Towards a Unified Specification Framework

Ilham W. Kurnia

joint work with: Arnd Poetzsch-Heffter, Yannick Welsch

University of Kaiserslautern

May 11, 2010

http://www.hats-project.eu

Overview

- Example (subject-observer pattern)
- Trace-based specification
- State-based specification
- Their relation

Subject-Observer Pattern [GOF]

```
interface IObserver {
  void notify(State s);
}

class Subject {
  IObserver o1, o2;

  Subject(IObserver o1,
          IObserver o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Subject-Observer Pattern [GOF]

```
interface IObserver {
  void notify(State s);
}

class Subject {
  IObserver o1, o2;

  Subject(IObserver o1,
          IObserver o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Assumption

- Different observers
- Non-null observers
- Deterministic
- Sequential

Subject-Observer Pattern [GOF]

```
interface IObserver {
  void notify(State s);
}

class Subject {
  IObserver o1, o2;

  Subject(IObserver o1,
          IObserver o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

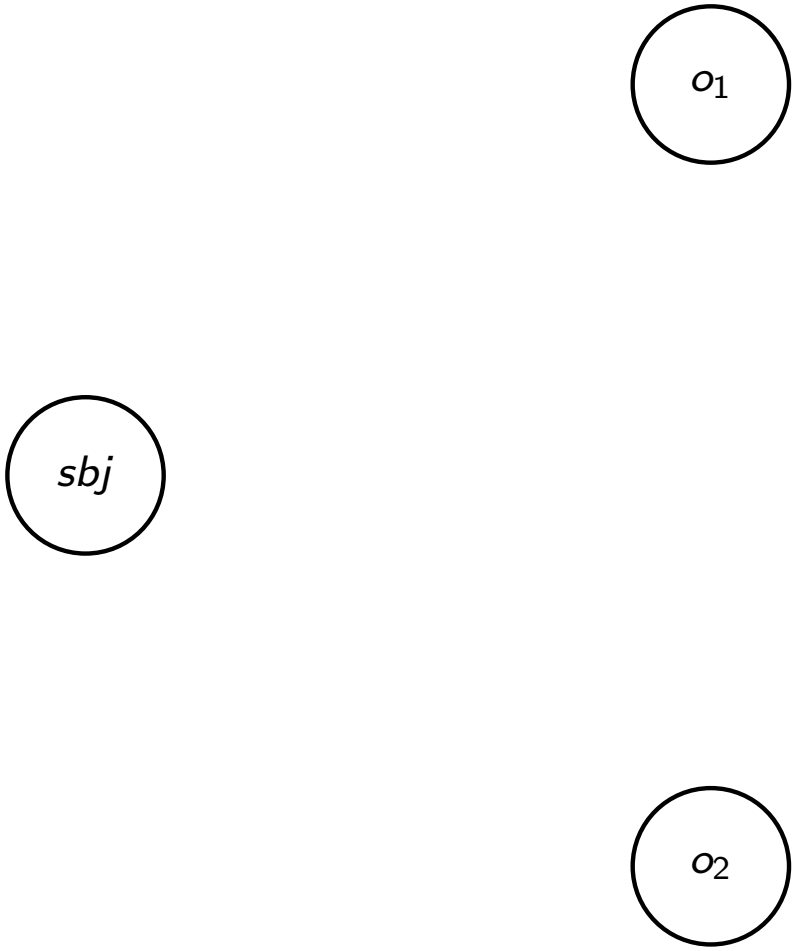
Assumption

- Different observers
- Non-null observers
- Deterministic
- Sequential

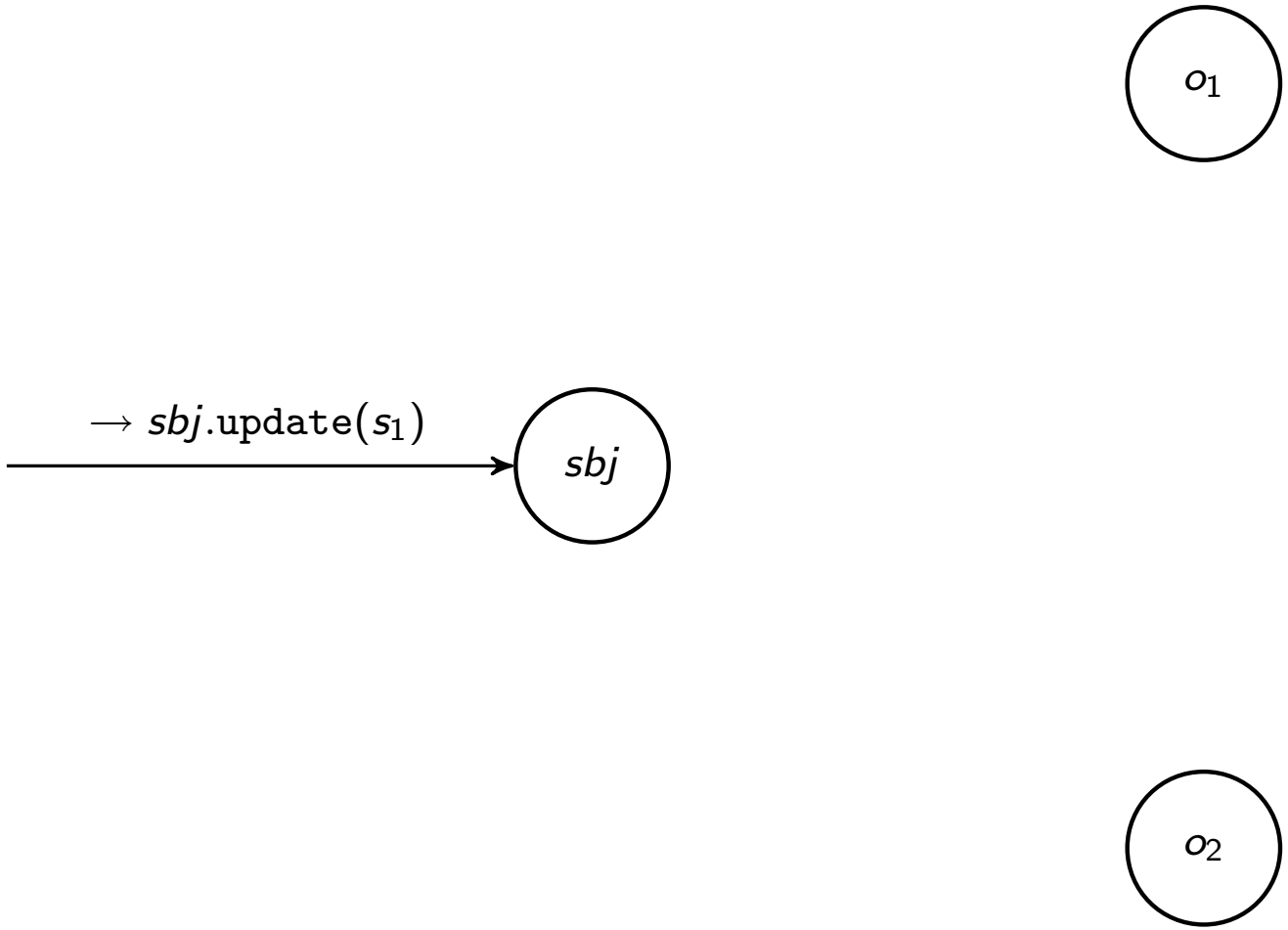
Messages

- $\longrightarrow$ `sbj.update(s)`
- $\longleftarrow$ `sbj.update()`
- $\longrightarrow$ `o.notify(s)`
- $\longleftarrow$ `o.notify()`

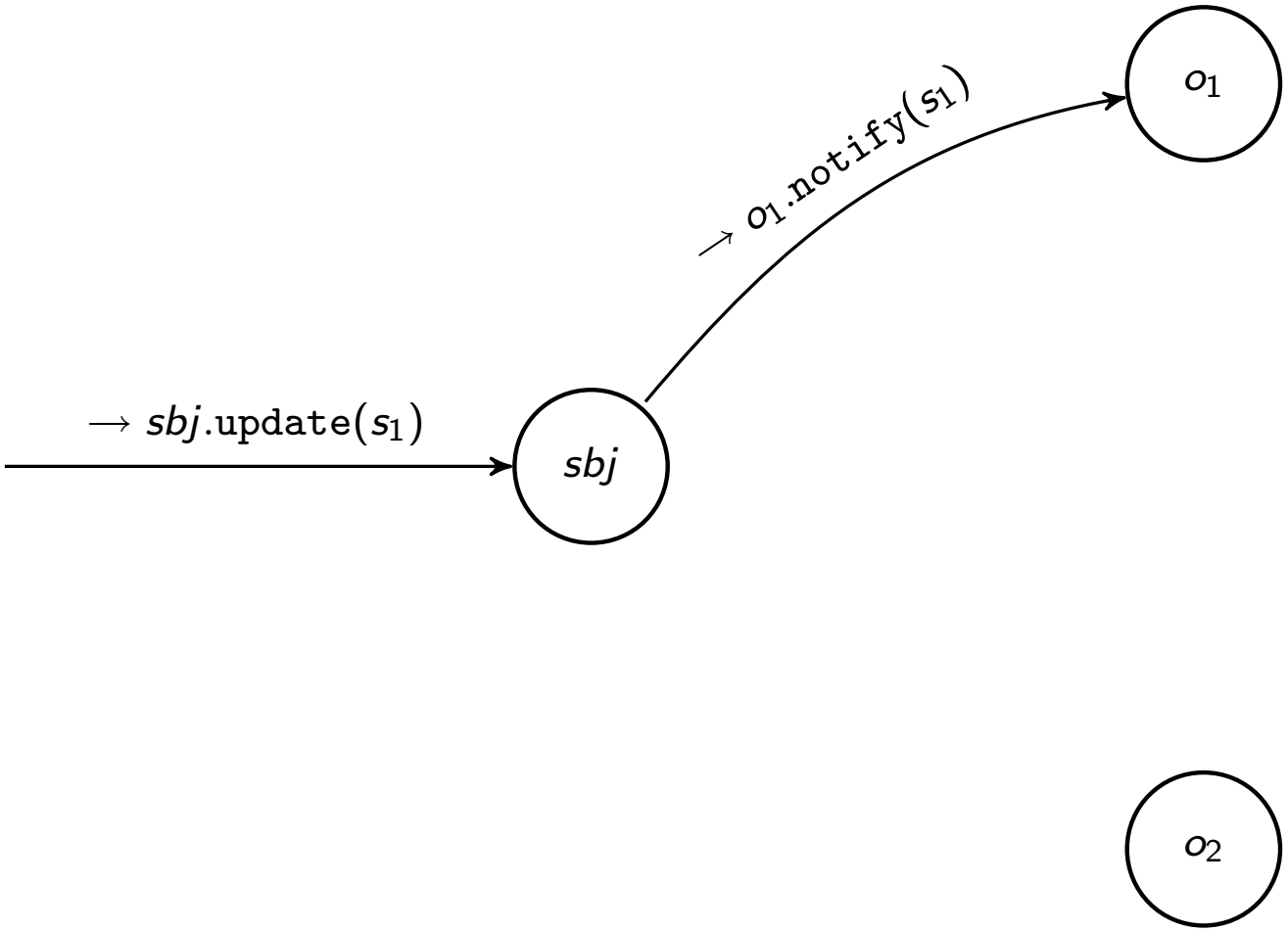
Scenarios



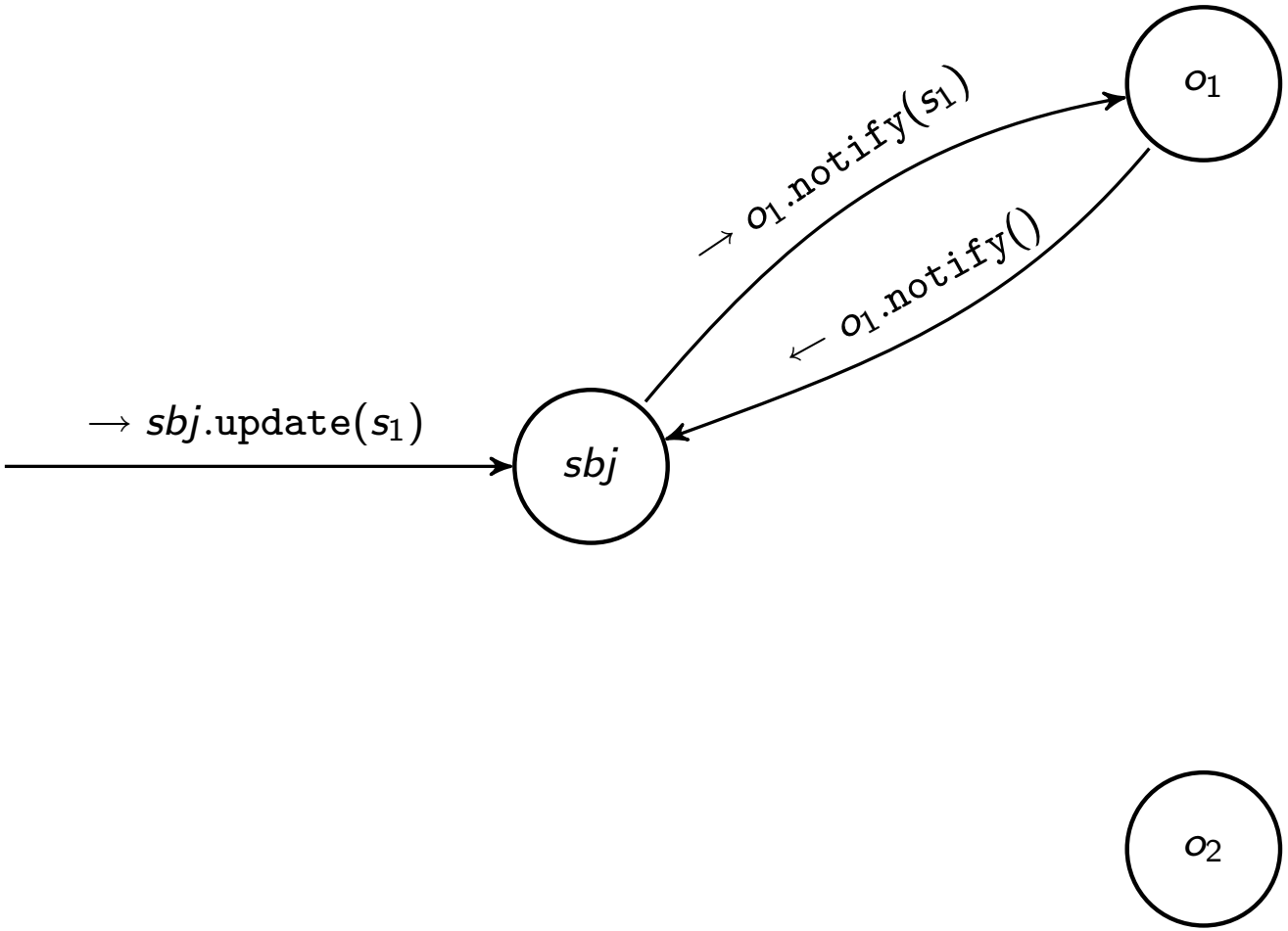
Scenarios



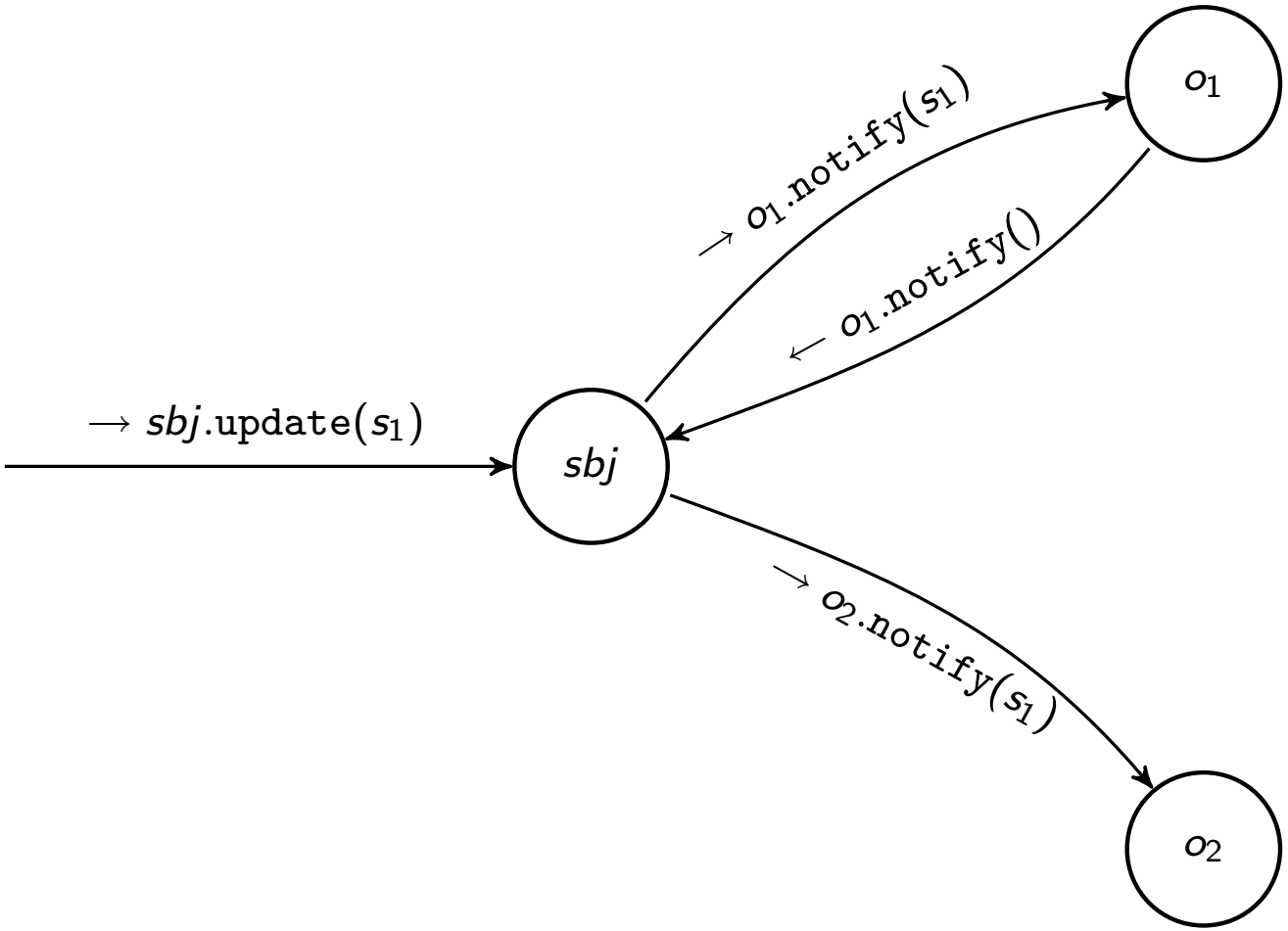
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
o_2

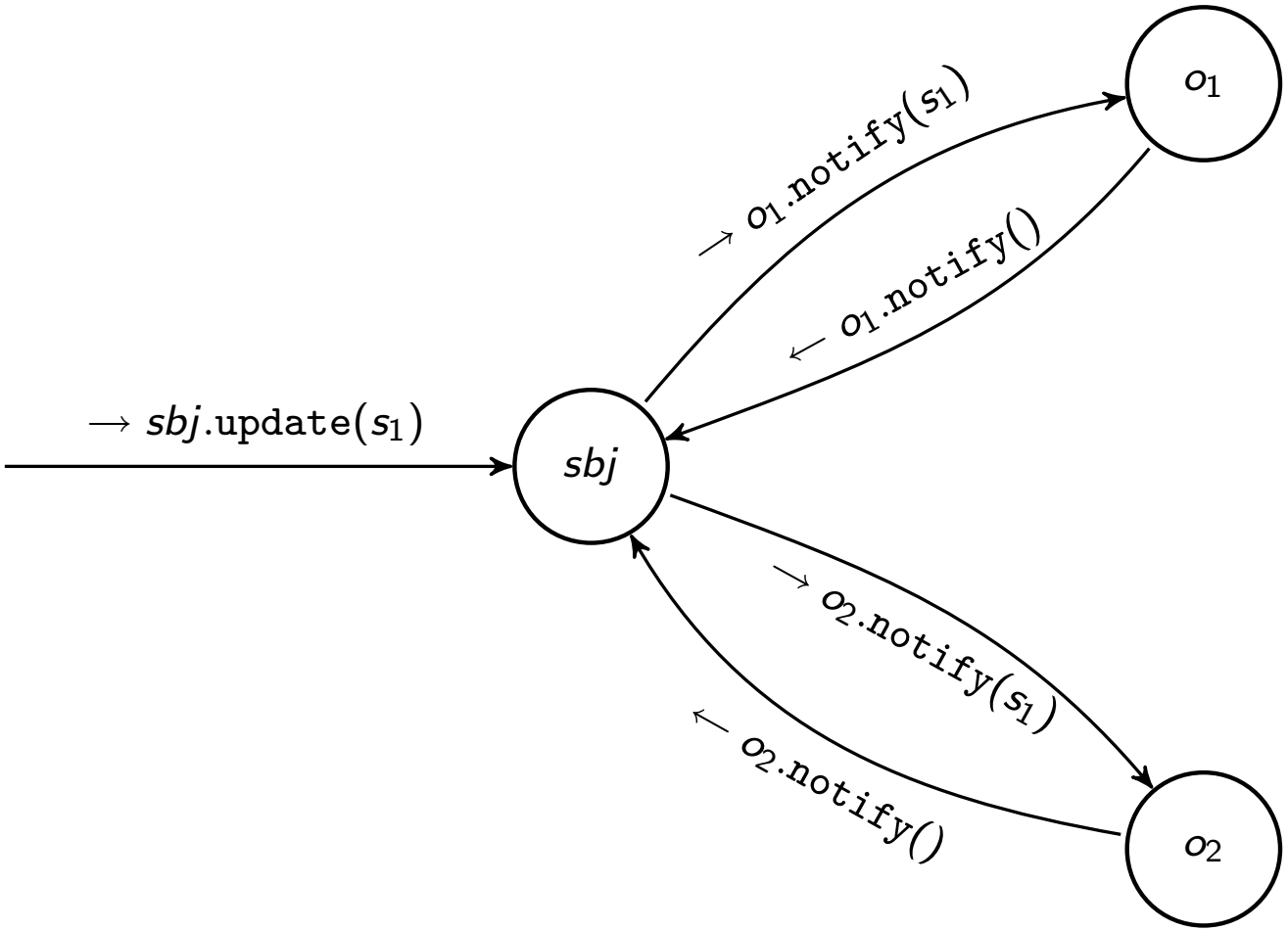
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
o_2

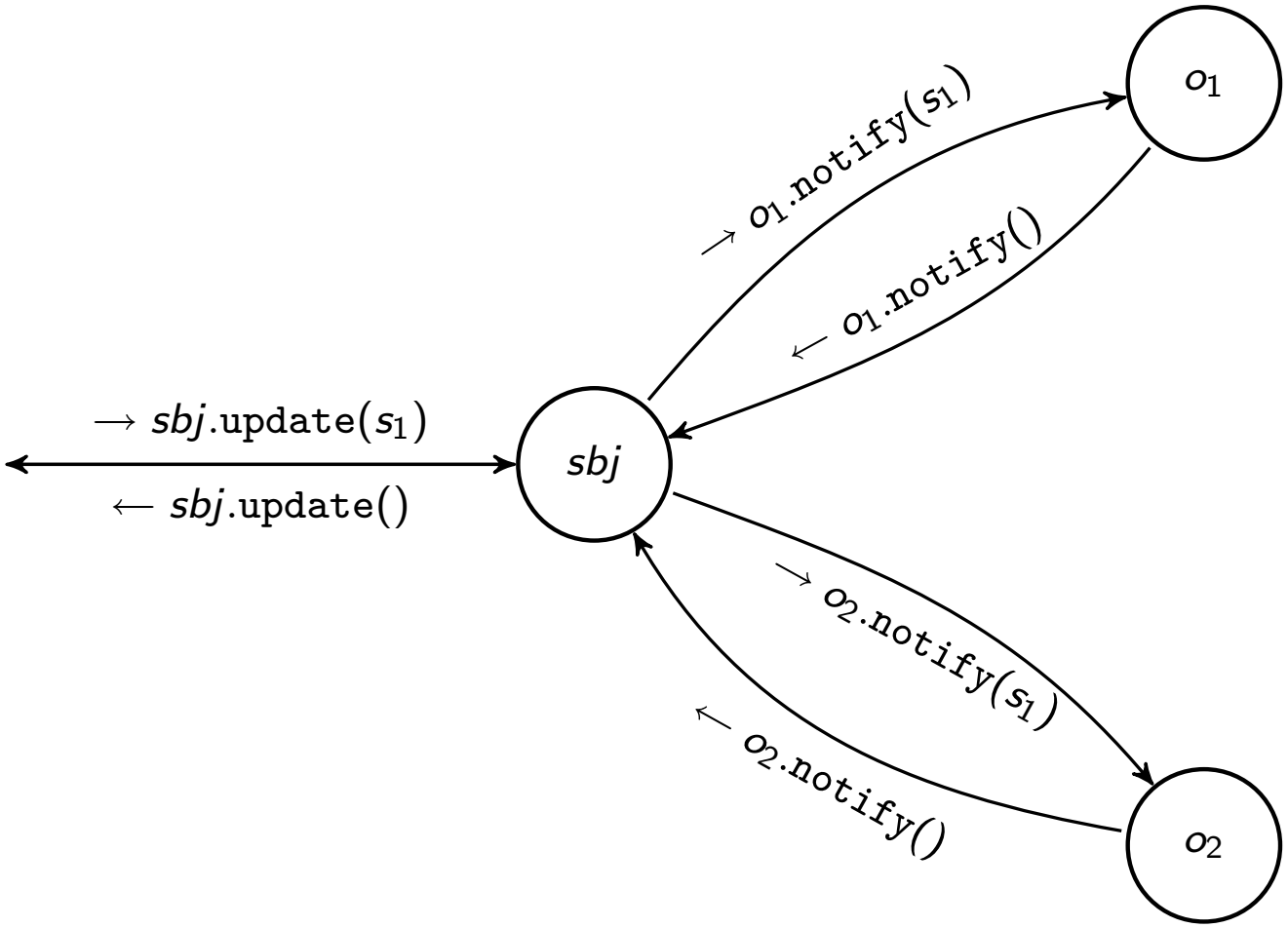
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
$\rightarrow o_2.\texttt{notify}(s_1)$
o_2

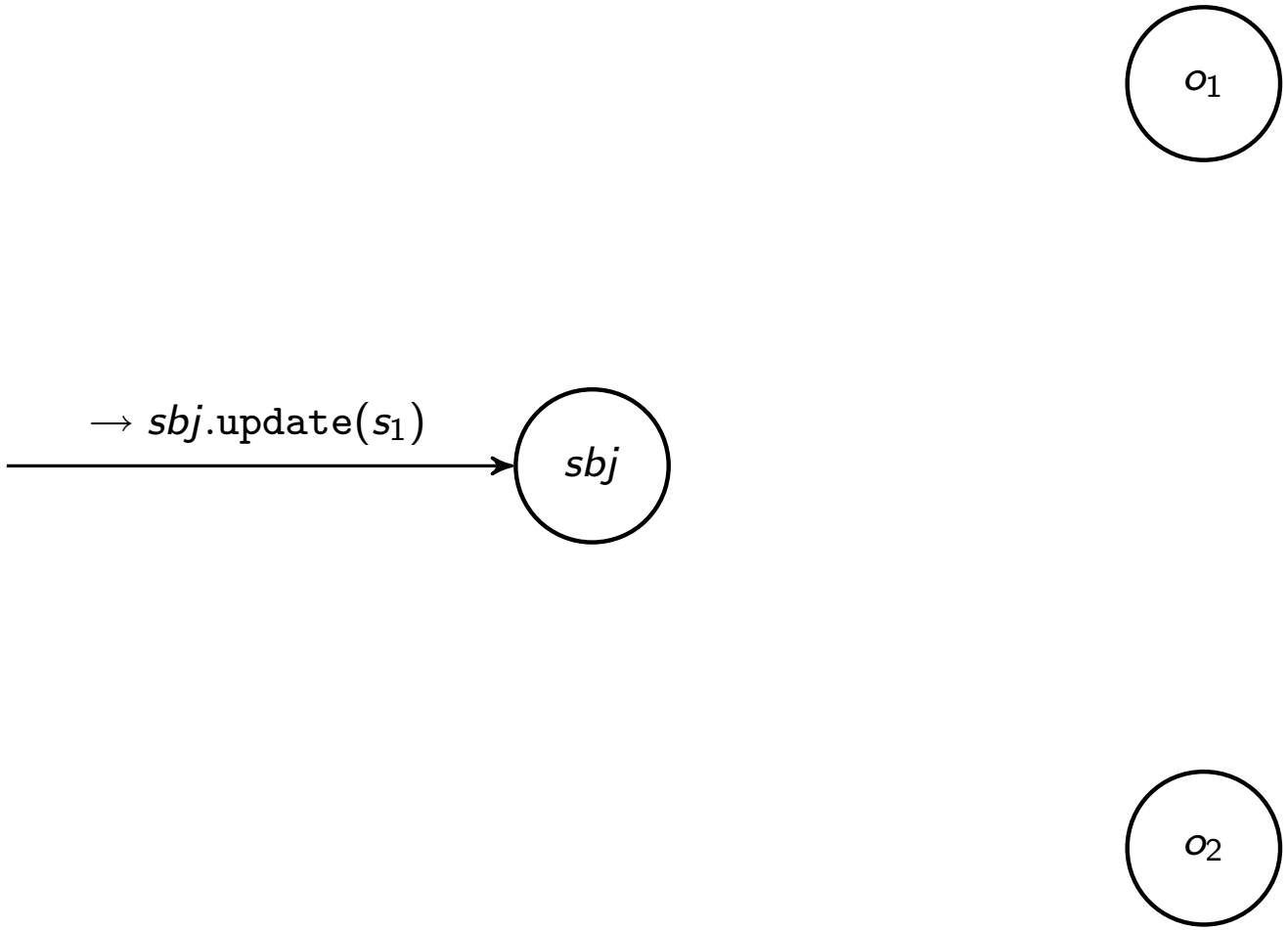
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
$\rightarrow o_2.\texttt{notify}(s_1)$
$\leftarrow o_2.\texttt{notify}()$
o_2

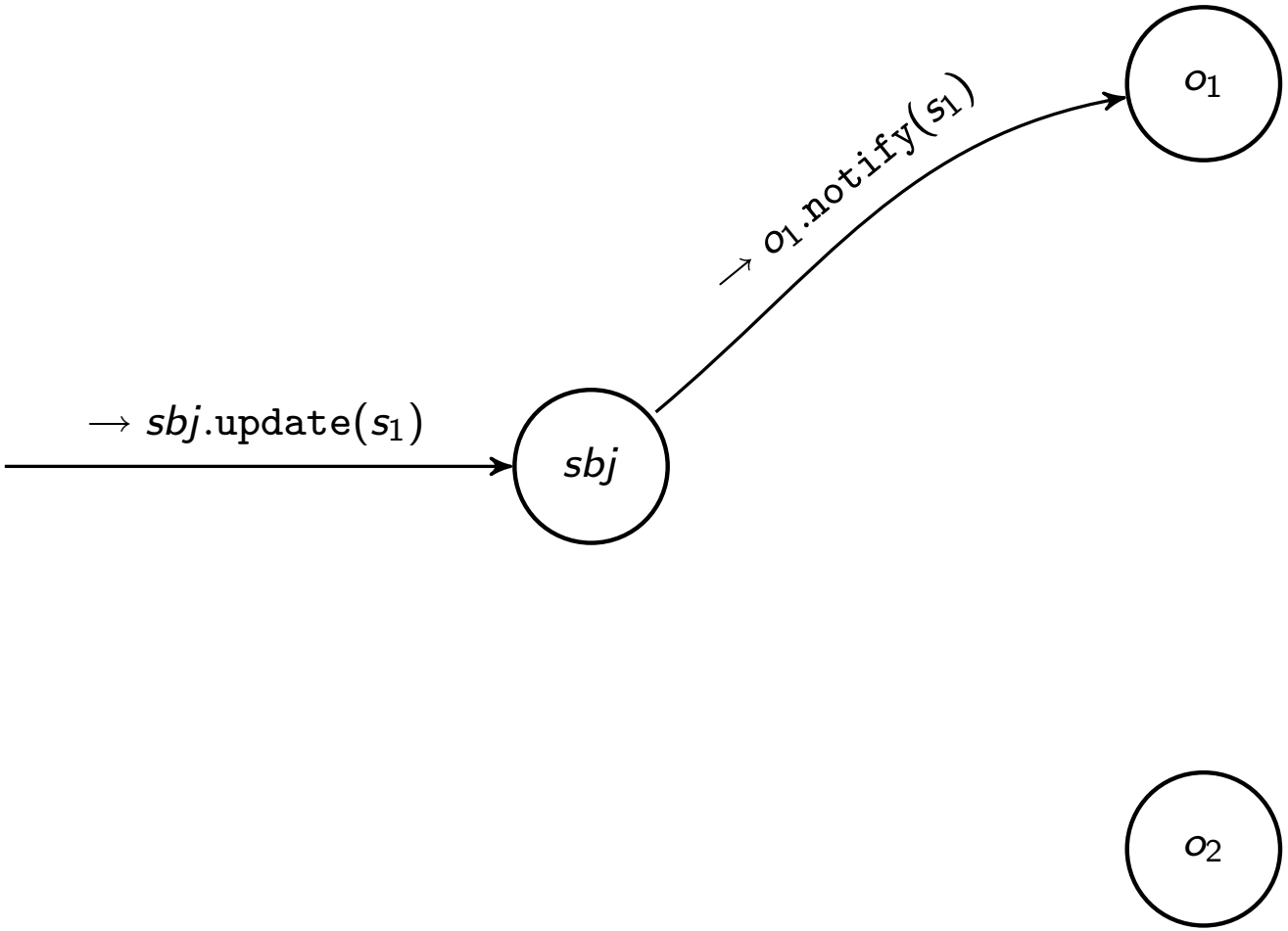
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
$\leftarrow sbj.\texttt{update}()$
sbj
$\rightarrow o_2.\texttt{notify}(s_1)$
$\leftarrow o_2.\texttt{notify}()$
o_2

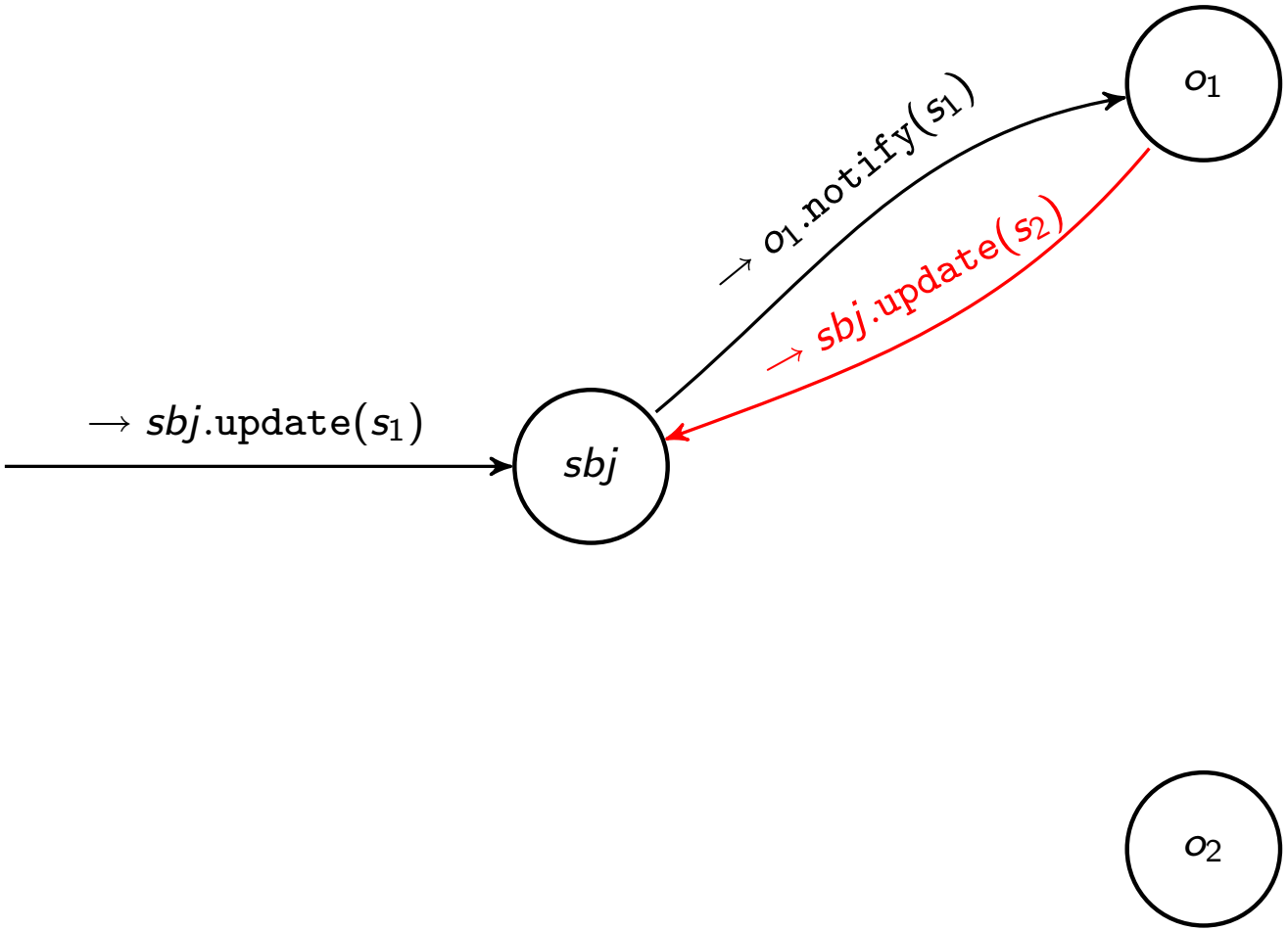
Scenarios



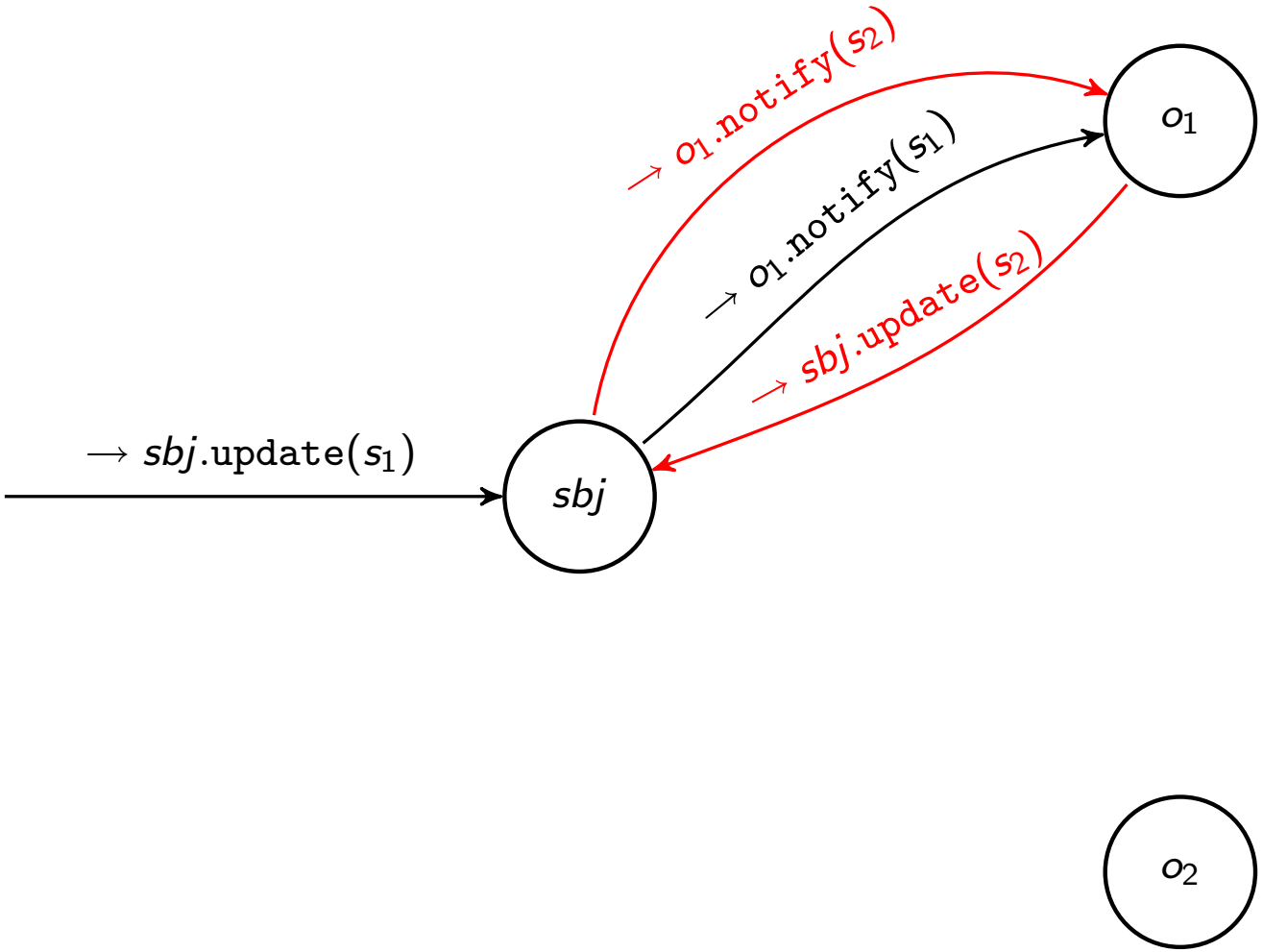
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
o_2

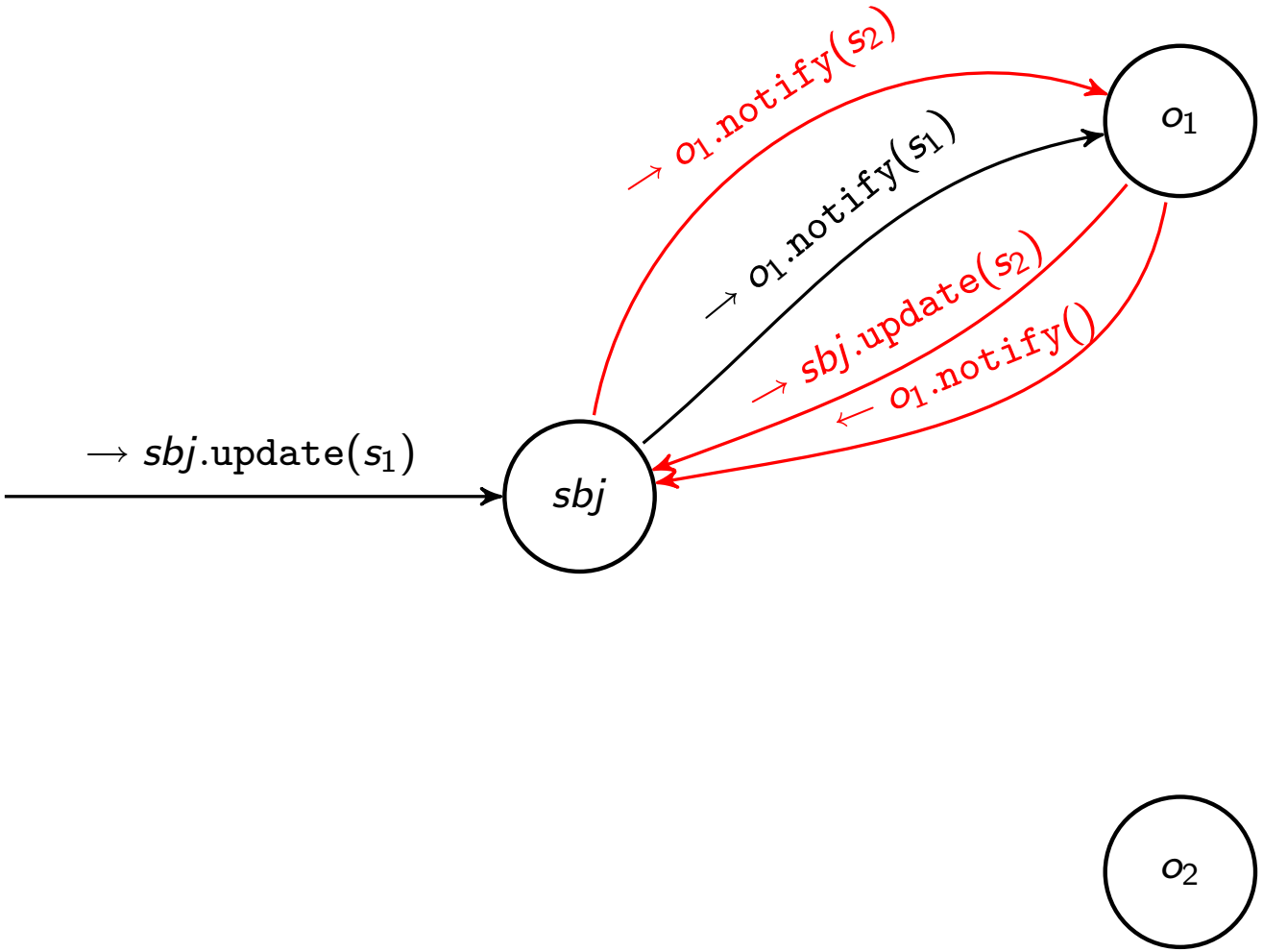
Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\rightarrow sbj.\texttt{update}(s_2)$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
o_2

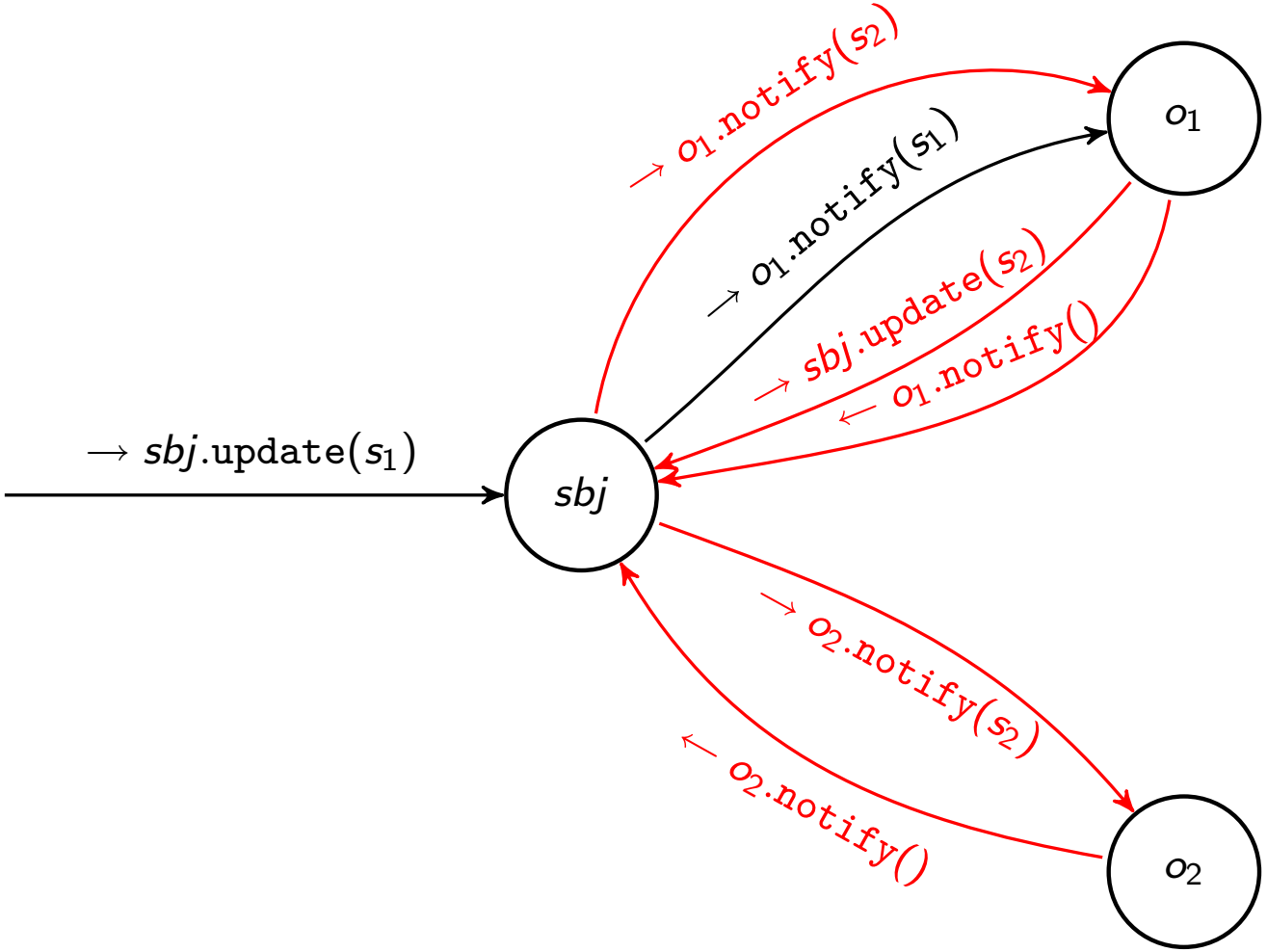
Scenarios


o_1.notify(s_2)
o_1.notify(s_1)
sbj.update(s_2)
$\longrightarrow$ sbj.update(s_1)
sbj
o_1
o_2

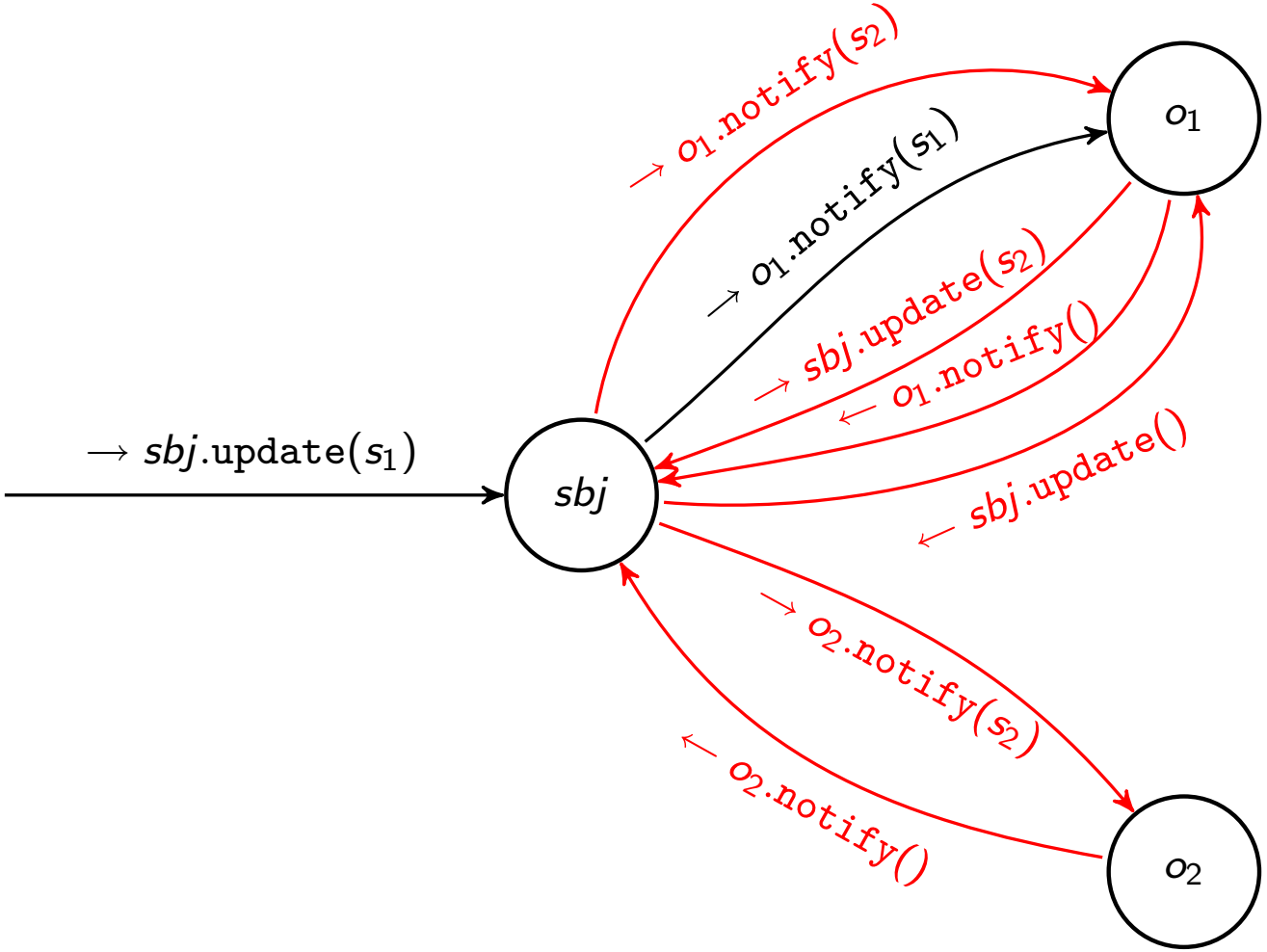
Scenarios


→ o1.notify(s2)
→ o1.notify(s1)
→ sbj.update(s2)
← o1.notify()
→ sbj.update(s1)
sbj
o1
o2

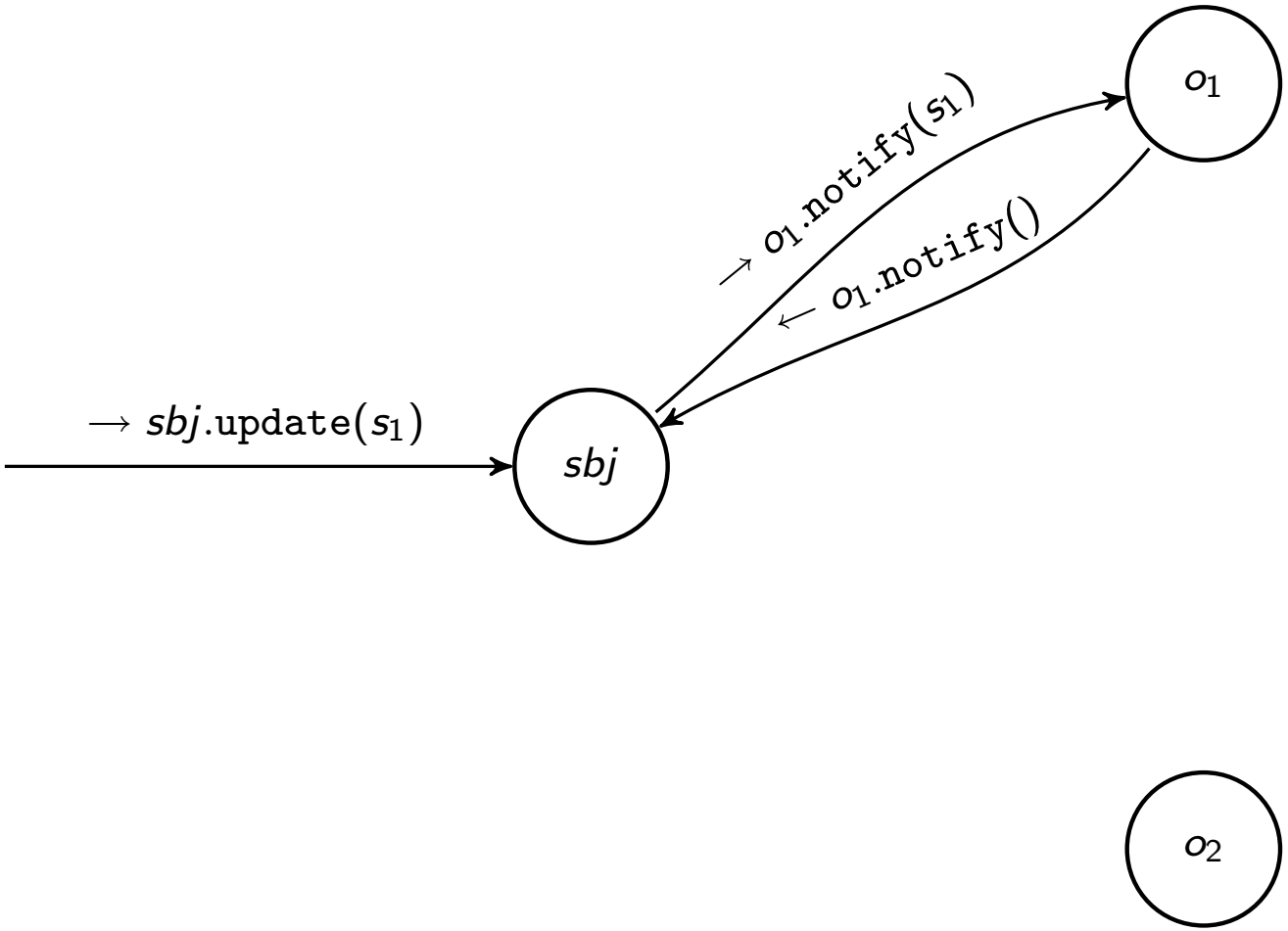
Scenarios


→ o1.notify(s2)
→ o1.notify(s1)
o1
→ sbj.update(s2)
← o1.notify()
→ sbj.update(s1)
sbj
→ o2.notify(s2)
← o2.notify()
o2

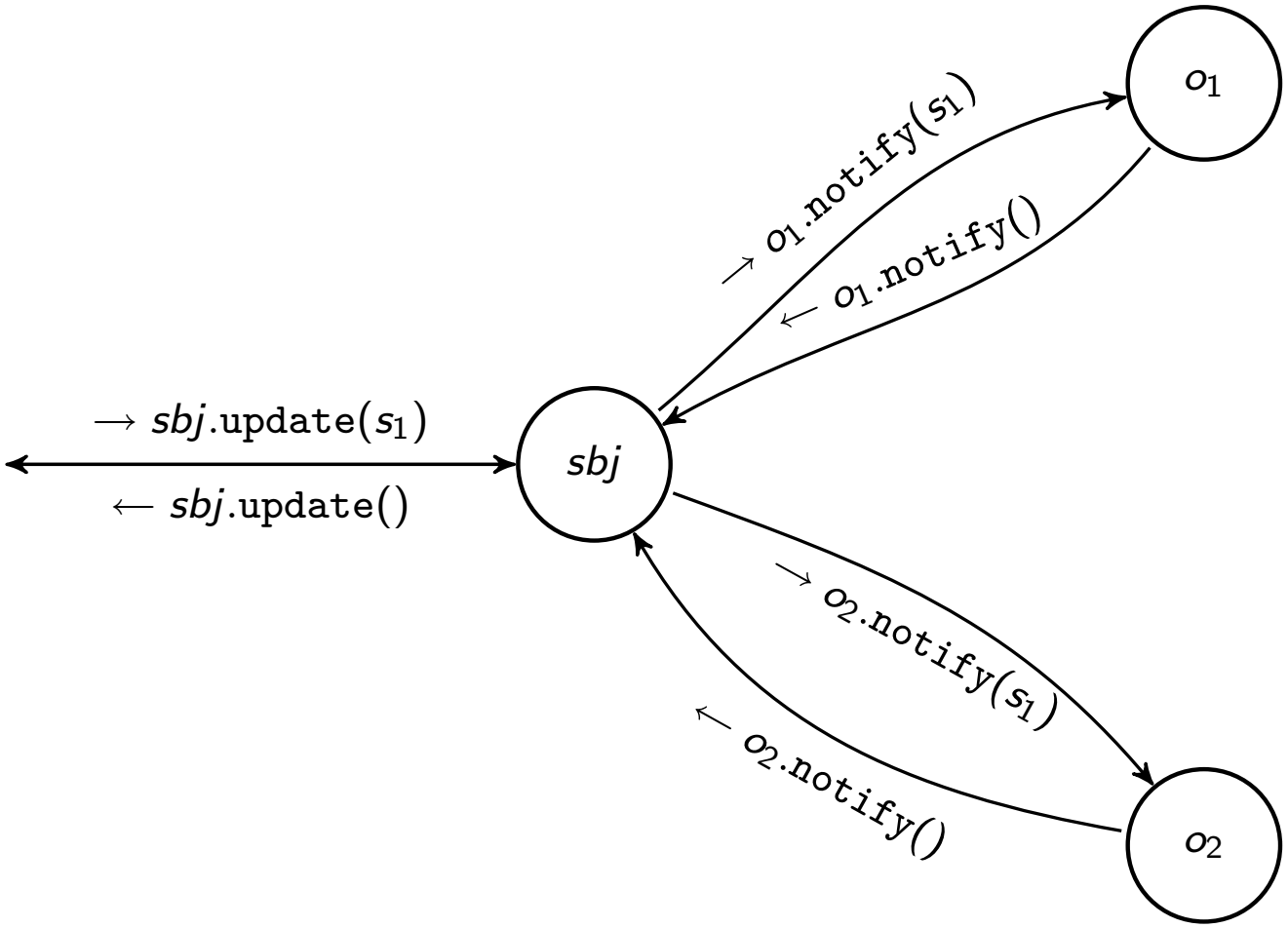
Scenarios


→ o1.notify(s2)
→ o1.notify(s1)
o1
→ sbj.update(s2)
← o1.notify()
→ sbj.update(s1)
sbj
← sbj.update()
→ o2.notify(s2)
← o2.notify()
o2

Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
sbj
o_2

Scenarios


o_1
$\rightarrow o_1.\texttt{notify}(s_1)$
$\leftarrow o_1.\texttt{notify}()$
$\rightarrow sbj.\texttt{update}(s_1)$
$\leftarrow sbj.\texttt{update}()$
sbj
$\rightarrow o_2.\texttt{notify}(s_1)$
$\leftarrow o_2.\texttt{notify}()$
o_2

Trace-Based Specification

trace spec `Subject` {

in $\rightarrow sbj.\texttt{Subject}(o_1', o_2')$ **out** $\leftarrow sbj.\texttt{Subject}()$
requires $o_1' \neq o_2' \neq \texttt{null}$;

in $\rightarrow sbj.\texttt{update}(s)$ **out** $\rightarrow o_1.\texttt{notify}(s)$
requires $sbj = \texttt{sbj}(h)$;
ensures $o_1 = \texttt{obs1}(h)$;

in $\leftarrow o.\texttt{notify}()$ **out** $\rightarrow o_2.\texttt{notify}(s)$
requires $o = \texttt{obs1}(h)$;
ensures $o_2 = \texttt{obs2}(h) \wedge s = \texttt{getS}(h)$;

in $\leftarrow o.\texttt{notify}()$ **out** $\leftarrow sbj.\texttt{update}()$
requires $o = \texttt{obs2}(h)$;
ensures $sbj = \texttt{sbj}(h)$;

}

Trace-Based Specification

```
Msg getS(List<Pair<Msg, Msg>> history) {
  for (int i = history.size() - 1; i >= 0; i--) {
    if (isUpdateMessage(history.get(i).first()) &&
        match(2 * i - 1, 2 * history.size()))
      return history.get(i).first();
  }
}
```

$$match(a,b) \stackrel{\text{def}}{=} \mu_a \in Msg^{in} \wedge \mu_b \in Msg^{out} \wedge header(\mu_a) = header(\mu_b) \wedge split(a+1, b-1), \text{and}$$

$$split(a,b) \stackrel{\text{def}}{=} a > b \vee match(a,b) \vee \exists\ a < c < b-1 \bullet split(a,c) \wedge split(c+1,b)$$

State-Based Specification

```
state spec Subject {
  Subject sbj;
  IObserver o1, o2;
  Stack<State> st;

  in → sbj.Subject(o1', o2') out ← sbj.Subject()
    requires o1' ≠ o2' ≠ null;
    ensures o1 = o1' ∧ o2 = o2' ∧ st = Stack.Empty();

  in → sbj.update(s) out → o1.notify(s)
    ensures st = old(st).push(s);

  in ← o.notify() out → o2.notify(s)
    requires o = o1;
    ensures s = old(st).top();

  in ← o.notify() out ← sbj.update()
    requires o = o2;
    ensures st = old(st).pop();
}
```

What If?

```
interface Observer {
  void notify(State s);
}

class Subject {
  Observer o1, o2;

  Subject(Observer o1,
          Observer o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Assumption

- Different observers
- Non-null observers
- Deterministic
- Sequential

What If?

```
interface Observer {
  void notify(State s);
}

class Subject {
  Observer o1, o2;

  Subject(Observer o1,
          Observer o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Assumption

- Non-null observers
- Deterministic
- Sequential

What If?

```
interface Observer {
  void notify(State s);
}

class Subject {
  Observer o1, o2;

  Subject(Observer o1,
          Observer o2) {
    this.o1 = o1;
    this.o2 = o2;
  }

  void update(State s) {
    o1.notify(s);
    o2.notify(s);
  }
}
```

Assumption

- Non-null observers
- Deterministic
- Sequential

- trace-based: additional function as a counter
- state-based: add e.g. `stack<Boolean>`

Trace- vs. State-Based Comparison

- Some properties are easier to specify in each of them
  - trace-based: after an update, two notifications follow
  - state-based: defining what to do at $\leftarrow o.\texttt{update}()$

Trace- vs. State-Based Comparison

- Some properties are easier to specify in each of them
  - trace-based: after an update, two notifications follow
  - state-based: defining what to do at $\leftarrow o.\texttt{update}()$
- Representing the current state
  - state-based: states only contain what are necessary, but needs to define the state changes
  - trace-based: no need to care about representation, but extracting necessary info can be difficult