State-Based Object Models Are More Abstract Than Trace-Based Models

Towards a Unified Specification Framework

Ilham W. Kurnia, Arnd Poetzsch-Heffter, Yannick Welsch

FoVeOOS 2010
Paris, France

June 29, 2010

Background

- State-based specification techniques
  - Close to implementation
  - Examples: Eiffel, JML, Spec#

Background

- State-based specification techniques
  - Close to implementation
  - Examples: Eiffel, JML, Spec#
- Trace-based specification techniques
  - No need to think about states
  - Examples: Jass (trace assertions), Java Jr. (program equivalence)

This Talk is About ...

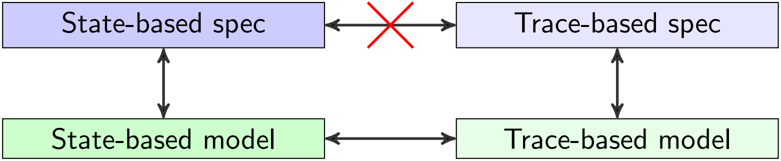
This talk is **not** about:

- Comparing specifications

This Talk is About ...

This talk is **not** about:

- Comparing specifications



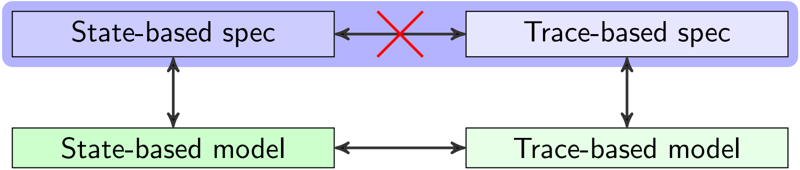
This talk is about:

- Comparing models via abstractions
- Applying simulation theory to this context

This Talk is About ...

This talk is **not** about:

- Comparing specifications, however ...



This talk is about:

- Comparing models via abstractions
- Applying simulation theory to this context

Our settings

- Object-based component
  A set of runtime objects.
- Sequential
  All method calls must fulfill call stack property
- Deterministic
  Given a state and a particular call, a component will always produce the same behavior.

Subject-Observer Pattern [GOF]

```
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);
  }
}
```

Subject-Observer Pattern [GOF]

```
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);
  }
}
```

Subject-Observer Pattern [GOF]

```
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);
  }
}
```

Subject-Observer Pattern [GOF]

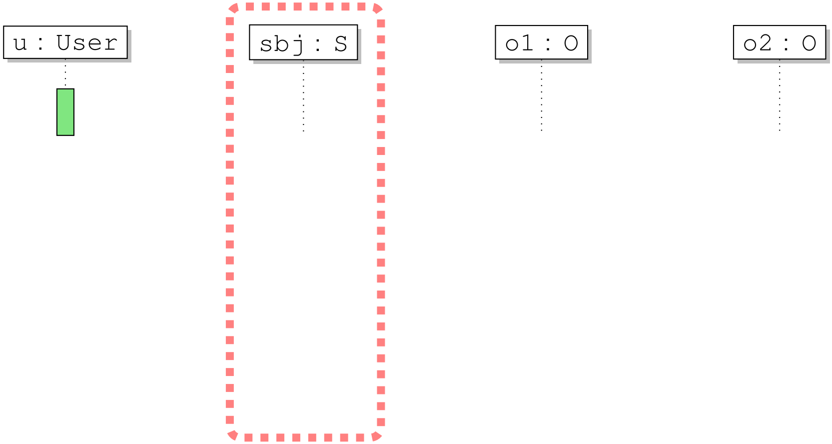
```
interface Observer {
  void notify(State s);
}

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);
  }
}
```

$o1 \neq o2 \neq$ **null**

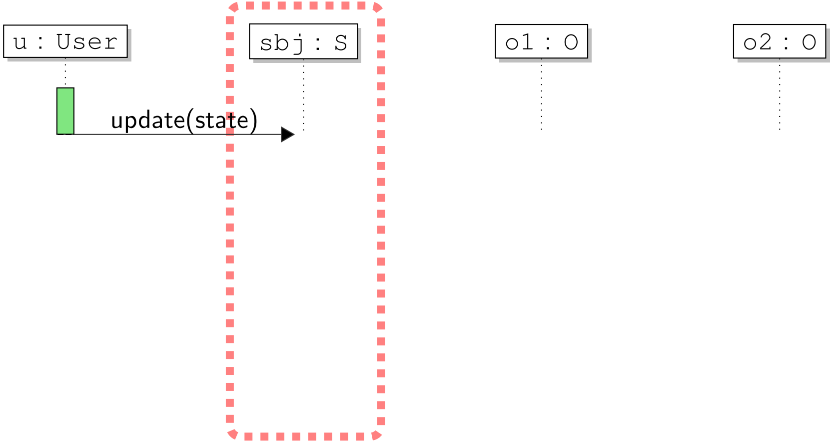
Subject-Observer Pattern [GOF]

```
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);
  }
}
```

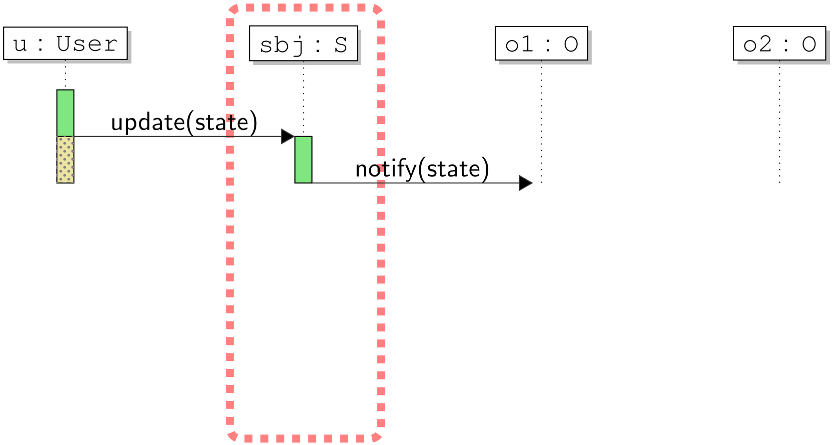
Subject Component Behavior



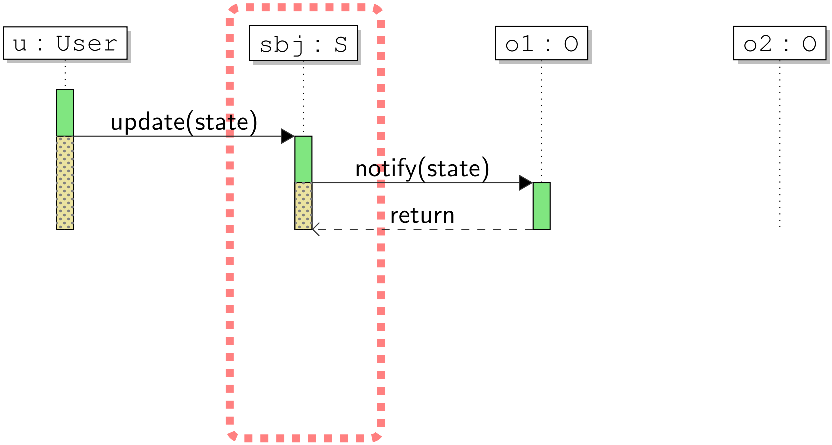
Subject Component Behavior



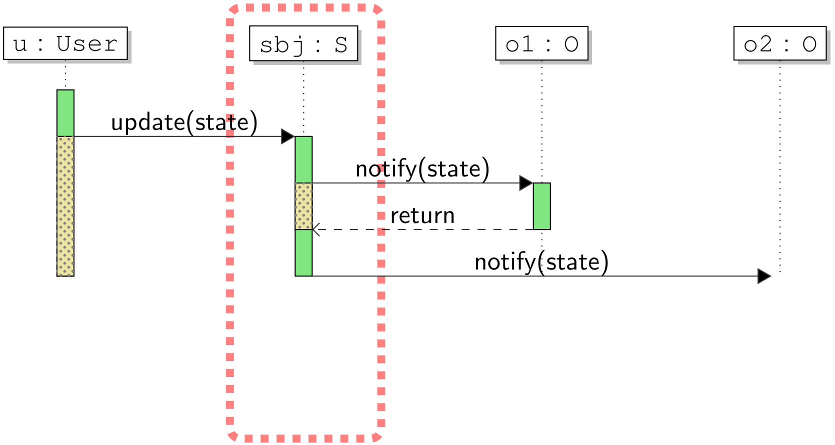
Subject Component Behavior



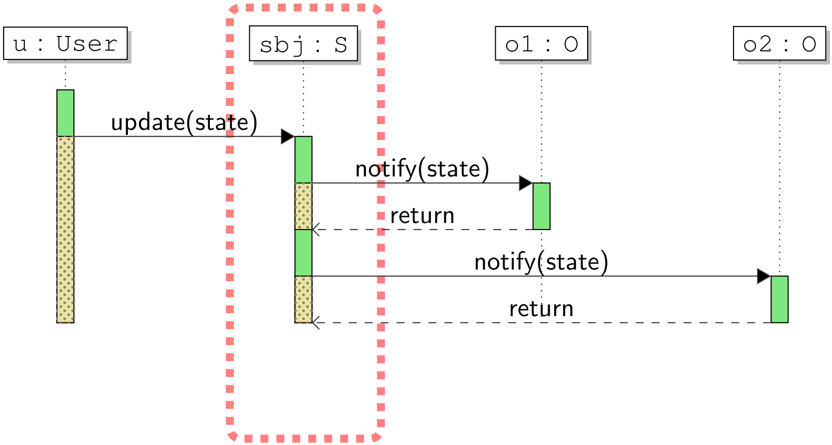
Subject Component Behavior



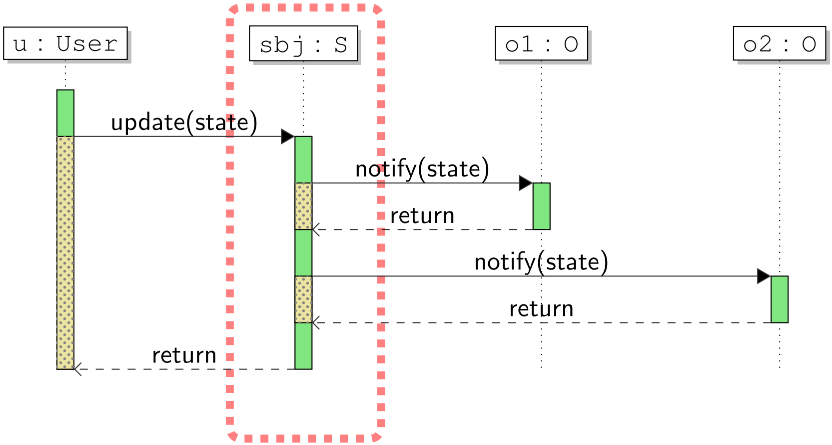
Subject Component Behavior



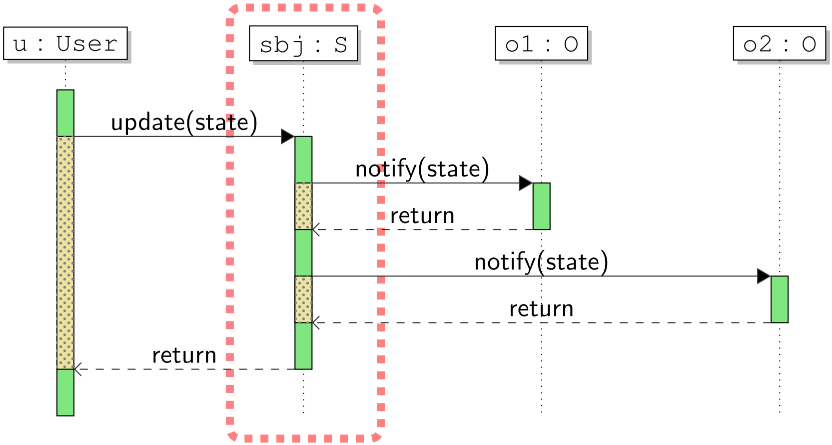
Subject Component Behavior



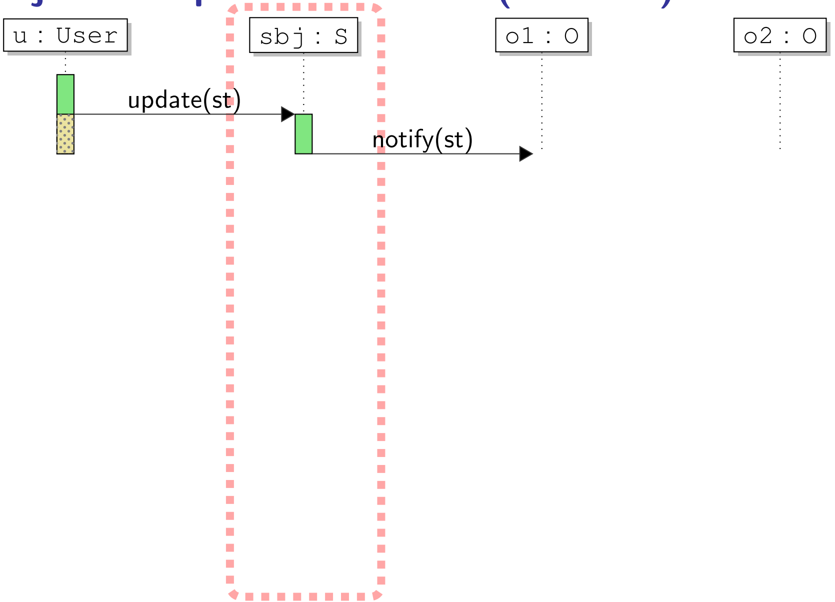
Subject Component Behavior



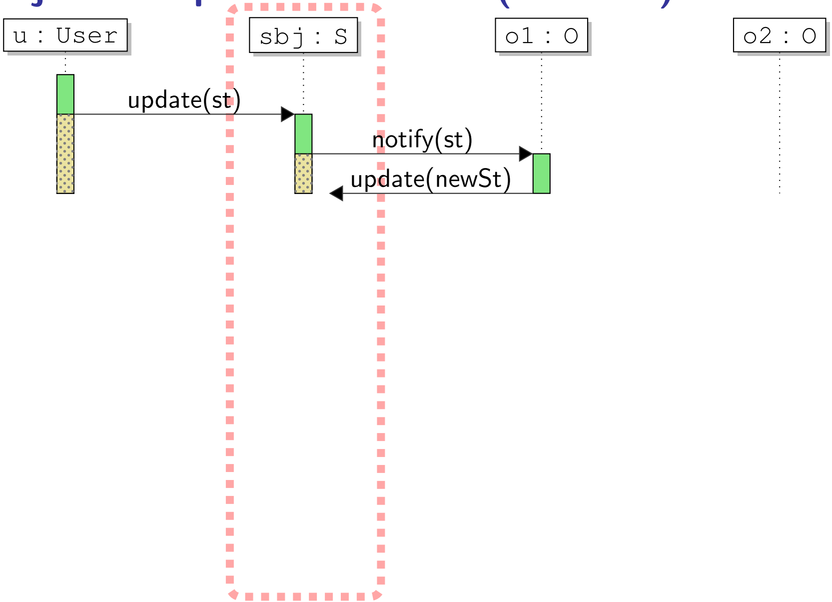
Subject Component Behavior



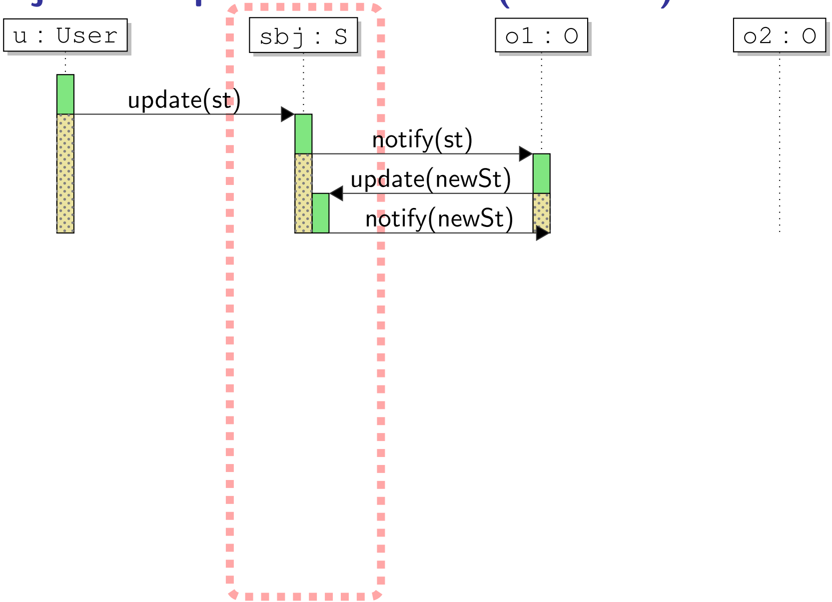
Subject Component Behavior (Callback)



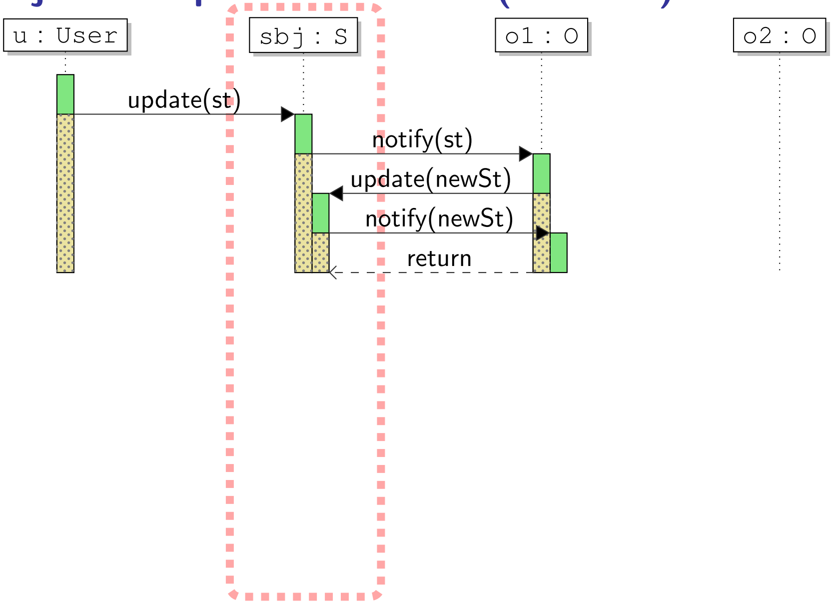
Subject Component Behavior (Callback)



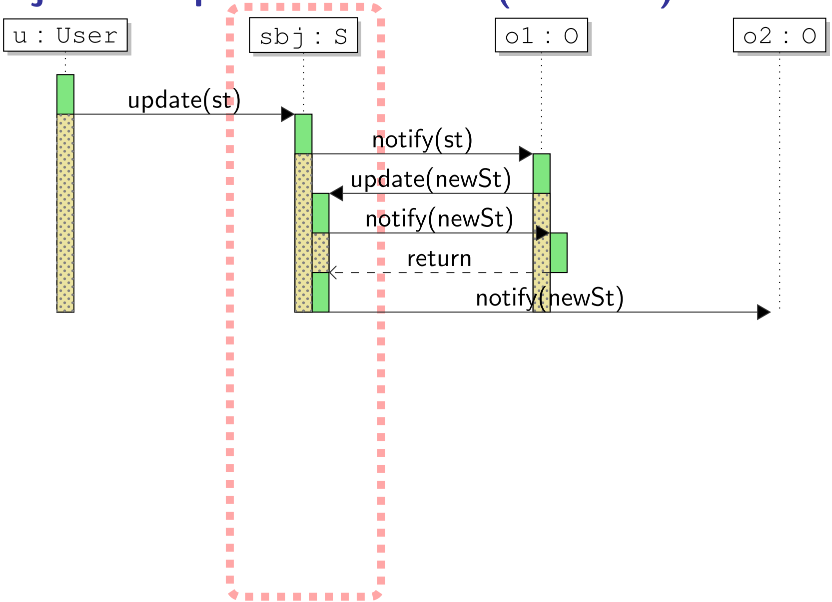
Subject Component Behavior (Callback)



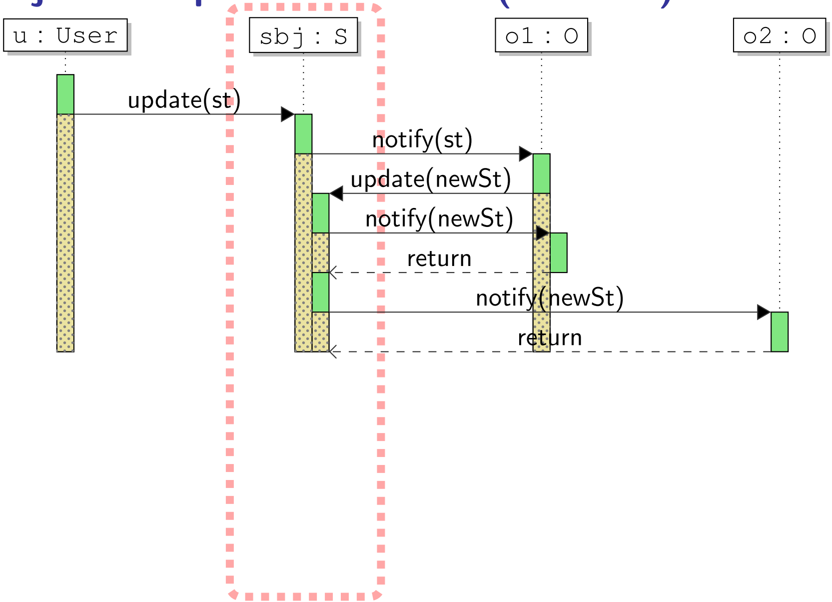
Subject Component Behavior (Callback)



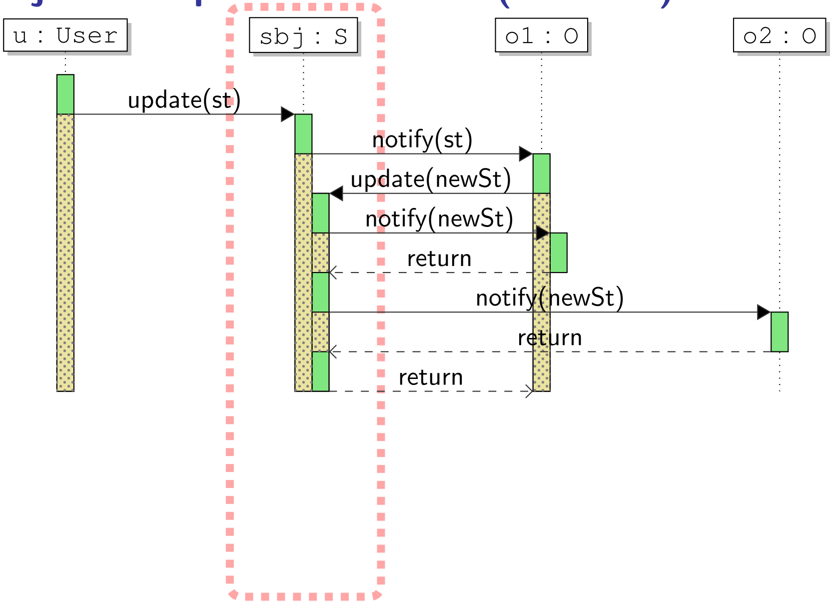
Subject Component Behavior (Callback)



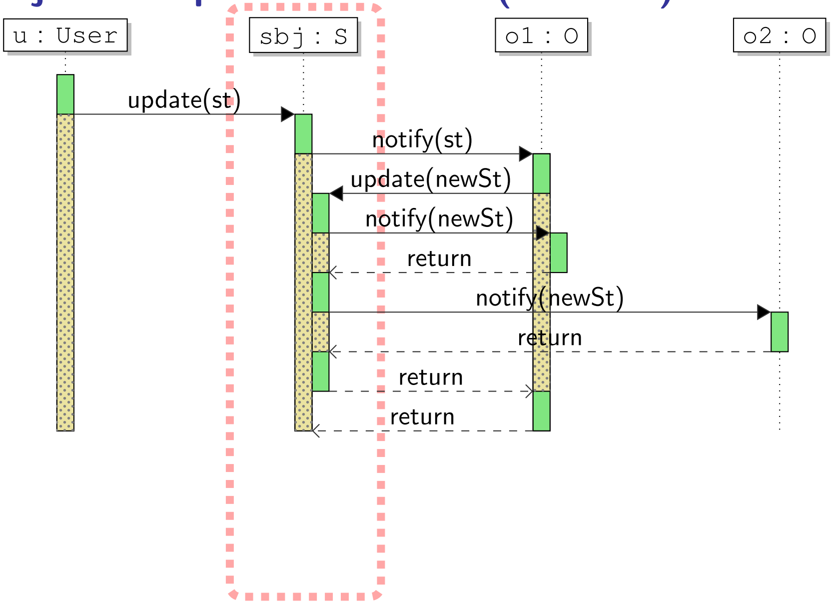
Subject Component Behavior (Callback)



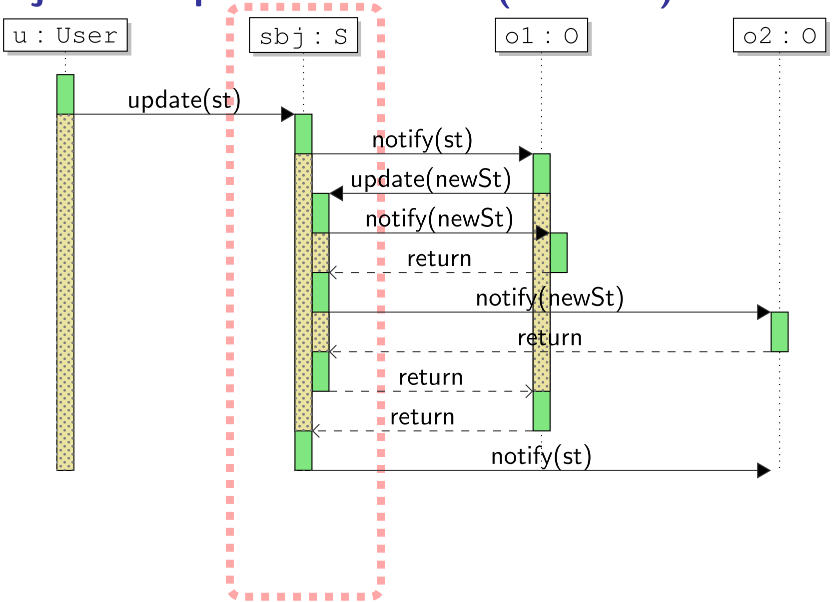
Subject Component Behavior (Callback)



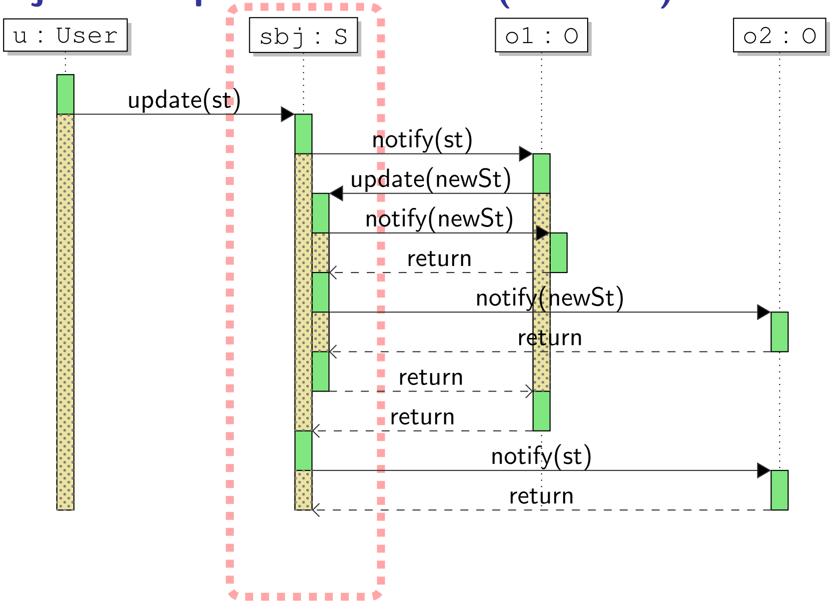
Subject Component Behavior (Callback)



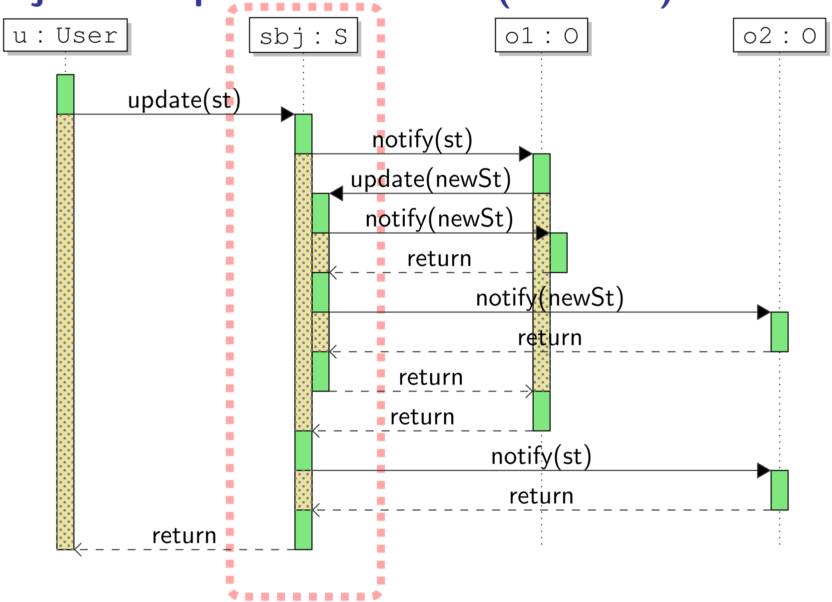
Subject Component Behavior (Callback)



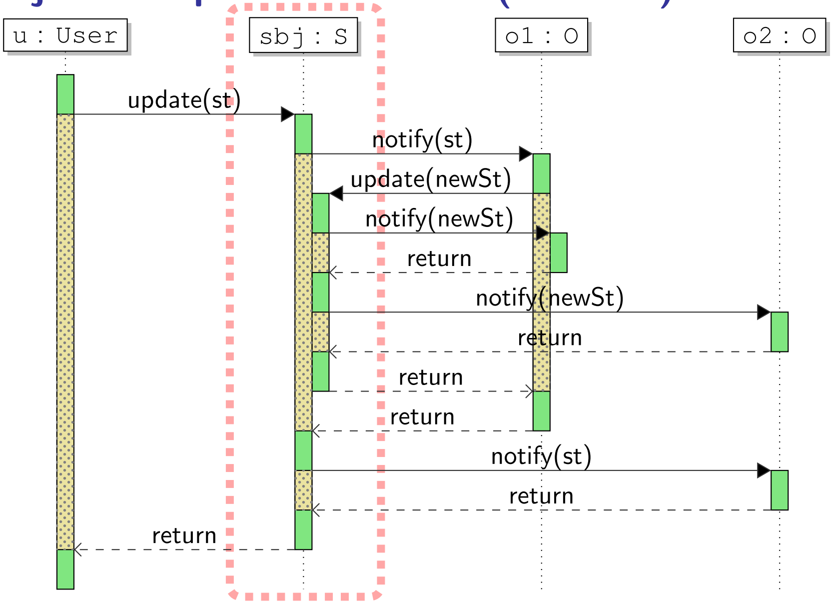
Subject Component Behavior (Callback)



Subject Component Behavior (Callback)



Subject Component Behavior (Callback)



Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

- A set of messages.

$$\begin{aligned} Msg = {} & \{\rightarrow sbj.\mathtt{Subject}(o_1, o_2), \leftarrow sbj.\mathtt{Subject}()\} \\ & \cup \{\rightarrow sbj.\mathtt{update}(s), \leftarrow sbj.\mathtt{update}()\} \\ & \cup \{\rightarrow o.\mathtt{notify}(s), \leftarrow o.\mathtt{notify}()\} \end{aligned}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

- A set of messages.

Direction

$$\begin{aligned} Msg = \{ & \rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\} \\ \cup \{ & \rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\} \\ \cup \{ & \rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\} \end{aligned}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

- A set of messages.

Callee

$$\begin{aligned} Msg = \{&\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\} \\ \cup \{&\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\} \\ \cup \{&\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\} \end{aligned}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

- A set of messages.

Method

$$Msg = \{\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\}$$
$$\cup \{\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\}$$
$$\cup \{\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

- A set of messages.

Parameters

$$\begin{aligned} Msg = &\{\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\} \\ &\cup \{\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\} \\ &\cup \{\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\} \end{aligned}$$

Model ingredients

- Object universe:

$$O = \{sbj, o_1, o_2, st_1, st_2, st_3, \ldots\}$$

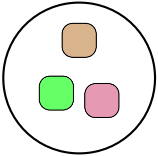
- A component = a set of objects (subset of O)

$$C = \{sbj\}$$

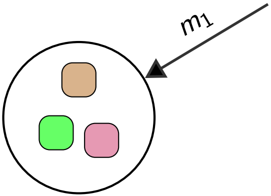
- A set of messages. Callee + Method = Header

$$Msg = \{\rightarrow sbj.\texttt{Subject}(o_1, o_2), \leftarrow sbj.\texttt{Subject}()\}$$
$$\cup \{\rightarrow sbj.\texttt{update}(s), \leftarrow sbj.\texttt{update}()\}$$
$$\cup \{\rightarrow o.\texttt{notify}(s), \leftarrow o.\texttt{notify}()\}$$

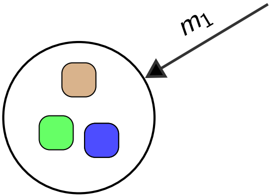
Model Representation



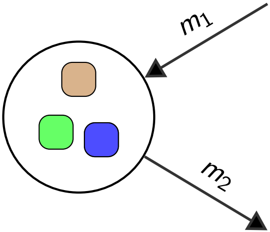
Model Representation



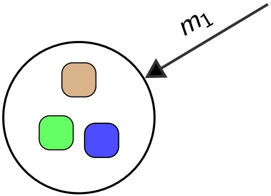
Model Representation



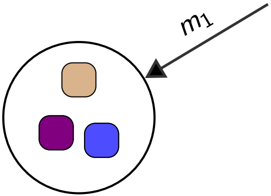
Model Representation



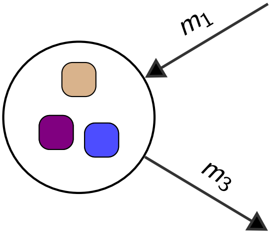
Model Representation



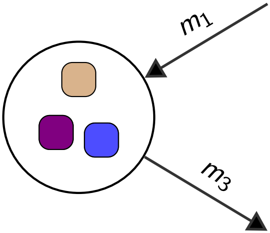
Model Representation



Model Representation



Model Representation

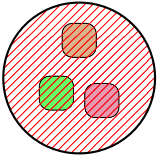


State-Based Models

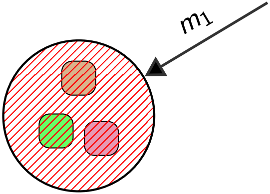
Transition system of (abstract) states

$$\mathcal{M}(Msg, O) = \langle S, \Theta, s_0 \rangle$$

Model Representation

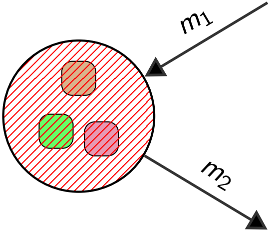


Model Representation



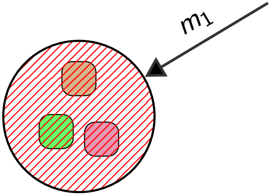
$t =$

Model Representation



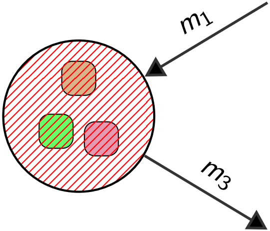
$$t = (m_1, m_2)$$

Model Representation



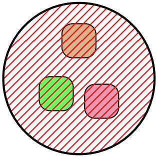
$$t = (m_1, m_2)$$

Model Representation



$$t = (m_1, m_2), (m_1, m_3)$$

Model Representation



$t = (m_1, m_2), (m_1, m_3)$

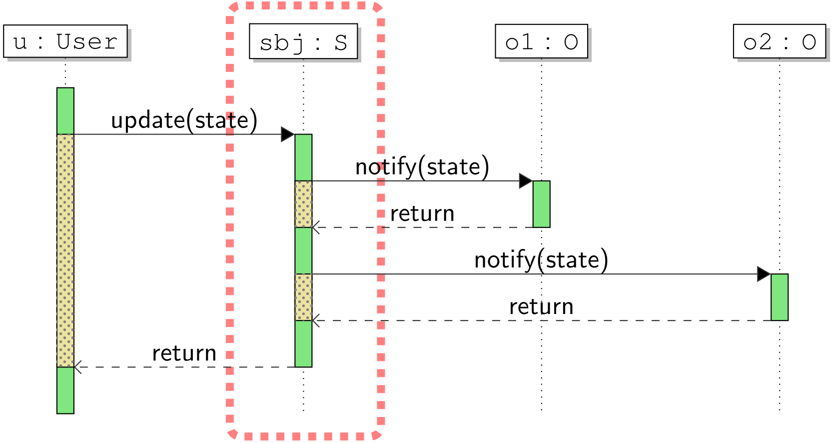
Trace-Based Models

A trace-based model = a set of traces

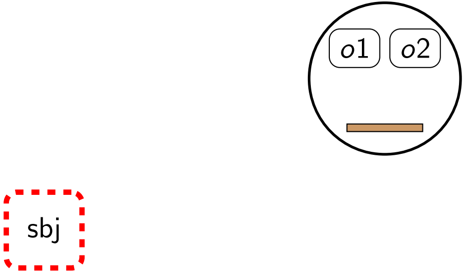
$$\mathcal{T}(Msg, O) = Traces(O)$$

A trace is a sequence of pairs of messages

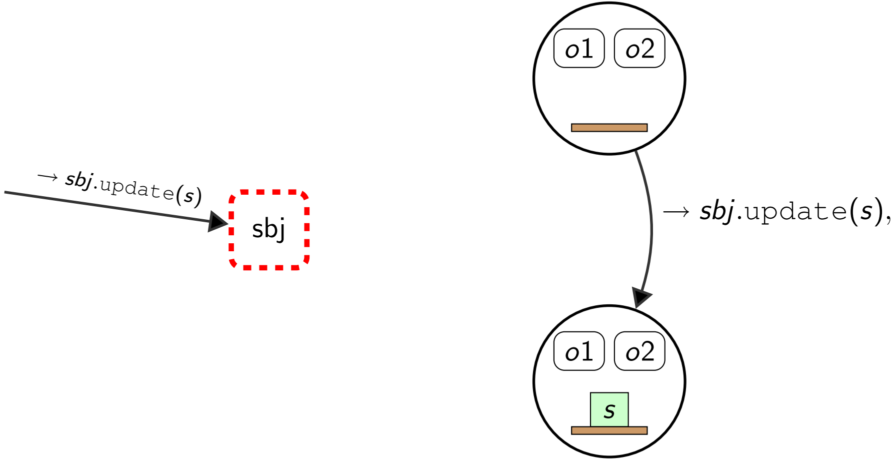
Subject Component Behavior



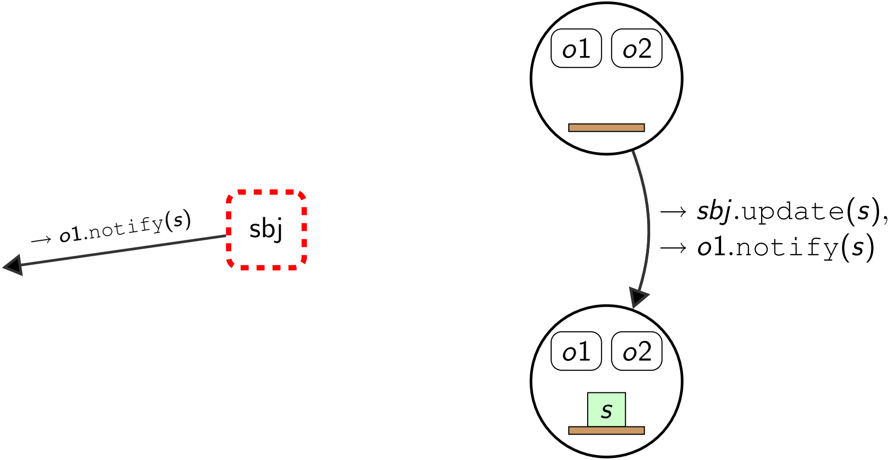
Subject Component State-Based Model



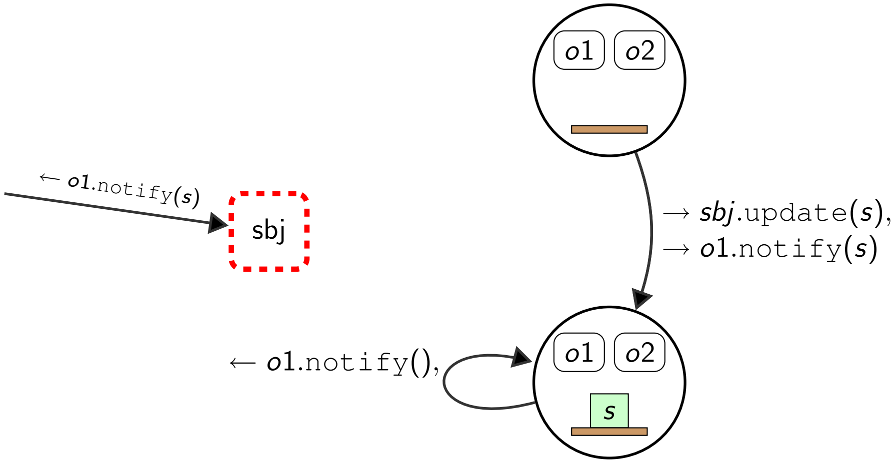
Subject Component State-Based Model



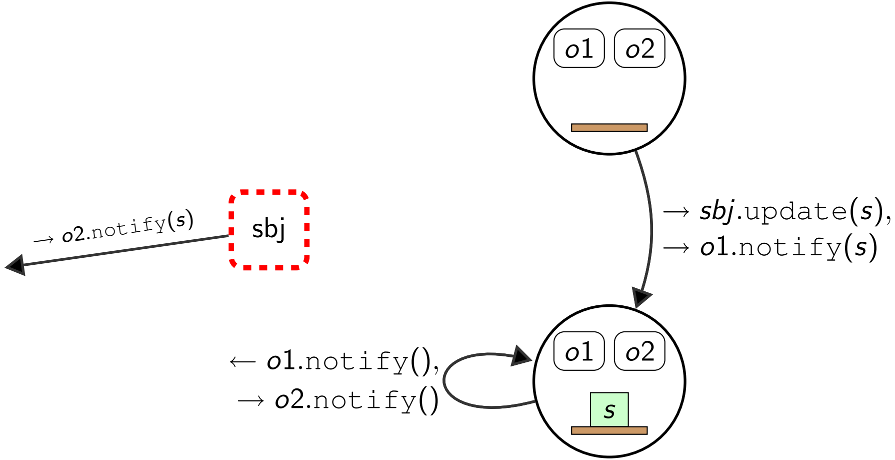
Subject Component State-Based Model



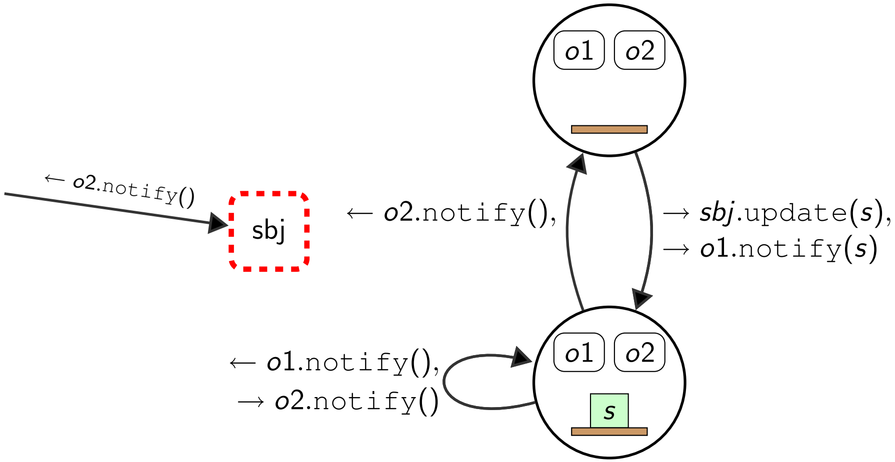
Subject Component State-Based Model



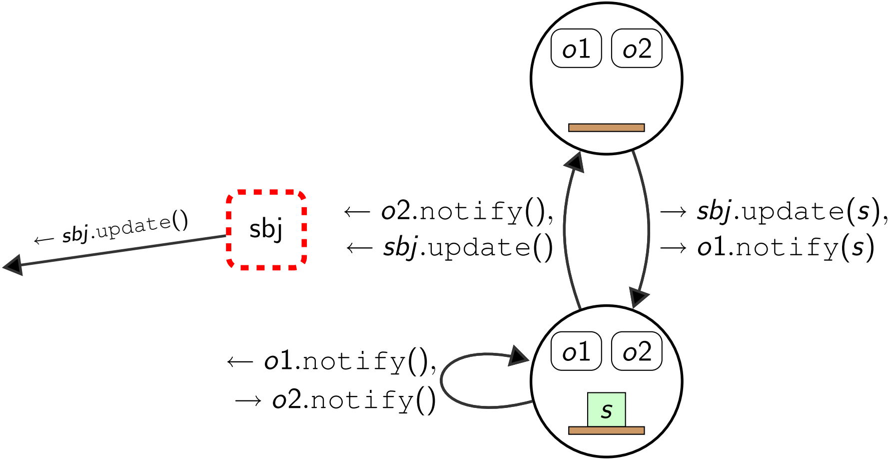
Subject Component State-Based Model



Subject Component State-Based Model



Subject Component State-Based Model



Subject Component Trace-Based Model

$$\mathcal{T} = \{[\ldots, (\rightarrow \mathit{sbj.update}(s), \rightarrow o1.\texttt{notify}(s))],$$

Subject Component Trace-Based Model

$$\begin{aligned}\mathcal{T} = \{&[\ldots, (\rightarrow \mathit{sbj}.\mathit{update}(s), \rightarrow o1.\texttt{notify}(s))],\\ &[\ldots, (\rightarrow \mathit{sbj}.\mathit{update}(s), \rightarrow o1.\texttt{notify}(s)),\\ &\quad(\leftarrow o1.\texttt{notify}(), \rightarrow o2.\texttt{notify}(s))],\end{aligned}$$

Subject Component Trace-Based Model

$$
\begin{aligned}
\mathcal{T} = \{&[\ldots, (\rightarrow \mathit{sbj}.\mathit{update}(s), \rightarrow o1.\texttt{notify}(s))],\\
&[\ldots, (\rightarrow \mathit{sbj}.\mathit{update}(s), \rightarrow o1.\texttt{notify}(s)),\\
&\quad (\leftarrow o1.\texttt{notify}(), \rightarrow o2.\texttt{notify}(s))],\\
&[\ldots, (\rightarrow \mathit{sbj}.\mathit{update}(s), \rightarrow o1.\texttt{notify}(s)),\\
&\quad (\leftarrow o1.\texttt{notify}(), \rightarrow o2.\texttt{notify}(s))],\\
&\quad (\leftarrow o2.\texttt{notify}(), \leftarrow \mathit{sbj}.\texttt{update}())],\\
&\ldots\}
\end{aligned}
$$

Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$
is disallowed.

Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$ is disallowed.

Trace-based models

Description: mutually recursive functions, *match* and *partition*.

$$t = \rightarrow a.m, \rightarrow b.n, \rightarrow c.o, \rightarrow d.p, \leftarrow d.p, \rightarrow e.q, \leftarrow e.q, \leftarrow c.o$$

Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$ is disallowed.

Trace-based models

Description: mutually recursive functions, *match* and *partition*.

$$t = \rightarrow a.m, \rightarrow b.n, \rightarrow c.o, \rightarrow d.p, \leftarrow d.p, \rightarrow e.q, \leftarrow e.q, \leftarrow \mathbf{c.o}$$

Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\mathtt{update}(s), \rightarrow o1.\mathtt{notify}(s), \leftarrow o2.\mathtt{notify}()$
is disallowed.

Trace-based models

Description: mutually recursive functions, *match* and *partition*.

$$t = \rightarrow a.m, \rightarrow b.n, \underbrace{\rightarrow \mathbf{c.o}, \rightarrow d.p, \leftarrow d.p, \rightarrow e.q, \leftarrow e.q, \leftarrow \mathbf{c.o}}$$

Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$ is disallowed.

Trace-based models

Description: mutually recursive functions, *match* and *partition*.

$$t = \rightarrow a.m, \rightarrow b.n, \underbrace{\rightarrow \mathbf{c.o}, \underbrace{\rightarrow d.p, \leftarrow d.p}, \underbrace{\rightarrow e.q, \leftarrow e.q}, \leftarrow \mathbf{c.o}}$$

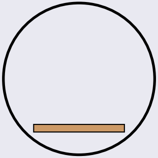
Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$ is disallowed.

State-based models

Create a call stack restrictor model.



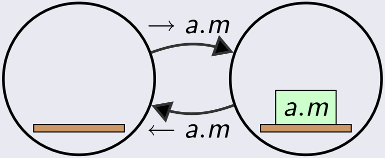
Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$
is disallowed.

State-based models

Create a call stack restrictor model.



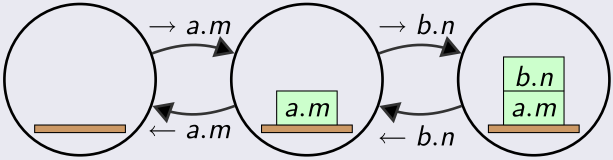
Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$
is disallowed.

State-based models

Create a call stack restrictor model.



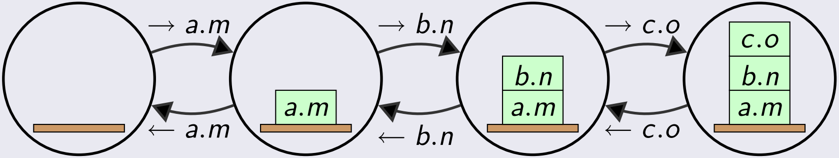
Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$
is disallowed.

State-based models

Create a call stack restrictor model.



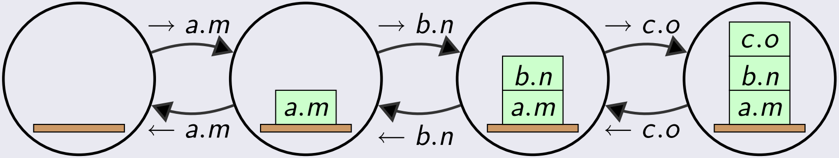
Well-Formedness Condition

All traces must follow the call stack property.

e.g. $t = \ldots, \rightarrow sbj.\texttt{update}(s), \rightarrow o1.\texttt{notify}(s), \leftarrow o2.\texttt{notify}()$
is disallowed.

State-based models

Create a call stack restrictor model.



- extend this to pairs of messages
- take the synchronous product

Trace-Based Models as State-Based Models

Lemma

Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.

Trace-Based Models as State-Based Models

Lemma

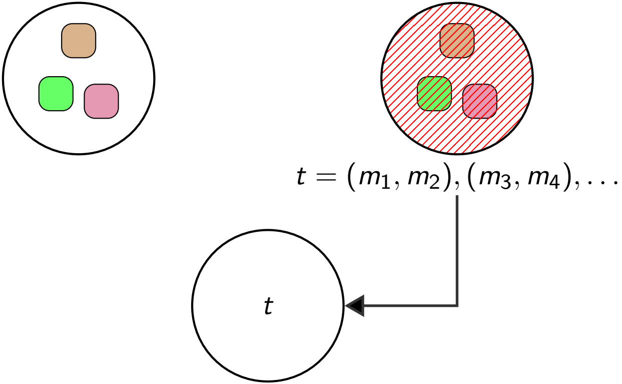
Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.

$$t = (m_1, m_2), (m_3, m_4), \ldots$$

Trace-Based Models as State-Based Models

Lemma

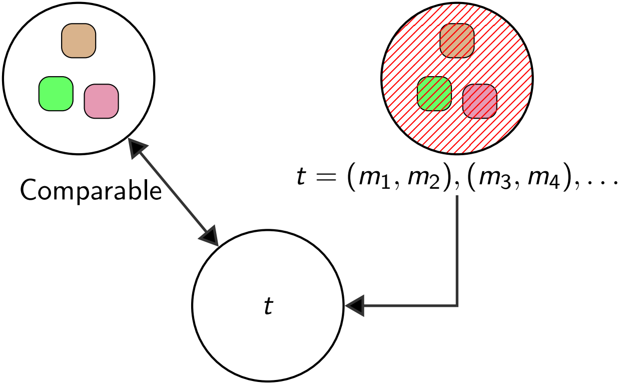
Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.



Trace-Based Models as State-Based Models

Lemma

Every trace-based model $\mathcal{T}$ can be canonically represented as a state-based model $\mathcal{M} = \langle S, \Theta, s_0 \rangle$.



Model Abstraction

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

$\mathcal{M}_2$ is *more abstract* than $\mathcal{M}_1$

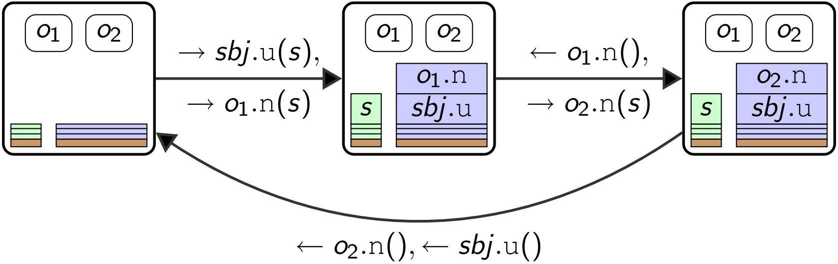
iff

there is $\alpha : S_1 \rightarrow S_2$ such that

- $s_{0,2} = \alpha(s_{0,1})$, and
- if $s_1 \xrightarrow{m_a, m_b}_1 s_1'$, then $\alpha(s_1) \xrightarrow{m_a, m_b}_2 \alpha(s_1')$.

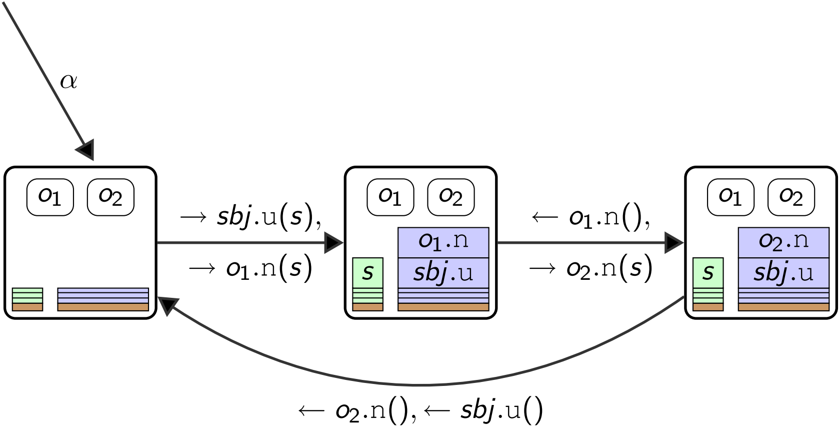
Abstraction Example

$$t = t', \rightarrow sbj.\mathtt{u}(s), \rightarrow o_1.\mathtt{n}(s), \leftarrow o_1.\mathtt{n}(), \rightarrow o_2.\mathtt{n}(s), \leftarrow o_2.\mathtt{n}(), \leftarrow sbj.\mathtt{u}()$$



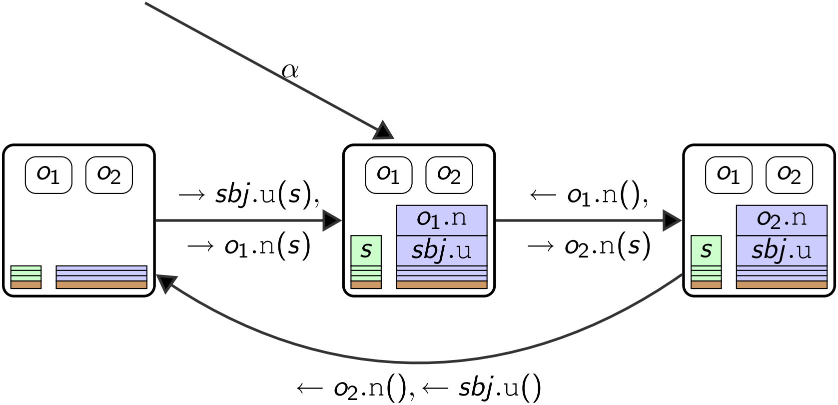
Abstraction Example

$$t = t', \rightarrow sbj.\mathrm{u}(s), \rightarrow o_1.\mathrm{n}(s), \leftarrow o_1.\mathrm{n}(), \rightarrow o_2.\mathrm{n}(s), \leftarrow o_2.\mathrm{n}(), \leftarrow sbj.\mathrm{u}()$$



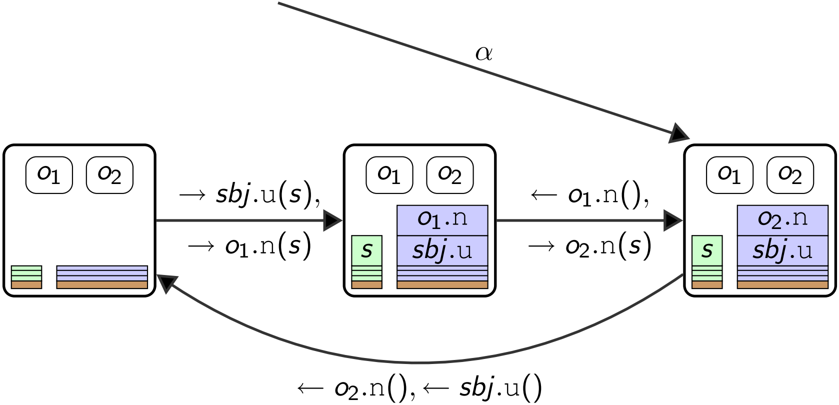
Abstraction Example

$$t = \underbrace{t', \rightarrow sbj.\mathtt{u}(s), \rightarrow o_1.\mathtt{n}(s)}, \leftarrow o_1.\mathtt{n}(), \rightarrow o_2.\mathtt{n}(s), \leftarrow o_2.\mathtt{n}(), \leftarrow sbj.\mathtt{u}()$$



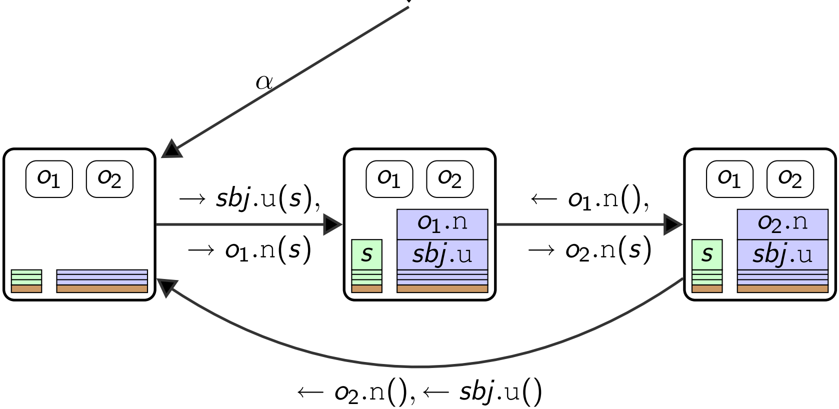
Abstraction Example

$$t = \underbrace{t', \rightarrow sbj.\mathtt{u}(s), \rightarrow o_1.\mathtt{n}(s), \leftarrow o_1.\mathtt{n}(), \rightarrow o_2.\mathtt{n}(s)}, \leftarrow o_2.\mathtt{n}(), \leftarrow sbj.\mathtt{u}()$$



Abstraction Example

$$t = \underbrace{t', \rightarrow sbj.\mathrm{u}(s), \rightarrow o_1.\mathrm{n}(s), \leftarrow o_1.\mathrm{n}(), \rightarrow o_2.\mathrm{n}(s), \leftarrow o_2.\mathrm{n}(), \leftarrow sbj.\mathrm{u}()}$$



Model Comparison

Theorem

For any trace-based model $\mathcal{T}$, there is state-based model $\mathcal{M}$ which is more abstract than $\mathcal{T}$. The converse does not hold.

Intuition:

$\Rightarrow$ Use previous lemma and take α as the identity function.

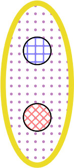
$\Leftarrow$ Counter example: subject component.

Main cause: sequential setting

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

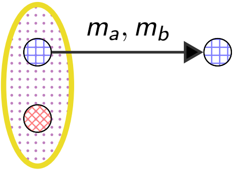
1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2.  $\in \mathcal{R}$

If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

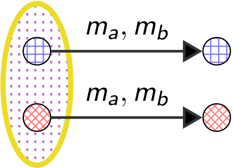
1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2. 

If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

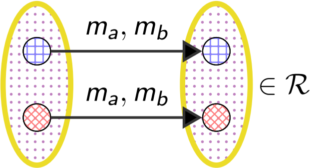
1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2. 

If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

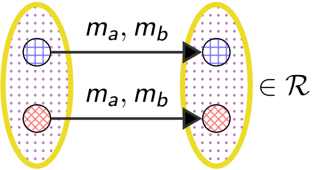
1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2. 

If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2. 

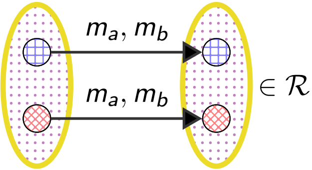
If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Abstraction function is an instance of simulation relation.

Simulation [Milner 1971]

Given: $\mathcal{M}_1 = \langle S_1, \Theta_1, s_{0,1} \rangle$ and $\mathcal{M}_2 = \langle S_2, \Theta_2, s_{0,2} \rangle$

A *simulation* for $(\mathcal{M}_1, \mathcal{M}_2)$ is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

1. $(s_{0,1}, s_{0,2}) \in \mathcal{R}$,
2. 

If such $\mathcal{R}$ exists for $(\mathcal{M}_1, \mathcal{M}_2)$, $\mathcal{M}_1 \preceq \mathcal{M}_2$.

Abstraction function is an instance of simulation relation.

Simulation retains the same behavior in our settings [KučeraMayr02]

Most Abstract Model [Grumberg and Bustan 2003]

Simulation equivalence: $\mathcal{M}_1 \simeq \mathcal{M}_2 \equiv \mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$

Purpose:

- build equivalence classes of states of $\mathcal{M}$
- build quotient models

Most Abstract Model [Grumberg and Bustan 2003]

Simulation equivalence: $\mathcal{M}_1 \simeq \mathcal{M}_2 \equiv \mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$

Purpose:

- build equivalence classes of states of $\mathcal{M}$
- build quotient models

Theorem

Let $\mathcal{M}/\simeq$ be the quotient of $\mathcal{M}$ under $\simeq$, then $\mathcal{M}/\simeq$ is the most abstract model.

Conclusion and Future work

Conclusion

- State-based object models are more abstract than trace-based models
- Most abstract models can be built using simulation quotient.

Conclusion and Future work

Conclusion

- State-based object models are more abstract than trace-based models
- Most abstract models can be built using simulation quotient.

Future Work

- Generalize settings (trace restriction as precondition, full OO, nondeterminism, concurrency).
- Compare specifications and build common framework.

Generalized State-Based Subject Specification

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

  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();
}
```