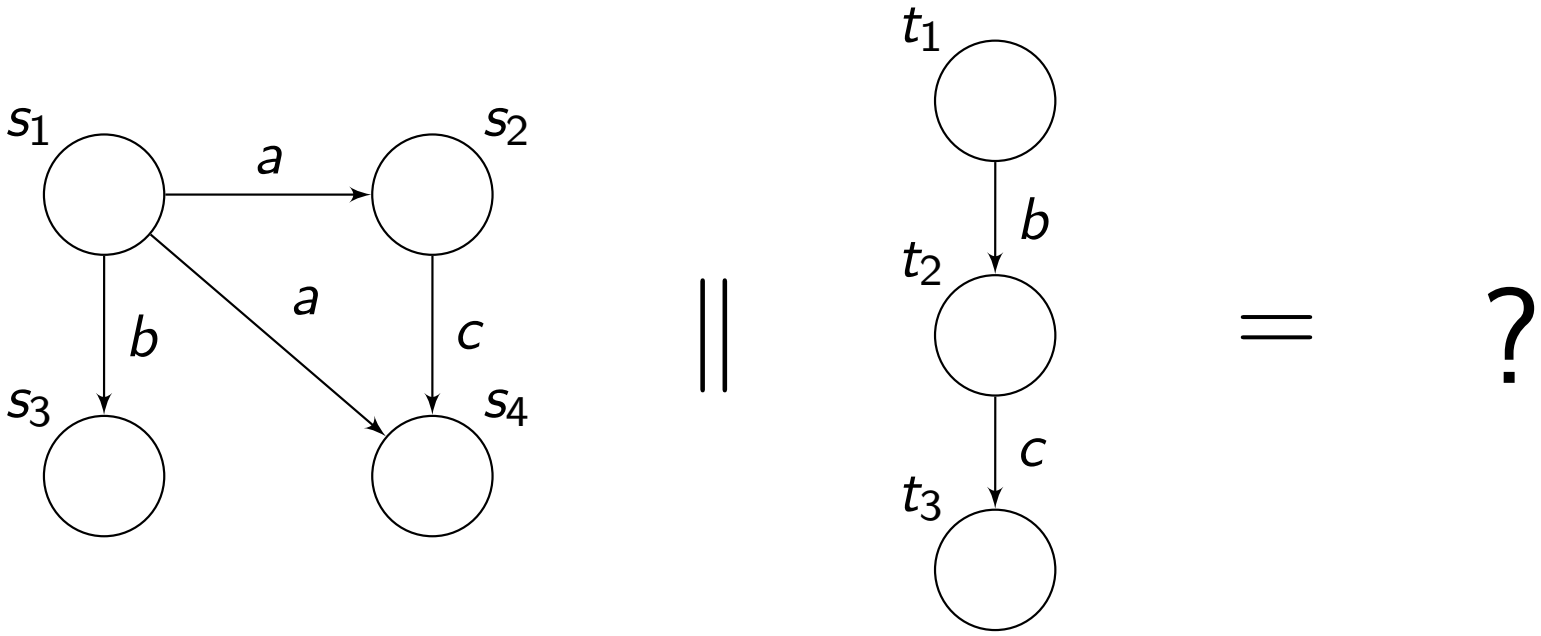
Synchronous products of rewrite systems

▶ Óscar Martín ◀
Alberto Verdejo
Narciso Martí-Oliet
Univ. Complutense de Madrid

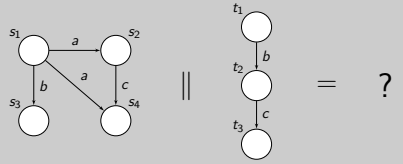
ATVA 2016
Chiba

Aim: modular specification in rewriting logic



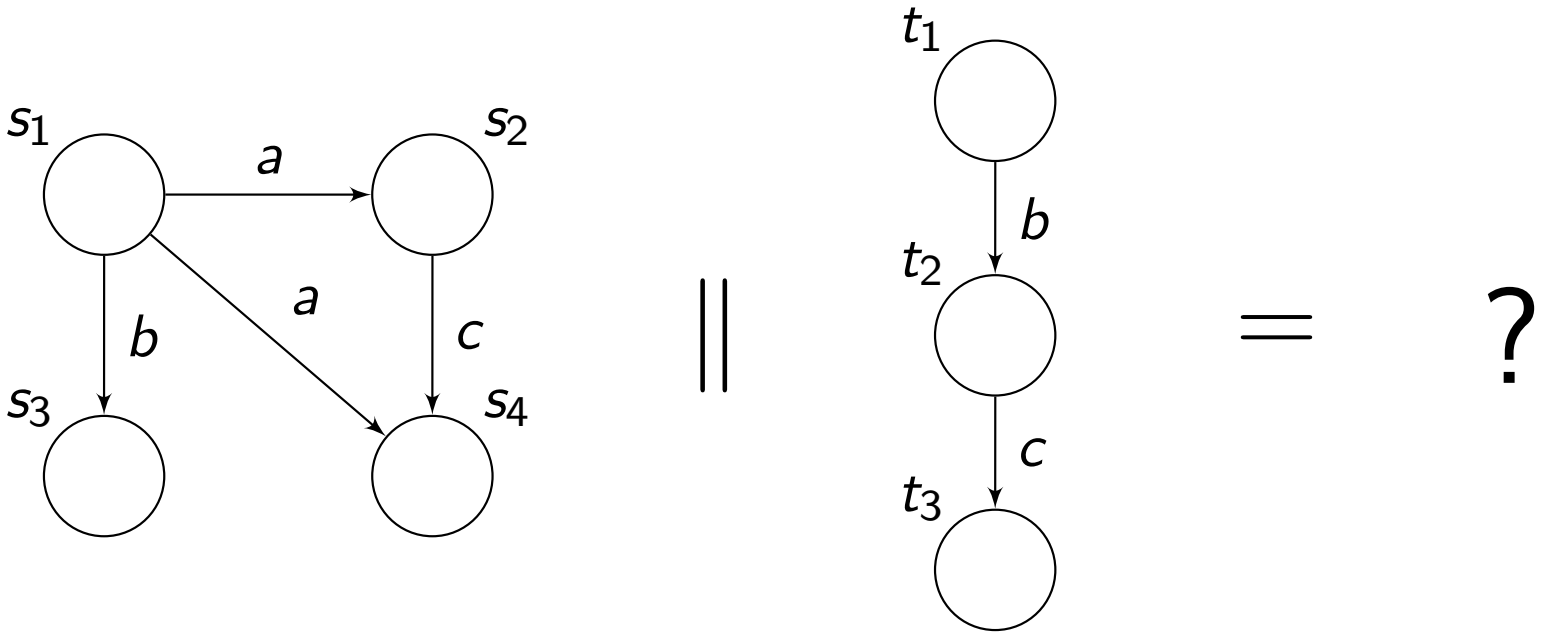
Aim: modular specification in rewriting logic

Aim: modular specification in rewriting logic



The idea is a well-known one: we are given the description of two systems in some way and we want to figure out what it means to compose them, to make them evolve in parallel and synchronized. It is the concept called *synchronous product* in automata theory; and the same concept called *parallel composition* of processes in process algebras.

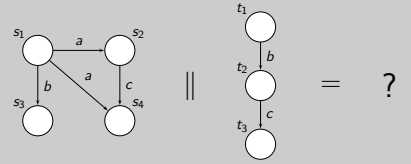
Aim: modular specification in rewriting logic



Rewriting logic (and Maude)

Aim: modular specification in rewriting logic

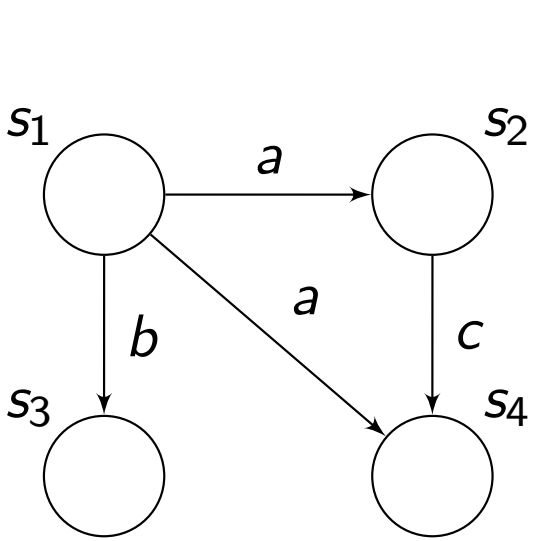
Aim: modular specification in rewriting logic



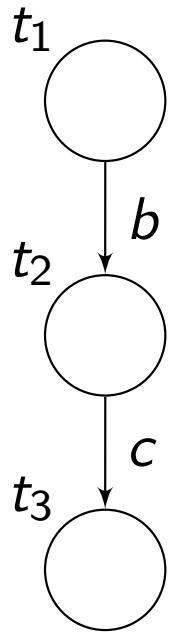
Rewriting logic (and Maude)

Our aim is to take to concept to rewriting logic. Rewriting logic is a very expressive formalism for the specification of systems that, up to now, lacks a concept of parallel composition of modules. Maude is the language based on rewriting logic that we use in our work.

Aim: modular specification in rewriting logic



$\|$



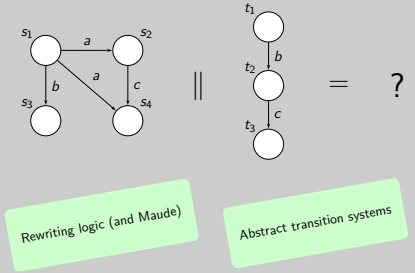
$=$?

Rewriting logic (and Maude)

Abstract transition systems

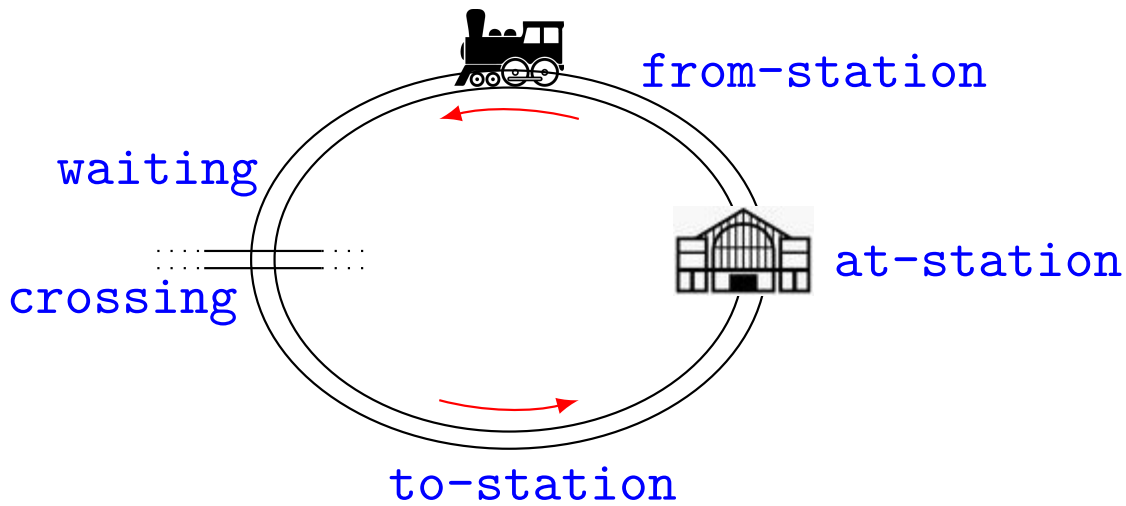
Aim: modular specification in rewriting logic

Aim: modular specification in rewriting logic

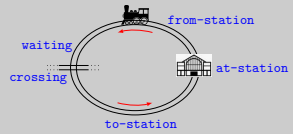


We also want to be somewhat more abstract and define the same concept for a suitable kind of transition systems that I will make explicit later.

This is a very simple Maude module



This is a very simple Maude module

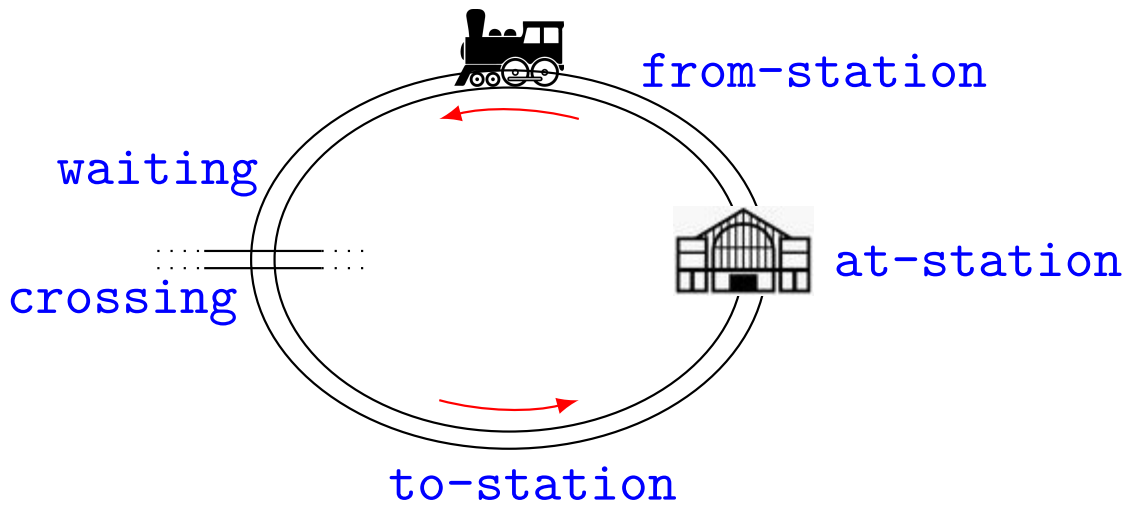


I will use my first few slides to go deeper into this motivation and, at the same time, refreshing you on rewriting logic and Maude, in case you need it.

This is a very simple system. It consists of a train that goes around a circular railway. We have identified five states on it: being at the station, coming from the station, waiting before the crossing, crossing, and going back to the station.

This is a very simple Maude module

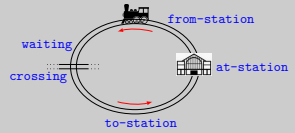
```
mod RAILWAY is
   sort State .
   ops waiting crossing to-station
       at-station from-station : -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```



This is a very simple Maude module

This is a very simple Maude module

```
mod RAILWAY is
   sort State .
   ops waiting crossing to-station
       at-station from-station : -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```



This is the Maude module that models such system. Its name is `RAILWAY`. We start declaring a sort (or type) `State` and five constants of that sort to represent the five states of the system. Then, there are five rules. Rules describe the dynamics of the system. For instance, the first one, whose label is `wc`, allows the system to move from state `waiting` to state `crossing`. Usually, there are also equations in a Maude module.

A little bit more complex

```
mod RAILWAYx2 is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

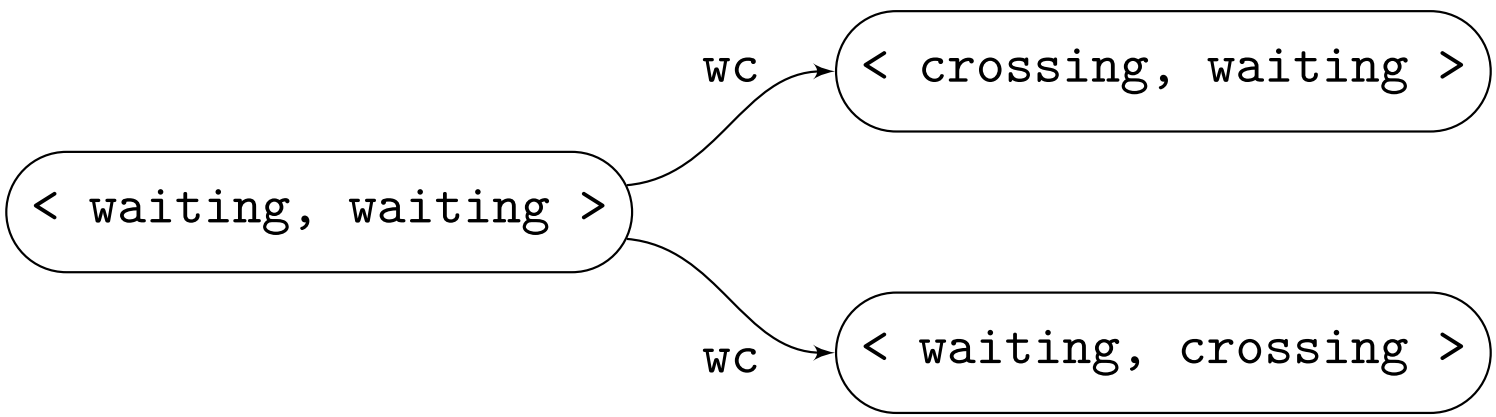
A little bit more complex

```
mod RAILWAYx2 is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

As there is a crossing with another railway in the previous system, we want to model that railway as well. We need two sorts now, `TrainState` for the state of each individual train, and `State` for the global state. We have the same five constants as before, but now they are of sort `TrainState`. There is a new operator for building terms of sort `State`. And we have exactly the same five rules as before.

A little bit more complex

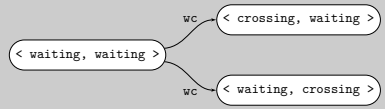
```
mod RAILWAYx2 is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```



A little bit more complex

A little bit more complex

```
mod RAILWAYx2 is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   rl [wc] : waiting => crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```



The same five rules work because, in rewriting logic, rules can be applied to subterms. For instance, starting from a state `< waiting, waiting >`, the rule labeled `wc` can be applied either to the first or to the second component of the global state.

Adding mutual exclusion

```
mod SAFE-RAILWAYS is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   var T : TrainState .
   crl [t1wc] : < waiting, T > => < crossing, T >
                if T =/= crossing .
   crl [t2wc] : < T, waiting > => < T, crossing >
                if T =/= crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

Adding mutual exclusion

Adding mutual exclusion

```
mod SAFE-RAILWAYS is
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_> : TrainState TrainState -> State .
   var T : TrainState .
   crl [t1wc] : < waiting, T > => < crossing, T >
                if T =/= crossing .
   crl [t2wc] : < T, waiting > => < T, crossing >
                if T =/= crossing .
   rl [ct] : crossing => to-station .
   rl [ta] : to-station => at-station .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

We need to ensure somehow that both trains do not go into the crossing at the same time. We can do so by adding some control in the rule that allows trains into the crossing. For that, we need the whole state to appear in that rule, so that we need to split rule `wc` into two, one to control each train.

Counting stations and getting a mess

```
mod COUNTING-SAFE-RAILWAYS is
   including NAT .
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_,_> : TrainState TrainState Nat -> State .
   var T : TrainState .
   var N : Nat .
   crl [t1wc] : < waiting, T, N > => < crossing, T, N >
                if T =/= crossing .
   crl [t2wc] : < T, waiting, N > => < T, crossing, N >
                if T =/= crossing .
   rl [ct] : crossing => to-station .
   rl [t1ta] : < to-station, T, N >
            => < at-station, T, N + 1 > .
   rl [t2ta] : < T, to-station, N >
            => < T, at-station, N + 1 > .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

Counting stations and getting a mess

```
mod COUNTING-SAFE-RAILWAYS is
   including NAT .
   sorts TrainState State .
   ops waiting crossing to-station
       at-station from-station : -> TrainState .
   op <_,_,_> : TrainState TrainState Nat -> State .
   var T : TrainState .
   var N : Nat .
   crl [t1wc] : < waiting, T, N > => < crossing, T, N >
                if T =/= crossing .
   crl [t2wc] : < T, waiting, N > => < T, crossing, N >
                if T =/= crossing .
   rl [ct] : crossing => to-station .
   rl [t1ta] : < to-station, T, N >
            => < at-station, T, N + 1 > .
   rl [t2ta] : < T, to-station, N >
            => < T, at-station, N + 1 > .
   rl [af] : at-station => from-station .
   rl [fw] : from-station => waiting .
endm
```

Let's add some other complexity: a counter for the number of times the trains stop at their stations. For that we need to add a natural number to the state. And we need rule `ta` to use the whole state, so that we need to split it into two, one for each train.

This is starting to look messy, even though the system is still quite simple. This is the usual problem with monolithic specifications.

The (well-known) solution is modularity

```
mod RAILWAY1 is
   sort State .
   ops w c t a f : -> State .
   rl [t1wc] : w => c .
   rl [t1ct] : c => t .
   rl [t1ta] : t => a .
   rl [t1af] : a => f .
   rl [t1fw] : f => w .
endm
```

```
mod RAILWAY2 is
   sort State .
   ops w c t a f : -> State .
   rl [t2wc] : w => c .
   rl [t2ct] : c => t .
   rl [t2ta] : t => a .
   rl [t2af] : a => f .
   rl [t2fw] : f => w .
endm
```

```
mod MUTEX is
   sort State .
   ops one none : -> State .
   rl [t1wc] : none => one .
   rl [t2wc] : none => one .
   rl [t1ct] : one => none .
   rl [t2ct] : one => none .
endm
```

```
mod COUNTER is
   inc NAT .
   sort State .
   subsort Nat < State .
   rl [t1ta] : N => N + 1 .
   rl [t2ta] : N => N + 1 .
endm
```

The (well-known) solution is modularity

The (well-known) solution is modularity

```
mod RAILWAY1 is
   sort State .
   ops w c t a f : -> State .
   rl [t1wc] : w => c .
   rl [t1ct] : c => t .
   rl [t1ta] : t => a .
   rl [t1af] : a => f .
   rl [t1fw] : f => w .
endm
```

```
mod RAILWAY2 is
   sort State .
   ops w c t a f : -> State .
   rl [t2wc] : w => c .
   rl [t2ct] : c => t .
   rl [t2ta] : t => a .
   rl [t2af] : a => f .
   rl [t2fw] : f => w .
endm
```

```
mod MUTEX is
   sort State .
   ops one none : -> State .
   rl [t1wc] : none => one .
   rl [t2wc] : none => one .
   rl [t1ct] : one => none .
   rl [t2ct] : one => none .
endm
```

```
mod COUNTER is
   inc NAT .
   sort State .
   subsort Nat < State .
   rl [t1ta] : N => N + 1 .
   rl [t2ta] : N => N + 1 .
endm
```

This is the usual solution: modular, or component-wise, specification. Our framework does not allow for parametric specifications. Thus, we need the two trains modeled by two separate modules. They have rules with different labels, but are otherwise equal.

The control needed for mutual exclusion is in module `MUTEX`. It has only two states, representing whether one or no trains are crossing. Note that the rule labels in this module are the same that appear in the railway systems. This is the whole point: rules with the same label can only be run synchronously, at the same time. Thus, rule `t1wc`, that allows train one to get into the crossing, can only be run when `MUTEX` is in state `none`. As that rule label does not occur in the other two systems, they have no concern about it.

The module `COUNTER` deals just with counting stops.

Each rule in `MUTEX`, and also in `COUNTER`, is somewhat duplicated: once for each railway system.

This is how our implementation is used

```
mod COMPOSED-MODULE is
   including ((RAILWAY1 || RAILWAY2) || MUTEX) || COUNTER .
   ...
   ... < < < crossing, T >, M >, N > ...
   ...
endm
```

This is how our implementation is used

This is how our implementation is used

```
mod COMPOSED-MODULE is
   including ((RAILWAY1 || RAILWAY2) || MUTEX) || COUNTER .
   ...
   ... < < < crossing, T >, M >, N > ...
   ...
endm
```

We have implemented on Maude the operator ||, that takes two Maude modules and produces one representing their synchronous product. Such composition can be imported in other module as shown and, then, composed states can be used.

This is how our implementation is used

```
mod COMPOSED-MODULE is
   including ((RAILWAY1 || RAILWAY2) || MUTEX) || COUNTER .
   ...
   ... < < < crossing, T >, M >, N > ...
   ...
endm


rl [t1wc] : < < < waiting, T >, none >, N >
         => < < < crossing, T >, one >, N > .
```

This is how our implementation is used

This is how our implementation is used

```
mod COMPOSED-MODULE is
   including ((RAILWAY1 || RAILWAY2) || MUTEX) || COUNTER .
   ...
   ... < < < crossing, T >, M >, N > ...
   ...
endm

rl [t1wc] : < < < waiting, T >, none >, N >
        => < < < crossing, T >, one >, N > .
```

Internally, the || operator generates rules like the one shown, with the same label taken from the components, and showing the synchronized changes. The user does not see this rule, nor needs it. The user only needs to note which rules have equal labels.

The other half: syncing also on states

Precedents:

- L^2TS (De Nicola & Vaandrager, 1995)
- ESTL (Kindler & Vesper, 1998)
- TLR* (Meseguer, 2008)
- egalitarian structures (ourselves, 2016)

The other half: syncing also on states

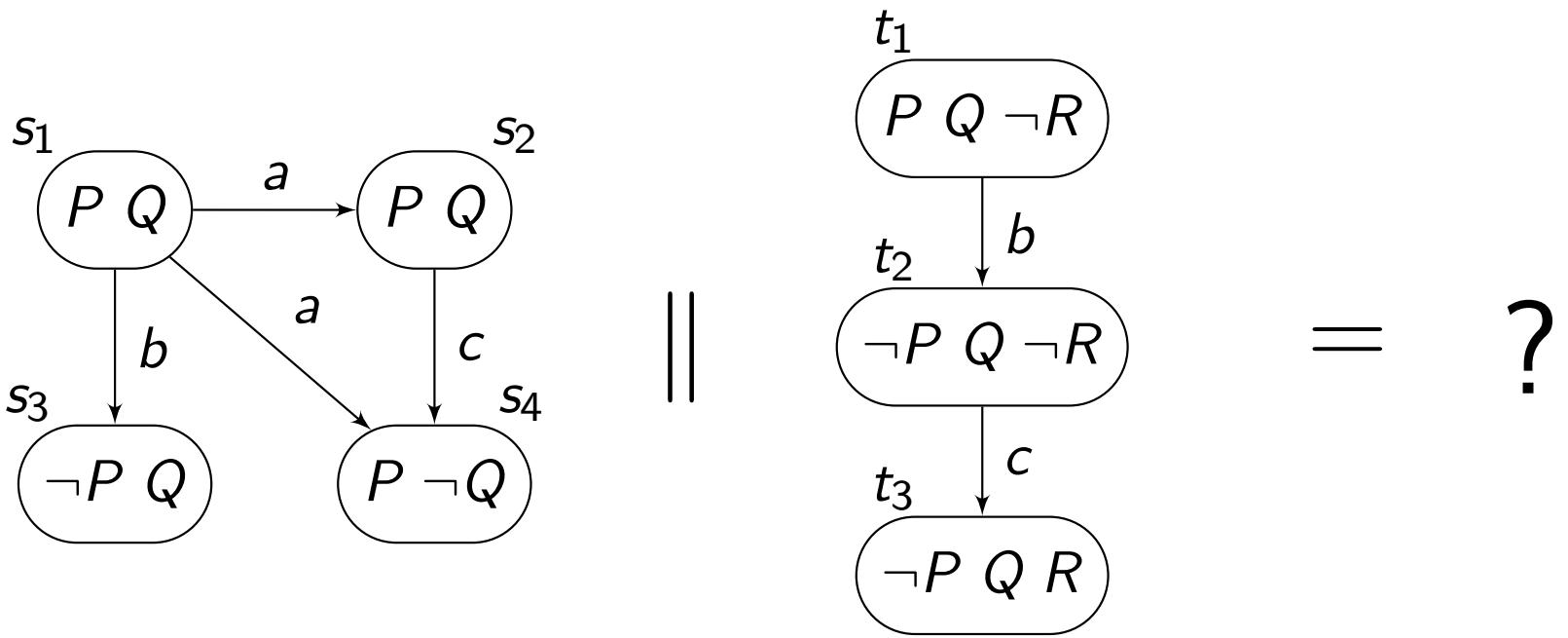
The synchronizations shown up to now are all on actions. But since the start, we wanted to synchronize our systems using both transitions and states.

I have listed a few of the precedents on using transitions and states. The two in the middle are temporal logics; the other two are transition structures. The important thing about these proposals is that they include rationals on why it is convenient to be able to refer to states and transitions at the same time. For instance, some temporal properties of systems are more naturally and simply expressed by formulas that can use both. We think that the same is true for synchronization purposes.

The other half: syncing also on states

Precedents:

- L^{2}TS (De Nicola & Vaandrager, 1995)
- ESTL (Kindler & Vesper, 1998)
- TLR* (Meseguer, 2008)
- egalitarian structures (ourselves, 2016)

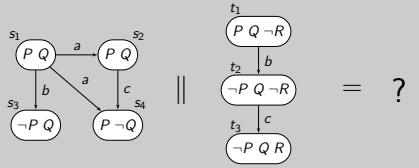


The other half: syncing also on states

The other half: syncing also on states

Precedents:

- L²TS (De Nicola & Vaandrager, 1995)
- ESTL (Kindler & Vesper, 1998)
- TLR* (Meseguer, 2008)
- egalitarian structures (ourselves, 2016)



These are L²TSs, doubly labeled transition structures. Their transitions are labeled by atomic action identifiers, and their states are labeled by the atomic propositions they satisfy. These are the kind of transition systems useful to us, as I will explain shortly. So the question is how to synchronize this kind of systems. The short answer is: transitions by equal labels; states by agreement on their common propositions.

Syncing on states

```
mod RAILWAYS-EXT is
   including SATISFACTION .
   including RAILWAY1 || RAILWAY2 .
   op safe : -> Prop .
   var S : State .
   eq < crossing, crossing > |= safe = false .
   eq S |= safe = true [owise] .
endm
```

```
mod MUTEX2 is
   including SATISFACTION .
   op o : -> State .
   op safe : -> Prop .
   eq o |= safe = true .
endm
```

```
Syncing on states
  mod RAILWAYS-EXT is
    including SATISFACTION .
    including RAILWAY1 || RAILWAY2 .
    op safe : -> Prop .
    var S : State .
    eq < crossing, crossing > |= safe = false .
    eq S |= safe = true [owise] .
  endm
  mod MUTEX2 is
    including SATISFACTION .
    op o : -> State .
    op safe : -> Prop .
    eq o |= safe = true .
  endm
```

As an example of synchronization on states, this is an implementation of the mutual exclusion control using states. As this synchronization is on propositions, we need to extend (*not* modify) the two-railways system. We define on it the proposition `safe`, as true everywhere except when both trains are crossing at the same time. Then we specify a new module `MUTEX2` with an only state `o` and a proposition also with the name `safe` that is always true.

States from the component systems can only be visited at the same time if they agree on their common propositions. The only common proposition is `safe`, and it is always true in the unique state of `MUTEX2`, so it must be also true in all states visited in the two railways. That is, a state with both trains crossing cannot be visited.

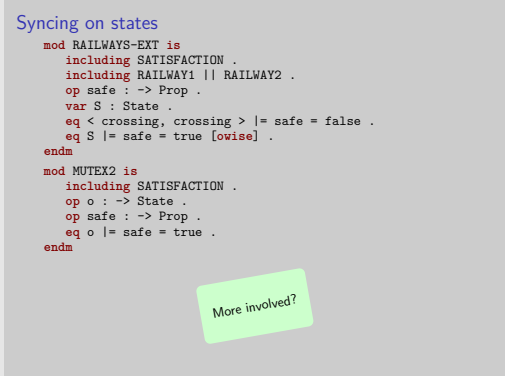
Syncing on states

```
mod RAILWAYS-EXT is
   including SATISFACTION .
   including RAILWAY1 || RAILWAY2 .
   op safe : -> Prop .
   var S : State .
   eq < crossing, crossing > |= safe = false .
   eq S |= safe = true [owise] .
endm
```

```
mod MUTEX2 is
   including SATISFACTION .
   op o : -> State .
   op safe : -> Prop .
   eq o |= safe = true .
endm
```

More involved?

Syncing on states



This can be seen as more involved. The idea is not: it only says that the state `<crossing, crossing>` cannot be visited. But the coding probably is.

Syncing on states

```
mod RAILWAYS-EXT is
   including SATISFACTION .
   including RAILWAY1 || RAILWAY2 .
   op safe : -> Prop .
   var S : State .
   eq < crossing, crossing > |= safe = false .
   eq S |= safe = true [owise] .
endm
```

```
mod MUTEX2 is
   including SATISFACTION .
   op o : -> State .
   op safe : -> Prop .
   eq o |= safe = true .
endm
```

More suitable

Syncing on states

```
mod RAILWAYS-EXT is
   including SATISFACTION .
   including RAILWAY1 || RAILWAY2 .
   op safe : -> Prop .
   var S : State .
   eq < crossing, crossing > |= safe = false .
   eq S |= safe = true [owise] .
endm
mod MUTEX2 is
   including SATISFACTION .
   op o : -> State .
   op safe : -> Prop .
   eq o |= safe = true .
endm
```

More suitable

But, more importantly, this is more correct. If, for instance, one of the trains could go backwards, and enter the crossing like this, the former control on actions would need to be modified to account for the new actions, but this control would still be valid.

Used in this way

```
mod COMPOSED-MODULE-2 is
   including (RAILWAYS-EXT || MUTEX2) || COUNTER .
   ...
endm
```

```
crl [t1wc] : < < < waiting, T >, M >, N >
          => < < < crossing, T >, M >, N >
          if compatible(< crossing, T >, M) .
```

Used in this way

Used in this way

```
mod COMPOSED-MODULE-2 is
   including (RAILWAYS-EXT || MUTEX2) || COUNTER .
   ...
endm

crl [t1wc] : < < < waiting, T >, M >, N >
          => < < < crossing, T >, M >, N >
          if compatible(< crossing, T >, M) .
```

The rules generated in this case look like this, where the function `compatible` tests whether the states agree on their common propositions.

From rewrite systems to L^{2}TSs

$(\Sigma, E \cup Ax, R) \quad \longmapsto \quad (S, \Lambda, \to, \mathrm{AP}, L)$:

- $S := T_{\Sigma/E \cup Ax, \mathtt{State}}$;
- Λ is the set of rule labels in R;
- $s \xrightarrow{\lambda} s'$ iff there is a rule in R with label λ that allows rewriting s to s' in one step;
- $\mathrm{AP} := T_{\Sigma/E \cup Ax, \mathtt{Prop}}$;
- $L(s) := \{p \in \mathrm{AP} \mid s \models p = \mathtt{true} \text{ modulo } E \cup Ax\}$.

From rewrite systems to L^{2}TSs

From rewrite systems to L^{2}TSs

$(\Sigma, E \cup Ax, R) \longmapsto (S, \Lambda, \rightarrow, \mathsf{AP}, L)$:

- $S := T_{\Sigma/E \cup Ax, \mathtt{State}}$;
- Λ is the set of rule labels in R;
- $s \xrightarrow{\lambda} s'$ iff there is a rule in R with label λ that allows rewriting s to s' in one step;
- $\mathsf{AP} := T_{\Sigma/E \cup Ax, \mathtt{Prop}}$;
- $L(s) := \{p \in \mathsf{AP} \mid s \models p = \mathtt{true} \text{ modulo } E \cup Ax\}$.

I want to be a little more formal in the final slides.
There is a natural translation from rewrite systems to L^{2}TSs (that can be seen as a semantics function):

- the set of states is given by the terms of sort `State` in the rewrite system;
- the alphabet of actions is the set of rule labels;
- a transition is given by a rewrite rule as written in the slide;
- the set of atomic propositions is given by the terms of sort `Prop` in the rewrite system;
- the labeling is given by the equations E as explained in the slide.

The synchronous product of L^2TSs

$$(S, \Lambda, \to, \mathrm{AP}, L) := (S_1, \Lambda_1, \to_1, \mathrm{AP}_1, L_1) \parallel (S_2, \Lambda_2, \to_2, \mathrm{AP}_2, L_2)$$

- $S := (S_1 \times S_2) \cap \approx_{\mathrm{AP}_1 \cap \mathrm{AP}_2}$;
- $\Lambda := \Lambda_1 \cup \Lambda_2$;
- regarding transitions:
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2' \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $s_2 \xrightarrow{\lambda} s_2'$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2 \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $\lambda \notin \Lambda_2$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1, s_2' \rangle$ iff $s_2 \xrightarrow{\lambda} s_2'$ and $\lambda \notin \Lambda_1$;
- $\mathrm{AP} := \mathrm{AP}_1 \cup \mathrm{AP}_2$;
- $L(\langle s_1, s_2 \rangle) := L_1(s_1) \cup L_2(s_2)$.

The synchronous product of L^2TSs

The synchronous product of L^2TSs

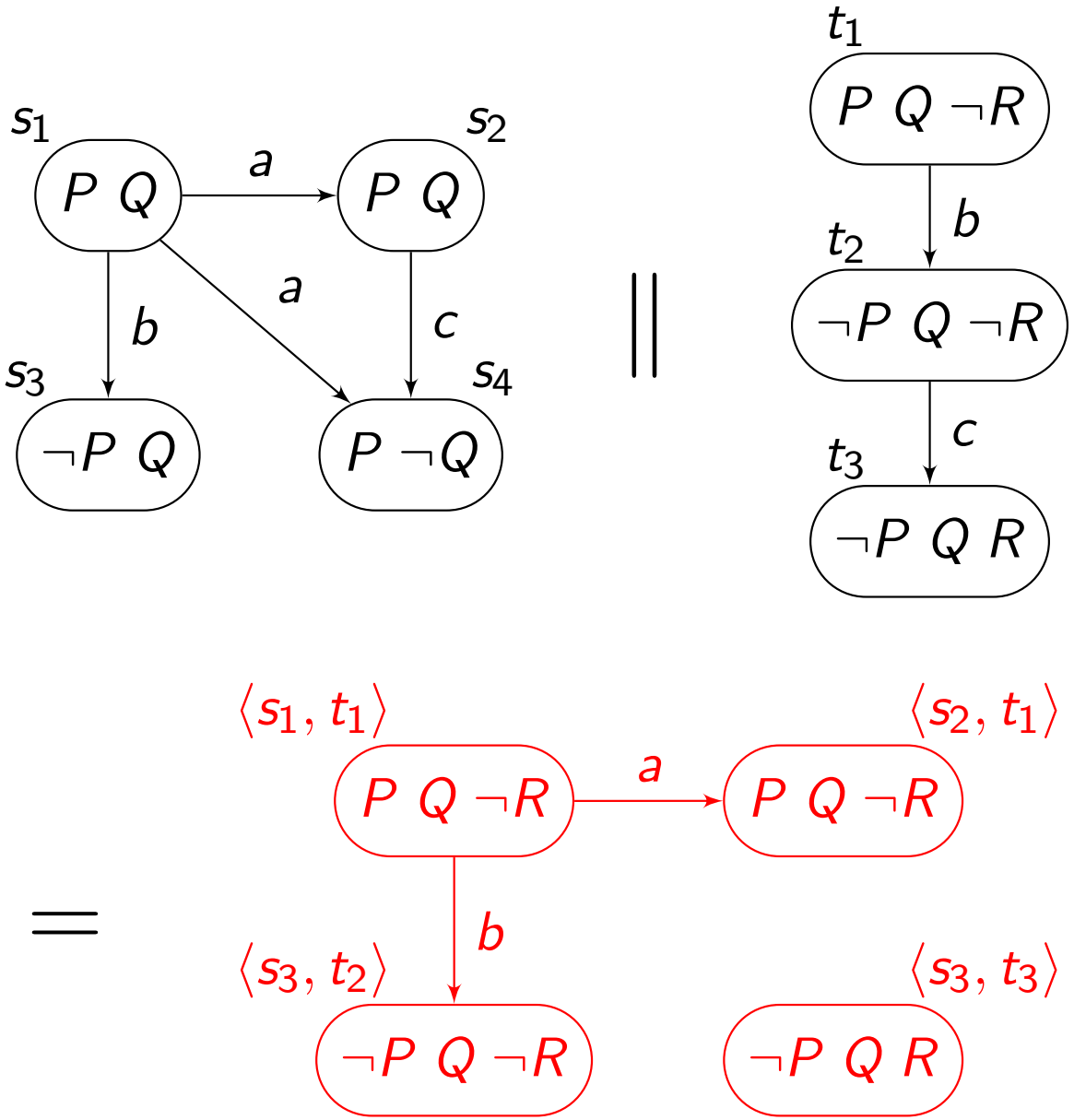
$(S, \Lambda, \to, AP, L) := (S_1, \Lambda_1, \to_1, AP_1, L_1) \| (S_2, \Lambda_2, \to_2, AP_2, L_2)$

- $S := (S_1 \times S_2) \cap \approx_{AP_1 \cap AP_2}$;
- $\Lambda := \Lambda_1 \cup \Lambda_2$;
- regarding transitions:
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2' \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $s_2 \xrightarrow{\lambda} s_2'$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1', s_2 \rangle$ iff $s_1 \xrightarrow{\lambda} s_1'$ and $\lambda \notin \Lambda_2$,
  - $\langle s_1, s_2 \rangle \xrightarrow{\lambda} \langle s_1, s_2' \rangle$ iff $s_2 \xrightarrow{\lambda} s_2'$ and $\lambda \notin \Lambda_1$;
- $AP := AP_1 \cup AP_2$;
- $L(\langle s_1, s_2 \rangle) := L_1(s_1) \cup L_2(s_2)$.

The definition of the synchronous product for rewrite systems happens to be too complex and long, due to the large number of cases and points that need to be taken into account. The same is true for its implementation. It is all in the paper. But the idea is simple and can be better appreciated with L^2TSs:

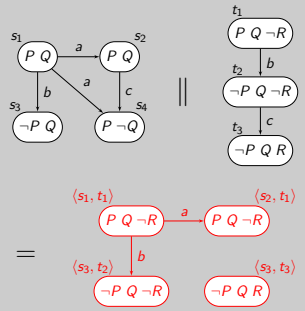
- the set of states is a subset of the Cartesian product, given by the compatibility relation $\approx$ on the common propositions;
- the alphabet of actions is the union of the individual alphabets;
- transitions are possible either if the same action is possible in both systems at the same moment or if an action is available in one system and does not exist in the alphabet of the other;
- atomic propositions and labeling as shown in the slide.

The synchronous product of L^{2}TSs: example



The synchronous product of L^2TSs: example

The synchronous product of L^2TSs: example



- State s_1 does not agree on P with t_2 nor t_3, so it only gives rise to $\langle s_1, t_1 \rangle$.
- Action b must be executed in both systems at the same time.
- Action a can be executed in the left system alone, as the right system does not care about it.
- Action c is present in both systems, but there is no way to execute it at the same time in both. Thus, c cannot happen in the composed system. (Note, by the way, that action c *is* in the alphabet of actions of the resulting system, even though it is not shown in the drawing.)

Future work

- Generalization: complex transition terms.
- Modular verification.
- Implementation of strategies.
- Explore intersections with:
  - runtime verification,
  - coordination models,
  - aspect-oriented programming,
  - behavioral programming,
  - DES,
  - ...

Future work

Future work

- Generalization: complex transition terms.
- Modular verification.
- Implementation of strategies.
- Explore intersections with:
  - runtime verification,
  - coordination models,
  - aspect-oriented programming,
  - behavioral programming,
  - DES,
  - ...

- The first thing on which we are working is generalizing all the above in several directions. The main one is treating transitions and states as equals. It is unfortunate that the only information we can use from transitions is the action identifier, while for states we have complex terms representing them and propositions defined on them. We want to find an appropriate term representation for transitions as well, and use it for synchronization.
- Modular verification seems to be a natural next step.
- Strategies, regarding non-deterministic systems, are a way to guide the system, restricting its capabilities, so as to get the system behave in certain ways and not others. We want to explore to what extent strategies can be implemented as rewrite systems that exert their control through synchronous products.
- Finally, it seems worth exploring the intersections of all the above with existing fields like the ones in the slide.

Thank you.

Have a good day.