueue::insert**

\( \text{size()} < \text{old(entries.size())} \)

**ModelQueue::add**

\( \text{result.size()} \geq \text{old(size())} \)
Base Pattern
Negation & Disjunction Patterns
Summary

• Systematic exploration of the satisfiability of subformulae
  – Why does the (top-level) formula have satisfiability value $\nu$?
• Achieved by automated analysis guided by “patterns”
• Aim to give specifier as rich feedback as possible
• Particularly interested in cases of vacuity
• A kind of “spec debugging”
• Small step towards provision of automated spec development environment (cf. Perry’s Eclipse IVE)
• Automated explorative analysis does not require expert direction
In The Same Spirit As

- *Soundness and Completeness Warnings in ESC/Java*
  - J. Kiniry, A. Morkan, and B. Denby, SAVCBS, 2006

- *Early Detection of JML Specification Errors Using ESC/Java2*
  - P Chalin, SAVCB, 2006

- *Vacuity Detection in Temporal Model Checking*
  - O. Kupferman and M Vardi, STTT, 1999

- *Extending Extended Vacuity*
  - A Gurfinkel and M Chechik, FMCAD, 2004
To Do

- Badly needs proper experimental evaluation
- Needs better implementation, maybe not using Alloy Analyzer
- Should be extended to real spec language such as JML or Spec#
  - current evaluation carried out with Loy, a toy JML
- Integration with an existing toolset