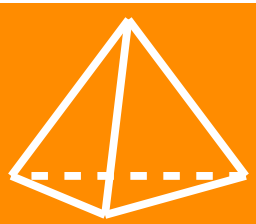
Counterexample-guided Design of Property-based Dynamic Software Updates

Min Zhang

2014.10.15



Research Center for Software Verification
Japan Advanced Institute of Science and Technology

Contents

1. Dynamic Software Updating (DSU)
   - Instantaneous DSU
   - Incremental DSU
2. Formalization and verification of instantaneous DSU
3. A case study

About DSU

Dynamic Software Updating

An updating technique for updating software systems that are running without incurring downtime.

Q: Why do we need DSU?

A: Two main reasons:

1. Software systems are inevitably subject to update
2. *For some software systems, shutting them down is expensive!*

Example (How much is an hour of downtime worth to you?)

According to a Yankee Group report, banks can lose as much as US $2.6 million per hour and brokerages as much as US $4.5 million per hour from downtime.

Existing works on DSU

More than 40 approaches have been proposed and prototypes have been implemented. (H. Seifzadeh, et al, *A survey of dynamic software updating*. J. of Software, Evolution, and Process. 25:535-568, 2013)

Table I. Evaluation of DSU research works based on capabilities and constraints.

	Scope*	Implemented	High level of Abs.	Upd. pre. started code	Simplicity	Consistency	Wait time and predict.†	Update duration	Code clean-up	Performance	Supported changes
Appavoo	OS	✓	×	×	✓	✓✓	P	✓	×	✓	×
Argus(A)	DS	✓	×	×	×	×	I	✓	×	×	×
Argus(S)	DS	✓	×	✓	×	✓	P	✓	×	✓	×
AutoPod	OS	✓	×	×	✓✓	✓✓	I	×	×	×	✓
Bialek	D	✓	×	×	✓	✓	U	✓	×	✓	✓
Boyapati	B	✓	×	×	×	✓	I	✓	×	✓	✓
Buisson	×	×	×	×	?	✓✓	I	?	?	?	?
DCF	D	✓	✓	×	✓✓	×	I	✓	×	×	✓
DRACO	DS	✓	×	✓	✓✓	✓✓	U	✓	×	✓	✓
Duggan	×	×	×	×	?	✓	I	✓✓	?	×	✓
DURTS	RT	✓	×	×	✓	×	I	✓✓	×	✓	×
DUSC	D	✓	×	×	✓✓	✓	I	✓	×	✓	×
DVM	D	✓	×	×	✓	✓	I	×	×	✓✓	✓
DYMOS	S	✓	×	×	✓	✓✓	P	✓	×	✓	✓
DynAMOS	OS	✓	×	✓	✓	✓✓	I	✓	×	×	✓
DynC++	S	✓	×	×	✓	×	I	✓	✓	✓	×
Ekiden	S	✓	×	×	✓✓	✓	I	×	—	✓✓	✓
EmbedDSU	RT	✓	×	×	✓✓	✓	P	✓	×	✓	✓
Fabry	B	×	×	×	?	✓	I	✓	×	✓	✓
Ginseng	S	✓	×	×	✓✓	✓✓	U	✓	×	✓	✓
Giuffrida	×	×	×	×	?	✓✓	P	?	?	?	×
Gracioli	RT	✓	×	×	✓✓	✓	I	✓	✓	✓	×

Motivation of the work

Three features of dynamic updating:

1. Doing dynamic update to a running system is dangerous!

   LOSE THE BATON!

2. Target systems are usually safety/mission-critical systems, while crashing them is expensive!

   LOSE THE GOLDEN MEDAL!

3. Designing a correct dynamic update is challenging!

   We do not even know what correctness means.

ON-THE-FLY PROGRAM MODIFICATION: SYSTEMS FOR DYNAMIC UPDATING

The rising cost of shutting down systems for maintenance and repair is forcing developers to look at ways to repair software as it runs. The authors briefly describe available updating systems and present a prototype.

MARK E. SEGAL
Bellcore
OPHIR FRIEDER
George Mason University

Advances in software-development technology and methods continue to help engineers build larger and more complex systems, but those systems are

ing, disabling a bank-transaction processing system may have significant economic consequences — particularly if the companies involved have a reputation for pro-

Studying dynamic updating at two levels

1. code-level: implementing a dynamic updating correctly
   - the code differences between old and new systems
   - how code is managed in memory by updating
   - how code is executed after updating, e.g.,
     - *whether a function refers to the data of correct type*,
     - *whether a function calls another of the correct version*.

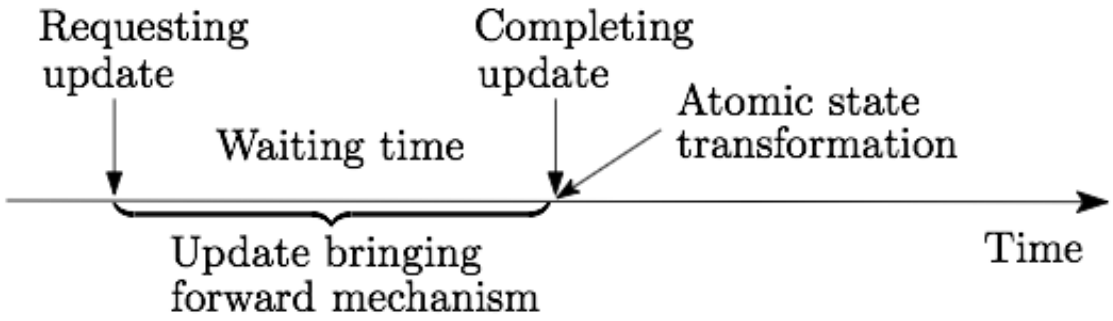
2. **design-level** (this talk): designing a dynamic updating correctly
   - static differences, e.g., system structures/states
   - dynamic differences, e.g., system behaviors
   - the behaviors during updating

> Software is built on abstractions. Pick the right ones, and programming will flow naturally from design; ...
>
> – Daniel Jackson

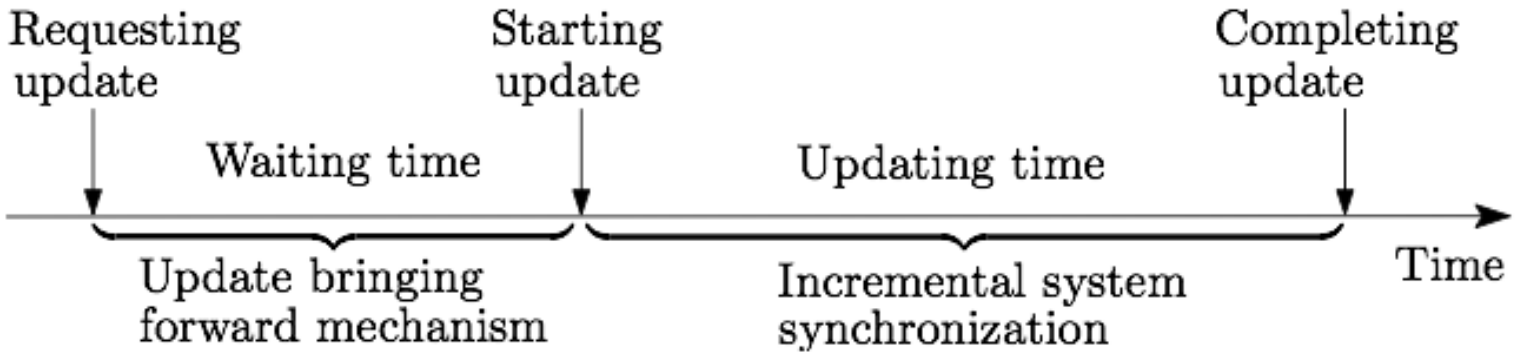
Instantaneous and incremental DSU

At the design level, we classify DSU into two classes according to update duration.

- **Instantaneous DSU** (this talk)



- Incremental DSU



Five requirements to design an instantaneous dynamic updating

1. a running software system (old system)
2. a new version of the running system (new system)
3. a set of updating points (a subset of old states where updates will be applied)
4. an (optional) update bringing forward mechanism, which is used to bring the running system to an update point as soon as possible.
5. a state transformation function, a (partial) function from old states to new states.

Challenges to the design of correct dynamic update

1. What does correct mean?
2. How to identify a set of updating points?
3. How to design an update bringing forward mechanism?
4. How to define a state transformation function?

Contributions of our work

1. We define the *correctness* in terms of the *properties* of the old and new systems.
2. We propose a *counterexample-guided approach* to designing correct dynamic updates by
   1. identifying a set of safe updating points
   2. defining a correct a state transformation function
   3. designing a correct update bringing forward mechanism

Formalization of instantaneous dynamic updates (I)
Formalization of software systems as Kripke structures

Definition (Kripke structure)

A Kripke structure $\mathcal{K}$ is a four-tuple (S, I, T, L) over a set AP of atomic propositions, where

- S: a finite set of states
- I: a set $I \subseteq S$ of initial states
- T: a transition relation $T \subseteq S \times S$, and T must be total, i.e., for any $s \in S$, there exists $s' \in S$, s.t. $(s, s') \in T$.
- L: a labeling function $S \to 2^{AP}$.

We assume that both the old and the new systems are finite-state systems.

Let $\mathcal{K}_1 = (S_1, I_1, T_1, L_1)$ and $\mathcal{K}_2 = (S_2, I_2, T_2, L_2)$ be two Kripke structures over two sets of atomic propositions AP_1 and AP_2 such that $\mathcal{K}_1$ and $\mathcal{K}_2$ model the old and the new systems respectively.

Formalization of instantaneous dynamic updates (II)

- A set S' of **update points** such that $S' \subseteq S_1$.
- A state transformation function $f : S' \to S_2$.
- Updating bringing forward mechanism: $T' \subseteq (S_1 \times S_1)$

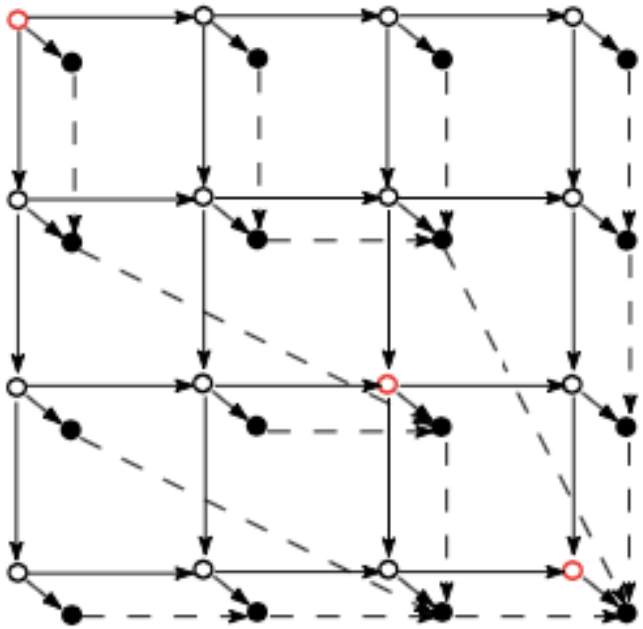
Formalization of update request (I)

Update request (generally update request can be made at any moment when the old system is running):

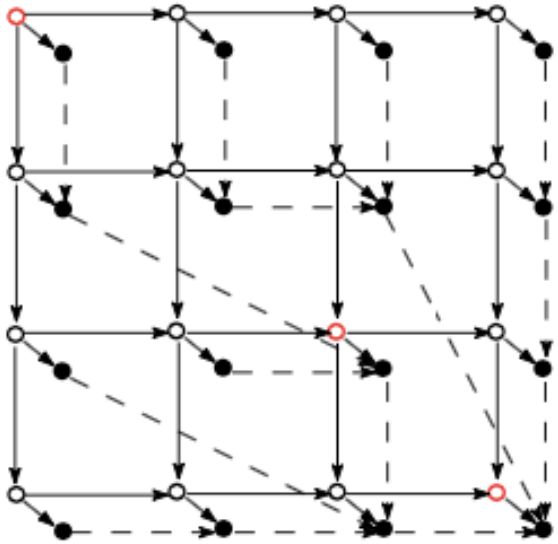
Front layer: old system

Back layer: update bringing forward mechanism

From the front layer to the back layer: update request



Formalization of update request (II)



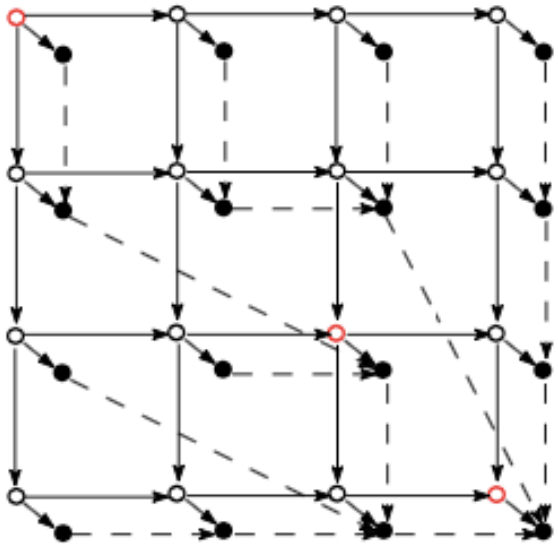
- Let S_1' be a mirror set of S_1, i.e., $S_1' \cap S_1 = \emptyset$ and there exists a bijection $\psi : S_1' \to S_1$. Let $L^\bullet : S_1 \cup S_1' \to 2^{AP_1 \cup \{req, upd\}}$

$$L^\bullet(s) = \begin{cases} L_1(s) & \text{if } s \in S_1 \\ L_1(\varphi^{-1}(s)) \cup \{req\} & \text{if } s \in S_1' \text{ and } \varphi^{-1}(s) \notin S' \\ L_1(\varphi^{-1}(s)) \cup \{req, upd\} & \text{if } s \in S_1' \text{ and } \varphi^{-1}(s) \in S' \end{cases}$$

$req \notin AP_1$: states where update request has been made;

$upd \notin AP_1$: update points

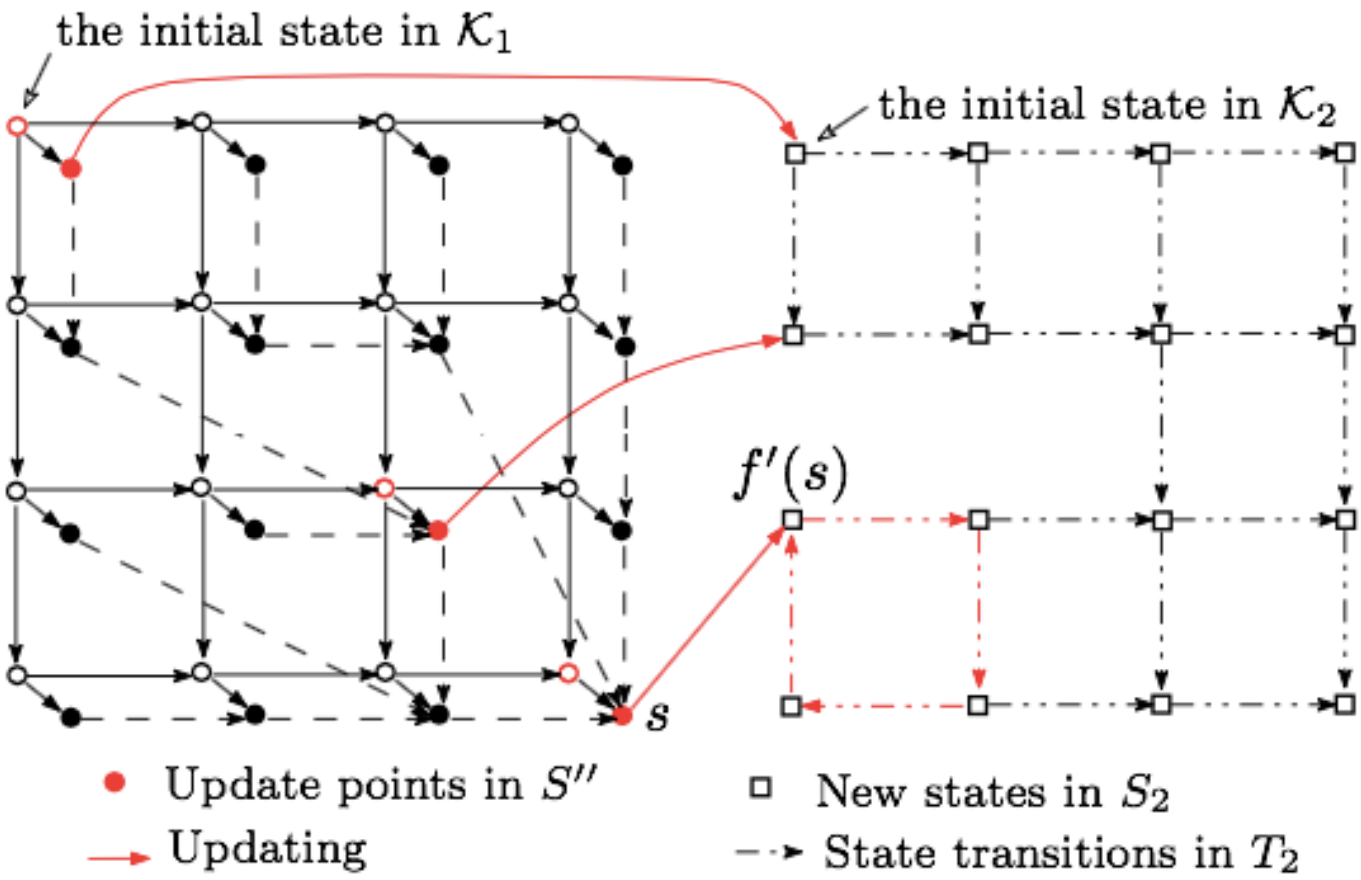
Formalization of update request (III)



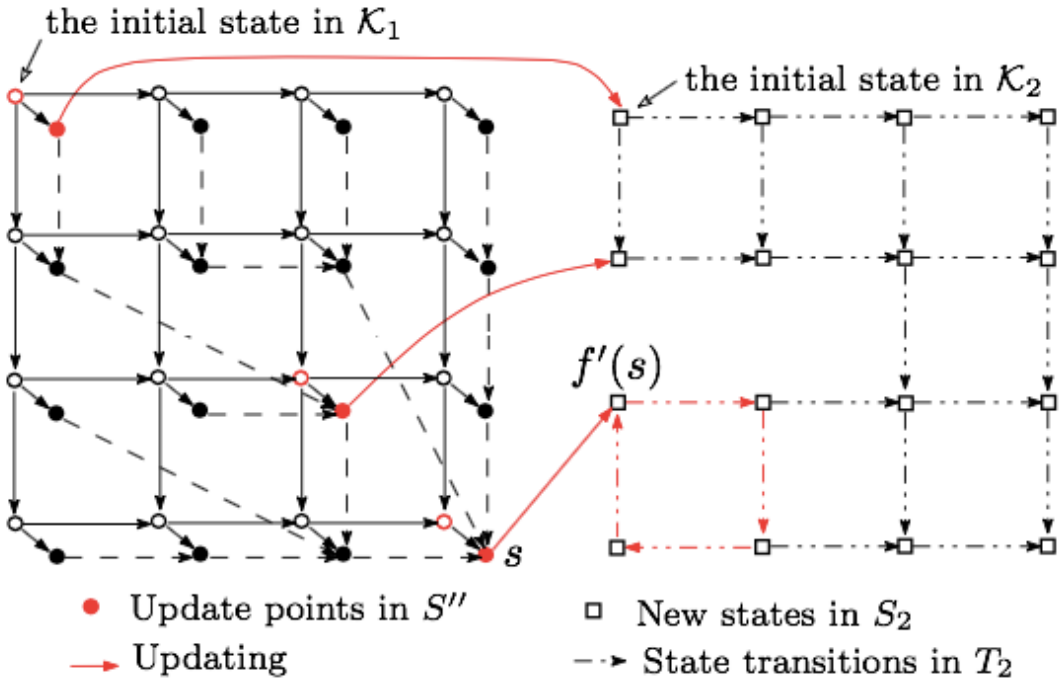
- Let $T'' \subseteq S'_1 \times S'_1$ s.t.
  $T'' = \{(s'_1, s'_2) | s'_1 \in S'_1, s'_2 \in S'_1, (\varphi^{-1}(s'_1), \varphi^{-1}(s'_2)) \in T'\}$.
- Let $T^\bullet = T_1 \cup T'' \cup \{(s, s') | s \in S_1, s' \in S'_1, s' = \varphi(s)\}$.
- Let $S^\bullet = S_1 \cup S'_1$.

We obtain a Kripke structure $\mathcal{K}^\bullet = (S^\bullet, I_1, T^\bullet, L^\bullet)$ over $AP_1 \cup \{req, upd\}$.

Formalization of updating (I)



Formalization of updating (II)



Some assumptions:

1. $S_2 \cap S_1 = \emptyset$, $S_2 \cap S_1' = \emptyset$
2. $AP_2 \cap AP_1 = \emptyset$
3. *new* is an atomic proposition such that $new \notin AP_1$ and $new \notin AP_2$.

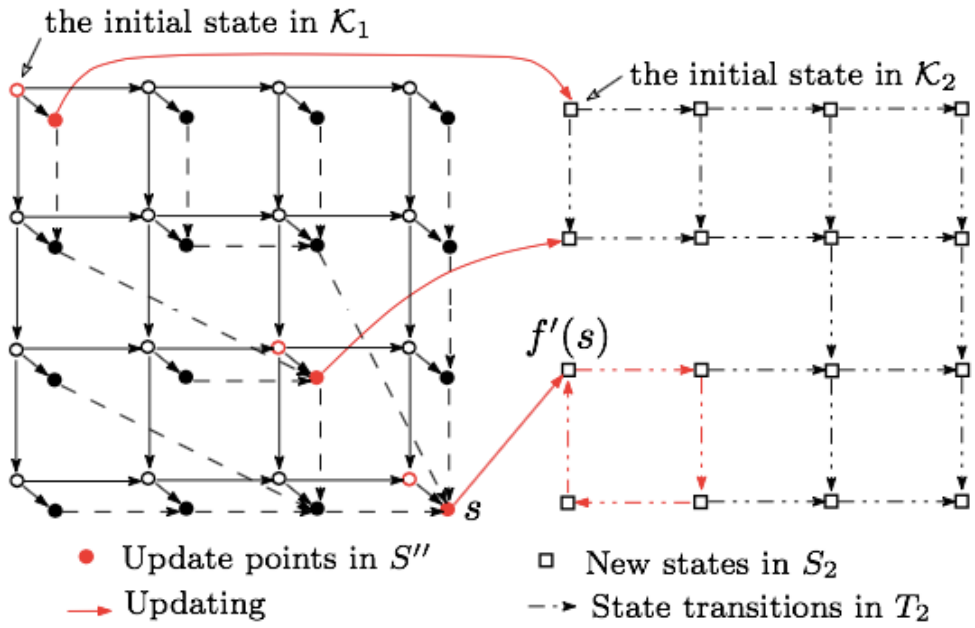
Formalization of updating (III)

Let $S^\dagger = S_1 \cup S_1' \cup S_2$

$AP^\dagger = AP_1 \cup AP_2 \cup \{new, req, upd\}$

$L^\dagger : S^\dagger \to 2^{AP^\dagger}$ s.t.

$$L^\dagger(s) = \begin{cases} L^\bullet(s) & \text{if } s \in S^\bullet \\ L_2(s) \cup \{new\} & \text{if } s \in S_2 \end{cases}$$



$S'' \subseteq S_1'$ s.t. $S'' = \{\varphi(s) | s \in S'\}$

$f' : S'' \to S_2$ s.t. $f'(s) = f(\varphi^{-1}(s))$, and

$T^\dagger = T^\bullet \cup T_2 \cup \{(s, f'(s)) | s \in S''\}$.

Finally, we obtain $\mathcal{K}^\dagger = (S^\dagger, I_1, T^\dagger, L^\dagger)$ over the set $AP^\dagger$.

Property-based Correctness of dynamic updating

Definition (Property-based correctness)

A dynamic update is correct if it satisfies a set P of given properties.

We classify the properties in P into three kinds:

1. common properties *which should be satisfied by all the dynamic updates*
2. dependent properties: *which specify the relation between the old system and the new system*
3. independent properties: *the properties of the new system which are independent from those of the old system*

Common properties

Definition (Updatability)

Once an update request is made, the running system must eventually reach an update point.

LTL formula of updatability:

$$\Box((\neg req \wedge \bigcirc req) \rightarrow \bigcirc \Diamond upd)$$

- $\Box$: Globally
- $\bigcirc$: Next
- $\Diamond$: Finally

Updatability is a property of update bringing forward mechanism.

Dependent properties

Given two LTL properties p_1 and p_2 which are built out of the atomic propositions in AP_1 and AP_2, if p_1 holds in the old system, then p_2 holds in the running system after it is updated.

Example (Connection preservation)

For a dynamic update to an FTP server a property that the update should satisfy is that all the connections should be preserved, i.e., if the number of connections in the old system is n, so is the number of connections after update.

A dependent property with p_1 and p_2 can be formalized as an LTL formula w.r.t. $\mathcal{K}^\dagger$, i.e., $(p_1 \ \mathbf{U} \ req) \rightarrow ((\neg new \wedge \bigcirc new) \rightarrow \bigcirc p_2)$

Independent properties

The properties of the new systems which are independent from those in the old systems.

Example (Dynamic update for bug-fixing)

Some properties are supposed to hold after the update, regardless of the situation when the update is applied.

Let p be an LTL property of the new property which is independent from the properties of the old one, its corresponding LTL formula w.r.t. $\mathcal{K}^{\dagger}$ is $(\neg new \wedge \bigcirc new) \rightarrow \bigcirc p$.

P-Correctness of a dynamic update

Definition (P-Correctness)

Suppose that $\mathcal{K}^\dagger$ is a Kripke structure of a dynamic update and $\mathcal{P}$ is a set of LTL formulas for a given set P of properties. The dynamic update is called *P-correct* iff for each p in $\mathcal{P}$, $\mathcal{K}^\dagger \models p$.

An algorithm of designing a P-correct dynamic update

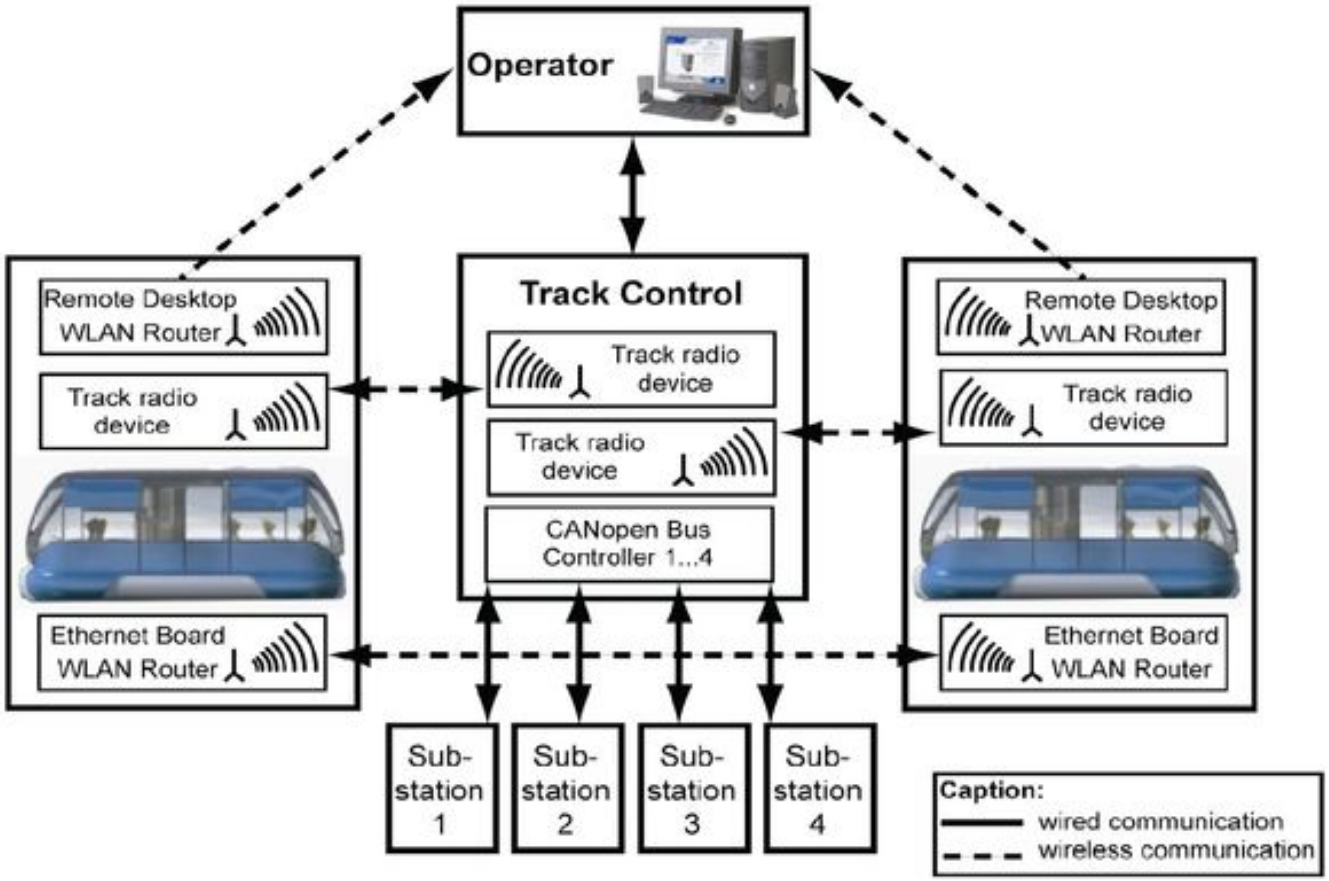
input : $\mathcal{K}_1, \mathcal{K}_2, T', f, P$

output : A set S' of update points, a refined update bringing forward mechanism T'' and a refined state transformation function f'

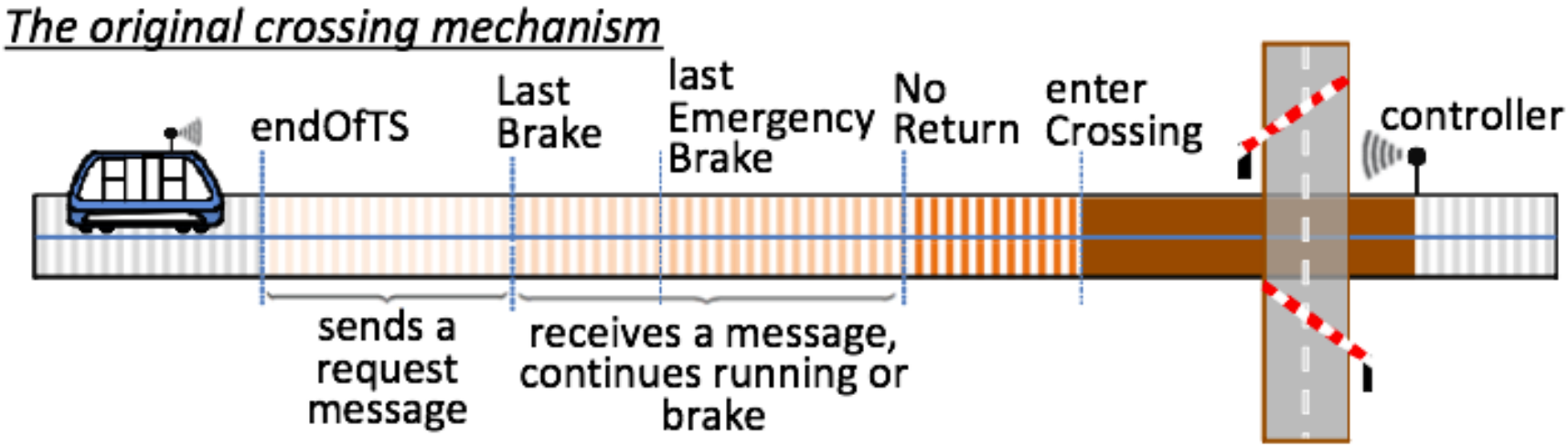
1 $S' := S_1$; $f' := f$; $T'' := T'$;
2 **while** *true* **do**
3 Synthesize a Kripke structure $\mathcal{K}^\dagger$ with $\mathcal{K}_1, \mathcal{K}_2, S', T'', f'$;
4 **if** $\mathcal{K}^\dagger \not\models \Box((\neg req \wedge \bigcirc req) \rightarrow \bigcirc\Diamond upd)$ **then** /* Check the updatability */
5 Refine S' and T'' based on the counterexample;
6 **else**
7 Construct LTL formulas $\mathcal{P}$ for P with respect to $\mathcal{K}^\dagger$; $hasCE := \text{false}$;
8 **foreach** p *in* $\mathcal{P}$ **do**
9 **if** $\mathcal{K}^\dagger \not\models p$ **then** /* Check if the counterexample is real for p */
10 **if** $p = \Box((p_1 \ \boldsymbol{U} \ req) \rightarrow ((\neg new \wedge \bigcirc new) \rightarrow \bigcirc p_2))$ **then**
11 **if** $\mathcal{K}_1 \models p_1 \wedge \mathcal{K}_2 \models p_2$ **then** { $hasCE := \text{true}$; break;} **else return** null ; /* The counterexample is real for p */
12 **else if** $p = \Box((\neg new \wedge \bigcirc new) \rightarrow \bigcirc p')$ **then**
13 **if** $\mathcal{K}_2 \models p'$ **then** { $hasCE := \text{true}$; break; } **else return** null; /* If p' is not satisfied by $\mathcal{K}_2$ */
14 **if** $hasCE$ **then** /* A real counterexample is found */
15 Refine S' and f' based on the counterexample;
16 **else** /* The algorithm terminates here */
17 break;
18 **return** S', T'' *and* f';

A Case Study

The Railcab system: *a conceptual driverless rail-bound transportation system proposed at the University of Paderborn*.



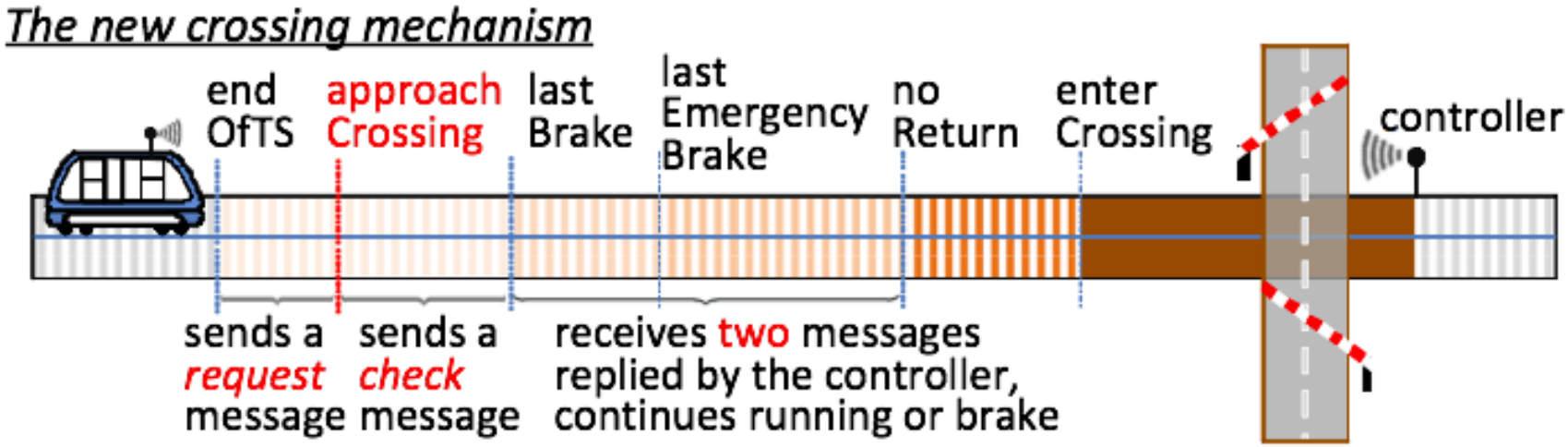
A buggy crossing mechanism in the Railcab system



1. At endOfTS, a Railcab sends a message to the controller to request for passing.
2. The controller replies a message to approve (if the gate is open) and reject (if the gate is closed) the request.
3. If the request is approved, the controller then closes the gate.

When a Railcab is crossing the gate, the gate may be still open.

A modification



1. A new signal trigger approchCrossing is added.
2. At approchCrossing, the Railcab sends a message to the controller to check the status of the gate.
3. If the message replied by the controller says the gate is close, it will pass, otherwise, it stops.

To design a dynamic update to the buggy mechanism

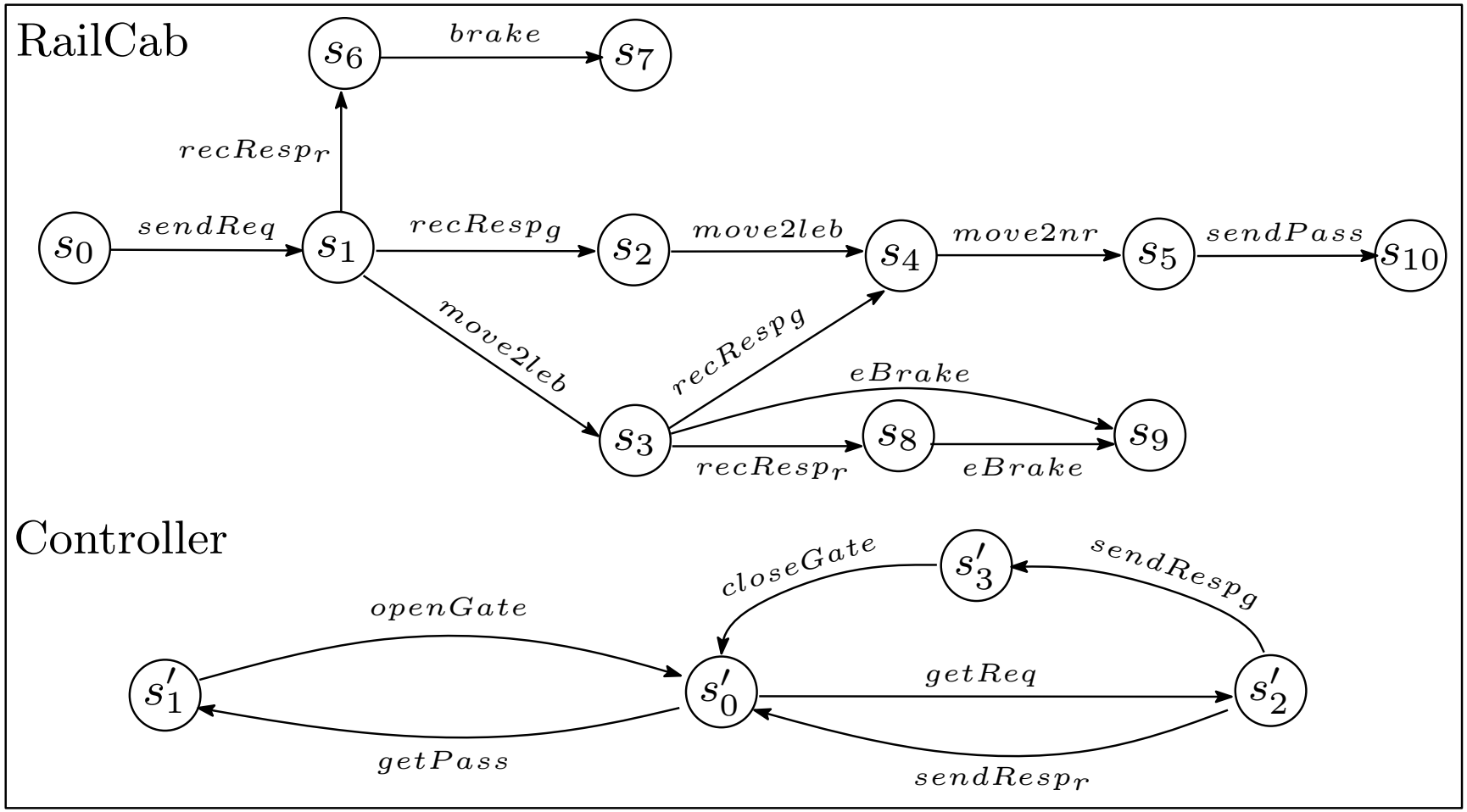
Some requirements to the update:

1. After update, the bug should be fixed, i.e., whenever the Railcab is passing the crossing, the gate must be closed.
2. The update should be transparent to the Railcab, i.e., not affecting the running Railcabs.

To design such an update:

1. What should be done to achieve the update, i.e., state transformation?
2. When is it safe to apply, i.e., update points?
3. How to apply as soon as possible, i.e., update bringing forward mechanism?

The Railcab system as a Distributed system



Specification of the buggy crossing mechanism in Maude (I)

Maude: an algebraic specification language based on rewriting logic.

We choose Maude because it supports LTL model checking.

A system state is represented as a multiset of sort `OldState`.

```
(gate:_)(conLoc:_)(cabLoc:_)(chan1:_)(chan2:_)
(cabSta:_)(pass:_)
```

- `gate`: the status of gate, i.e., open or close
- `conLoc`: the state of the controller
- `cabLoc`: the location of the Railcab
- `chan1`: the sequence of the messages sent from the Railcab to the controller
- `chan2`: the sequence of the messages sent from the controller to the Railcab
- `cabSta`: the status of the Railcab, i.e., running or braked
- `pass`: the request result: i.e., approved or rejected.

Specification of the buggy crossing mechanism in Maude (II)

Specification of the transitions as rewrite rules:

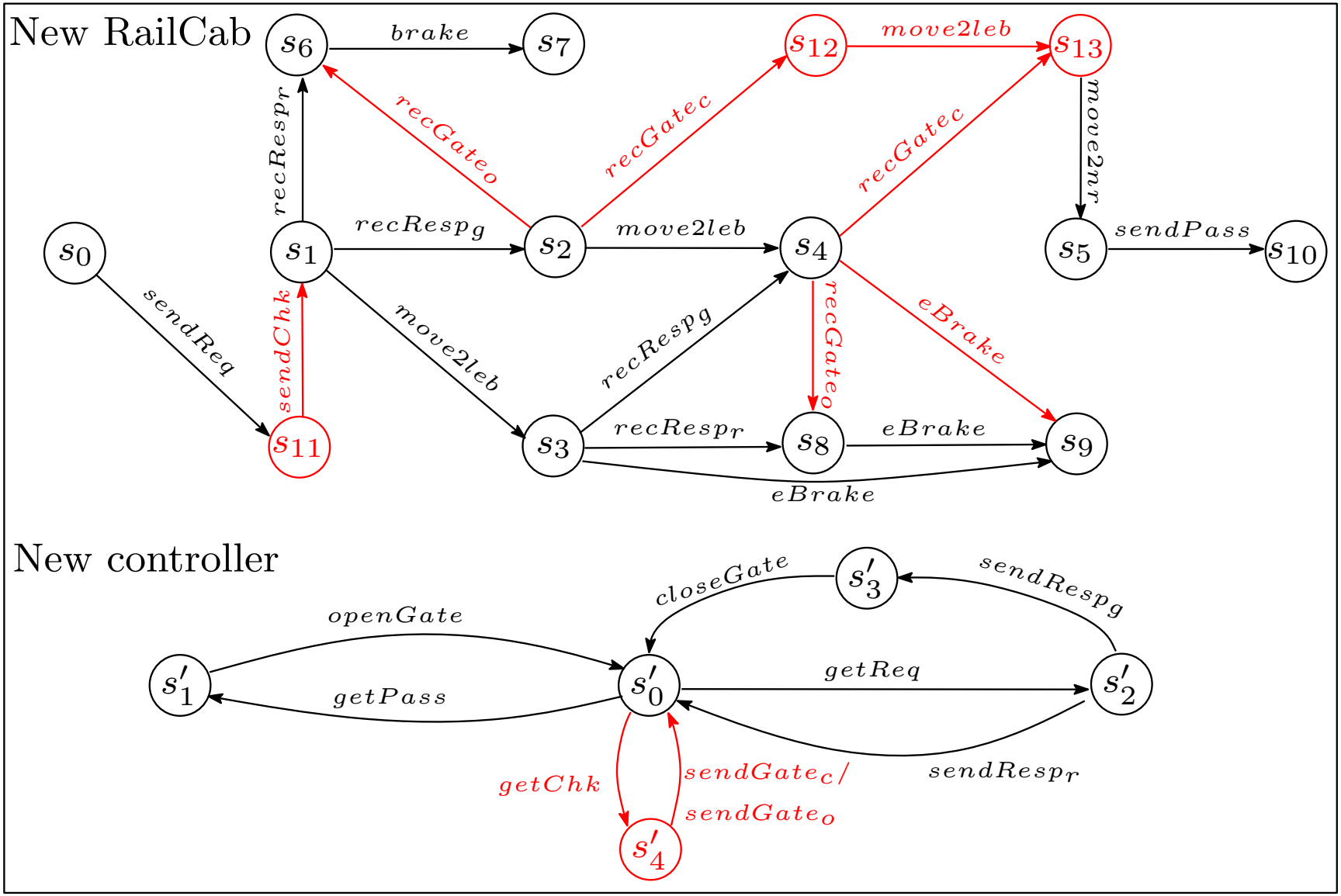
Example (sending a request message)

The Railcab sends a request to the controller when it is at the `endOfTS` location.

```
rl [sendReq]:(cabLoc: endOfTS)  (chan1: SQ) =>
             (cabLoc: lastBrake)(chan1: (reqMsg & SQ)) .
```

`SQ`: a variable of `MsgSeq` (sort of sequences of messages).

Formalization of the new crossing mechanism



Specification of the new crossing mechanism in Maude

Let NewState be the sort of states of the new mechanism.

```
(gate':_)(conLoc':_)(cabLoc':_)(chan1':_)(chan2':_)
(cabSta':_)(pass':_)(appRes:_)
```

appRes: to record the result of the gate status learned from the message replied by the controller, *unknown*, *open* or *close*.

Transitions in the new crossing mechanism can be formalized similarly.

An initial design of a dynamic update

- Update points: we simply assume that the update can take place at any moment
- State transformation: simply copy the old state to the new one and initialize `appRes` by *unknown*
- Update bringing forward: no, because the update is expected to be transparent to the Railcab.

Formalization of the initial design

- Formalization of the state transformation:

```
op f : OldState -> NewState .
eq f((gate: B)(conLoc: L1)(cabLoc: L2)(chan1: CH1)
     (chan2: CH2)(cabSta: S)(pass: P)) =
     (gate': B)(conLoc': L1)(cabLoc': L2)(chan1': CH1)
     (chan2':CH2)(cabSta': S)(pass': P)(appRes: unknown).
```

- Formalization of the update request and update application:

```
rl [request]: (req: false) => (req: true).
rl [apply]: (req: true) OS:OldState => f(OS) .
```

We declare a super sort `State`, i.e., `OldState NewState < State`.

We add a new field `(req:_)` to the old state to formalize the update request.

Formalization of the expected properties of the dynamic update

Crossing property

After updating, when the Railcab is at the `noReturn` point, the gate must be closed.

Passability

If the gate is open and the Railcab wants to pass the crossing, it must finally pass it.

- `@noRet`: the RailCab is at the `noReturn` point
- `passed`: the RailCab has already passed the crossing
- `gateClose`: the gate is closed
- `gateOpen`: the gate is open

$$\textit{Crossing property:} \quad p_1 \triangleq \Box(\texttt{@noRet} \rightarrow \texttt{gateClose})$$

$$\textit{Passability:} \quad p_2 \triangleq \Box(\texttt{gateOpen} \wedge \neg\texttt{passed} \rightarrow \Diamond\texttt{@noRet})$$

Verification by model checking

The LTL formulas to be model checked:

$$\textit{Updatability}: \quad \Box((\neg \textit{req} \wedge \bigcirc \textit{req}) \rightarrow \bigcirc \Diamond \textit{upd}) \tag{1}$$

$$\textit{Crossing property}: \quad \Box((\neg \texttt{new} \wedge \bigcirc \texttt{new}) \rightarrow \bigcirc p_1) \tag{2}$$

$$\textit{Passability}: \quad \Box((\neg \texttt{new} \wedge \bigcirc \texttt{new}) \rightarrow \bigcirc p_2) \tag{3}$$

Verification result of Formula 1: no counterexample.

Maude found a counterexample for Formula 2. The snippet of the counterexample:

```
...{(loc: noReturn)...(gate: false),update}
   {(loc': noReturn)...(gate': false),closeGate}...
```

Update takes place when the RailCab is at `noReturn` point.

Refinement of the dynamic update (I)

Exclude the states where the RailCab is at the `noReturn` point from the update points:

```
crl [apply]: (req: true) (cabLoc : L) OS:OldState =>
             f((cabLoc : L) OS)
if L =/= noReturn .
```

Model checking result with the refined dynamic update:

Formula 1: no counterexample

Formula 2: no counterexample

A counterexample is found for Formula 3:

```
...{(loc: lastBrake)...(gate:false),update}{(loc':lastBrake)
    ...(gate': false)(appRes:unknown),move2leb}
    {loc': lastEmergencyBrake)...(appRes: unknown),eBrake}
    ... {...,deadlock}
```

Refinement of the dynamic update (II)

Exclude the states where the RailCab is at the `lastBrake` point from the update points:

```
crl [apply]: (req: true) (cabLoc : L) OS:OldState =>
             f((cabLoc : L) OS)
if L =/= noReturn and L =/= lastBrake .
```

Model checking result with the refined dynamic update:

Formula 1: no counterexample

Formula 2: no counterexample

A counterexample is found for Formula 3:

```
...{(loc: lastEmergencyBrake)...(gate:false),update}{(loc':
    lastEmergencyBrake)...(gate': false)(appRes:unknown),
    eBrake}... {...,deadlock}
```

Refinement of the dynamic update (III)

Exclude the states where the RailCab is at the `lastBrake` point from the update points:

```
crl [apply]: (req: true) (cabLoc : L) OS:OldState =>
             f((cabLoc : L) OS)
if L =/= noReturn and L =/= lastBrake and
   L =/= lastEmergencyBrake .
```

Model checking result with the refined dynamic update:

- Formula 1: no counterexample
- Formula 2: no counterexample
- Formula 3: no counterexample

Another refinement

The reason for the counterexample is that when the RailCab is at `lastBrake` or `lastEmergencyBrake` location, the value of `appRes` is initialized to be `unknown`, regardless of the status of gate.

No chance for the RailCab to send the second message to check the status.

We can initialize `appRes` according to the status of the gate when the RailCab is at the `lastBrake` or `lastEmergencyBrake`.

```
op f' : OldState -> NewState .
ceq f'((gate: B)(conLoc: L1)(cabLoc: L2)(chan1: CH1)
       (chan2: CH2)(cabSta: S)(pass: P)) =
       (gate': B)(conLoc': L1)(cabLoc': L2)(chan1': CH1)
       (chan2':CH2)(cabSta': S)(pass': P)
       (appRes: if B then close else unknown)
  if L2 = lastBrake or L2 = lastEmergencyBrake .
```

No counterexample is found for the three formulas even if the update takes update when the RailCab is at `lastBrake` or `lastEmergencyBrake`.

Lessons learned from the case study

With the proposed formal approach, we can design a dynamic update

1. identifying a set of update points
2. defining an appropriate state transformation function
3. the dynamic update is verified to satisfy a set of desired properties
4. state transformation and update points depend on each other

Some limitations and possible extensions

The proposed approach is suited

1. to systems with finite state space
2. to closed systems (no interaction with environment)
3. to instantaneous dynamic update

Some possible extensions

1. for infinite-state system: abstraction or narrowing-based model checking
2. for open systems which need to interact the environment: Labeled Kripke structure
3. to incremental dynamic update: to formalize the synchronization mechanism between the old and new systems during updating.

Conclusion and future work

A formal approach to the design of dynamic update:

1. Instantaneous and incremental dynamic update
2. Property-based correctness of dynamic update
3. Counterexample-guided design of dynamic update
4. The dependency between update points and state transformation

Future work: to extend the proposed approach

1. for infinite-state system
2. for open systems which need to interact the environment
3. for incremental dynamic update

Q&A

ありがとうございました！