Exercise 1
Railway Safety Invariants

Exercise in UML-B Class and Context Diagram modelling

Colin Snook

Specification:
Railway Interlocking Safety Requirements

A Railway interlocking system controls trains passing through a track layout by changing the state of Signals which can be Proceed, Warning and Stop.

The signal immediately before another signal is said to be RearOf the second signal.

The track layout is divided into Routes. Each Route has an Entry signal at its start.

Some Routes Conflict with others (e.g. use the same section of track). A route is locked before it is used and then unlocked again.

The following safety requirements are specified:

SR1 - If a signal shows Stop, the signal RearOf it must show Stop or Warning

SR2 - If the entry signal of a route shows Proceed or Warning, then the Route is locked

SR3 - If a route is locked then no route that conflicts with it is locked

Instructions:
Railway Interlocking Safety Requirements (cont.)

Model this domain in just enough detail to be able to express the safety requirements.

Use a UML-B Context diagram for the static parts and a Class Diagram for the varying parts. (Link the Classes to the ClassTypes using the Instances property of the Class).

Add invariants to your model to reflect these requirements.

Add guards to your events to ensure the system does not violate the invariants.

Verify the model using the prover

Analysis

Our aim is to keep the model as simple as possible. We just want enough detail to be able to express the safety requirements as invariants and no more.

Looking at SR1 we need to model a set of Signals. Signals have exactly one associated RearOf signal. This is a constant* so we should model it in a Context Diagram (That means we will have to make Signal a ClassType in a Context Diagram).

We need to talk about the state of Signals (called 'aspect' in railway jargon) so we will need to define an enumerated type. We can do this with a Class Type that has instances set to {Proceed,Warning,Stop}.

Since the aspect of the Signal varies we will need to model that in a Class Diagram. So we need a Class signal that is linked to the ClassType Signal . We will need events to set each state. (Note that it is better to model separate setter events for each state so that we can put different guards on each of them)

---

* Actually rearOf might depend on the which routes are currently locked but for simplicity we assume it is a constant for now

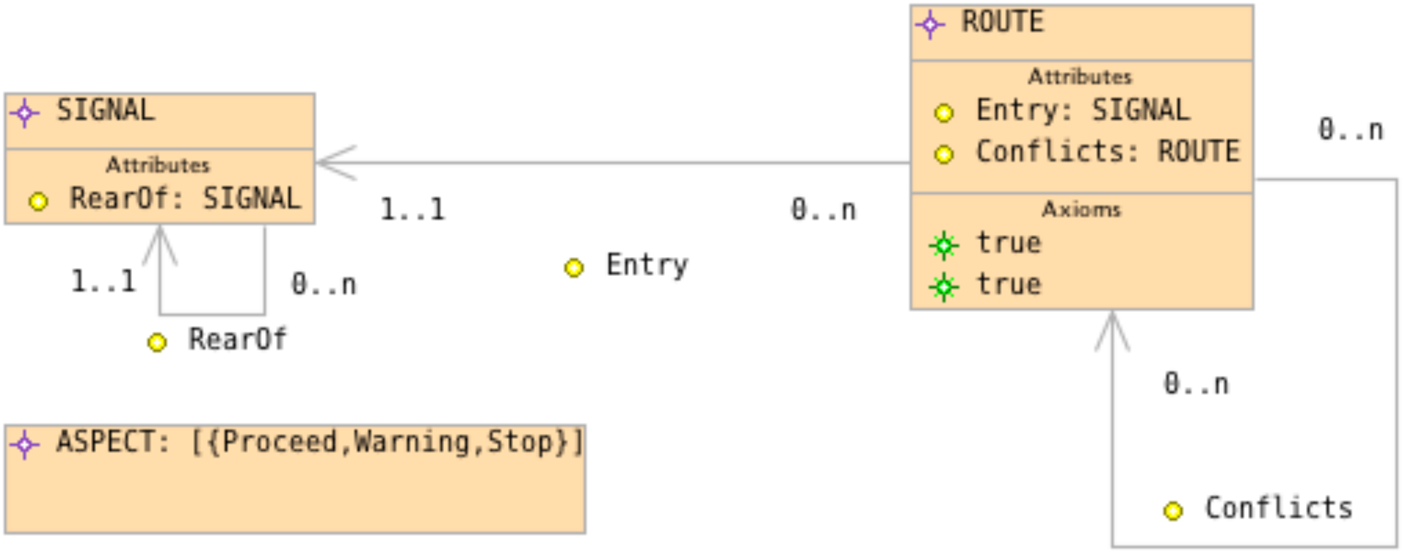
Analysis (cont.)

From SR2 we need to model a set of Routes. Routes have an associated Entry Signal which we can model as an association. Again this is a constant, so we will put it in a ClassType on the Context Diagram. Routes can be locked and unlocked. We could model that as a boolean variable attribute, ‘locked’, in a linked Class and we will need lock and unlock events to change it.

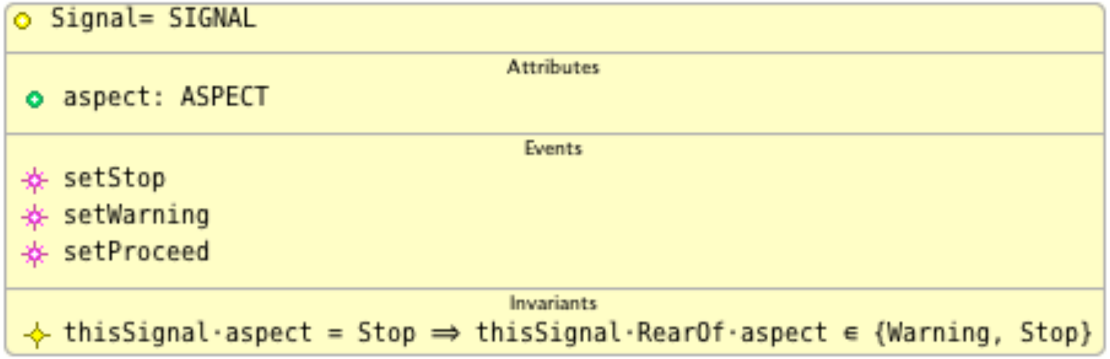
Finally, from SR3 routes may be in conflict with other routes. This is a constant association from Routes to Routes so we should put it as a ‘self’ loop in the ClassType for Routes. Since each route may have none or many conflicting routes we will make this a multiplicity many association (i.e. relation).

* Actually rearOf might depend on the which routes are currently locked but for simplicity we assume it is a constant for now

Context Diagram



Class Diagram



Signal= SIGNAL

Attributes

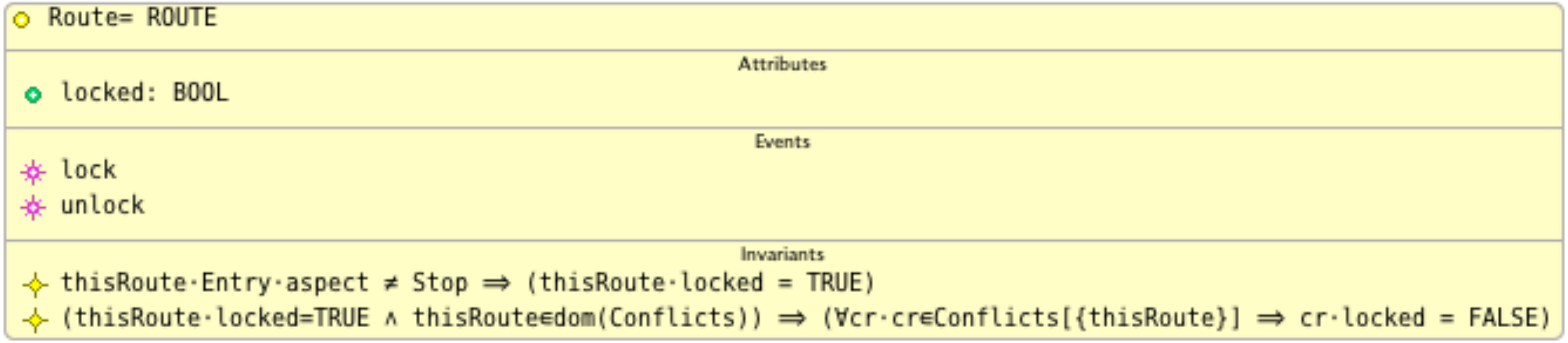
aspect: ASPECT

Events

setStop
setWarning
setProceed

Invariants

thisSignal·aspect = Stop ⇒ thisSignal·RearOf·aspect ∈ {Warning, Stop}



Route= ROUTE

Attributes

locked: BOOL

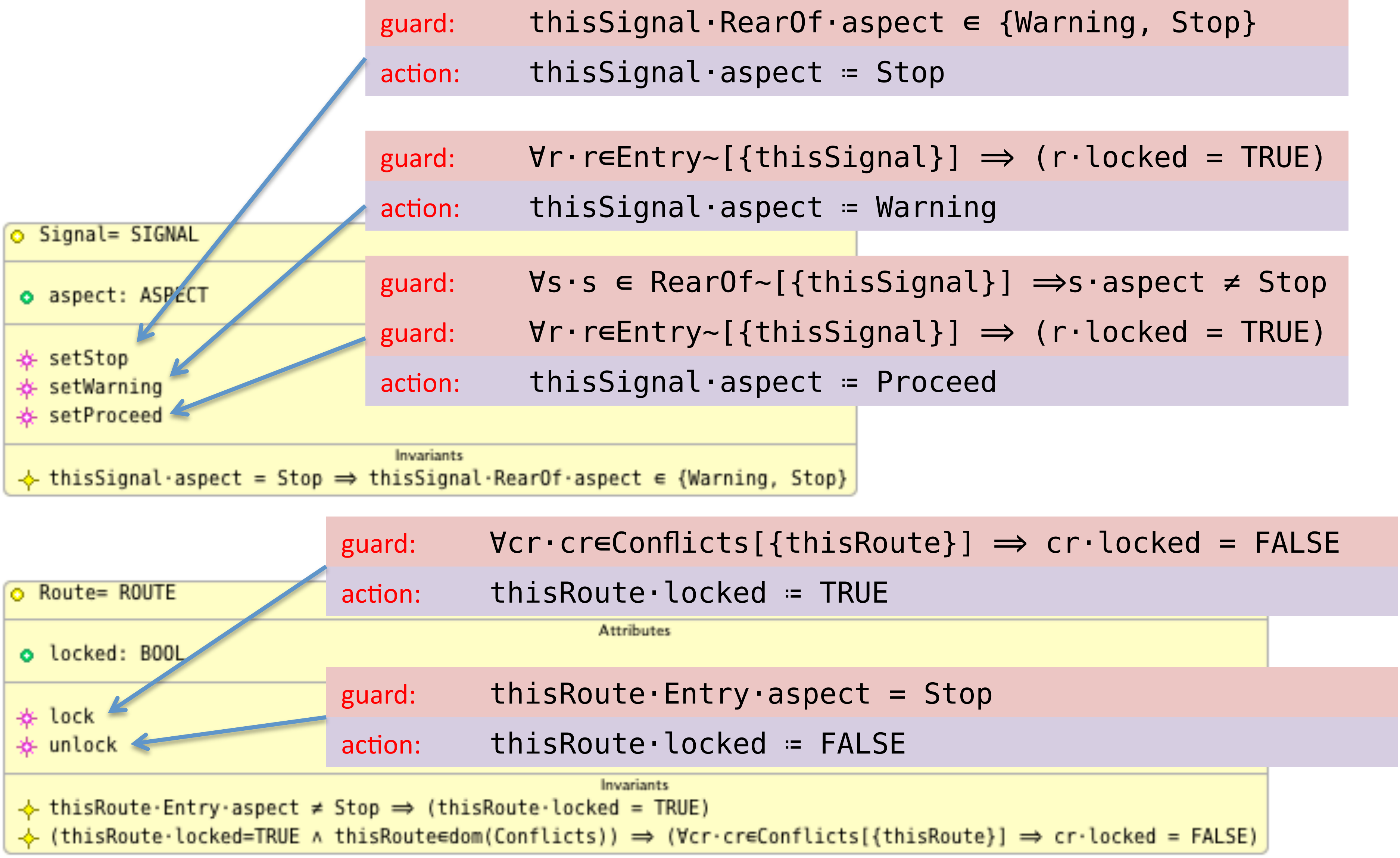
Events

lock
unlock

Invariants

thisRoute·Entry·aspect ≠ Stop ⇒ (thisRoute·locked = TRUE)
(thisRoute·locked=TRUE ∧ thisRoute∈dom(Conflicts)) ⇒ (∀cr·cr∈Conflicts[{thisRoute}] ⇒ cr·locked = FALSE)

Class Diagram with guards and actions



Signal= SIGNAL

aspect: ASPECT

setStop
setWarning
setProceed

Invariants

thisSignal·aspect = Stop ⇒ thisSignal·RearOf·aspect ∈ {Warning, Stop}

setStop:

guard: thisSignal·RearOf·aspect ∈ {Warning, Stop}
action: thisSignal·aspect ≔ Stop

setWarning:

guard: ∀r·r∈Entry∼[{thisSignal}] ⇒ (r·locked = TRUE)
action: thisSignal·aspect ≔ Warning

setProceed:

guard: ∀s·s ∈ RearOf∼[{thisSignal}] ⇒s·aspect ≠ Stop
guard: ∀r·r∈Entry∼[{thisSignal}] ⇒ (r·locked = TRUE)
action: thisSignal·aspect ≔ Proceed

Route= ROUTE

Attributes

locked: BOOL

lock
unlock

Invariants

thisRoute·Entry·aspect ≠ Stop ⇒ (thisRoute·locked = TRUE)
(thisRoute·locked=TRUE ∧ thisRoute∈dom(Conflicts)) ⇒ (∀cr·cr∈Conflicts[{thisRoute}] ⇒ cr·locked = FALSE)

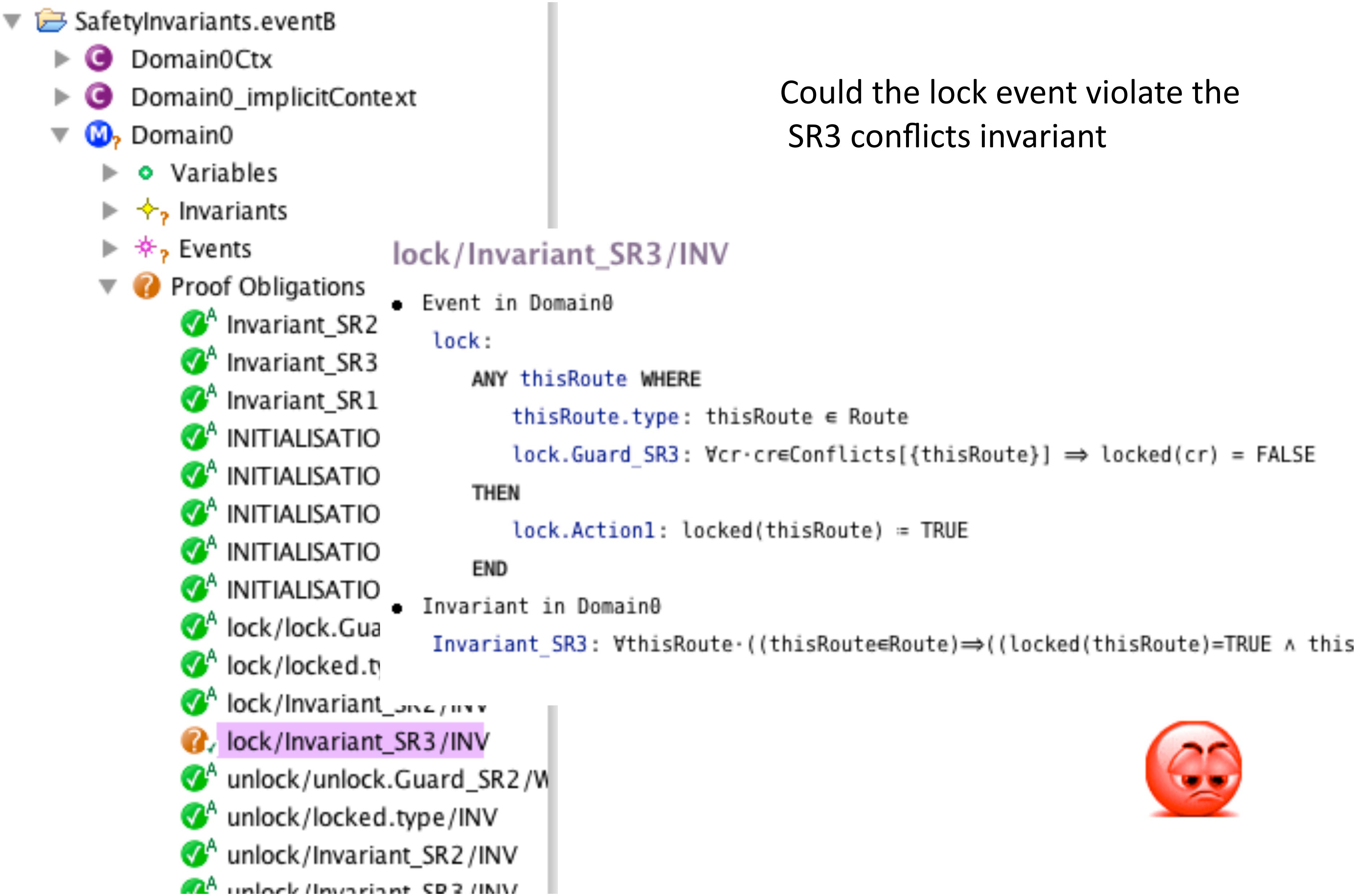
lock:

guard: ∀cr·cr∈Conflicts[{thisRoute}] ⇒ cr·locked = FALSE
action: thisRoute·locked ≔ TRUE

unlock:

guard: thisRoute·Entry·aspect = Stop
action: thisRoute·locked ≔ FALSE

One proof obligation does not prove automatically



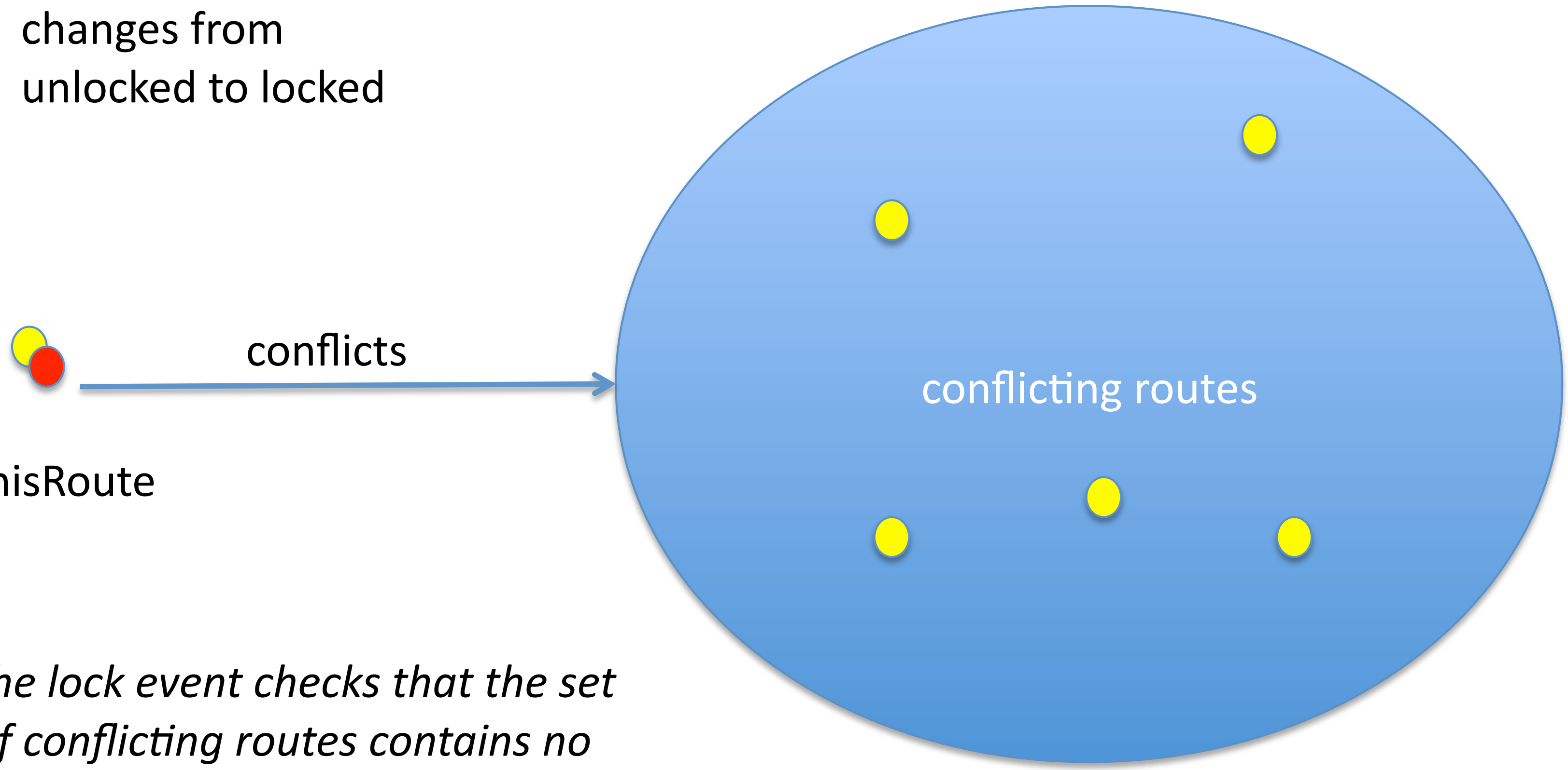
lock/Invariant_SR3/INV

```
• Event in Domain0
  lock:
     ANY thisRoute WHERE
        thisRoute.type: thisRoute ∈ Route
        lock.Guard_SR3: ∀cr·cr∈Conflicts[{thisRoute}] ⇒ locked(cr) = FALSE
     THEN
        lock.Action1: locked(thisRoute) ≔ TRUE
     END
• Invariant in Domain0
  Invariant_SR3: ∀thisRoute·((thisRoute∈Route)⇒((locked(thisRoute)=TRUE ∧ this
```

Could the lock event violate the
SR3 conflicts invariant

The Lock event

- 🔴 locked route
- 🟡 un-locked route
- 🟡🔴 changes from unlocked to locked



conflicts

thisRoute

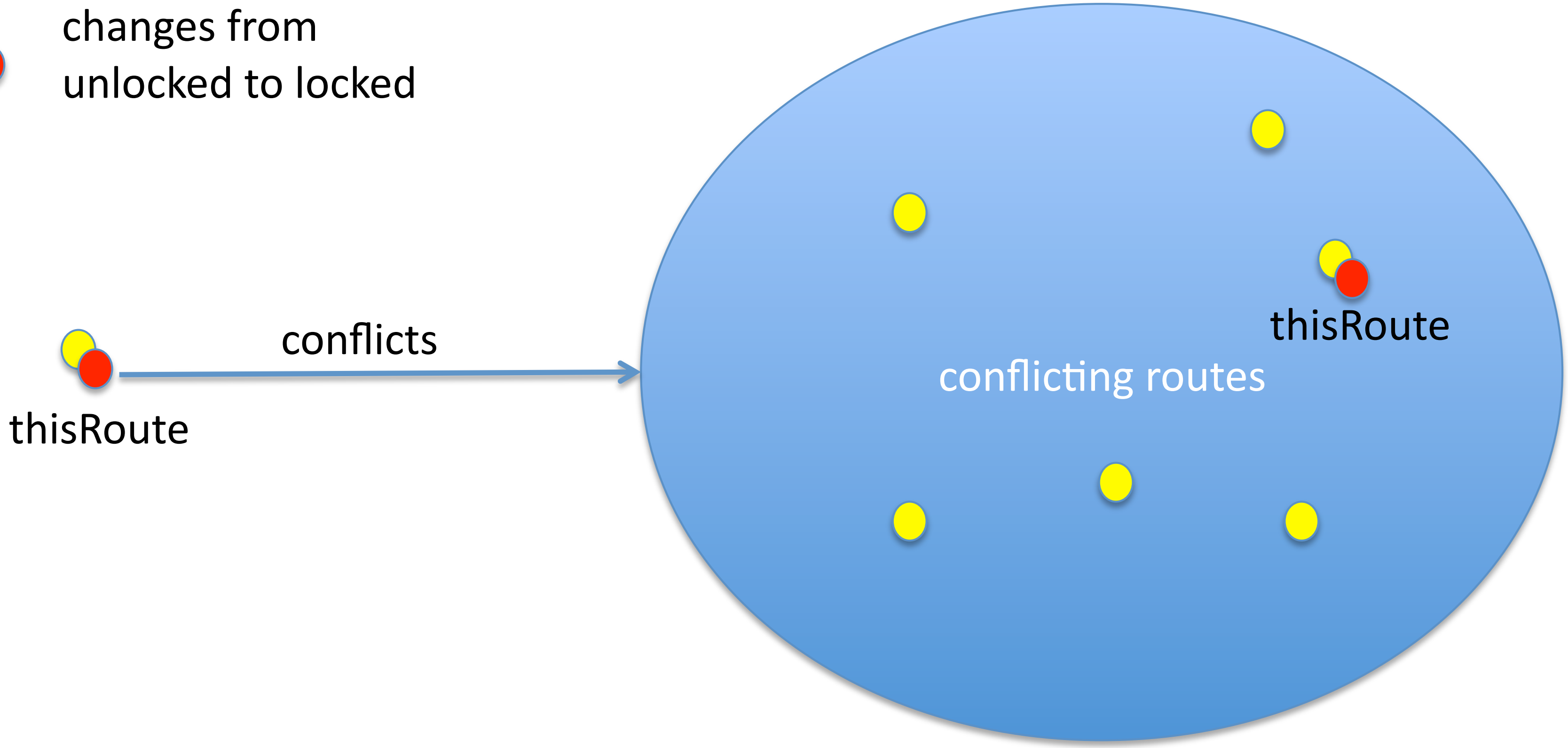
conflicting routes

the lock event checks that the set of conflicting routes contains no locked routes before locking ‘thisRoute’

A route that conflicts with itself

- 🔴 locked route
- 🟡 un-locked route
- 🟡🔴 changes from unlocked to locked

What if conflicts contains thisRoute?
then the invariant breaks



thisRoute

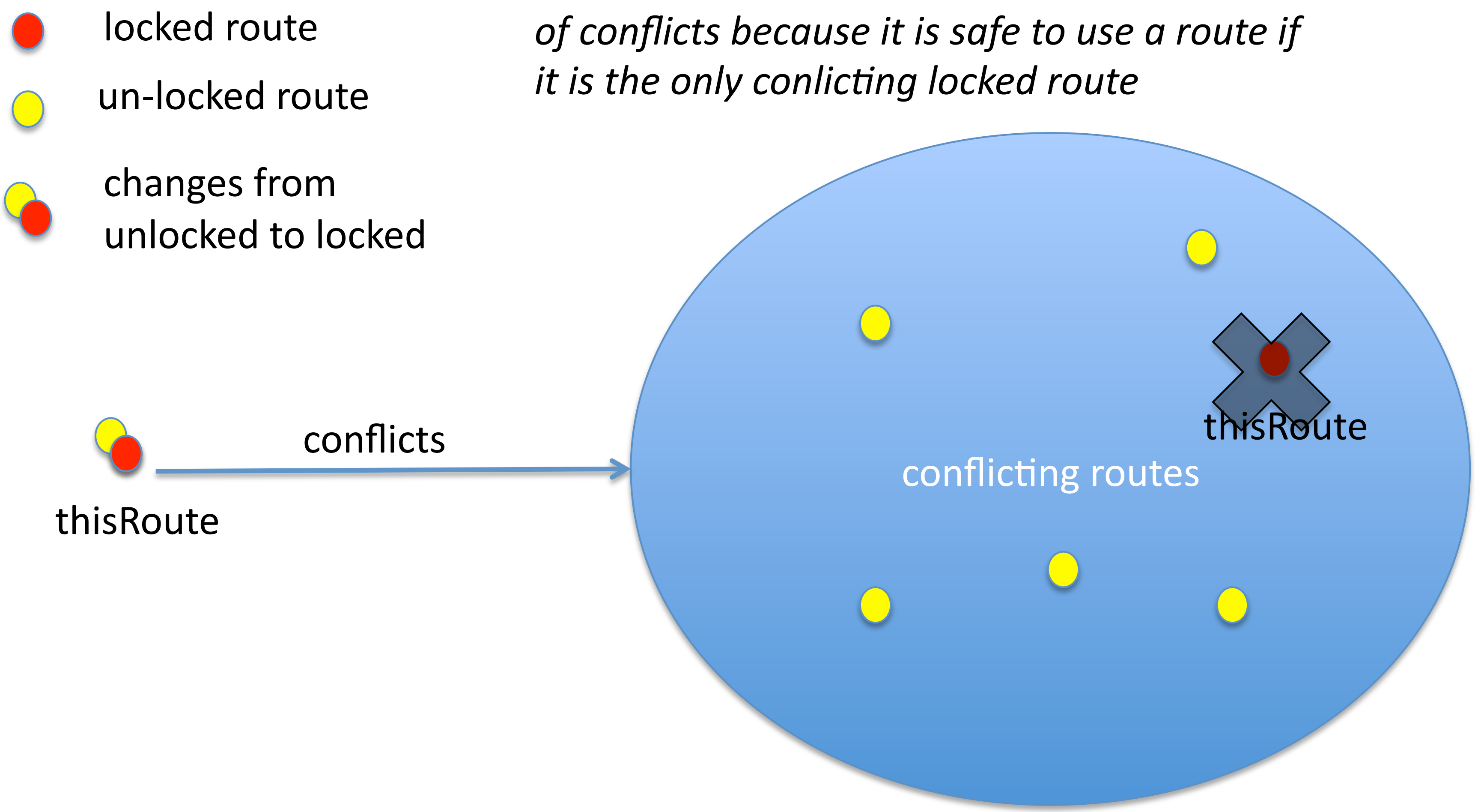
conflicts

thisRoute

conflicting routes

A route that conflicts with itself

We can disallow this in our definition of conflicts because it is safe to use a route if it is the only conlicting locked route



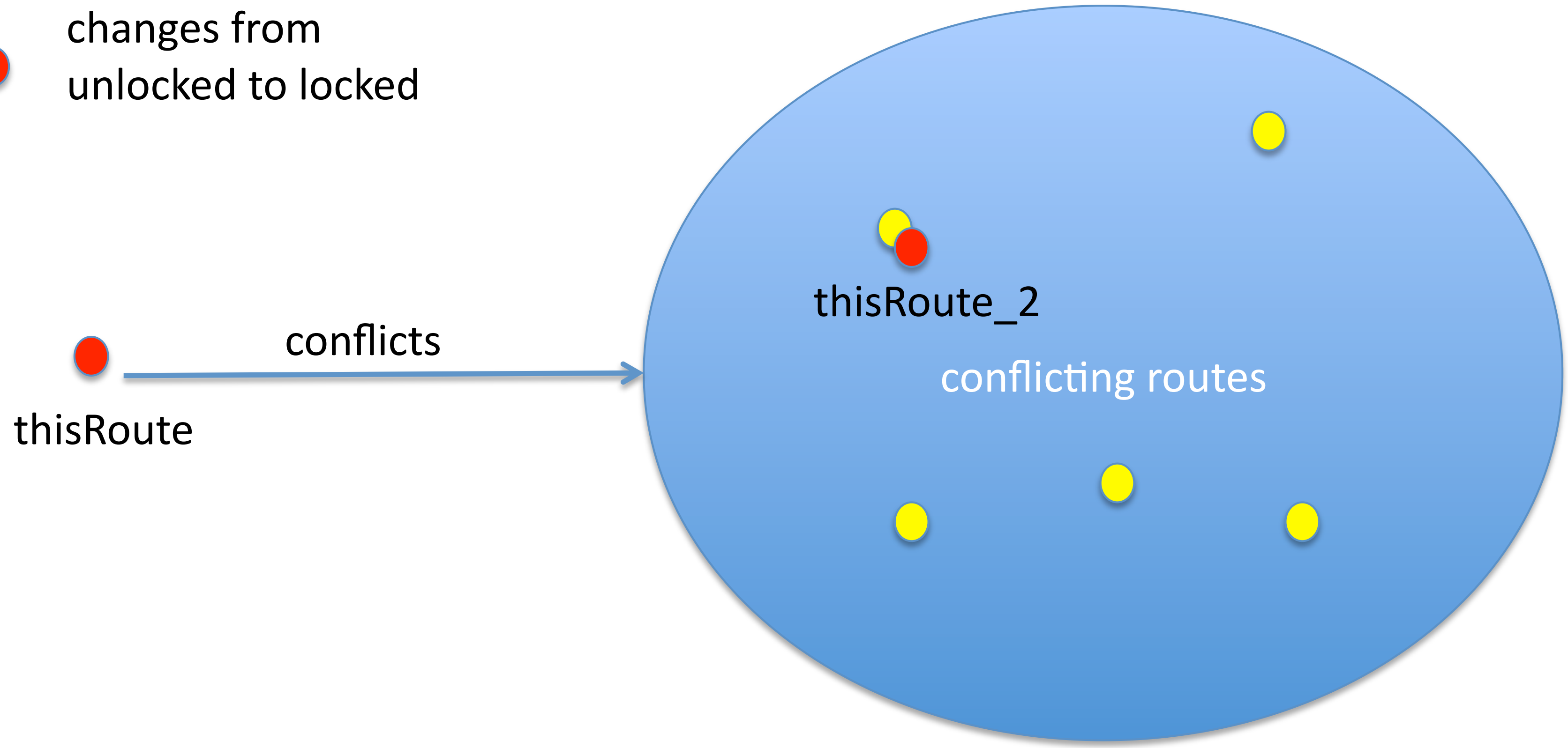
Symmetry

🔴 locked route

🟡 un-locked route

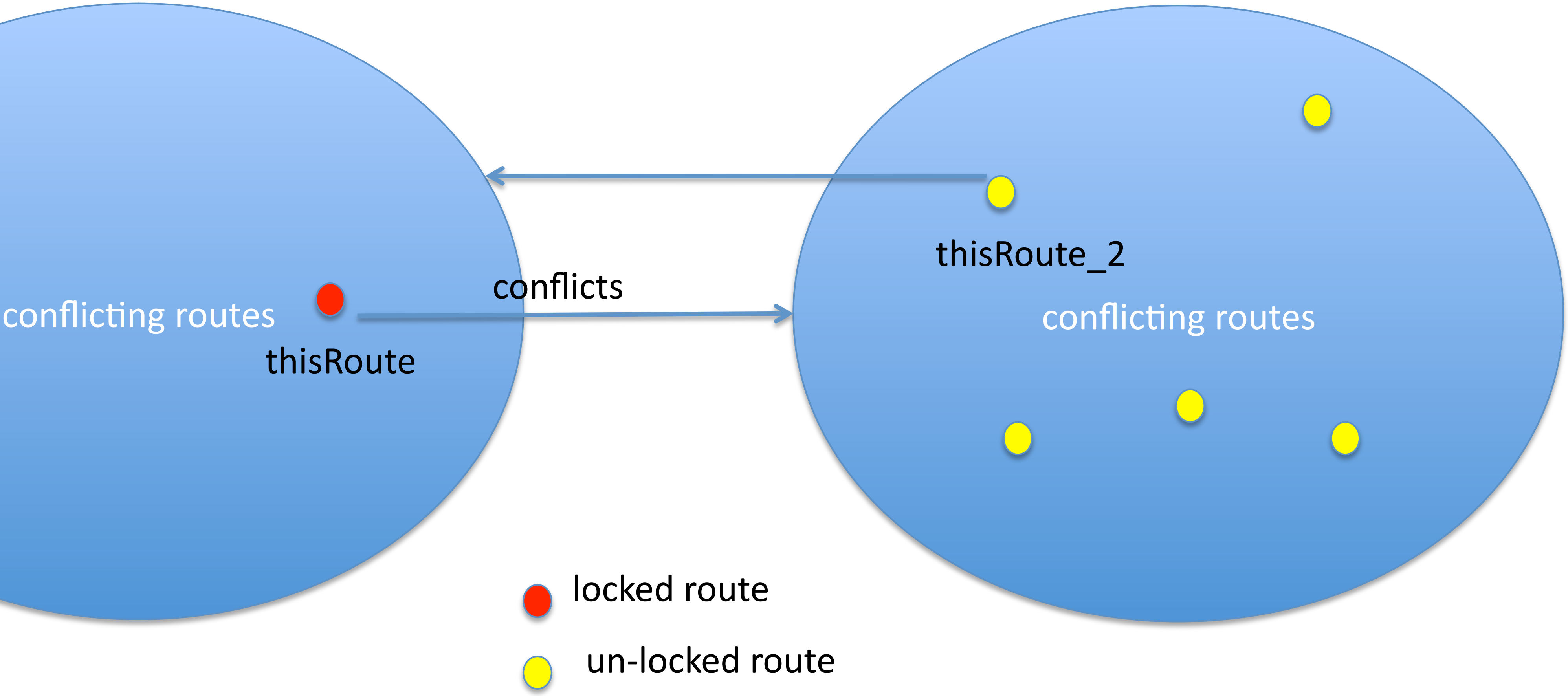
🟡🔴 changes from unlocked to locked

What if we lock one of the conflicting routes later on in another lock event…
then the invariant breaks



Symmetry

We prevent this by insisting on symmetry so that the guard in the lock event prevents us from locking thisRoute2 when thisRoute is already locked.
Our real-life' concept of conflicts is symmetric.



conflicting routes

thisRoute

conflicts

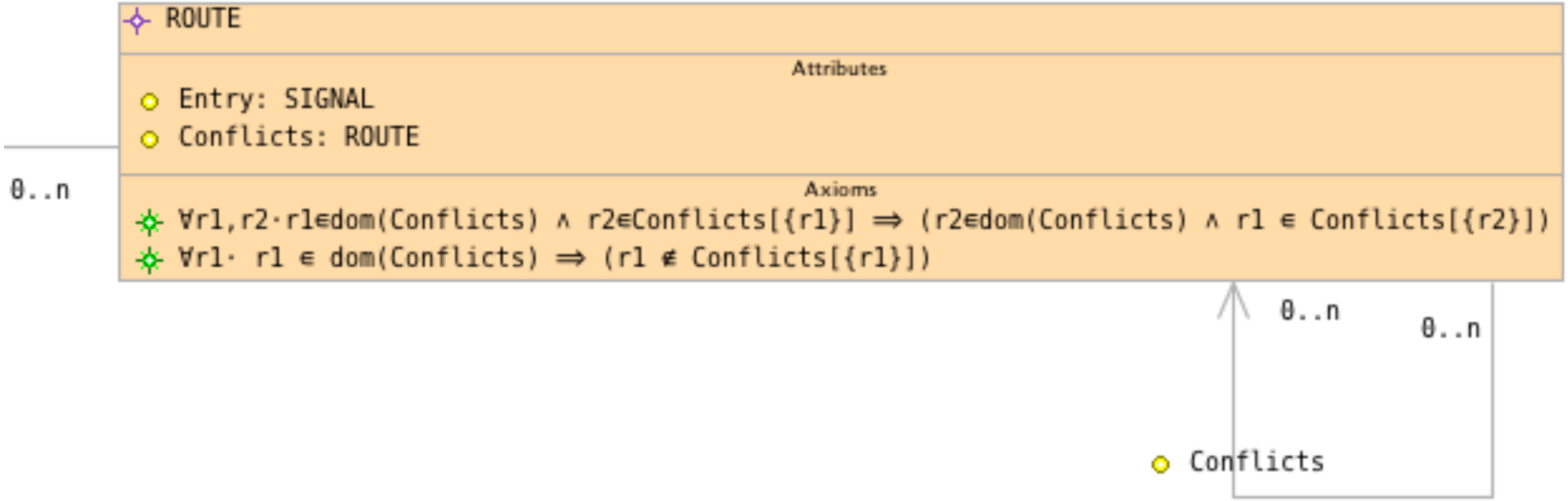
thisRoute_2

conflicting routes

🔴 locked route

🟡 un-locked route

New Axioms added to ROUTE



$\forall r1, r2 \cdot r1 \in dom(Conflicts) \wedge r2 \in Conflicts[\{r1\}] \Rightarrow (r2 \in dom(Conflicts) \wedge r1 \in Conflicts[\{r2\}])$

$\forall r1 \cdot r1 \in dom(Conflicts) \Rightarrow (r1 \notin Conflicts[\{r1\}])$

The axioms make our definition of Conflicts more precise.

Now everything proves