

Formal verification of Dynamic Software Updating in CafeOBJ

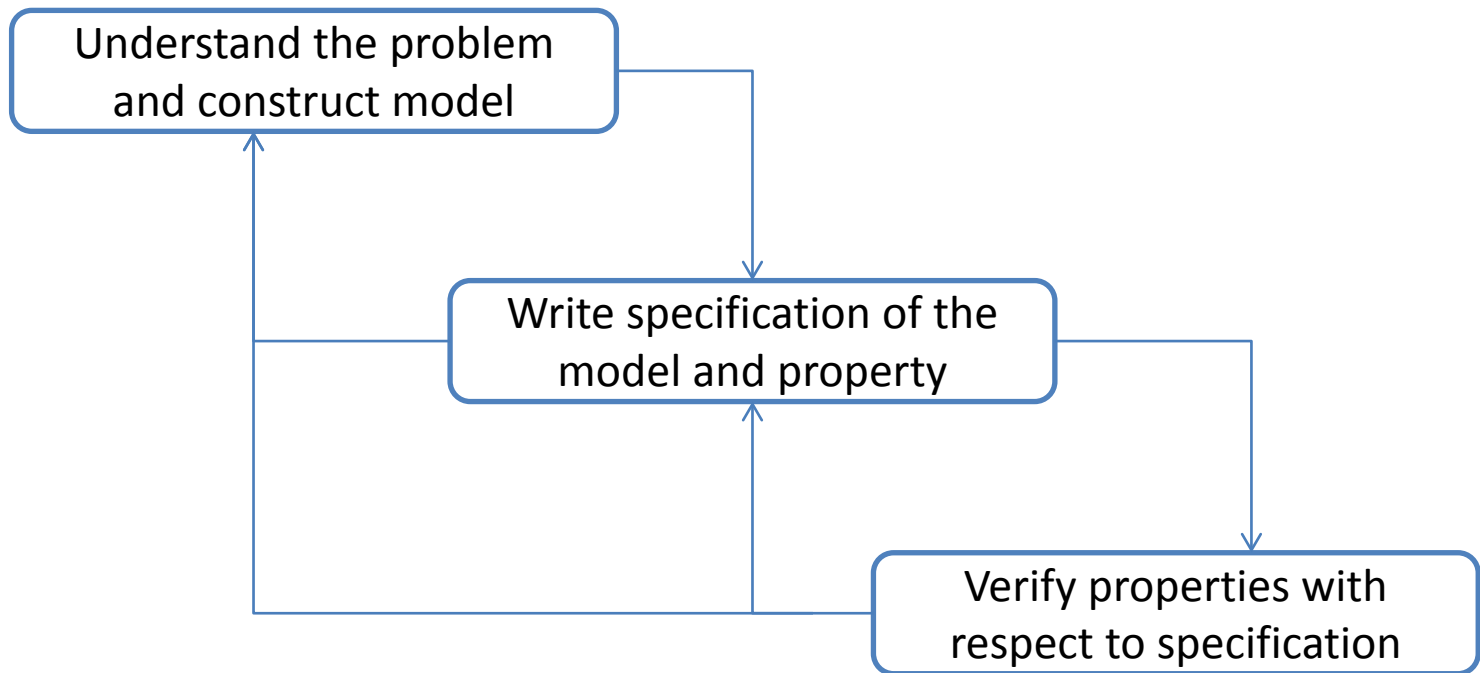
A case study on RailCab System

Zhang Min

Lecture Note 13
Formal Method (i613-1312)

Review I

- Methodology



Review II

- OTS-based system specification and verification
 - State is formalized as a set of observers
 - State transitions are declared by equations
 - Verification of invariant properties by CITP
- Trans-based system specification and verification
 - State is formalized as a multiset of observed values
 - State transitions are defined by CafeOBJ transitions
 - Verification by searching or model checking

Today's lecture

1. A crossing mechanism used in RailCab System
2. How to apply the modeling and verification method to the crossing mechanism
 - a. Modeling by OTS
 - b. Verifying the crossing property in CIP
3. Formal verification of dynamic software updating by model checking

Dynamic software updating

- A technique for updating a software when it is running without incurring downtime.
(update a running system without stopping it)
- It is useful to Systems that provide non-stoppable services
 - Web servers
 - bank system
 - traffic system

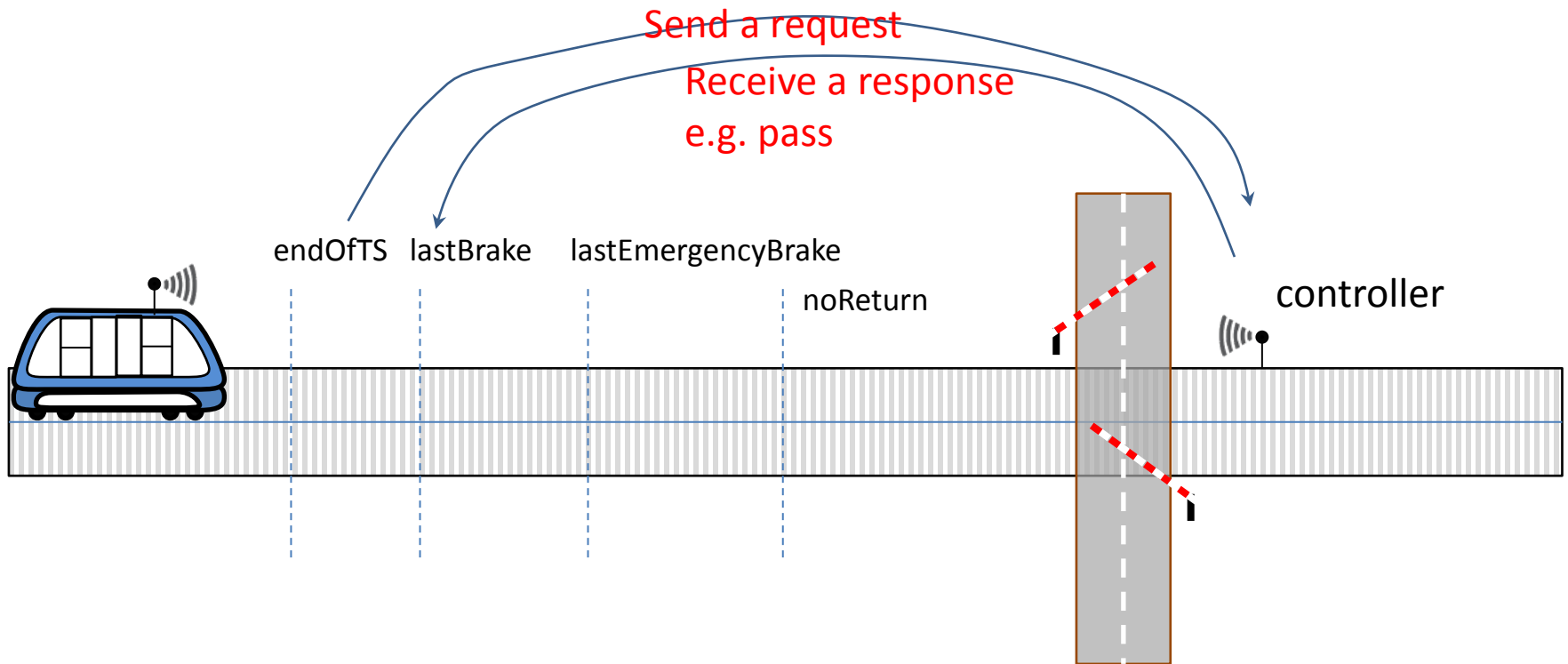
The RailCab System

- Conceptual transportation system
- A research project since 1997
- University of Paderborn in Germany
- Features:
 - Driverless
 - Work on demand
 - Intelligent
 - Contact-free



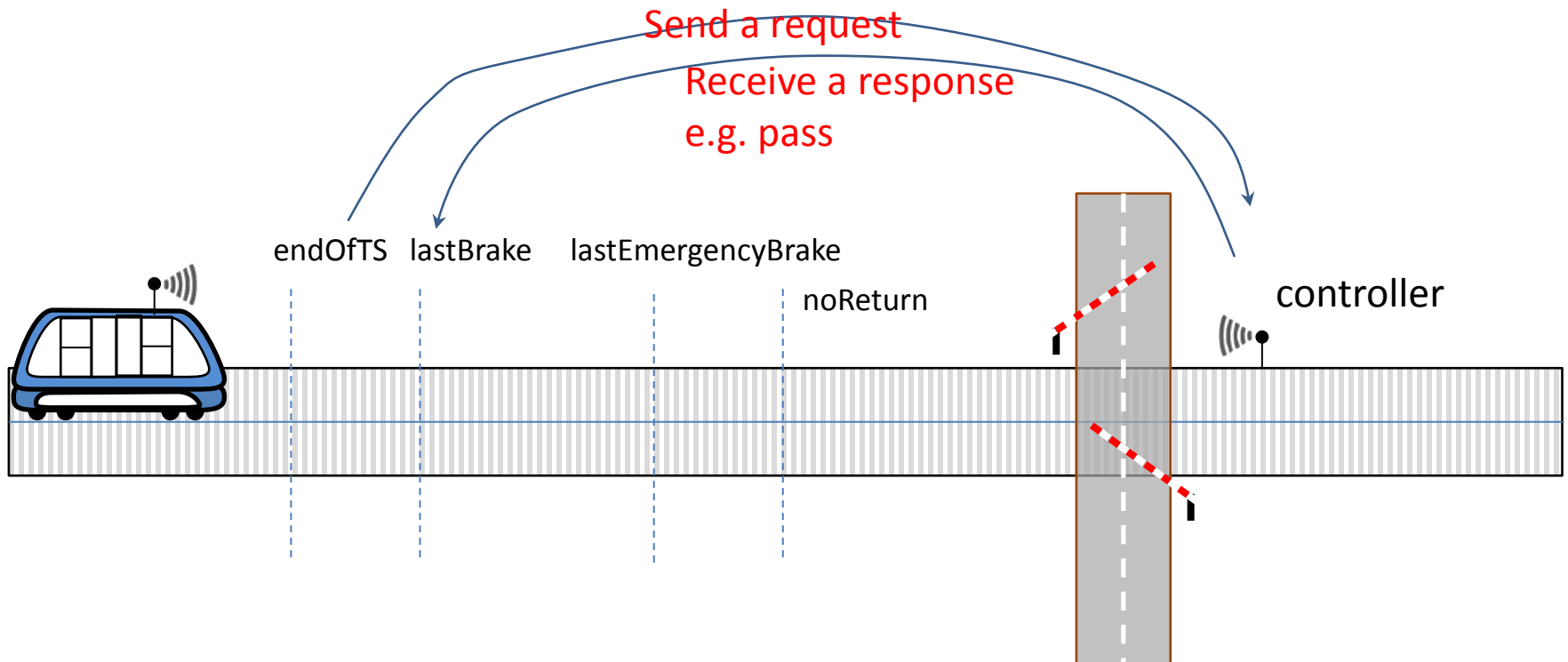
Crossing mechanism of RailCab

Basic mechanism



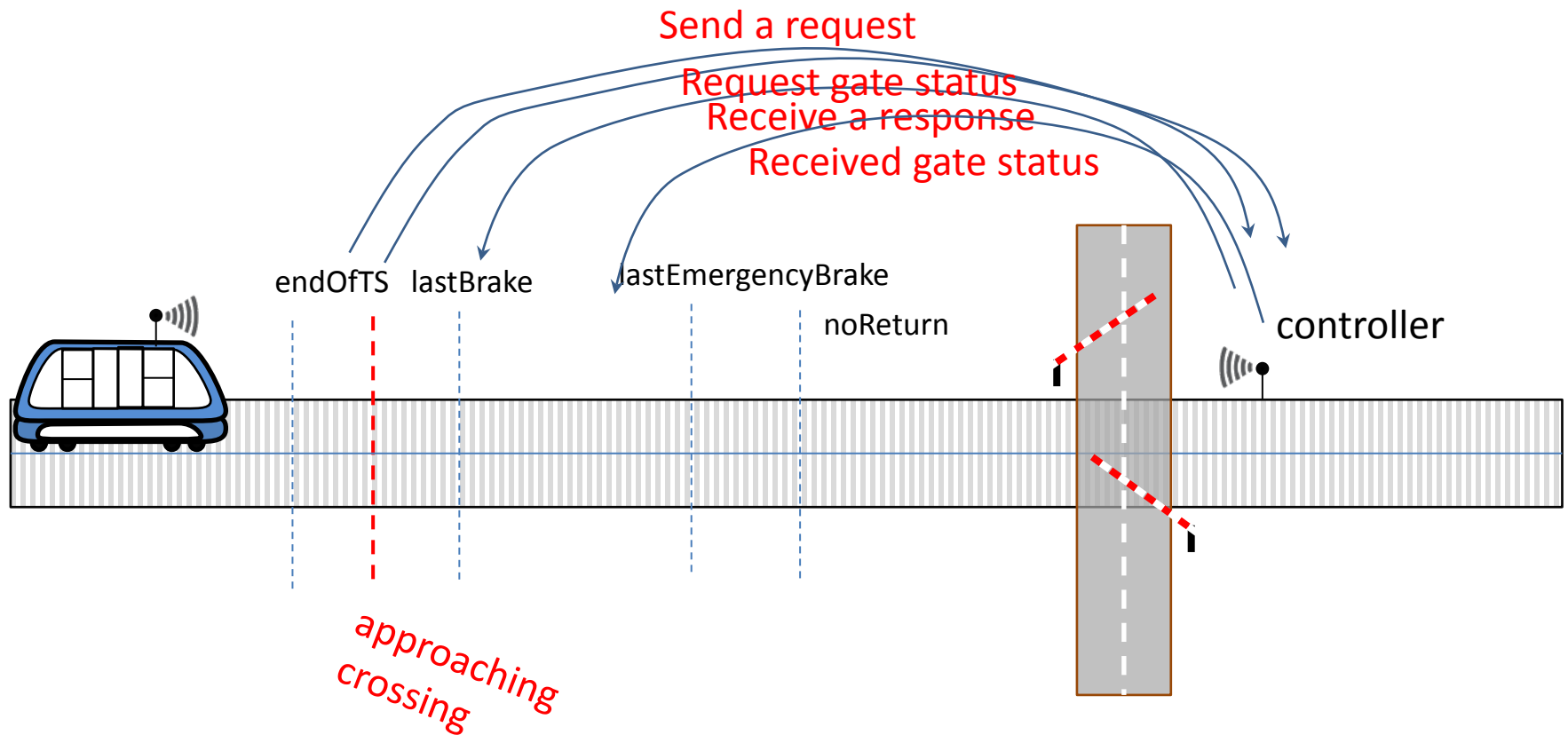
Crossing mechanism of RailCab (old)

Cross when gate is still open (**not safe**)



Crossing mechanism of RailCab (new)

New mechanism (two-time communication)

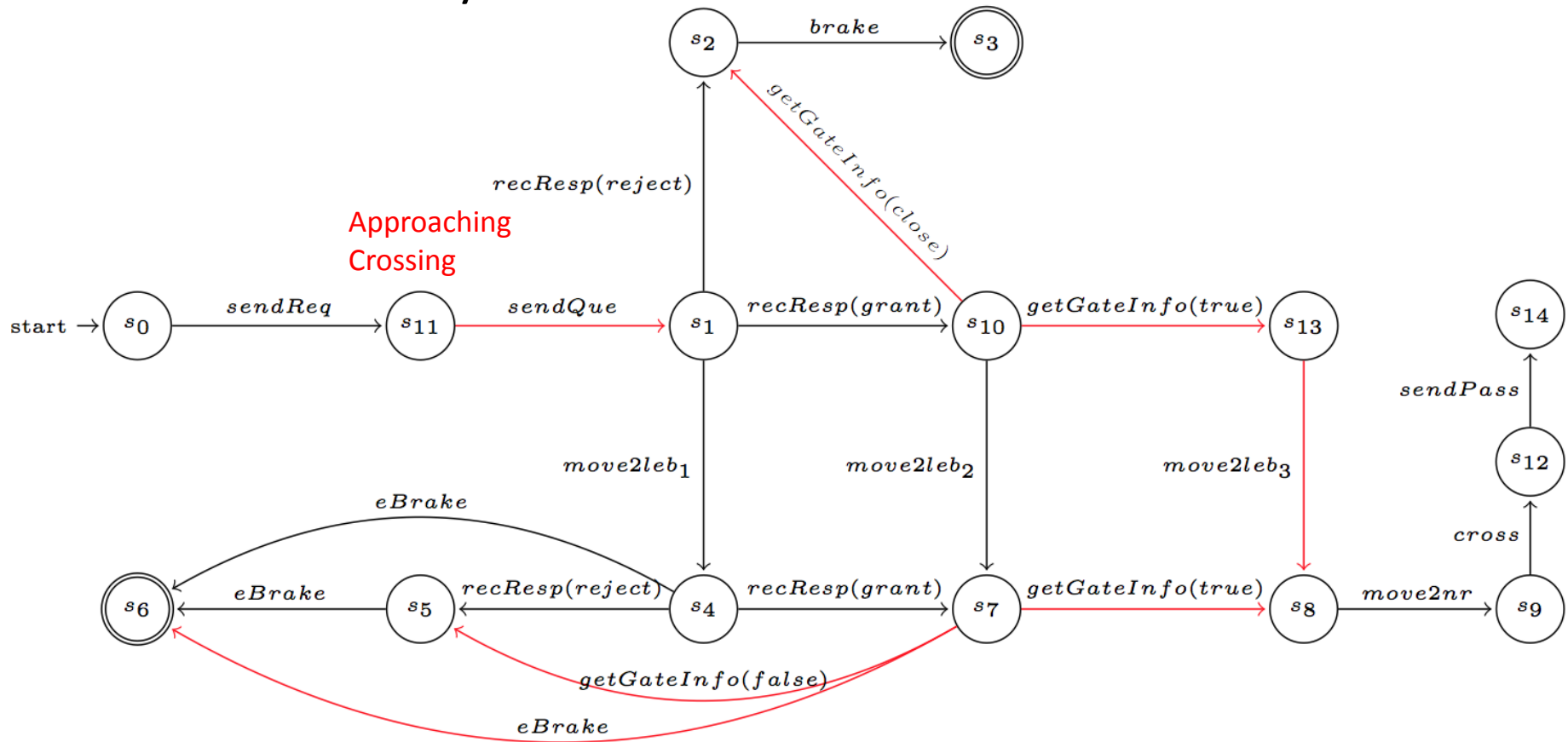


Verifying the crossing property in COTP

We want to verify that:

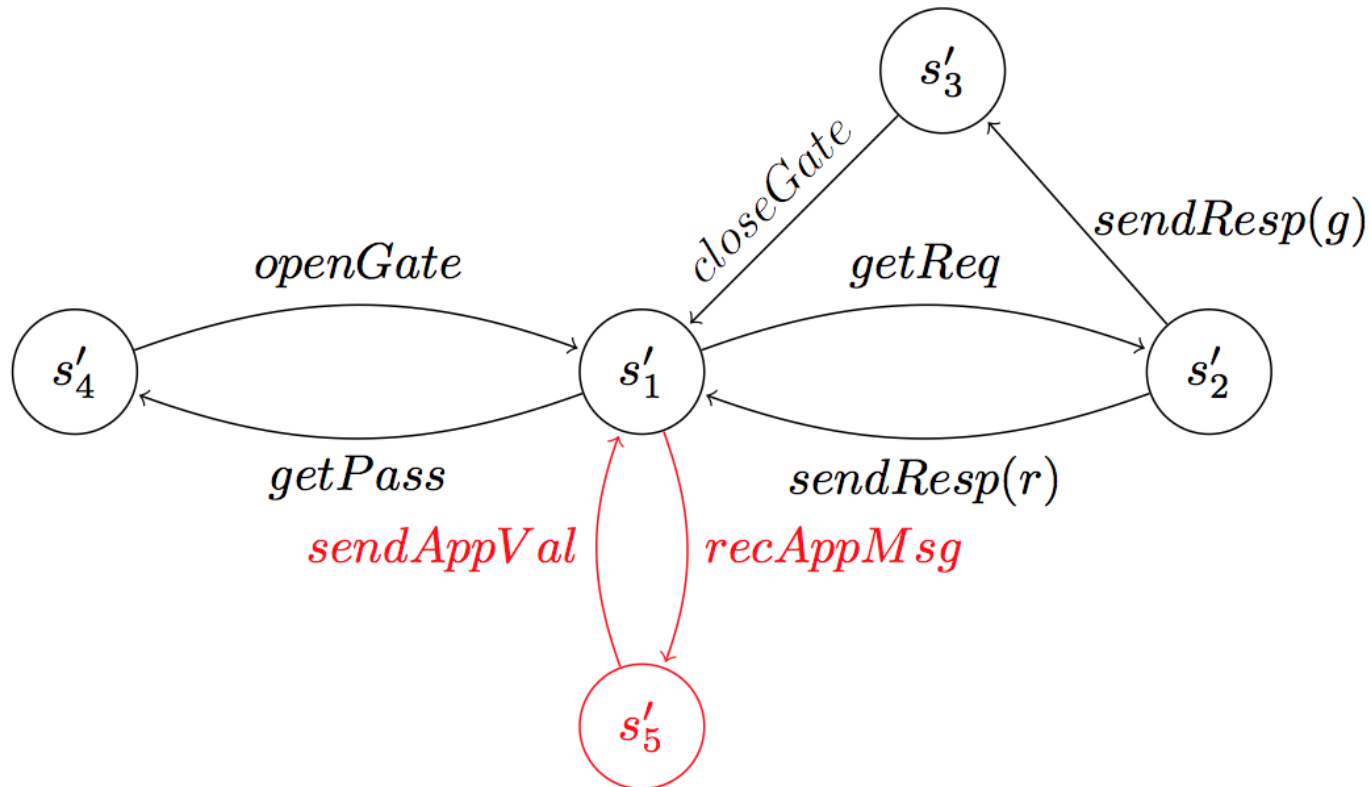
whenever the RailCab is at the noReturn location, gate must be closed for the new crossing mechanism.

Modeling the new system



Modeling the new system

The new Controller



Specifying the new system

Observers:

--- 8 observers

```
op loc : Sys -> Location .
op channel1 : Sys -> MsgSeq .
op channel2 : Sys -> MsgSeq .
op rStatus : Sys -> Status .
op gate : Sys -> Bool .
op pass : Sys -> Signal .
op conLoc : Sys -> Label .
op appResult : Sys -> Signal .
```

1. RailCab's location
2. Two communication channels
3. RailCab's status (running or braked)
4. Gate's status (closed or open)
5. Whether RailCab can pass
6. Controller's current state
7. Feedback of the gate status checking

Constructors:

--- 18 constructors

```
op init : Sys -> Sys [ctor] .
--- behavior of RailCab
op sendReq : Sys -> Sys [ctor] .
op recResp : Sys Signal -> Sys [ctor] .
op brake : Sys -> Sys [ctor] .
op move2leb : Sys -> Sys [ctor] .
op eBrake : Sys Signal -> Sys [ctor] .
op move2nr : Sys -> Sys [ctor] .
op cross : Sys -> Sys [ctor] .
op sendPass : Sys -> Sys [ctor] .
op sendAppReq : Sys -> Sys [ctor] .
op recAppResp : Sys Signal -> Sys [ctor] .
--- behavior of controller
op recReq : Sys -> Sys [ctor] .
op sendResp : Sys -> Sys [ctor] .
op closeGate : Sys -> Sys [ctor] .
op openGate : Sys -> Sys [ctor] .
op recAppReq : Sys -> Sys [ctor] .
op sendAppResp : Sys -> Sys [ctor] .
op getPass : Sys -> Sys [ctor] .
```

Specifying the new system

Definition of constructors by equations

An example of *recResp*, which receives the reply of from controller for the first communication.

```
--- recResp
--- when there is a response message at the head of channel2
ceq channel2(recResp(S,G)) = Q if (Q respMsg(G)) := channel2(S) [metadata "CA-" ] .
ceq channel2(recResp(S,G)) = channel2(S) if (Q passed ) := channel2(S) [metadata "CA-" ] .
ceq channel2(recResp(S,G)) = channel2(S) if (Q reqMsg ) := channel2(S) [metadata "CA-" ] .
ceq channel2(recResp(S,G)) = channel2(S) if (Q chkMsg ) := channel2(S) [metadata "CA-" ].
ceq channel2(recResp(S,G)) = channel2(S) if (Q gateMsg(G) ) := channel2(S) [metadata "CA-" ].

ceq pass(recResp(S,G)) = G      if (Q respMsg(G)) := channel2(S) [metadata "CA-" ] .
ceq pass(recResp(S,G)) = pass(S) if (Q passed ) := channel2(S) [metadata "CA-" ].
ceq pass(recResp(S,G)) = pass(S) if (Q reqMsg ) := channel2(S) [metadata "CA-" ].
ceq pass(recResp(S,G)) = pass(S) if (Q chkMsg ) := channel2(S) [metadata "CA-" ].
ceq pass(recResp(S,G)) = pass(S) if (Q gateMsg(G)) := channel2(S) [metadata "CA-" ].

eq rStatus(recResp(S,G)) = rStatus(S) .
eq loc(recResp(S,G)) = loc(S) .
eq gate(recResp(S,G)) = gate(S) .
eq conLoc(recResp(S,G)) = conLoc(S) .
eq channel1(recResp(S,G)) = channel1(S) .
eq appResult(recResp(S,G)) = appResult(S) .
```

Verification of the crossing property

We want to verify that:

whenever the RailCab is at the noReturn location, gate must be closed for the new crossing mechanism.

The goal to prove:

```
(goal RAILCAB-NEW |- ceq gate(S:Sys) = true if loc(S:Sys) = noReturn ;)
```

Proof:

```
(set ind on S:Sys .)
(apply SI .)    --- 18 subgoals generated
(auto .)
```

```
(apply CA CA IP RD .) --- apply 14 times
```

```
--- use lemma 1 to prove the case 1-1-7-1
```

```
(init ceq gate(S:Sys) = true if appResult(S:Sys) = grant . by S:Sys <- x#1 ; .)
```

```
--- use lemma 2 to prove the case 1-1-8-1
```

```
(init ceq true = false if conLoc(S:Sys) = s4 /\ loc(S:Sys) = noReturn . by S:Sys <- x#1 ; .)
```

```
--- the end of prove
```

Verification of the crossing property

Lemma-1:

It says that for any state if the feedback result of the second communication is grant in it, gate must be closed.

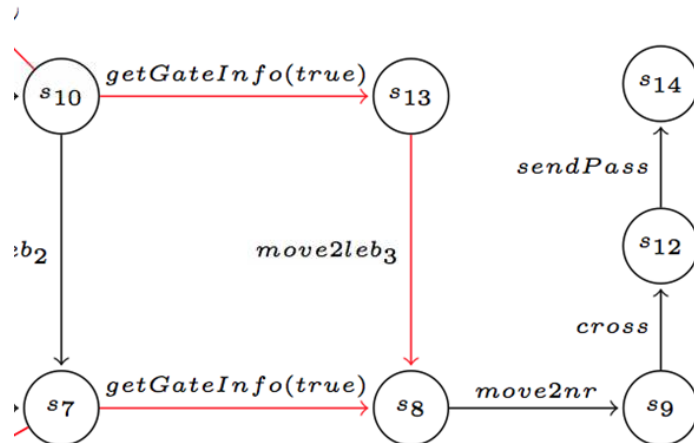
`(goal RAILCAB-NEW |- ceq gate(S:Sys) = true if appResult(S:Sys) = grant ;)`

Lemma-2:

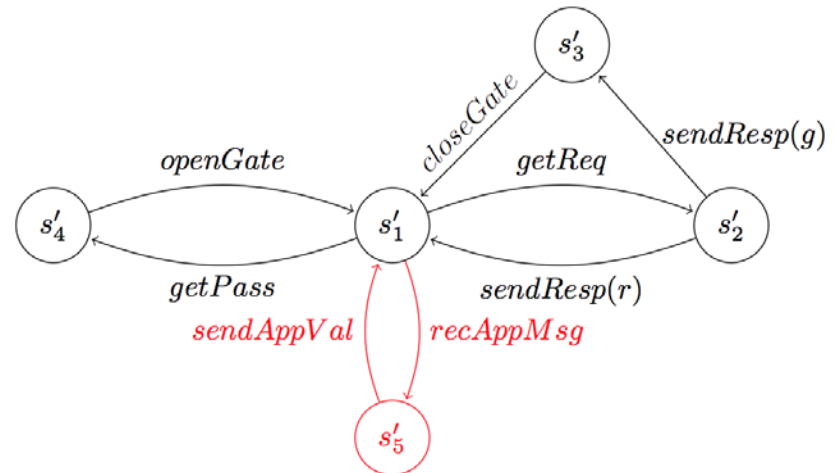
It is impossible that RailCab is at noReturn location, but the controller is at the s4.

`(goal RAILCAB-NEW |- ceq true = false if conLoc(S:Sys) = s4 /\ loc(S:Sys) = noReturn ;)`



RailCab

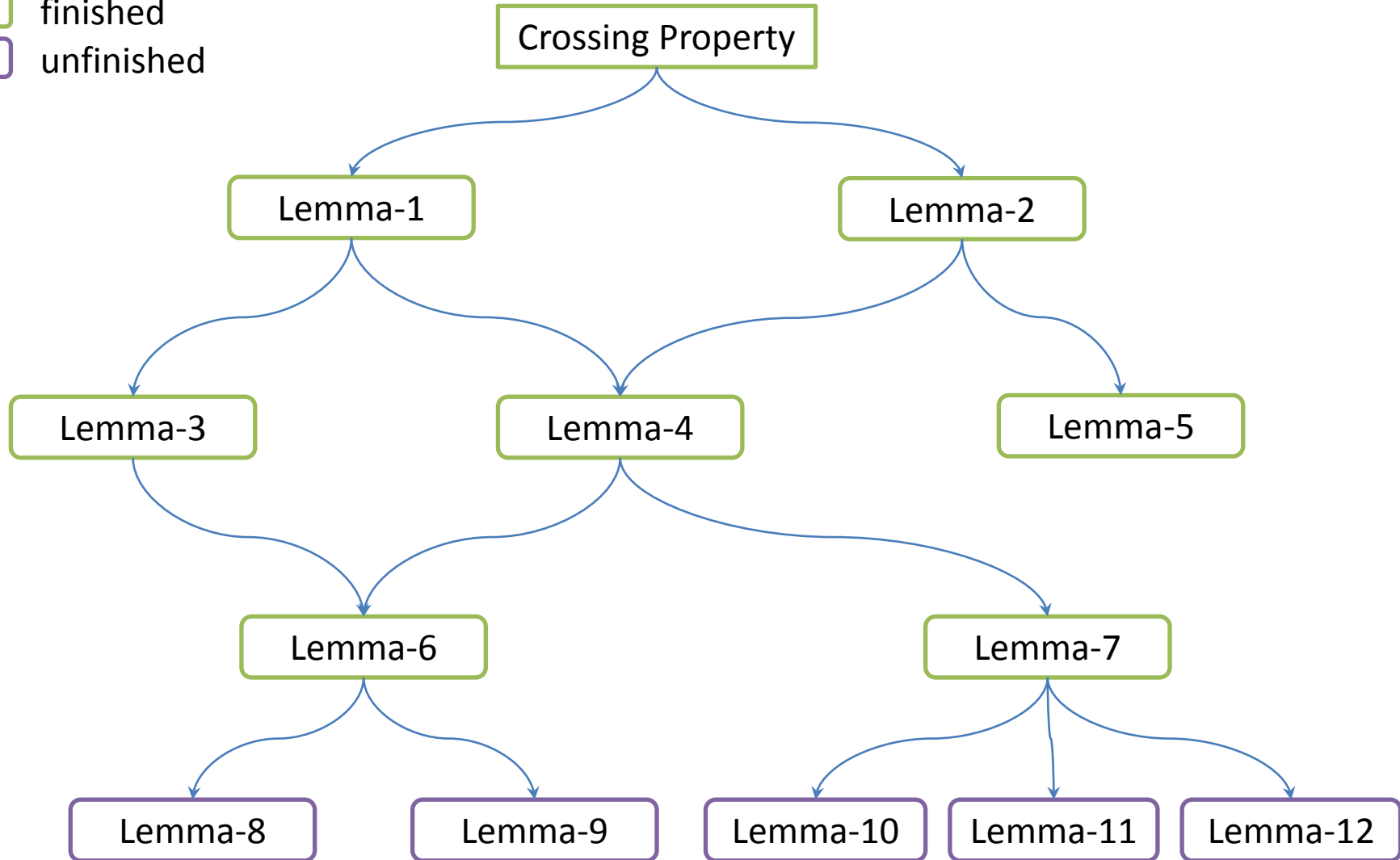


Controller



Overview of the proof

 finished
 unfinished



Lemmas

--- lemma-3

```
(goal RAILCAB-NEW |- ceq gate(S:Sys) = true if  
  channel2(S:Sys) = (Q:MsgSeq gateMsg(grant)) [label lemma-3 nonexec] ;)
```

--- lemma-4

```
(goal RAILCAB-NEW |- ceq true = false if  
  conLoc(S:Sys) = s4 /\ appResult(S:Sys) = grant ;)
```

--- lemma-5

```
(goal RAILCAB-NEW |- ceq channel1(S:Sys) = empty  
  if loc(S:Sys) = noReturn [label lemma-5] ;)
```

--- lemma-6

```
(goal RAILCAB-NEW |- ceq channel2(S:Sys) = empty if conLoc(S:Sys) = s4 ;)
```

--- lemma-7

```
(goal RAILCAB-NEW |- ceq channel1(S:Sys) = empty  
  if conLoc(S:Sys) = s1 /\ appResult(S:Sys) = grant [label lemma-7] ;)
```

--- lemma-8

```
(goal RAILCAB-NEW |- ceq (loc(S:Sys) ~ opposite) = false  
  if conLoc(S:Sys) = s4 [label lemma-8] .)
```

Dynamic update of RailCab System

Suppose that we need to dynamically update the RailCab System to the new version. We need to know:

1. What?
 - What are the **changes** (differences between the old system and the new one)
2. When?
 - In **which state** update should be applied to make the system after being updated safe?
 - What are the **criterion** of the safety?
3. How?
 - How changes are **applied** by updating

Changes between the old and new

- A new signal trigger (**approaching crossing**)
- Two new messages,
 - Request message of gate's status
 - Reply message of gate's status
- Four new behaviors
 - RailCab **sends** a request message of gate's status
 - RailCab **receives** a reply message of gate's status
 - Controller **receives** the request message
 - Controller **sends** a reply message
- Change of condition for braking
 - Not receives any of the two replies
 - Any of the reply is negative (rejected, or gate is open)
- Change of condition for passing
 - Both the two replies are positive (granted and gate is closed)

Criterion of safe updating (RailCab)

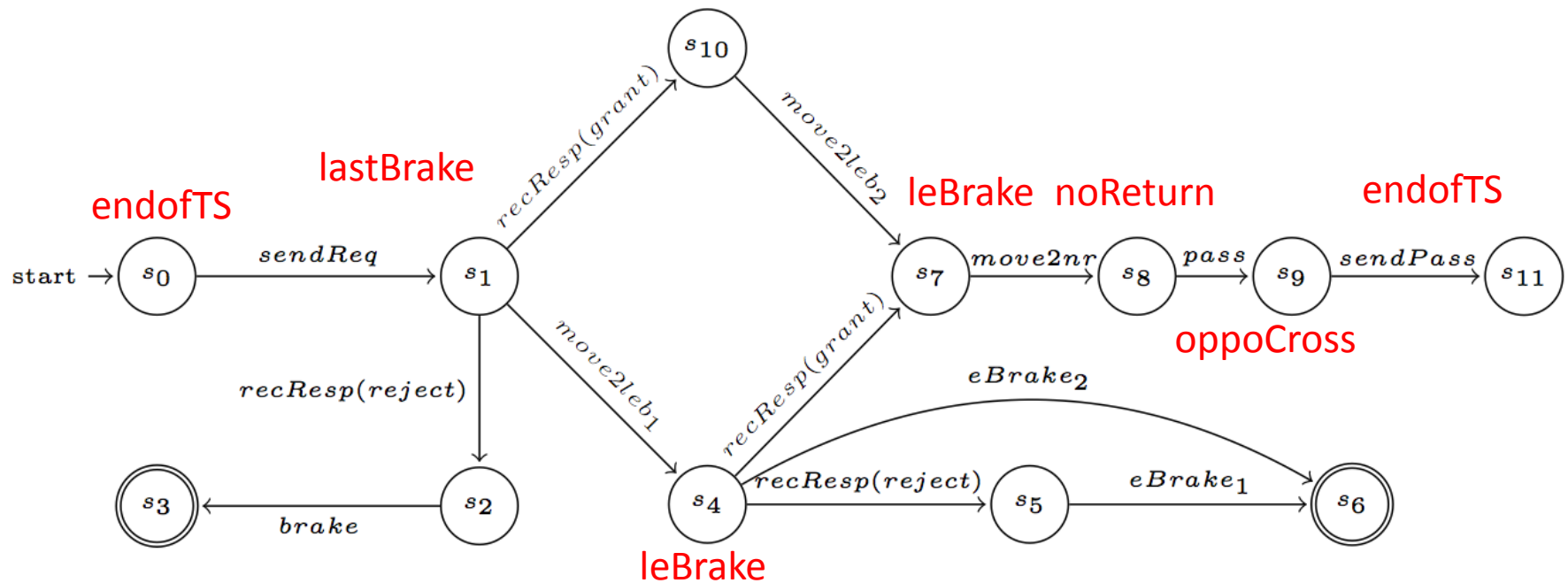
- A safe updating should satisfy the following three properties
 1. The updatable state should be **eventually** reachable
 2. After updating, it must be **safe** to cross the intersection i.e., gate must be closed.
 3. After updating, if RailCab can cross the intersection, it must eventually cross it.
- State preservation
 - State where update takes place **should be preserved** as much as possible.

Modeling and verifying the update

- Old and new systems are modeled as two **state transition systems**
- Updating is considered as a **transition** from an **old state to a new one**.
- By verification:
 - To **verify** whether an old state is a safe updating point.
 - To **find** a safe updating point

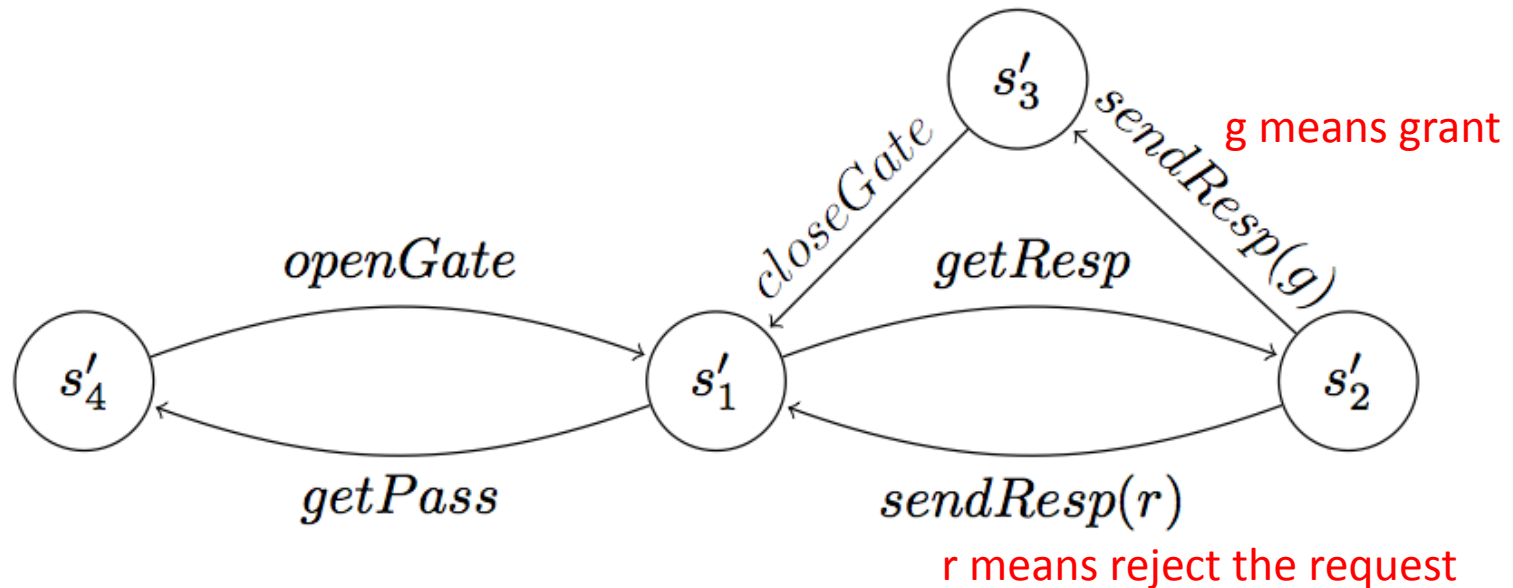
Modeling the old RailCab System

State transition system (RailCab)



Modeling the old RailCab System

State transition system (Controller)



Trans-based specification in CafeOBJ

Property to be verified:

Formalization of states

```
[OldState]
op __ : OldState OldState -> OldState {comm assoc} .

op loc-o:_ : Location -> OldState {constr} .
op channel1-o:_ : QMsg -> OldState {constr} .
op channel2-o:_ : QMsg -> OldState {constr} .
op rStatus-o:_ : Status -> OldState {constr} .
op conLoc-0:_ : Label -> OldState {constr} .
op gate-o:_ : Bool -> OldState {constr} .
op pass-o:_ : Signal -> OldState {constr} .
```

Formalization of behavior

```
trans [sendReq] : (loc-o: endOfTS) (channel1-o: NW) =>
  (loc-o: lastBrake) (channel1-o: (reqMsg & NW)) .

trans [recResp] :
  (channel2-o: (NW & respMsg(S))) (pass-o: S') =>
  (channel2-o: NW) (pass-o: S) .
```

Verification by searching in CafeOBJ

- Property to verify:
 - When RailCab is at the noReturn, gate must be closed.

There is never a state where RailCab is at the noReturn location but the gate is open, such that the state is reachable from initial state.

- Initial state:

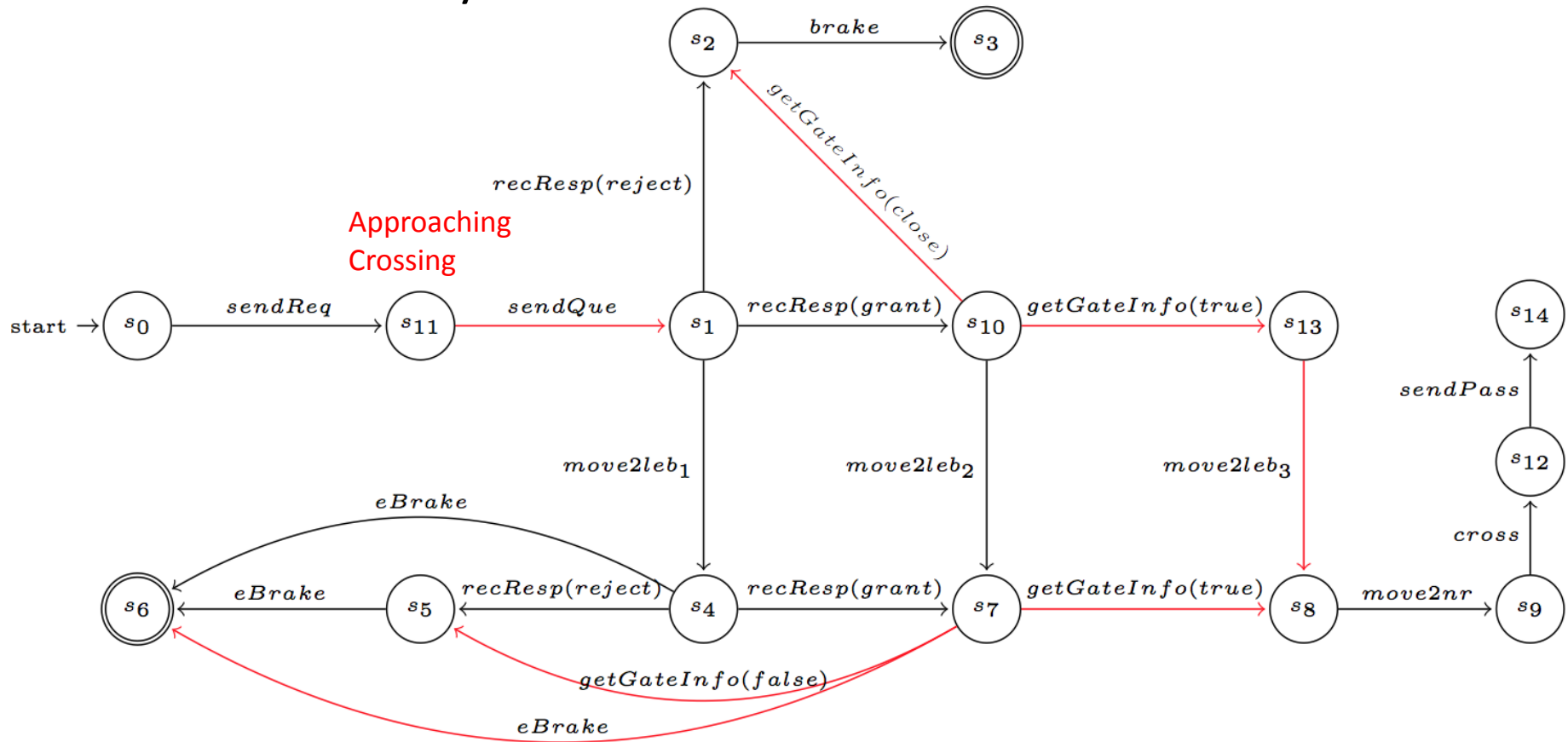
```
eq init-o = (loc-o: endOfTS) (rStatus-o: running) (pass-o: unknown)
             (channel1-o: empty) (channel2-o: empty) (conLoc-0: s1)
             (gate-o: false) .
```

- Searching:

```
open RAILCAB-OLD .
red init-o =(*,*)=>+ (gate-o: false) (loc-o: noReturn) S:OldState .
close
```

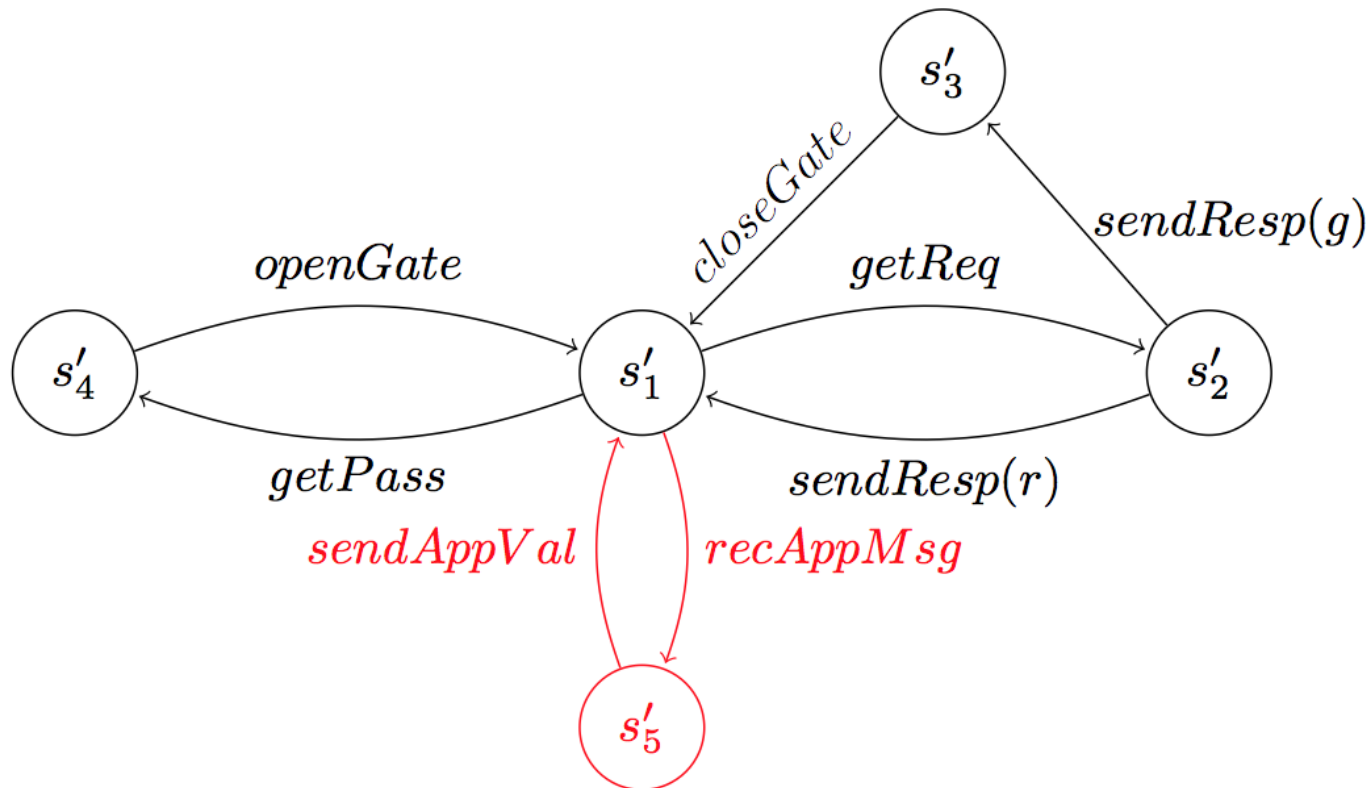
CafeOBJ returns a state that matches the pattern

Modeling the new system



Modeling the new system

The new Controller



Specifying the new system in CafeOBJ

Property to be verified:

- Formalization of states

```
[NewState]
op _,_ : NewState NewState -> NewState {comm assoc} .

op loc-n:_ : Location -> NewState {constr} .
op channel1-n:_ : QMsg -> NewState {constr} .
op channel2-n:_ : QMsg -> NewState {constr} .
op rStatus-n:_ : Status -> NewState {constr} .
op conLoc-N:_ : Label -> NewState {constr} .
op gate-n:_ : Bool -> NewState {constr} .
op pass-n:_ : Signal -> NewState {constr} .

-- newly added
op appResult:_ : Signal -> NewState {constr} .
```

- Formalization of behavior (**new**)

```
trans [recAppMsg] :
  (conLoc-N: s1), (channel1-n: (NW & chkMsg)) =>
    (conLoc-N: s5), (channel1-n: NW) .

trans [sendAppVal] : (conLoc-N: s5), (channel2-n: NW), (gate-n: B) =>
  (conLoc-N: s1), (channel2-n: (gateMsg(B) & NW)), (gate-n: B) .
```

Formalizing Dynamic updating (I)

- Declare a super sort **ONState**

```
mod! RAILCAB-UPDATE-1 {  
  inc(RAILCAB-NEW)  
  inc(RAILCAB-OLD)  
  
  [OldState NewState < ONState]
```

- Specify updating by transition from old state to new state

Example (**an offline-like updating**)

```
trans [update-1] :  
  (loc-o: LOC:Location) (rStatus-o: T:Status)  
  (pass-o: S:Signal) (channel1-o: CH1:QMsg)  
  (channel2-o: CH2:QMsg) (conLoc-0: L:Label)  
  (gate-o: B:Bool) =>  
  (loc-n: endOfTS), (rStatus-n: running),  
  (pass-n: unknown), (channel1-n: empty),  
  (channel2-n: empty), (conLoc-N: s1),  
  (gate-n: false), (appResult: unknown) .
```

**An arbitrary old
state**

**The new initial
state**

Formalizing Dynamic updating (II)

- A real dynamic updating

```
ctrans [update-2] :  
  (loc-o: LOC:Location) (rStatus-o: T:Status)  
  (pass-o: S:Signal) (channel1-o: CH1:QMsg)  
  (channel2-o: CH2:QMsg) (conLoc-o: L:Label)  
  (gate-o: B:Bool) =>  
  (loc-n: LOC), (rStatus-n: T),  
  (pass-n: S), (conLoc-n: L),  
  (channel1-n: CH1:QMsg),  
  (channel2-n: CH2:QMsg),  
  (gate-n: B), (appResult: (if B then grant else unknown fi))  
  if not (LOC = noReturn) .
```

1. When RailCab is not at the noReturn location
2. appResult is initialized according to gate's status
3. The old state is preserved

Verifying the correctness of updating

- We should verify the following three properties:
 1. Whether the updatable state is always eventually reached or RailCab is braked
 2. After being updated, whether the RailCab can always safely cross the intersection.
 3. After being updating, when the RailCab can cross the Intersection, whether it will eventually cross it.

We use Maude to verify these properties, because 1, and 3 are **liveness properties** which the current version of CafeOBJ does not support to verify.

Formalizing Dynamic updating (II)

We should verify the three properties:

```
--- define the condition of updatable state
op updatable : -> Prop .
ceq (loc-o: L) (rStatus-o: running) OS:OldState != updatable = true
  if not (L = noReturn) .
```

- Property 1:

```
--- formula of the properties
--- 1. Updatability, <> updatable or <> braked

red modelCheck(init-o, (<> updatable) ∨ (<> braked)) .
```

- Property 2:

```
--- a. [](@noReturn -> closedGate)
red modelCheck(init-o, [] (@noReturn -> closedGate)) .
```

A counterexample is found

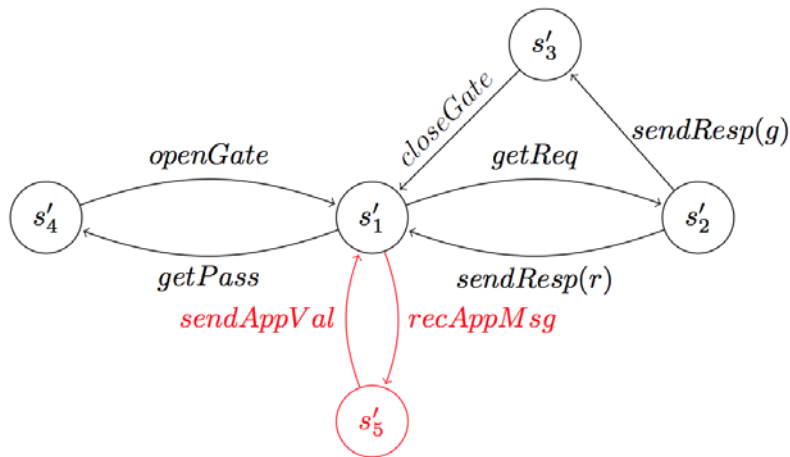
Counterexample

The updating point:

{loc-o: **leBrake**
channel1-o: **reqMsg**
channel2-o: empty
rStatus-o: running
conLoc-O: **s4**
gate-o: **true**
pass-o: unknown}

update

{loc-n: **leBrake**,
channel1-n: **reqMsg**,
channel2-n: empty,
rStatus-n: running,
conLoc-n: **s4**,
gate-n: **true**,
pass-n: unknown,
appResult: grant}



New controller

Transitions leading to counterexample:

1. openGate
2. getReq
3. sendResp(g)
4. recResp(g) by Railcab
Pass-n: grant

A correct updating

The updating point:

```
ctrans [update-3] :  
  (loc-o: LOC:Location) (rStatus-o: T:Status)  
  (pass-o: S:Signal) (channel1-o: empty)  
  (channel2-o: CH2:QMsg) (conLoc-o: s1)  
  (gate-o: B:Bool) =>  
    (loc-n: LOC), (rStatus-n: T),  
    (pass-n: S), (conLoc-N: s1),  
    (channel1-n: empty),  
    (channel2-n: CH2:QMsg),  
    (gate-n: B), (appResult: (if B then grant else unknown fi))  
  if not (LOC = noReturn) .
```

We can verify that the three properties are satisfied by the update!

Sufficient conditions for safe updating:

1. *All the messages sent by Railcab must be processed by the controller.*
2. Controller is at *s1*.
3. RailCab is not at *noReturn* location

Summary

In this lecture,

1. How to model the crossing mechanism of RailCab system and verify its crossing property in CILP
2. How to model dynamic software updating
 - a. Formalize both old and new systems as two independent state transition systems
 - b. Formalize dynamic updating as transitions from old state to new states
3. How to formalize the correctness of a dynamic update
4. How to find correct updating points

Materials used in this case study

File name	Content
<i>railcab-trans-old.cafe</i>	CafeOBJ specification of the old RailCab's crossing mechanism
<i>railcab-trans-new.cafe</i>	CafeOBJ specification of the old RailCab's crossing mechanism
<i>railcab-trans-update.cafe</i>	Three modules specifying the updates in the lecture
<i>railcab-trans-old.maude</i>	Maude specification of the old RailCab's crossing mechanism
<i>railcab-trans-new.maude</i>	Maude specification of the new RailCab's crossing mechanism
<i>railcab-trans-update-1.maude</i>	The second update and its verification
<i>railcab-trans-update-2.maude</i>	The third update and its verification
<i>railcab-citp-new.maude</i>	The OTS-based specification used for theorem proving in CITP
<i>inv.maude</i>	The proof of Crossing Property in CITP
<i>lemma-*.maude</i>	Lemmas and their proofs

Reference:

Steve Eker, et. al, *The Maude LTL model checker*, ENTCS, Vol. 71, pp. 162-187, Elsevier, 2003.