Using Analysis Patterns to Uncover Specification Errors

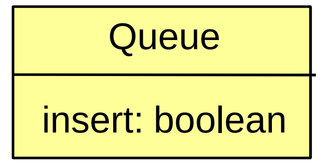
William Heaven Alessandra Russo

Dept. of Computing
Imperial College London

Motivating Example

Method *insert* in *Queue*:

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



Motivating Example

Method *insert* in *Queue*:

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

Method *add* in *ModelQueue*:

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

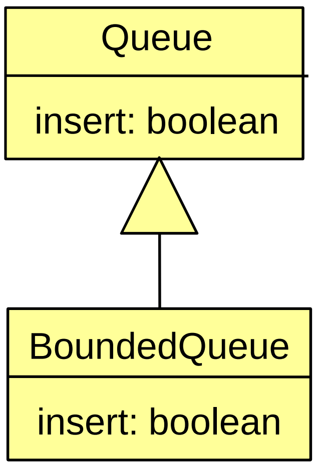
Motivating Example

Method *insert* in *Queue*:

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

Overriding method *insert* in *BoundedQueue*:

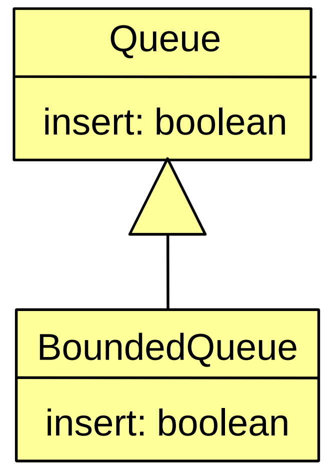
```
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size(){ .. }

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



Motivating Example

Method *insert* in *Queue*:

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



Overriding method *insert* in *BoundedQueue*:

```
/*@ 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

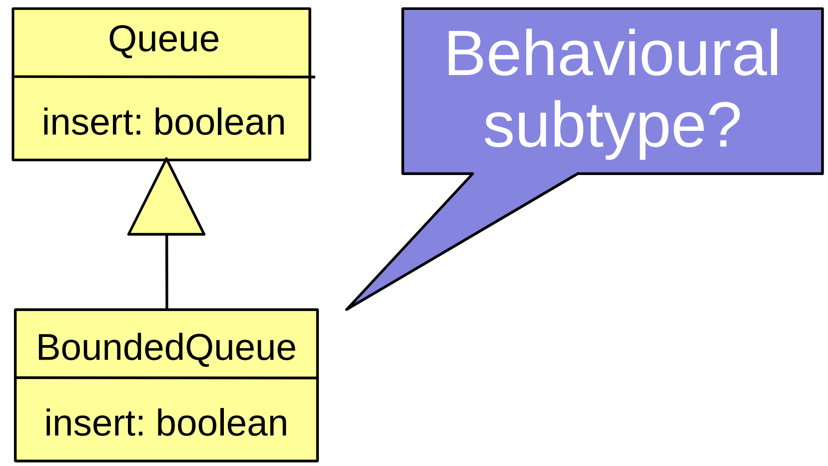
Motivating Example

Method *insert* in *Queue*:

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

Overriding method *insert* in *BoundedQueue*:

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

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



To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}} \Rightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

Motivating Example

To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}} \Longrightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

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

Motivating Example

To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}}$$

$$\Rightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

⇒ *VALID*

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

Motivating Example

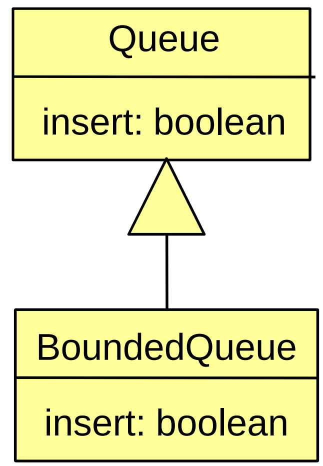
Method *insert* in *Queue*:

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

Overriding method *insert* in *BoundedQueue*:

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

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

What if we'd made a typo?



Behavioural subtype?

To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}} \Rightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

Motivating Example

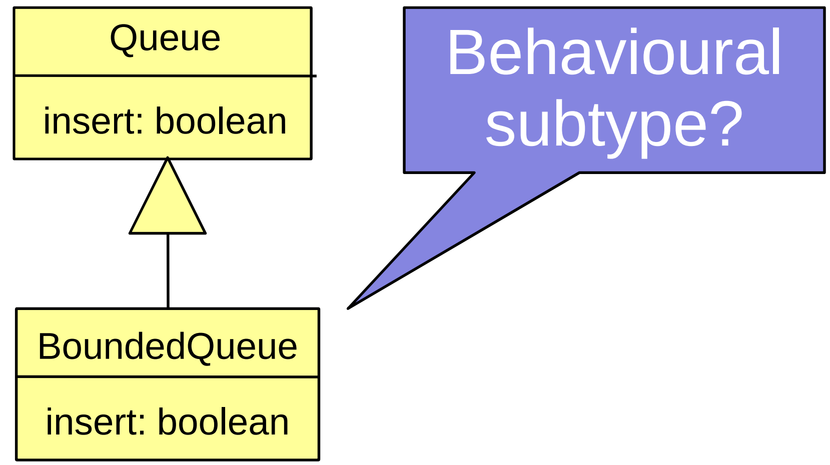
Method *insert* in *Queue*:

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

Overriding method *insert* in *BoundedQueue*:

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

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

What if we'd made a typo?



To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}} \Rightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

Motivating Example

To check:

$$BoundedQueue::insert_{POST} \Rightarrow Queue::insert_{POST}$$

- With the typo we get the same positive result ..

Motivating Example

To check:

$$\textit{BoundedQueue::insert}_{\mathrm{POST}} \Rightarrow \textit{Queue::insert}_{\mathrm{POST}}$$

VALID

- With the typo we get the same positive result ..
- This is perfectly correct. Indeed, the implication's still valid.

Motivating Example

To check:

$\boldsymbol{BoundedQueue::insert_{\mathrm{POST}}}$

$\boldsymbol{\Rightarrow Queue::insert_{\mathrm{POST}}}$

VACUOUS

- $BoundedQueue::insert_{\mathrm{POST}} \Rightarrow Queue::insert_{\mathrm{POST}}$

Unsatisfiable!

Motivating Example

Method *insert* in *Queue*:

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

Method *add* in *ModelQueue*:

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

Overriding method *insert* in *BoundedQueue*:

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

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

These formulae are inconsistent

Motivating Example

Method *insert* in *Queue*:

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

Method *add* in *ModelQueue*:

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

Overriding method *insert* in *BoundedQueue*:

```
/*@ 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())

Motivating Example

Method *insert* in *Queue*:

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

Method *add* in *ModelQueue*:

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

Overriding method *insert* in *BoundedQueue*:

```
/*@ 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())

Motivating Example

Method *insert* in *Queue*:

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

Method *add* in *ModelQueue*:

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

Overriding method *insert* in *BoundedQueue*:

```
/*@ 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:

$$SAT: \Phi \rightarrow \{\mathbf{s}, \mathbf{u}\}$$

$SAT(\varphi) = \mathbf{s}$ iff φ is satisfiable [written $\mathbf{s}(\varphi)$]
$SAT(\varphi) = \mathbf{u}$ otherwise [written $\mathbf{u}(\varphi)$]

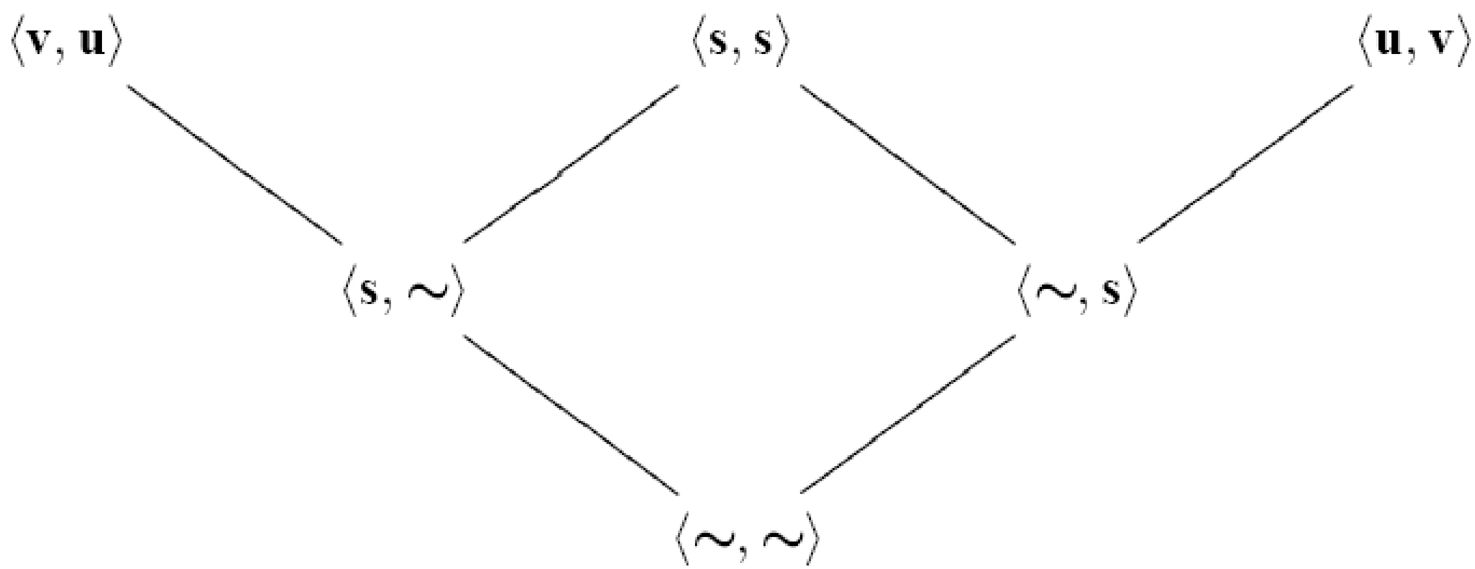
- And define further *satisfiability values* in terms of **s** and **u** ..

$$\mathbf{v}(\varphi) \text{ iff } \mathbf{u}(\neg \varphi)$$

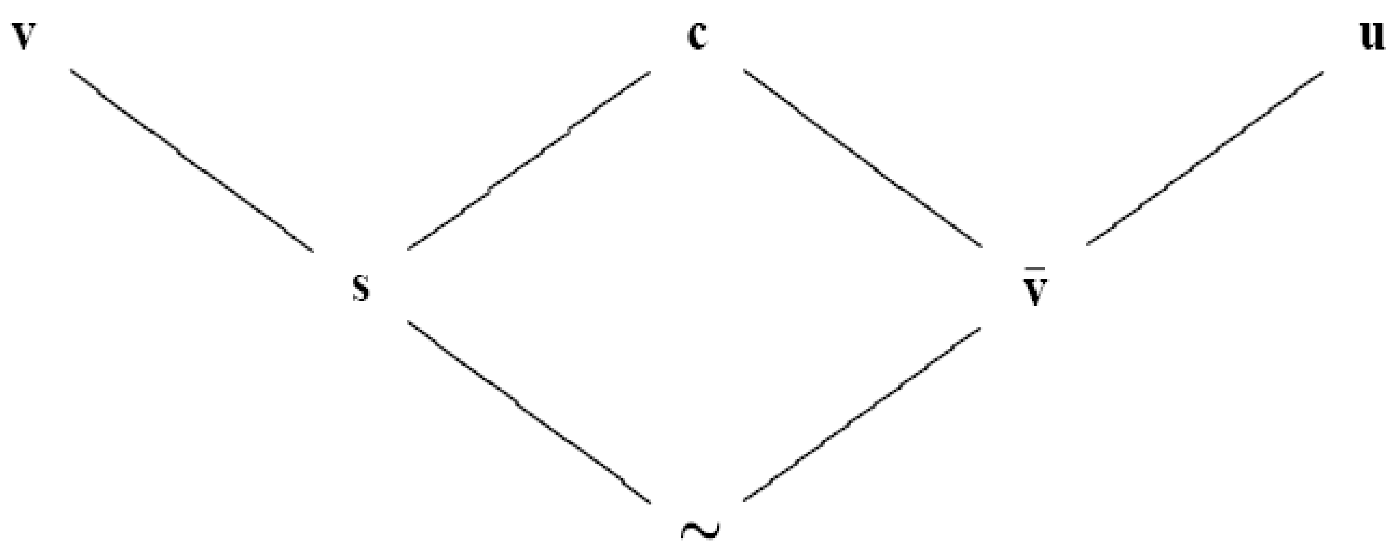
$$\mathbf{c}(\varphi) \text{ iff } \mathbf{s}(\varphi) \wedge \mathbf{s}(\neg \varphi)$$

$$\bar{\mathbf{v}}(\varphi) \text{ iff } \mathbf{s}(\neg \varphi)$$

Ordering of φ,¬φ pairs



Ordering of satisfiability values



$$\mathbf{v} > \mathbf{s}, \quad \mathbf{c} > \mathbf{s}, \quad \mathbf{c} > \bar{\mathbf{v}}, \quad \mathbf{u} > \bar{\mathbf{v}}, \quad \mathbf{s} > \sim, \quad \bar{\mathbf{v}} > \sim$$

Obtaining Values

- In addition to obtaining values through the oracle *SAT*, values may also be inferred: e.g. knowing **s**(φ) and **s**(¬ φ) gives **c**(φ) and **c**(¬ φ)
- To permit inference, we store obtained values in a lookup table:

$$SAT_{\text{Table}} : \Phi \rightarrow \{\mathbf{v}, \mathbf{c}, \mathbf{u}, \mathbf{s}, \bar{\mathbf{v}}, \sim\}$$

- 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:

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

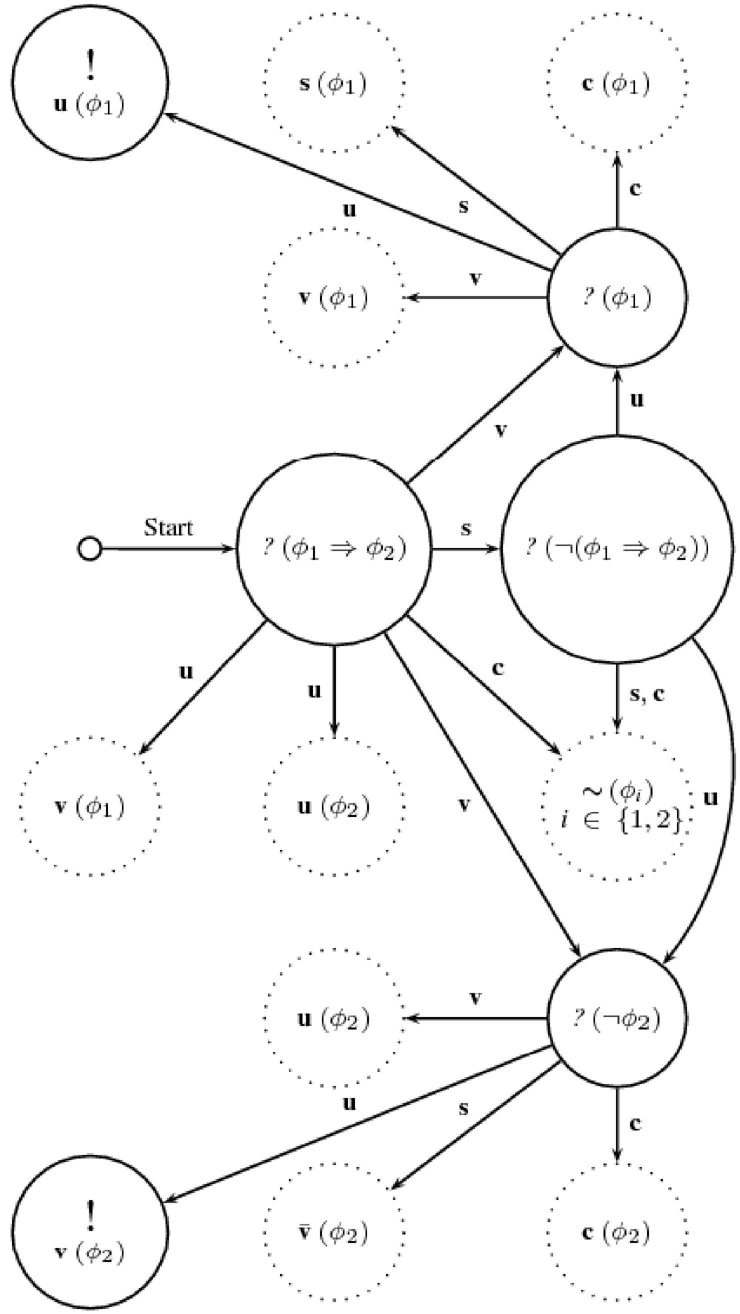
- *?* (φ) denotes a *satisfiability query*

Implication Pattern

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

$\Rightarrow$

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

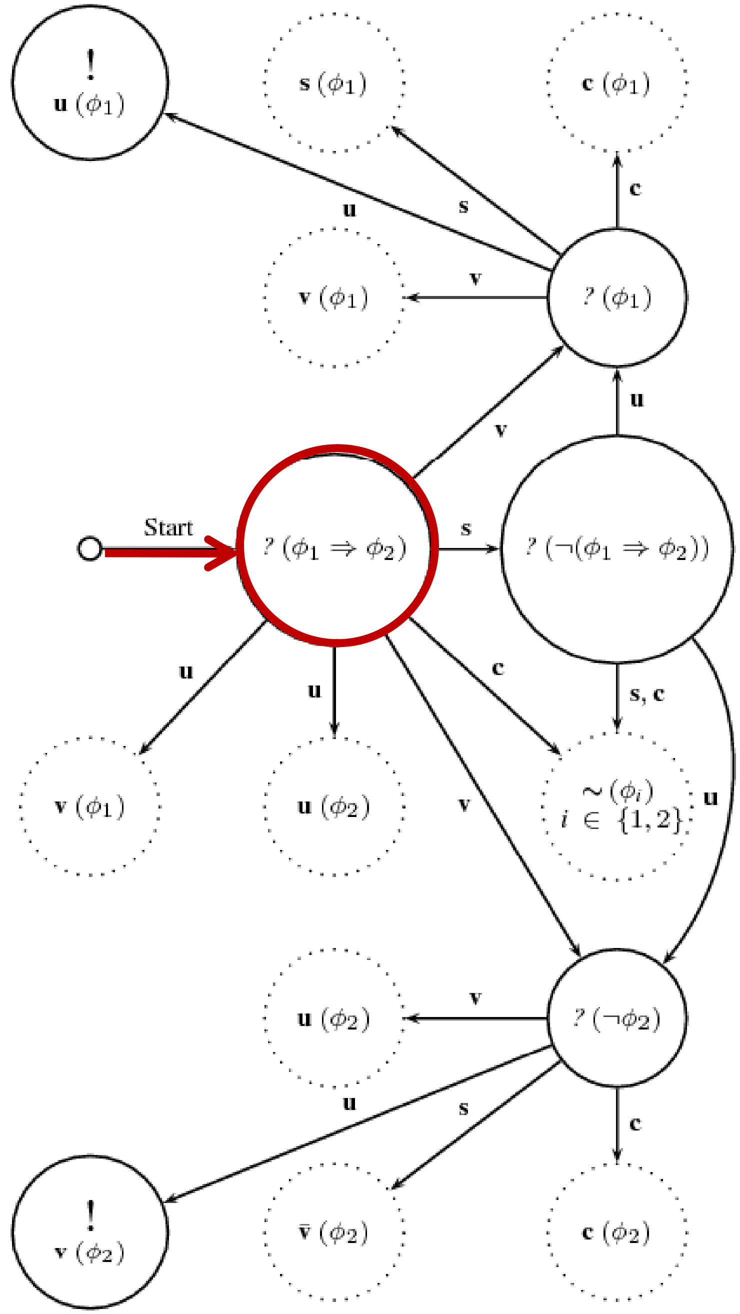


Implication Pattern

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

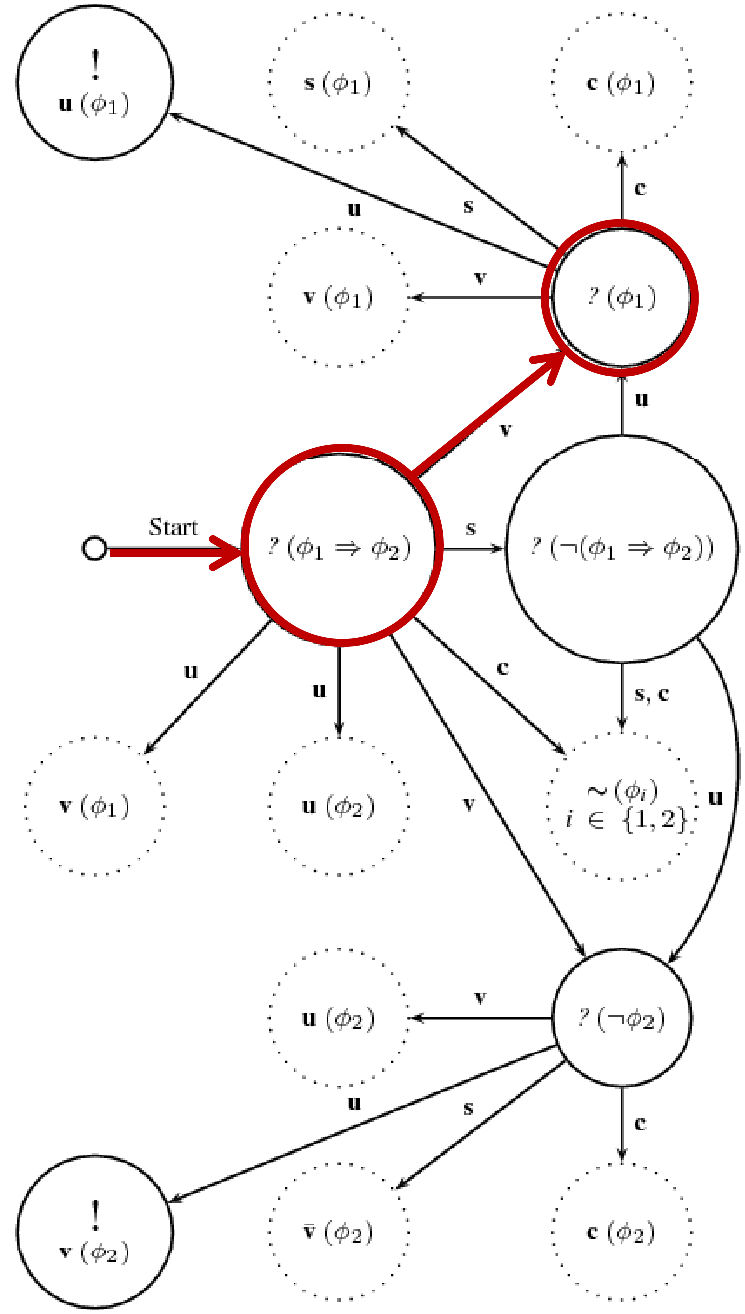
$$\Rightarrow$$

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



Implication Pattern

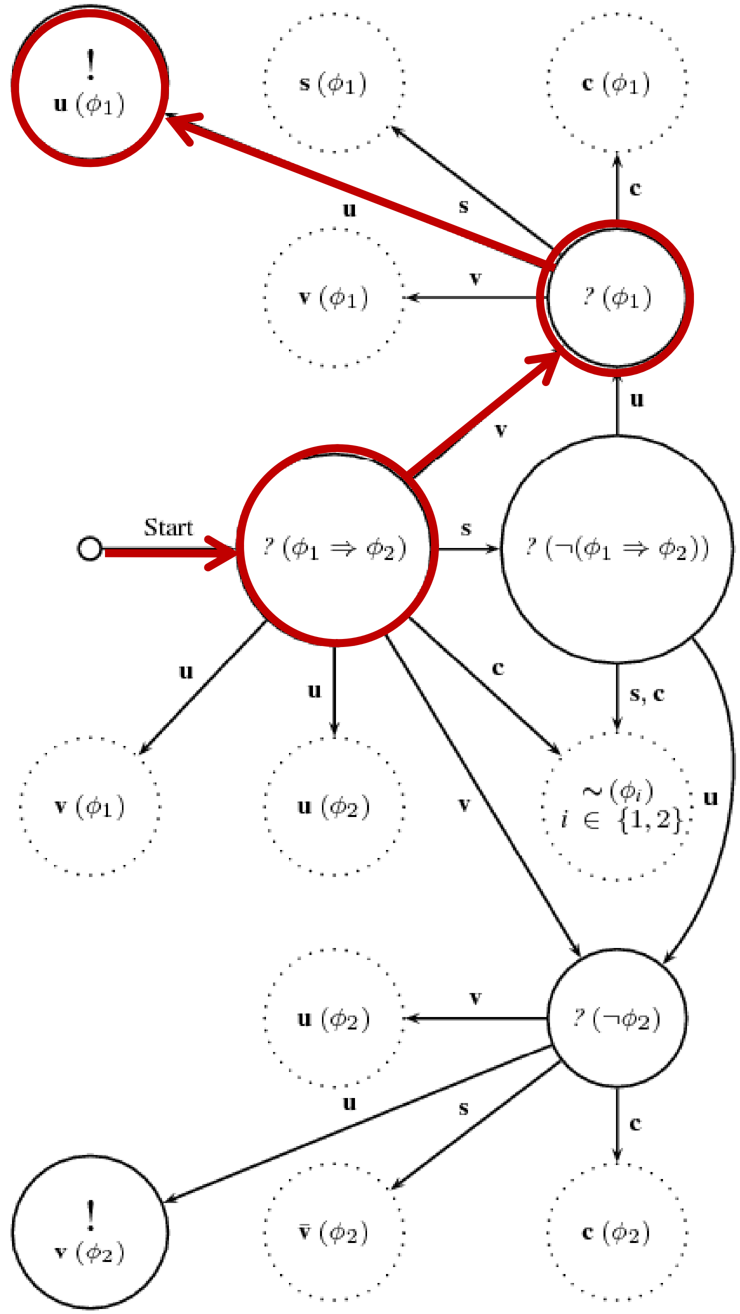
(result ==> contains(e))
&& (entries == \old(entries.add(e)))
&& ensures size() < \old(entries.size())
&& size() <= MAX



Implication Pattern

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

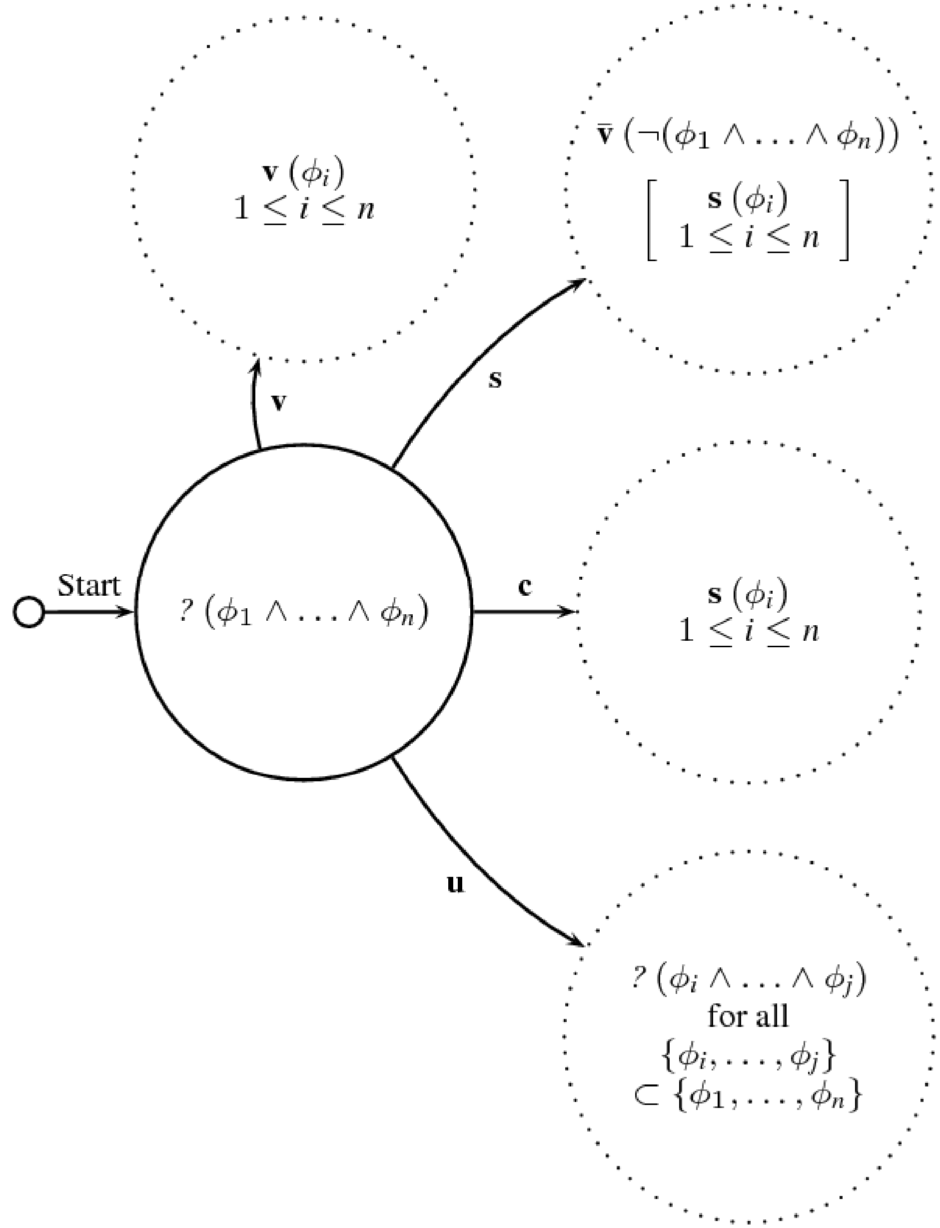
!



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

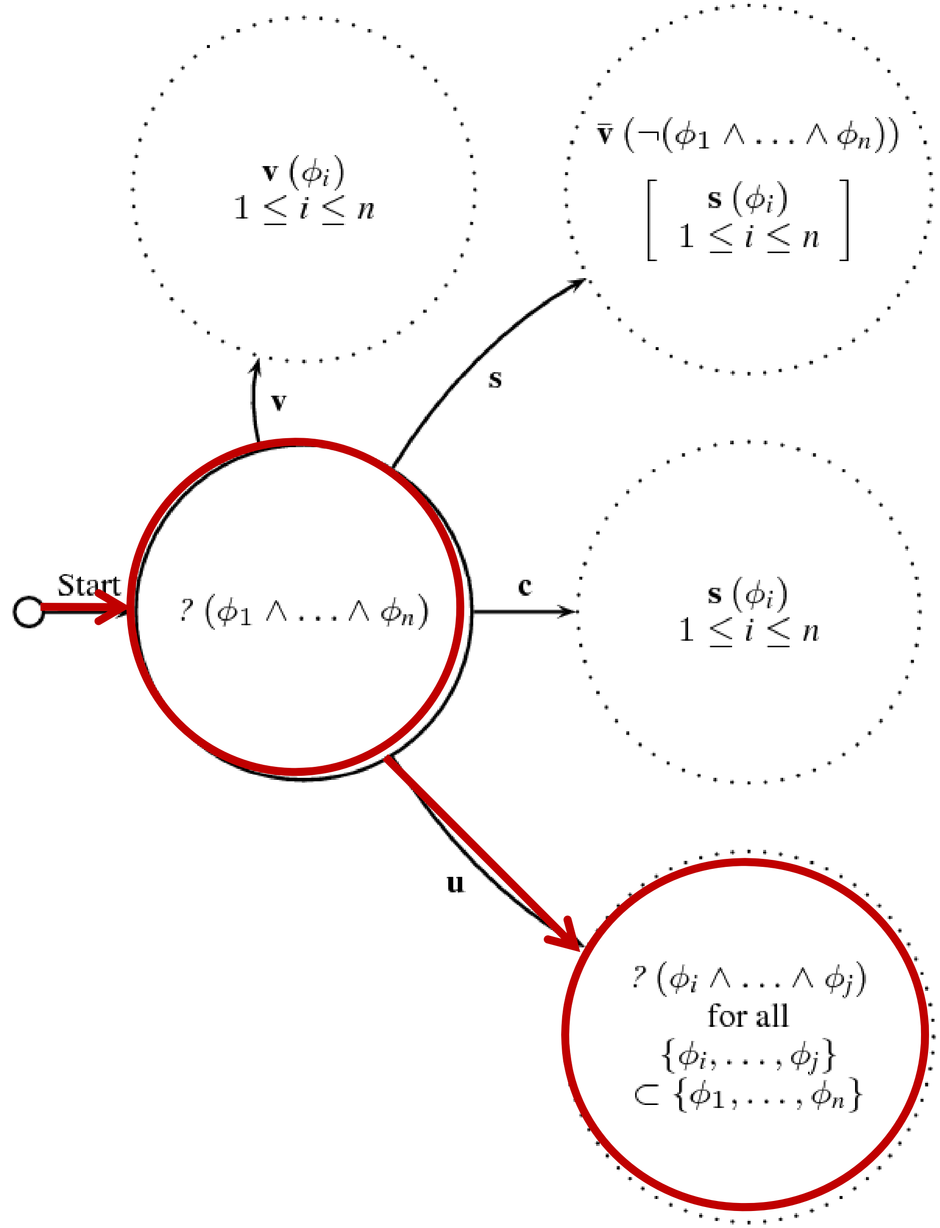
Conjunction Pattern

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



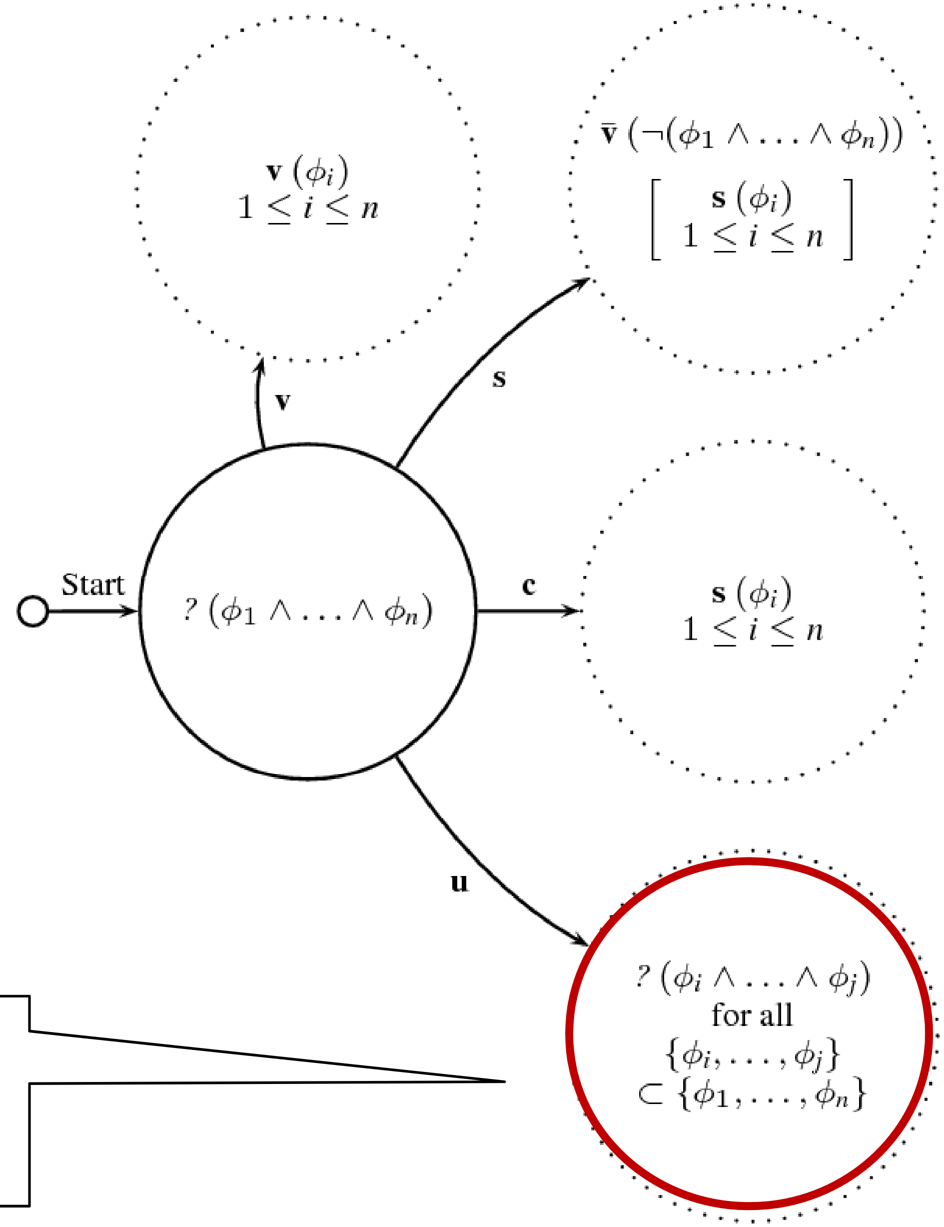
Conjunction Pattern

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



Conjunction Pattern

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

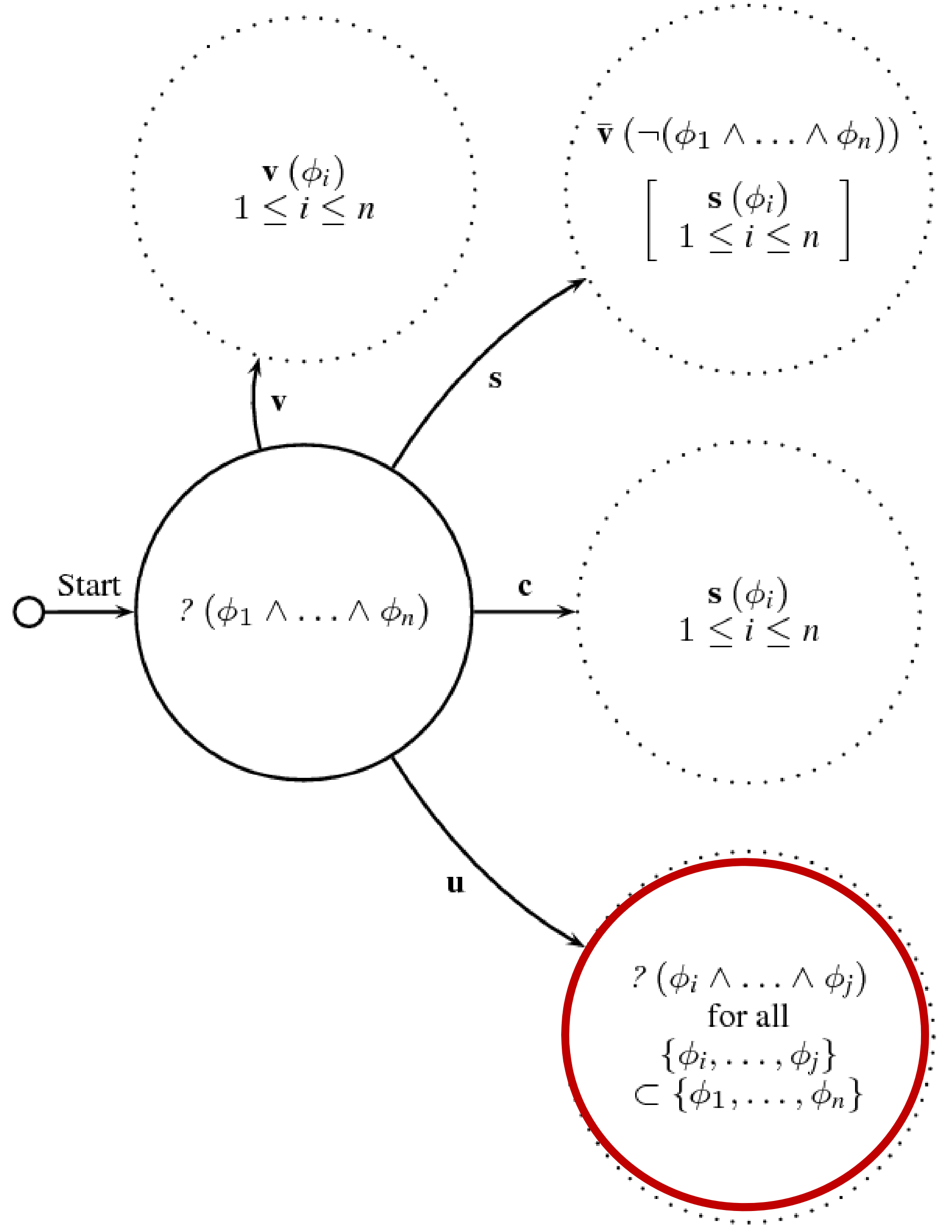


Try combinations of conjunct until you find the inconsistent set

Conjunction Pattern

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

This includes exploring the specifications of *contains*, *add* and both *sizes*, with pure method specs treated as special kinds of conjunction



Conjunction Pattern

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

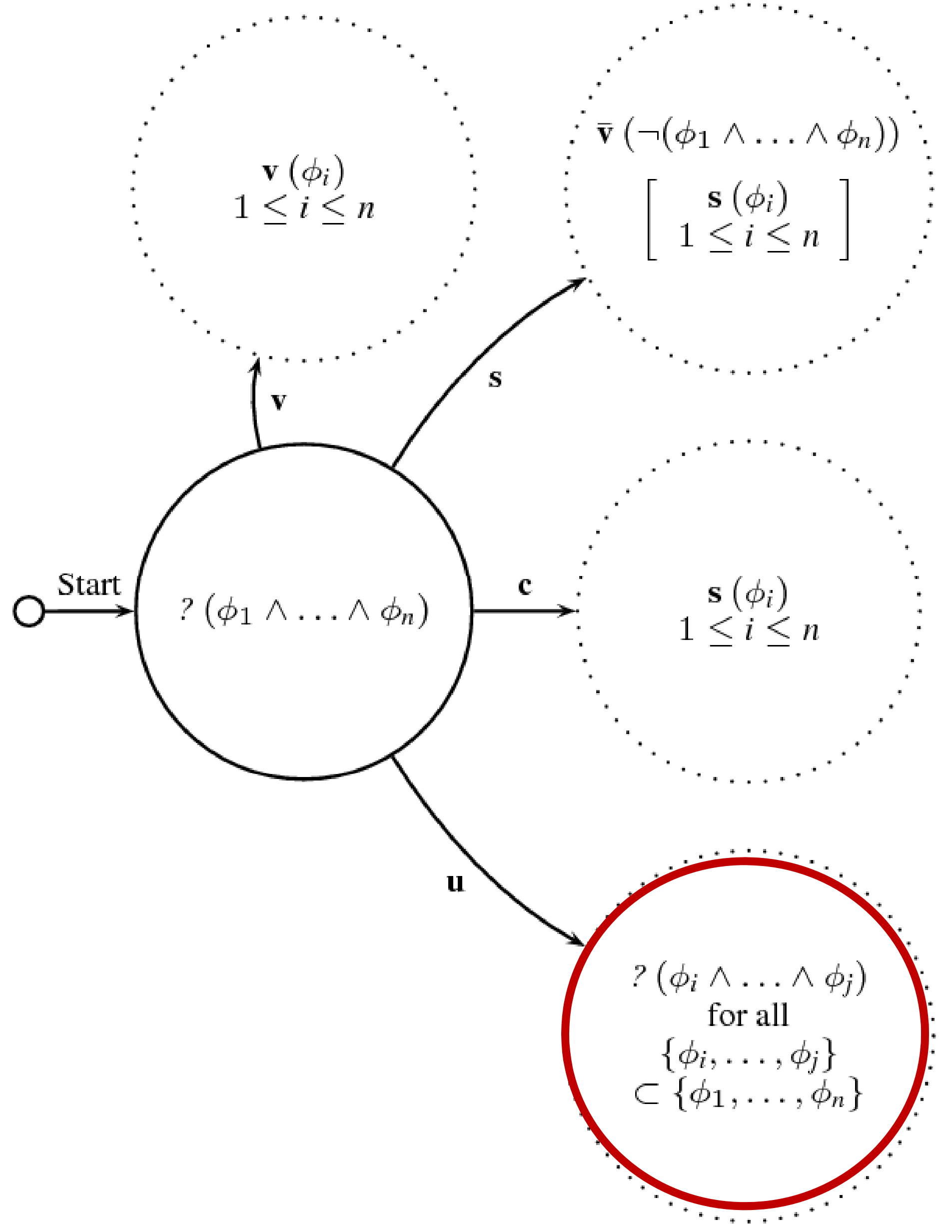
Terminates with inconsistent pair:

BoundedQueue::insert

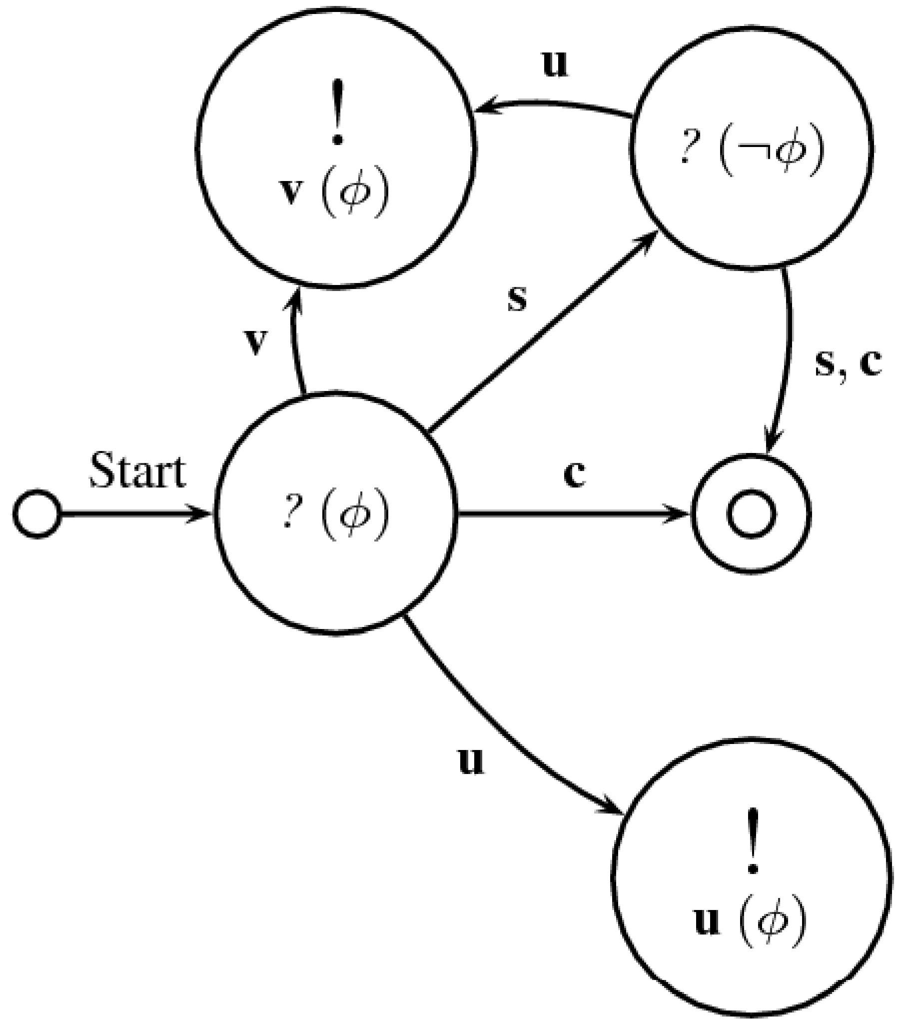
size() < \old(entries.size())

ModelQueue::add

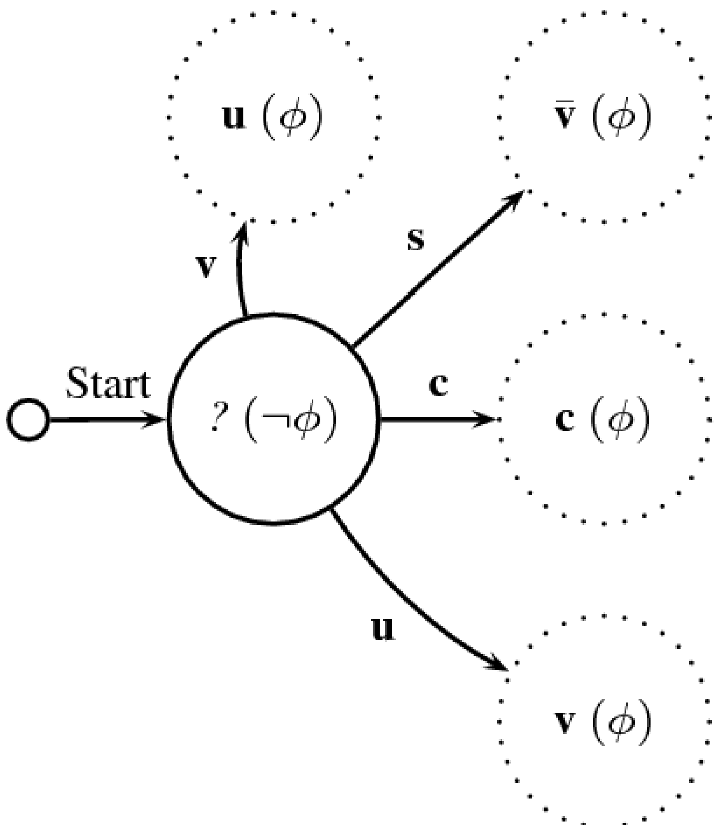
result.size() >= \old(size())

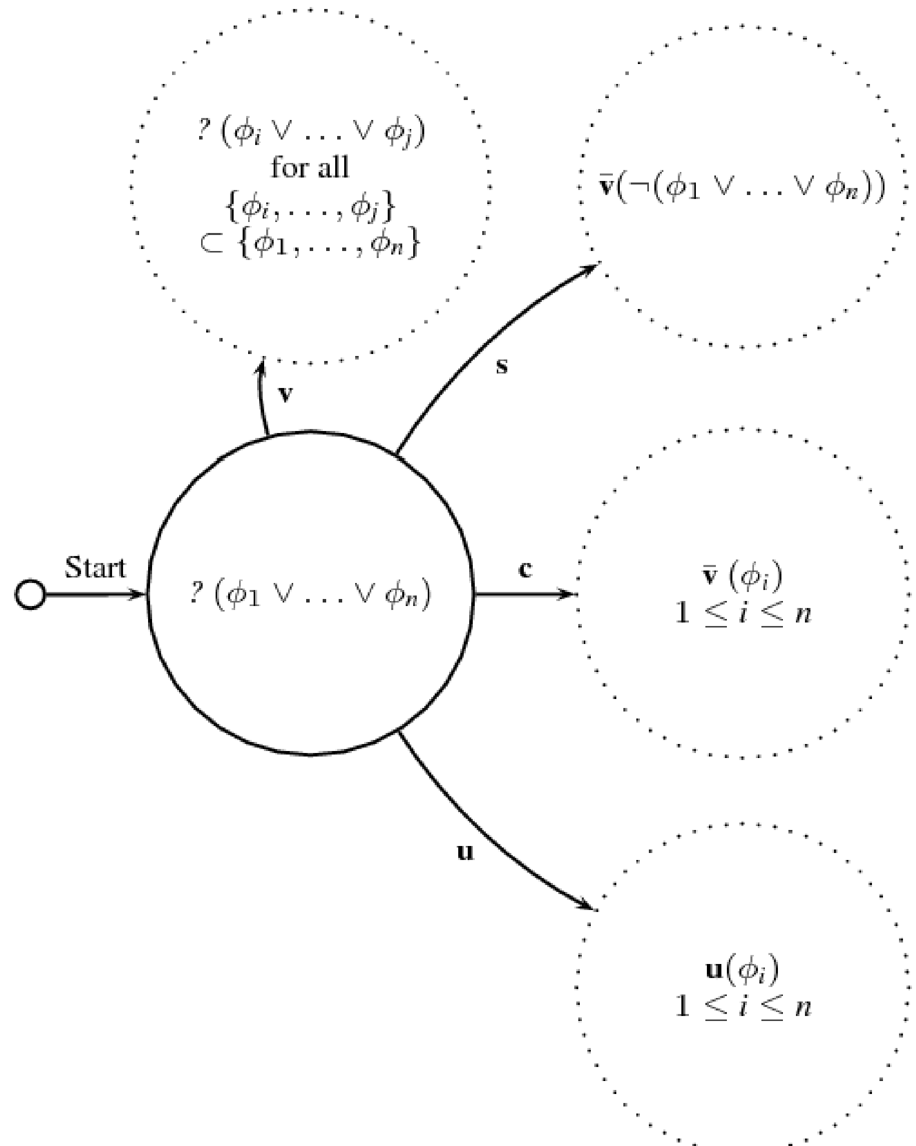


Base Pattern



Negation & Disjunction Patterns





Summary

- Systematic exploration of the satisfiability of subformulae
  - Why does the (top-level) formula have satisfiability value *v*?
- 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