Egalitarian state-transition structures

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

WRLA 2016
Eindhoven

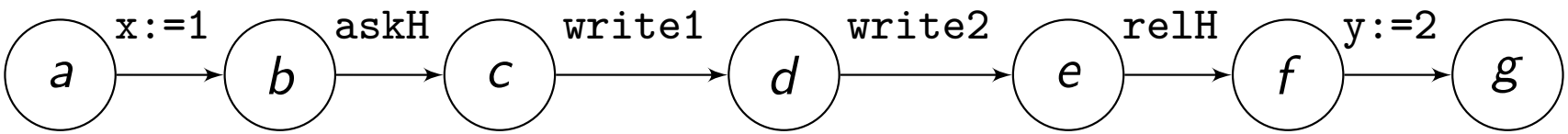
Egalitarian state-transition structures

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

WRLA 2016
Eindhoven

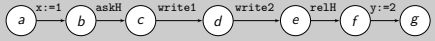
This talk is about a kind of structure that we want to propose that we call *egalitarian*. They are egalitarian in the sense that we consider states and transitions at the same level—with the same rights, if you want. I will explain in which precise sense this is so, and why we think these structures are useful or convenient.

Why we need to be egalitarian



$\Box(d \rightarrow \diamondsuit c)$

Why we need to be egalitarian



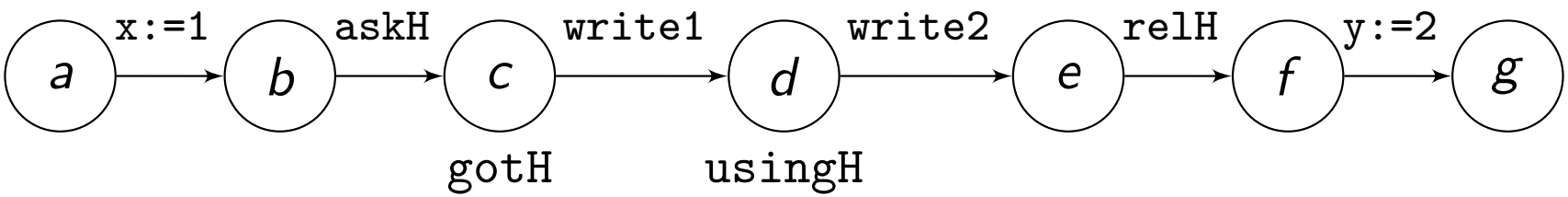
$\Box(d \rightarrow \diamondminus c)$

I will begin with an example. This is a very simple program. There are a few instructions. Two of them are writes to a file, and so there is an instruction for asking for the handle to the file and another for releasing the handle. There are some other instructions we don't care about. We have assigned labels identifying the states between the instructions.

A desirable property of such a process is that it cannot write unless it previously has got the handle. We can express this with a formula like $\Box(d \rightarrow \diamondminus c)$, using the past-time temporal operator $\diamondminus$. That is, "if we are at state d, writing to the file, at some previous time we were at state c, that is, we got the handle".

However, this formula is very closely tied to this particular part of this particular process, with the particular identifiers we have chosen for its states.

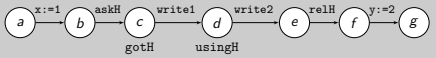
Why we need to be egalitarian



$\Box(d \rightarrow \diamondsuit c)$

$\Box(\texttt{usingH} \rightarrow \diamondsuit \texttt{gotH})$

Why we need to be egalitarian



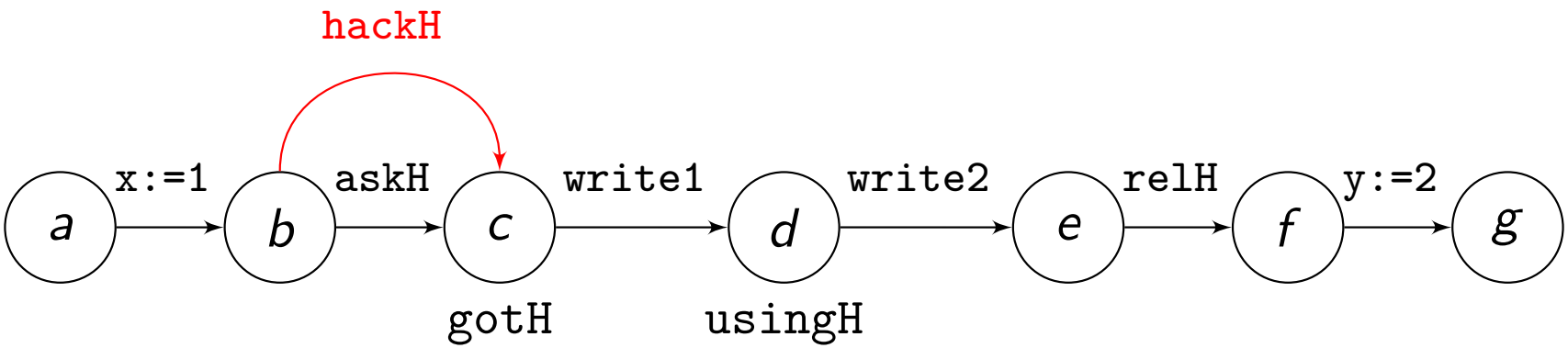
$\Box(d \to \Diamond c)$

$\Box(\texttt{usingH} \to \Diamond \texttt{gotH})$

Instead of using such a formula, we usually define propositions, say `usingH` and `gotH`, and write the formula $\Box(\texttt{usingH} \to \Diamond\!\!\!\!-\,\texttt{gotH})$. These are true of the states shown in the drawing, and probably also of some others not shown. Indeed, such a formula can be used on any process on which the two propositions can be given a meaning.

Up to now, all has been state-based, and this is known to be not enough in some cases.

Why we need to be egalitarian

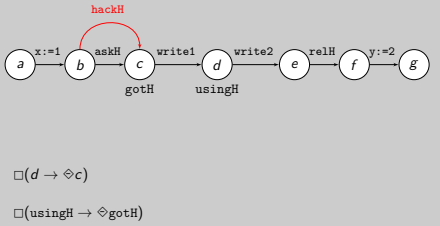


$\Box(d \rightarrow \diamondsuit c)$

$\Box(\texttt{usingH} \rightarrow \diamondsuit \texttt{gotH})$

Why we need to be egalitarian

Why we need to be egalitarian

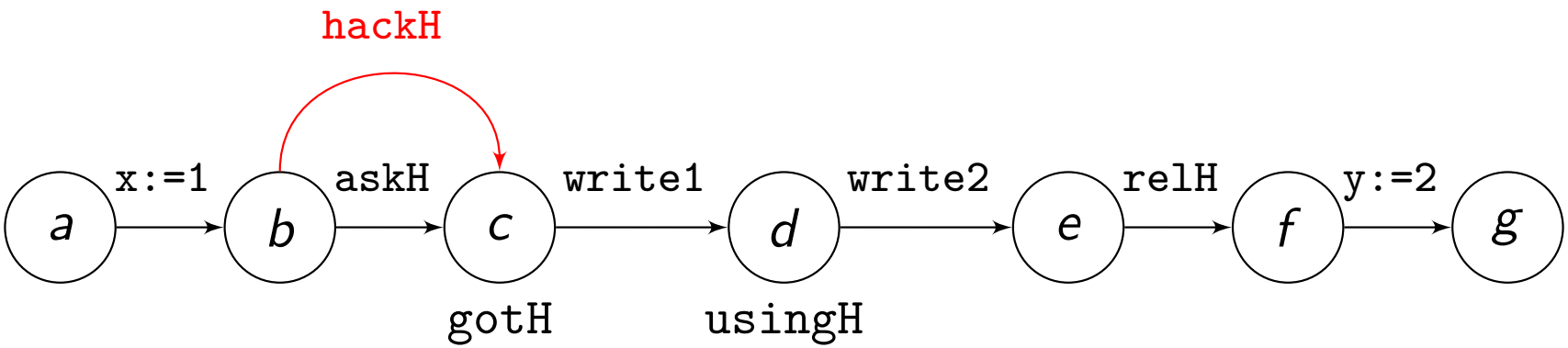


$\Box(d \to \Diamond c)$

$\Box(\texttt{usingH} \to \Diamond \texttt{gotH})$

Suppose, for instance, that a new way to get the handle is discovered to exist—hacking it in some way. This is undesirable, and we want to rule it out with our formula. But this is not possible in a state-based setting. We need to refer to actions.

Why we need to be egalitarian



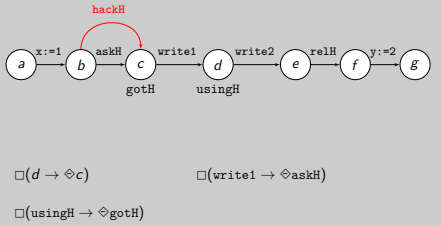
$\Box(d \rightarrow \diamondminus c)$

$\Box(\texttt{write1} \rightarrow \diamondminus \texttt{askH})$

$\Box(\texttt{usingH} \rightarrow \diamondminus \texttt{gotH})$

Why we need to be egalitarian

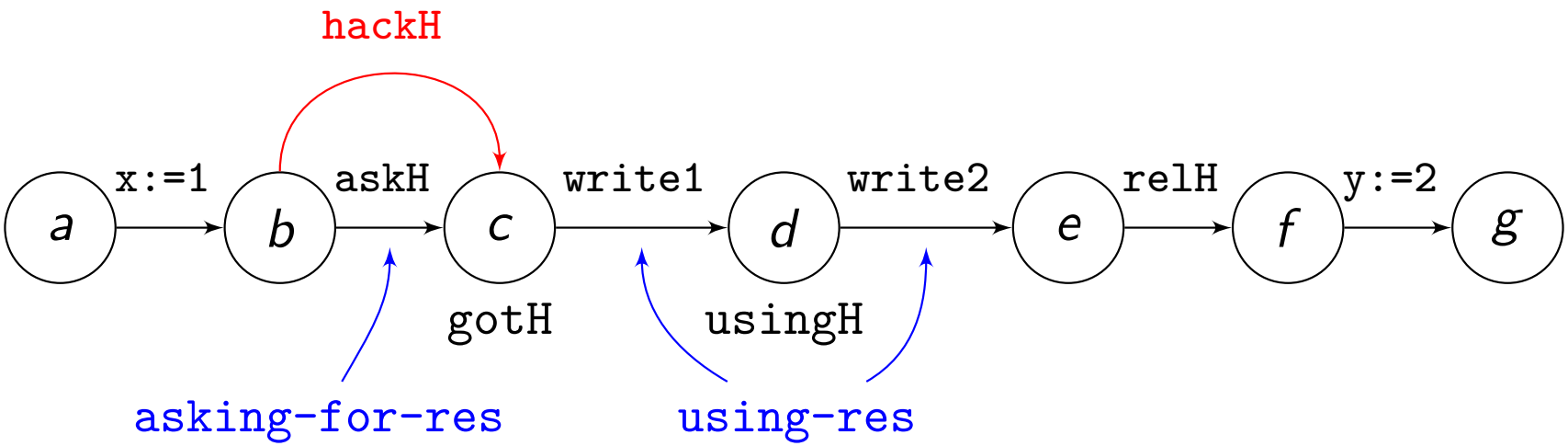
Why we need to be egalitarian



Some logics of actions allow writing formulas like $\Box(\texttt{write1} \rightarrow \diamondsuit\texttt{askH})$, where we refer not to the fact that we got the handle, but to the way we got it. Note that also the other part of the formula, the fact that we are writing to the file, is better considered as action-based.

But this new formula has the same problem as the original one: it uses literally identifiers from the text of the program and, thus, is only useful for this particular part of this particular path of this process.

Why we need to be egalitarian



$\Box(d \to \diamondminus c)$

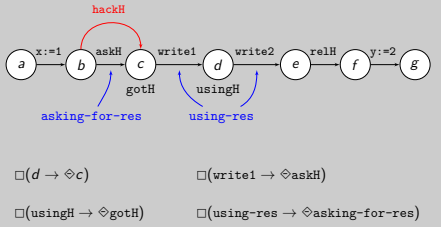
$\Box(\texttt{write1} \to \diamondminus \texttt{askH})$

$\Box(\texttt{usingH} \to \diamondminus \texttt{gotH})$

$\Box(\texttt{using-res} \to \diamondminus \texttt{asking-for-res})$

Why we need to be egalitarian

Why we need to be egalitarian



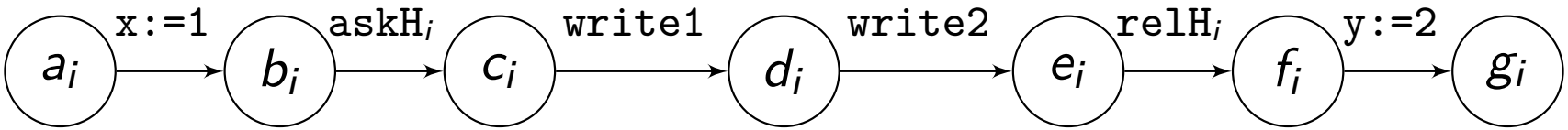
What we need are propositions on transitions, say `using-res` and `asking-for-res` (using the resource, or the handle if you prefer, and asking for it). We define them to be true on the transitions shown in the drawing, but probably also on other transitions not shown, and we write the appropriate formula $\Box(\texttt{using-res} \rightarrow \diamondminus\texttt{asking-for-res})$.

Why we need to be egalitarian

$$\Box(\texttt{using-res} \rightarrow \diamondminus\texttt{asking-for-res})$$

The point to note is that this is a very general and meaningful formula. Thanks to a smart choice of names for the propositions, this formula can be understood by itself, even if we don't know to which system it is to be applied. This is one of our main points in this talk: being egalitarian entails, in particular, using propositions on transitions.

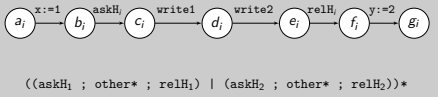
Strategies also benefit from propositions on transitions



$$((\mathtt{askH}_1\ ;\ \mathtt{other}^*\ ;\ \mathtt{relH}_1)\ |\ (\mathtt{askH}_2\ ;\ \mathtt{other}^*\ ;\ \mathtt{relH}_2))^*$$

Strategies also benefit from propositions on transitions

Strategies also benefit from propositions on transitions

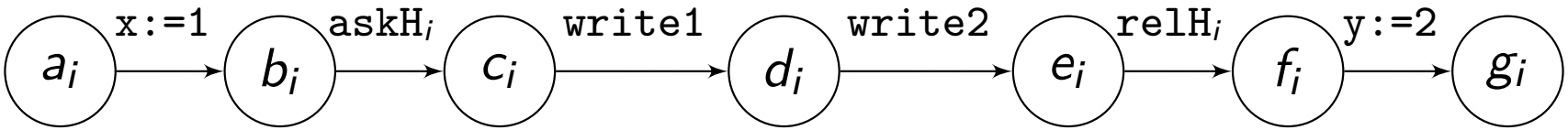


Let me show you another example, this one related to strategies. We now have two copies of the same previous process—that's why we have included subscripts. Both processes write to the same file, so we need a way to ensure mutual exclusion. We can use a strategy in the shape of a regular expression like the one in the slide.

Here, `other` is a shorthand for the disjunction of all instructions different from `askH` and `relH`. So, after process 1 has asked for the file handle and got it, this expression only allows actions not related to the handle, until process 1 releases it. In particular, process 2 is not allowed to ask for the handle until process 1 releases it. And vice versa.

The problem with this expression is the same as above: it is only useful for these particular actions.

Strategies also benefit from propositions on transitions



((askH$_1$; other* ; relH$_1$) | (askH$_2$; other* ; relH$_2$))*

```
ops enter exit : -> Prop .
eq askH |= enter = true .
eq relH |= exit = true .
eq I:Instruction |= P:Prop = false [owise] .
```

Strategies also benefit from propositions on transitions

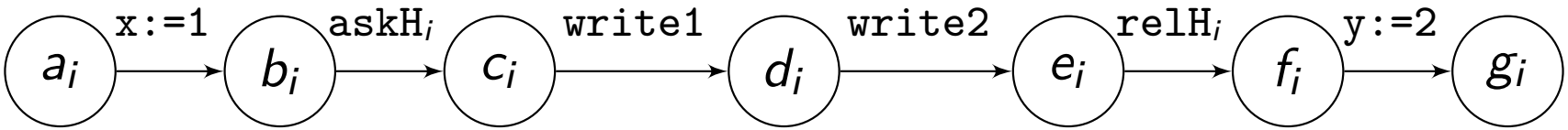
Strategies also benefit from propositions on transitions

$((\text{askH}_1 \ ; \ \text{other}* \ ; \ \text{relH}_1) \mid (\text{askH}_2 \ ; \ \text{other}* \ ; \ \text{relH}_2))*$

```
ops enter exit : -> Prop .
eq askH |= enter = true .
eq relH |= exit = true .
eq I:Instruction |= P:Prop = false [owise] .
```

Using a Maude-like syntax, we can define propositions on transitions called `enter` and `exit` to be true respectively of `askH` and `relH`, and false everywhere else.

Strategies also benefit from propositions on transitions



$$((\text{askH}_1 \ ; \ \text{other}* \ ; \ \text{relH}_1) \mid (\text{askH}_2 \ ; \ \text{other}* \ ; \ \text{relH}_2))*$$

```
ops enter exit : -> Prop .
eq askH |= enter = true .
eq relH |= exit = true .
eq I:Instruction |= P:Prop = false [owise] .
```

$$(\text{enter} \ ; \ (\neg\text{enter})* \ ; \ \text{exit})*$$

Strategies also benefit from propositions on transitions

Strategies also benefit from propositions on transitions

$((askH_1 \ ; \ other* \ ; \ relH_1) \mid (askH_2 \ ; \ other* \ ; \ relH_2))*$

```
ops enter exit : -> Prop .
eq askH |= enter = true .
eq relH |= exit = true .
eq I:Instruction |= P:Prop = false [owise] .
```

$(enter \ ; \ (\neg enter)* \ ; \ exit)*$

Now, we can write this very simple strategy expression, that has the same mutual exclusion effect: “once an action satisfying `enter` occurs, only actions not satisfying `enter` are allowed until an `exit` occurs”.

Strategies also benefit from propositions on transitions

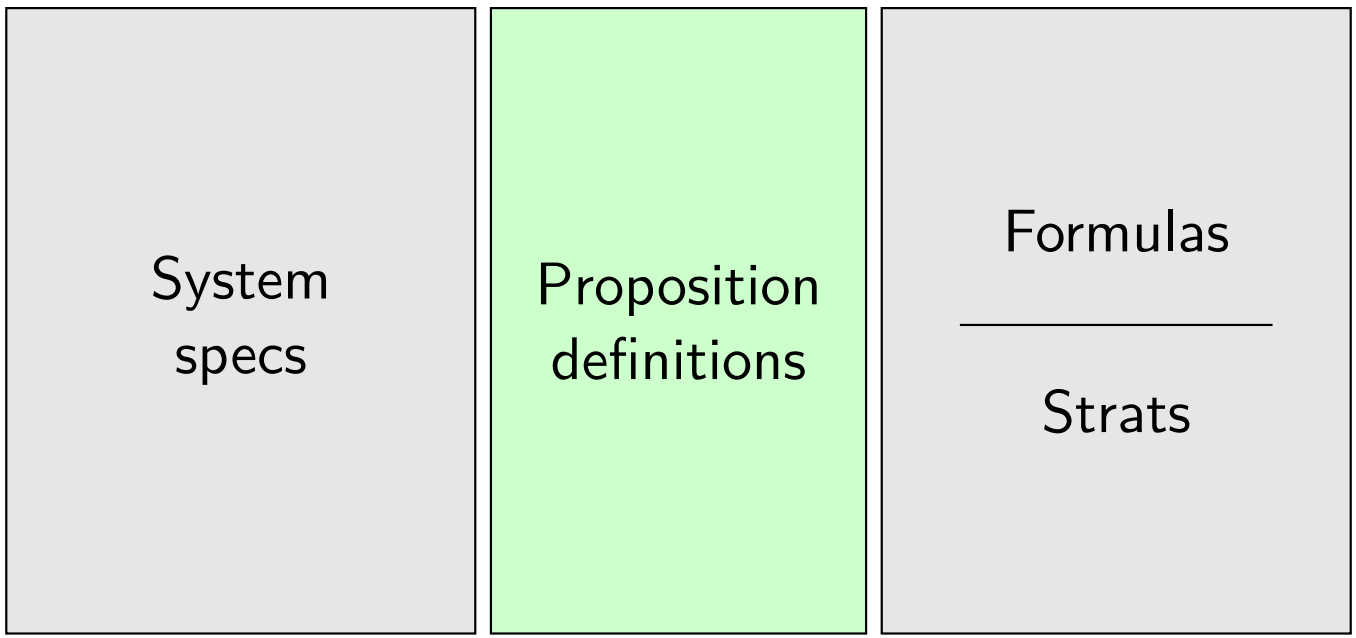
$$(\texttt{enter} \;;\; (\neg\texttt{enter})^* \;;\; \texttt{exit})^*$$

Strategies also benefit from propositions on transitions

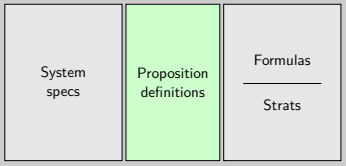
(enter ; (¬enter)* ; exit)*

Again, I want to point out that this expression is valid for any system in which these two propositions can be defined. And it is meaningful by itself, and can be understood even not knowing to which system it is going to be applied.

Decoupling is our aim



Decoupling is our aim

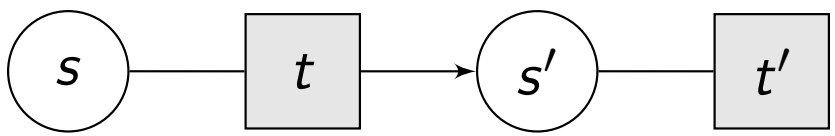


Decoupling is a key word. System specification is an independent task. The specification of temporal properties by means of formulas, and the specification of control by means of strategies should be also independent tasks. Independent from the particular systems to which they can be applied in the future. This independence can be achieved thanks to the interface provided by propositions, both on states and on transitions.

We propose *egalitarian structures*

$$(S, T, R, \mathrm{AP}, L)$$

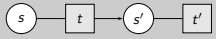
- S: set of states;
- T: set of transitions;
- $R \subseteq (S \times T) \cup (T \times S)$;
- AP: set of atomic propositions;
- $L : (S \cup T) \to 2^{\mathrm{AP}}$.



We propose *egalitarian structures*

$(S, T, R, \mathrm{AP}, L)$

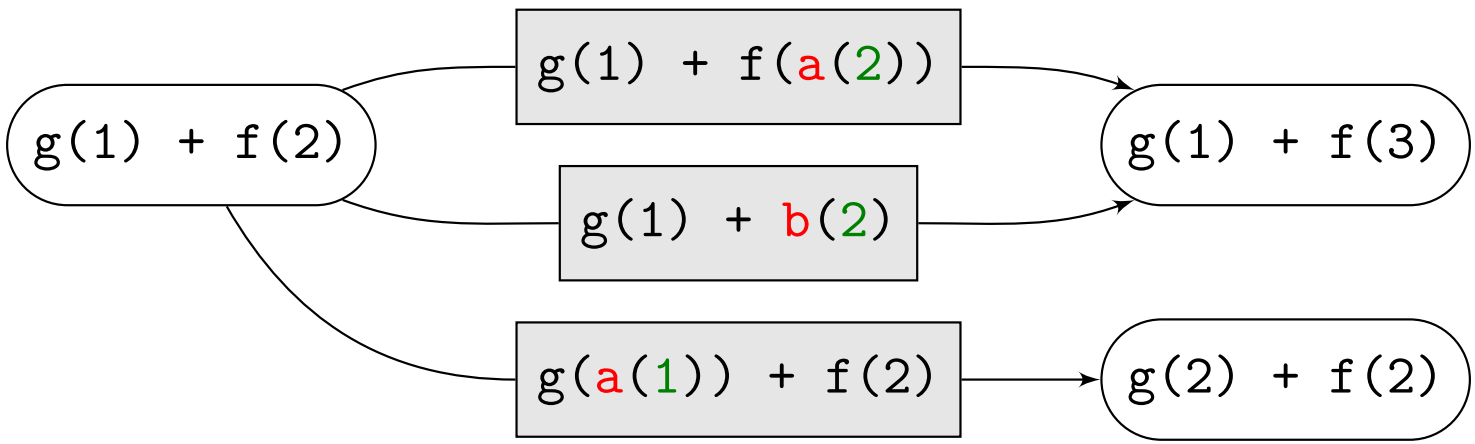
- S: set of states;
- T: set of transitions;
- $R \subseteq (S \times T) \cup (T \times S)$;
- AP: set of atomic propositions;
- $L : (S \cup T) \to 2^{\mathrm{AP}}$.



That was the problem we think we have identified. Now I present our proposal. An egalitarian structure is formally given by a tuple $(S, T, R, \mathrm{AP}, L)$. Note that T is an independent set of objects: a transition is not given as a pair of states, or anything similar, but as a new kind of object. The adjacency relation R is bipartite: from a state to a transition, and from a transition to a state. We draw it as shown: with rounded shapes for states and square ones for transitions—Petri-net style. Then we have a set of atomic propositions and the labeling function. Note that we use the same set of propositions on states and on transitions. I will be back to this point several times in the rest of the talk. We think this is just natural. For instance, it is often the case that a property of a system begins to hold at a certain point and keeps holding for some lapse of time. All the states in this lapse of time satisfy the property; it seems suitable that the same property also holds while transitions are being executed within this lapse of time.

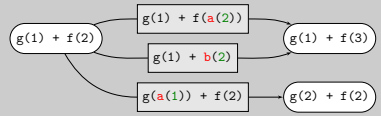
Rewrite systems are egalitarian

```
ops f g : Nat -> SomeSort .
op _+_ : SomeSort SomeSort -> State .
var N : Nat .
rl [a] : N => N + 1 .
rl [b] : f(N) => f(3) .
```



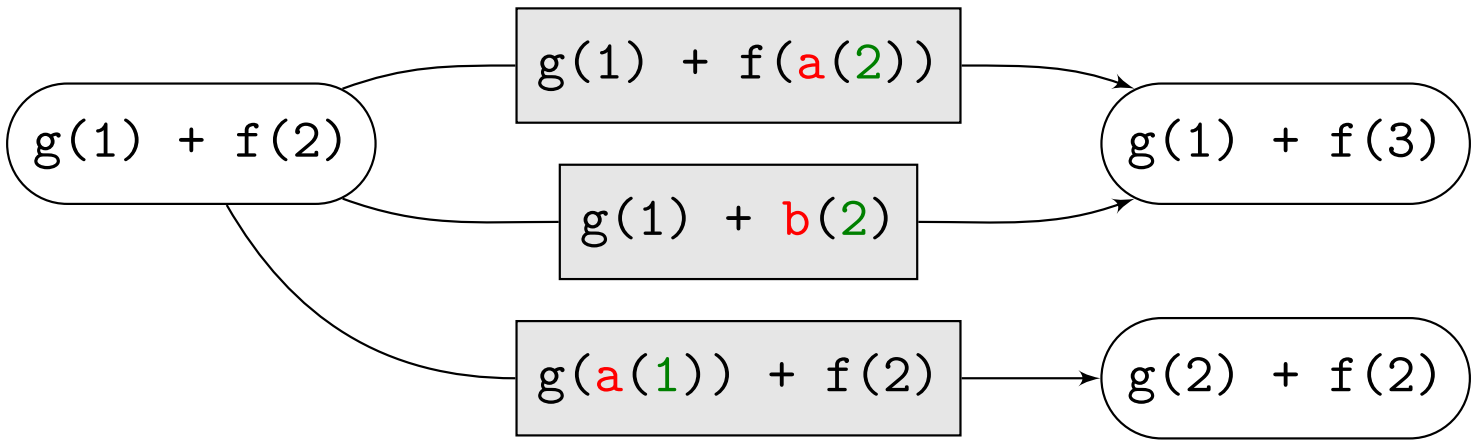
Rewrite systems are egalitarian

```
ops f g : Nat -> SomeSort .
op _+_ : SomeSort SomeSort -> State .
var N : Nat .
rl [a] : N => N + 1 .
rl [b] : f(N) => f(3) .
```



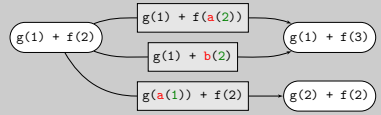
In rewriting logic, transitions are represented by so-called proof terms, in the same way as states are represented by terms of the appropriate sort. The diagram shows the proof terms for the three possible transitions from the chosen initial state. A proof term includes the label of the rule being applied, the values that instantiate the variables in the rule, and the context, that is, the part of the term that does not change. Thus, as both kinds of objects are equally represented by terms, they are ready to be treated just the same.

Rewrite systems are egalitarian




```
op has-g1 : Prop .
var E : Elem .
eq g(1) + E |= has-g1 = true .
eq E |= has-g1 = false [owise] .
```

Rewrite systems are egalitarian



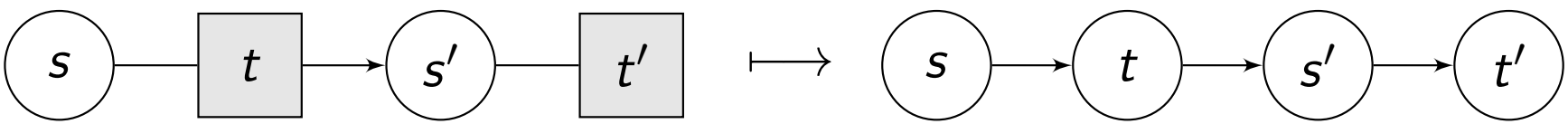

```
op has-g1 : Prop .
var E : Elem .
eq g(1) + E |= has-g1 = true .
eq E |= has-g1 = false [owise] .
```

For instance, in this toy example, suppose that we are interested in defining a proposition `has-g1` that is true of terms of the shape `g(1)`+ something. This definition does the job. Note that it is valid for states and for transitions, at the same time, with an only, uniform definition. The two upper states and the two upper transitions satisfy it; the others don't.

Egalitarian structures can be *split*

Egalitarian structures $\longrightarrow$ Kripke structures

$$(S, T, R, \mathrm{AP}, L) \longmapsto (S \cup T, R, \mathrm{AP}, L)$$



Egalitarian structures can be *split*

Egalitarian structures can be *split*

Egalitarian structures $\longrightarrow$ Kripke structures

$(S, T, R, \mathrm{AP}, L) \longmapsto (S \cup T, R, \mathrm{AP}, L)$

It is possible to translate an egalitarian structure to an equivalent Kripke structure. Formally, it is only necessary to transform transitions in new states. Graphically, we just redraw squares as circles.

The point to note here is that the resulting Kripke structure and the original, egalitarian one are *isomorphic*. All the information present in one is also in the other. Any property we express about the Kripke structure (by means of a temporal formula, for instance) is immediately translatable to a property about the egalitarian one. Any verification task performed on the Kripke structure is verifying something about the egalitarian structure. In this way, we can use any non-egalitarian tool, either theoretical or practical, with an egalitarian aim.

Rewrite systems can be *split*

```
rl [a] : N => N + 1 .        ⟼    rl [a1] : N => a(N) .
                                   rl [a2] : a(N) => N + 1 .

rl [b] : f(N) => f(3) .      ⟼    rl [b1] : f(N) => b(N) .
                                   rl [b2] : b(N) => f(3) .
```

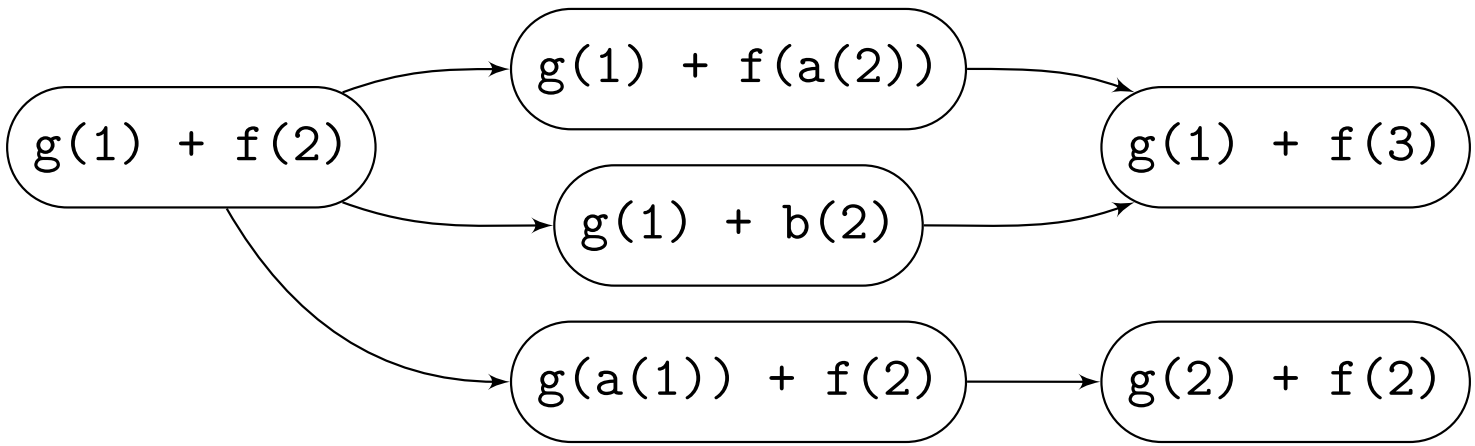
Rewrite systems can be *split*

```
rl [a] : N => N + 1 .     ⟼   rl [a1] : N => a(N) .
                               rl [a2] : a(N) => N + 1 .

rl [b] : f(N) => f(3) .   ⟼   rl [b1] : f(N) => b(N) .
                               rl [b2] : b(N) => f(3) .
```

There is a translation between rewrite systems, that we also call *split*, that corresponds to the one in the previous slide. The idea is transforming each rule into two. Note that the left hand side of the original rule occurs as left hand side of the first *half rule*; and the right hand side occurs as such in the second *half rule*. In between, we have added a term formed using the original rule label and the list of variables in it.

Rewrite systems can be *split*

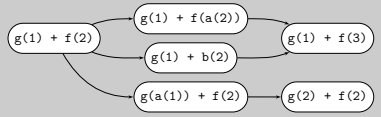
```
rl [a] : N => N + 1 .      ⟼   rl [a1] : N => a(N) .
                                rl [a2] : a(N) => N + 1 .

rl [b] : f(N) => f(3) .    ⟼   rl [b1] : f(N) => b(N) .
                                rl [b2] : b(N) => f(3) .
```



Rewrite systems can be *split*

Rewrite systems can be *split*

```
rl [a] : N => N + 1 .     ⟼   rl [a1] : N => a(N) .
                               rl [a2] : a(N) => N + 1 .

rl [b] : f(N) => f(3) .   ⟼   rl [b1] : f(N) => b(N) .
                               rl [b2] : b(N) => f(3) .
```



The interesting thing about the resulting split system is that its standard Kripke-structure semantics is the split of the egalitarian semantics of the original rewrite system. Note that the drawing is the split of the one shown in a previous slide—just with rectangles turned into rounded shapes.

Although rewriting logic is egalitarian in nature, Maude is not so, as proof terms are not Maude objects. The split transformation makes proof terms appear, and allows us to refer to states and to transitions in the same way, as we want.

Temporal logics for egalitarian structures

- Using LTL, CTL, μ-calculus... on the split system.
- Translating:

$$(\mathrm{split}, \sigma) : \mathsf{RwS} \times \mathsf{TLR}^* \to \mathsf{RwS} \times \mathsf{CTL}^*,$$

such that

$$\mathcal{R}, e \models \varphi \quad \Leftrightarrow \quad \mathrm{split}(\mathcal{R}), e \models \sigma(\varphi).$$

When we want to use temporal logics on egalitarian structures, there are two ways to do it. The first one is to use a logic for Kripke structures and use it on the split system, as explained above. The second one is to use a temporal logic of an egalitarian nature and translate its formulas so that the satisfaction relation is preserved.

TLR* is not really egalitarian, but it is able to talk about actions as well as states, so we show a suitable translation for it.

Temporal logics for egalitarian structures

- Using LTL, CTL, μ-calculus... on the split system.
- Translating:

$$(\mathrm{split}, \sigma) : \mathsf{RwS} \times \mathsf{TLR}^* \to \mathsf{RwS} \times \mathsf{CTL}^*,$$

such that

$$\mathcal{R}, e \models \varphi \quad \Leftrightarrow \quad \mathrm{split}(\mathcal{R}), e \models \sigma(\varphi).$$

$\sigma : \mathsf{TLR}^* \to \mathsf{CTL}^*$:

- $\sigma(P) = P$, if P has sort `SProp`;
- $\sigma(P) = \bigcirc P$, if P has sort `TProp`;
- $\sigma(\neg\varphi) = \neg\sigma(\varphi)$;
- $\sigma(\varphi_1 \vee \varphi_2) = \sigma(\varphi_1) \vee \sigma(\varphi_2)$;
- $\sigma(\bigcirc\varphi) = \bigcirc \bigcirc \sigma(\varphi)$;
- $\sigma(\varphi_1 \,\mathbf{U}\, \varphi_2) = (\mathtt{isState} \to \sigma(\varphi_1)) \,\mathbf{U}\, (\mathtt{isState} \wedge \sigma(\varphi_2))$;
- $\sigma(\mathbf{E}\,\varphi) = \mathbf{E}\,\sigma(\varphi)$.

Temporal logics for egalitarian structures
- Using LTL, CTL, μ-calculus... on the split system.
- Translating:

$$(\mathrm{split}, \sigma) : \mathrm{RwS} \times \mathrm{TLR}^* \to \mathrm{RwS} \times \mathrm{CTL}^*,$$

such that

$$\mathcal{R}, e \models \varphi \quad \Leftrightarrow \quad \mathrm{split}(\mathcal{R}), e \models \sigma(\varphi).$$

$\sigma : \mathrm{TLR}^* \to \mathrm{CTL}^*$:
- $\sigma(P) = P$, if P has sort SProp;
- $\sigma(P) = \bigcirc P$, if P has sort TProp;
- $\sigma(\neg\varphi) = \neg\sigma(\varphi)$;
- $\sigma(\varphi_1 \vee \varphi_2) = \sigma(\varphi_1) \vee \sigma(\varphi_2)$;
- $\sigma(\bigcirc\varphi) = \bigcirc\bigcirc\sigma(\varphi)$;
- $\sigma(\varphi_1 \,\mathbf{U}\, \varphi_2) = (\mathtt{isState} \to \sigma(\varphi_1)) \,\mathbf{U}\, (\mathtt{isState} \wedge \sigma(\varphi_2))$;
- $\sigma(\mathbf{E}\,\varphi) = \mathbf{E}\,\sigma(\varphi)$.

We use here a particular flavor of TLR*: while the standard definition of TLR* uses patterns for proof terms (so-called *spatial actions*) we use directly propositions for transitions. However, this does not turn TLR* into a fully egalitarian temporal logic, because in TLR* propositions and formulas can only be evaluated on states. The way to assert something about a transition is to assert it on its origin state.

So, while a proposition on states (sort `SProp`) is left unchanged by the translation, a proposition on transitions (`TProp`) is translated with a *next* operator $\bigcirc$, to make it jump to the next state (of the split system, that represents a transition of the original one).

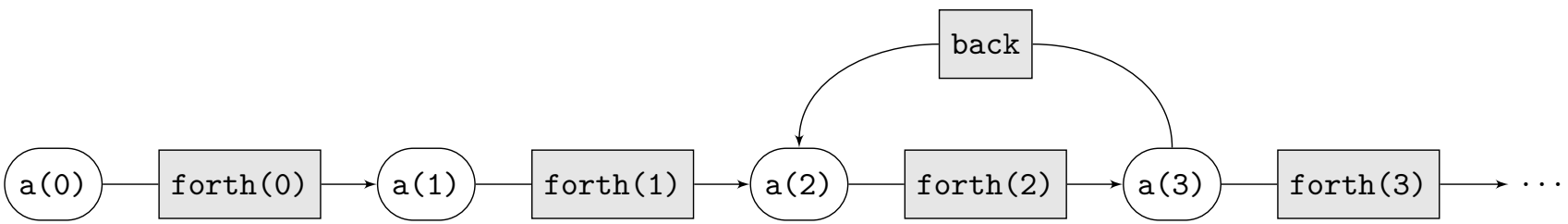
Also, the *next* operator in $\bigcirc\varphi$ has to be duplicated, to make the system jump over the transition-state to the next genuine state.

Something similar happens with the translation of the *until* operator: we have to be sure that the formulas are always evaluated on genuine states.

For this, we need a particular proposition, `isState`, true of states that represent states, false otherwise.

Our implementation

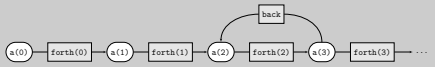
```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [forth] : a(N) => a(N + 1) if N < 100 .
   rl  [back]  : a(3) => a(2) .
endm)
```



Our implementation

Our implementation

```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [forth] : a(N) => a(N + 1) if N < 100 .
   rl  [back]  : a(3) => a(2) .
endm)
```



We have implemented a module operator that allows us to work in an egalitarian way on Maude modules. I will show you how it works. This is the example I will use. All states are built with the operator a and a natural number. There are two rules: one that allows adding one to the number, the other is a loop back from 3 to 2.

The property in which we will be interested is that after performing rule `back` it is not possible to reach a state with number less than two. This is quite obvious, but we want to model check for it, anyway.

Our implementation

```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [forth] : a(N) => a(N + 1) if N < 100 .
   rl  [back]  : a(3) => a(2) .
endm)


(mod EXAMPLE-PROPS is
   pr SPLIT[EXAMPLE] .
   ...
endm)
```

Our implementation

```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [forth] : a(N) => a(N + 1) if N < 100 .
   rl  [back]  : a(3) => a(2) .
endm)

(mod EXAMPLE-PROPS is
   pr SPLIT[EXAMPLE] .
   ...
endm)
```

We have implemented a module operator called `SPLIT`. The slide shows the way to use it. Instead of extending the original module as usual, with definitions of propositions, initial states, and so on, the user has to extend the split system. After importing `SPLIT[EXAMPLE]`, users have state terms and transition terms at their disposal.

Our implementation

```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [forth] : a(N) => a(N + 1) if N < 100 .
   rl  [back]  : a(3) => a(2) .
endm)
```

```
(mod SPLIT[EXAMPLE] is
   pr NAT .
   sorts StateEXAMPLE TransEXAMPLE State .
   subsorts StateEXAMPLE TransEXAMPLE < State .
   op a : Nat -> StateEXAMPLE .
   op forth : Nat -> TransEXAMPLE .
   op back : -> TransEXAMPLE .
   var N : Nat .
   crl [forth1] : a(N) => forth(N) if N < 100 .
   rl  [forth2] : forth(N) => a(N + 1) .
   rl  [back1]  : a(3) => back .
   rl  [back2]  : back => a(2) .
endm)
```

Our implementation

```
(mod EXAMPLE is
  pr NAT .
  sort State .
  op a : Nat -> State .
  var N : Nat .
  crl [forth] : a(N) => a(N + 1) if N < 100 .
  rl  [back]  : a(3) => a(2) .
endm)
(mod SPLIT[EXAMPLE] is
  pr NAT .
  sorts StateEXAMPLE TransEXAMPLE State .
  subsorts StateEXAMPLE TransEXAMPLE < State .
  op a : Nat -> StateEXAMPLE .
  op forth : Nat -> TransEXAMPLE .
  op back : -> TransEXAMPLE .
  var N : Nat .
  crl [forth1] : a(N) => forth(N) if N < 100 .
  rl  [forth2] : forth(N) => a(N + 1) .
  rl  [back1]  : a(3) => back .
  rl  [back2]  : back => a(2) .
endm)
```

The user does not even see the result of the split, but I will show you what is happening behind the scenes. The `SPLIT` operator assumes the argument module has a sort named `State`. The split module has a sort `StateEXAMPLE` that is a renaming of that sort `State`. It also has `TransEXAMPLE` to represent transitions in the original module. There is also a sort `State` that is not the original module's one, but a new one that includes states and transitions of the original module.

There are two operators that build transition terms, with the names of the original module's rules. And there are the four rules that result from the splitting of the two original ones.

Our implementation

```
(mod EXAMPLE-PROPS-LTLR is
   pr SPLIT[EXAMPLE] .
   inc MODEL-CHECKER .
   inc LTLR .

   op going-back : -> TProp .
   eq back |= going-back = true .
   eq T:TransEXAMPLE |= going-back = false [owise] .

   var N : Nat .
   op at-Nlt2 : -> SProp .
   eq a(N) |= at-Nlt2 = (N < 2) .
   eq S:StateEXAMPLE |= at-Nlt2 = false [owise] .

   eq S:StateEXAMPLE |= isState = true .
   eq T:TransEXAMPLE |= isState = false .
endm)


red modelCheck(a(0), LTLR([] (going-back -> [] ~ at-Nlt2))) .
```

Our implementation

```
(mod EXAMPLE-PROPS-LTLR is
   pr SPLIT[EXAMPLE] .
   inc MODEL-CHECKER .
   inc LTLR .

   op going-back : -> TProp .
   eq back |= going-back = true .
   eq T:TransEXAMPLE |= going-back = false [owise] .

   var N : Nat .
   op at-Nlt2 : -> SProp .
   eq a(N) |= at-Nlt2 = (N < 2) .
   eq S:StateEXAMPLE |= at-Nlt2 = false [owise] .

   eq S:StateEXAMPLE |= isState = true .
   eq T:TransEXAMPLE |= isState = false .
endm)

red modelCheck(a(0), LTLR([] (going-back -> [] ~ at-Nlt2))) .
```

I will show you two ways of performing the model checking. Let me advance that we prefer the second one. In the first case, we declare and define a proposition on transitions (`TProp`) called `going-back`, true when executing rule `back` and false when executing `forth`. We also use a proposition on states called `at-Nlt2`, true of a state iff its number is less than two. The sorts `SProp` and `TProp` are defined in the module `LTLR`, that is part of our implementation.

We can now write `[] (going-back -> [] ~ at-Nlt2)` which is a TLR* formula. More precisely, a formula from LTLR, the linear-time subset of TLR*. We need to translate it to LTL to be able to use Maude's model checker. The function `LTLR`, also defined in the module by the same name, does the translation. It is the restriction of the function σ above to the linear-time setting. Note that this provides a model checker for LTLR as a by-product.

As explained, the translation needs to use the proposition `isState`, so we have defined it.

Our implementation

```
(mod EXAMPLE-PROPS-LTL is
   pr SPLIT[EXAMPLE] .
   inc MODEL-CHECKER .

   op going-back : -> Prop .
   eq back |= going-back = true .
   eq S:State |= going-back = false [owise] .

   var N : Nat .
   op at-Nlt2 : -> Prop .
   eq a(N) |= at-Nlt2 = (N < 2) .
   eq forth(N) |= at-Nlt2 = (N < 2) .
   eq S:State |= at-Nlt2 = false [owise] .
endm)


red modelCheck(a(0), [] (going-back -> [] ~ at-Nlt2)) .
```

Our implementation

```
(mod EXAMPLE-PROPS-LTL is
   pr SPLIT[EXAMPLE] .
   inc MODEL-CHECKER .

   op going-back : -> Prop .
   eq back |= going-back = true .
   eq S:State |= going-back = false [owise] .

   var N : Nat .
   op at-Nlt2 : -> Prop .
   eq a(N) |= at-Nlt2 = (N < 2) .
   eq forth(N) |= at-Nlt2 = (N < 2) .
   eq S:State |= at-Nlt2 = false [owise] .
endm)

red modelCheck(a(0), [] (going-back -> [] ~ at-Nlt2)) .
```

In the second case we define the same propositions but on all states and transitions. Proposition `going-back`, that was previously defined only on transitions, is now defined also on states. While staying on a state we are going nowhere, in particular we are not going back, so we define this proposition as false on every state. Note that the sort `State` appearing there is the one from the split module, that includes states and transitions from the original module.

Also, the proposition `at-Nlt2` can be naturally given a meaning on transitions as shown: true only when applying rule `forth` with a number less that 2.

With propositions defined in this way, we can write an LTL formula (the same one as above) that can be directly fed to the model checker.

This is the very nature of being egalitarian: not only using propositions on states and on transitions, but doing it uniformly, with no visible discrimination.

Future work

- Be egalitarian!
- Strategies.

Future work

I finish with a glimpse of our future work in two items. The first one looks rather like a New Year's resolution. As it is written, it is a little strong. I would rather say: "consider the possibility that being egalitarian in your particular setting pays off".

The second item is more concrete. We are interested in strategies, in controlling systems by means of strategies, and we feel that being egalitarian will allow our strategies to be more general, more meaningful, and maybe even easier to implement.

Thank you.

Have a good day.