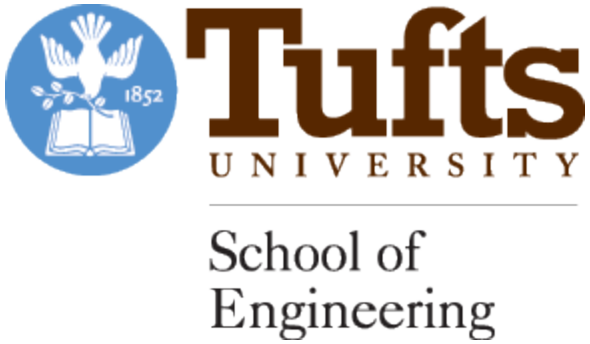
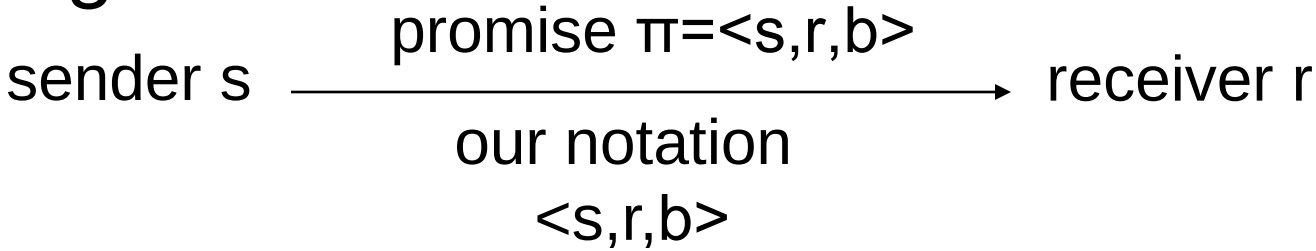


Modeling change without breaking promises

Alva Couch
Hengky Susanto
Marc Chiarini
Tufts University

Promises

- A promise is a one-sided agreement from the sender to conform to some limits upon the sender's behavior.
- Sender agrees to some behavior b (called a **promise body**)
- Receiver simply observes and is not obligated.



Conditional promises

- A **conditional promise** constrains the sender's behavior only under certain conditions.
- In our calculus of conditions, **only other promises can be conditions**.
- The notation **$\langle s_2, r_2, b_2 \rangle \mid \langle s_1, r_1, b_1 \rangle$** means that s_2 promises b_2 to r_2 **only if it observes** that s_1 has promised b_1 to r_1.
- Subtle: the above is really **one promise** with a special body: **$\langle s_2, r_2, (b_2 \mid \langle s_1, r_1, b_1 \rangle) \rangle$**

One problem with promises...

- ... is that they aren’t valid “forever”.
- If conditions change, an agent must “break promises”.
- A **broken promise** occurs when an agent promises something **contradictory** to a prior promise it has made.
- Note that a promise may also be **unfulfilled**; this is different from breaking a promise.

Semantics of broken promises

- The “contradiction” that signals that a promise is broken can be complex.
- A promise body can be thought of as a set of **prolog-style facts**.
- A **broken promise** is one in which the facts are **logically inconsistent** with those of some prior promise.

Example of a broken promise

- **fileservice(100ms)** – I promise to give you file service with an average response time of 100ms.
- **fileservice(70ms)** – better, not a broken promise.
- **fileservice(200ms)** – worse, and **breaks both other promises.**
- Semantics of broken promises are **complex** and depend upon semantics of promise bodies!

How not to break promises

- **Scope promises** in time and by events.
- Avoid having to infer contradictions to invalidate promises.
- Really, this is part of the **type system** of promise bodies.
- But we can separate this scoping from the type system via a simple notation.

Operative and inoperative promises

- A promise is **operative** (at a particular time) if it holds at that time, and **inoperative** otherwise.
  1. Unconditional promises are operative until they are broken.
  2. Conditional promises are operative if their conditions are operative.

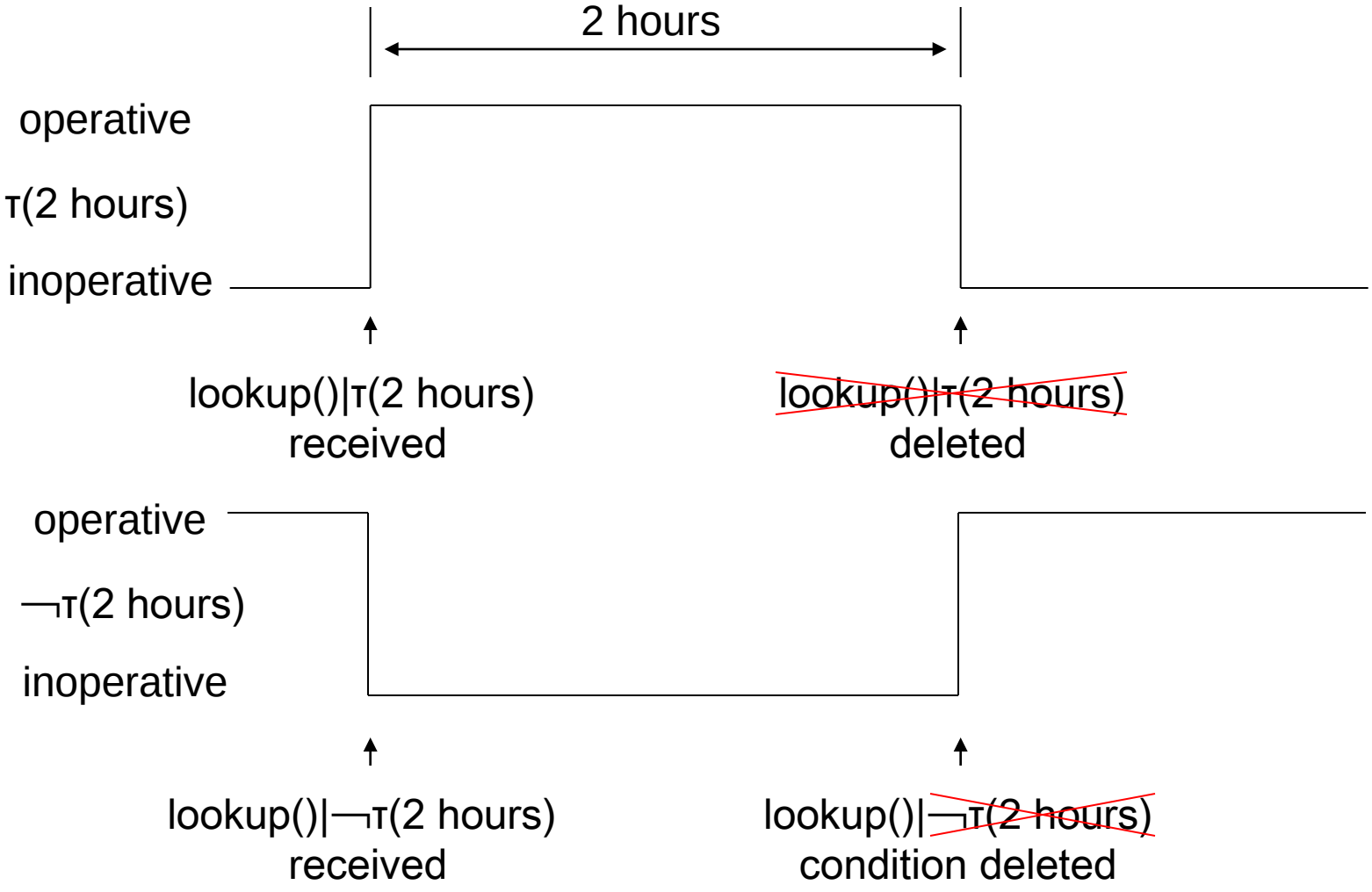
α and τ

- Two new promise bodies:
  - **τ(increment)** is operative from current time to current time + increment
  - **α(promise)** is operative until receipt of the specified promise.
- And one new operator:
  - **¬(p)** is operative whenever p is not operative.

Implicit sender and receiver

- **<s,r,(b|τ(1 second))>**
  means b is operative for one second only.
- We can “factor” τ out of the promise body:
  <s,r,b>|<s,r,τ(1 second)>
- But only s,r make sense as sender and receiver of τ. Thus we can write:
  <s,r,b>|τ(1 second)
  without confusion

Timing diagrams



Leasing and gating

- τ is operative for a given amount of time.
  - So τ can be used to simulate leasing.
- α is operative until a given promise is received.
  - So α can be used to simulate gating, in which receipt of one promise activates or deactivates another.

Leasing

- **<s,r,dhcp(192.138.177.3)> | τ(2 hours)**

  a DHCP lease grants use of an IP address **for two hours**.

- **<s,r,fileservice()>|—¬τ(1 hour), τ(3 hours)**

  s offers r fileservice **one hour from now**, **for two hours.**

- (a list of conditions is a **conjunction**)

Gating

- **<s,r,fileservice()> | α(<r,s,stop()>)**

  offer fileservice until told to stop offering it.

- **<r,s,stop()>|τ(0)**

  stop offering file service any more.

  (**τ(0)** becomes operative and then non-operative **at the same time step** and “gates” the transition.)

  (**stop()** is an **abstract promise** whose meaning is just to gate another one)

Type factoring

Consider the promise system

- <s,r,dhcp(192.138.178.1)> | τ(2 hours)
- <r,s,dns()> | α(<s,r,dns()>)

At any time, this system can be reduced to an equivalent one free of α and τ.

The reduction differs, depending upon time and events.

Before 2 hours are up and <s,r,dns()> not received

Reduced system:

- <s,r,dhcp(192.138.178.1)> | τ(2 hours)
- <r,s,dns()> | α(<s,r,dns()>)

After 2 hours are up and <s,r,dns()> not received

Reduced system:

- ~~<s,r,dhcp(192.138.178.1)> | τ(2 hours)~~
- <r,s,dns()> | ~~α(<s,r,dns()>)~~

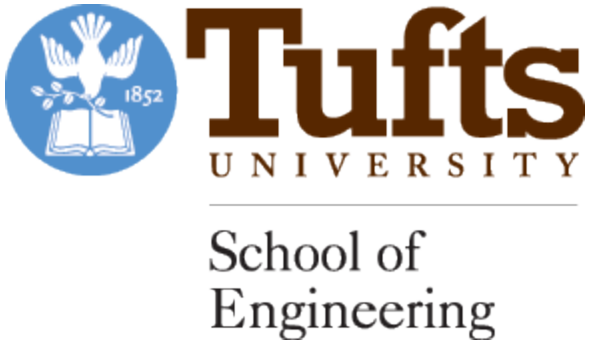
After 2 hours are up and after <s,r,dns()> received

Reduced system:

- ~~<s,r,dhcp(192.138.178.1)> | τ(2 hours)~~
- ~~<r,s,dns()> | α(<s,r,dns()>)~~

Claims

- α and τ are the minimal necessary operators for accomplishing change in promise networks without breaking promises. They are:
  - **self-erasing** when purpose is complete
  - **scalable** to use in complex tasks
  - **flexible**; any sequence of promise states can be managed in the promise space of the recipient.
  - **external** to the type system of promise bodies.



Modeling change without breaking promises

Alva Couch
Hengky Susanto
Marc Chiarini
Tufts University