# Egalitarian state-transition systems (extended version)*

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet
{omartins,jalberto,narciso}@ucm.es

Technical Report 01/16

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid, Spain

March 2016

* Partially supported by MINECO Spanish project StrongSoft (TIN2012–39391–C04–04), Comunidad de Madrid program N-GREENS Software (S2013/ICE-2731), and UCM-Santander grant GR3/14.

**Abstract.** We argue that considering transitions at the same level as states, as first-class citizens, is advantageous in many cases. Namely, the use of atomic propositions on transitions, as well as on states, allows temporal formulas and strategies to be more powerful, general, and meaningful. We define *egalitarian* structures and logics, and show how they generalize well-known state-based, event-based, and mixed ones. We present translations from egalitarian to non-egalitarian settings that, in particular, allow the model checking of LTLR formulas using Maude's LTL model checker. We have implemented these translations as a prototype in Maude itself.

# Table of Contents

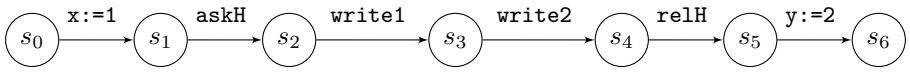
## 1 Introduction

There is a case of discrimination in computer science that favors states against transitions (call them events or actions, if you prefer). It is not unlike the discrimination by gender in some human societies. Considering transitions just as a means to go from a state to another is as unfair as considering women just as a means for passing genes from father to son. We want to show that this discrimination (against transitions) hinders specification and programming tasks.

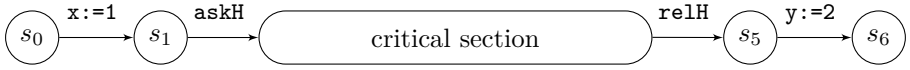
There exist, certainly, transition-oriented formalisms as well as state-oriented ones. State-oriented structures (like Kripke structures) give names to states and assert that some atomic propositions are true on each state. State-oriented temporal logics (like LTL and CTL) use these propositions as basic formulas. On the other hand, transition-oriented structures (like labeled transition systems, LTSs) also give names to states; transitions are associated to a non-unique action name. The only way to identify a transition is by looking at the adjacent states. Action names are used in formulas in transition-oriented temporal logics, like HML (Hennessy-Milner logic) [10] and ACTL* [6], but they are not formulas by themselves. In both types of logics, formulas are evaluated on states or on processes starting at an initial state. The idea of evaluating a formula on a transition sounds odd.

The origin of this discriminating view is a model of information systems in which indivisible and instantaneous events occur that change the system's state. We argue that, as successful as this model has been, it is not the whole story.

An imperative program is a sequence of instructions and it is natural to see a state in the gap between two consecutive instructions. The evolution of (part of) one such program can be represented like this:




We used `askH` as an abbreviation for "ask for handle to file" and `relH` for "release handle". The $s_i$ are names for states. Suppose now that this is not the only program or process running in the system, and the two `write` instructions involve a shared resource and need to be executed in mutual exclusion. There is a critical section, and it is natural to consider it as an unrefined composed state. This would be a bird's-eye view of the program:

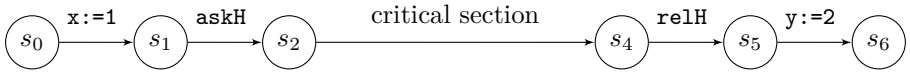



Now, we can define the proposition $\texttt{in-crit}_1$ to be true in the critical section; and $\texttt{in-crit}_2$ for the other program. We can assert mutual exclusion by the LTL formula

$$\text{MUTEX} := \Box(\neg\texttt{in-crit}_1 \vee \neg\texttt{in-crit}_2)$$

and perform verification as needed.

There is a better way. There is no reason why states $s_2$ and $s_4$ should be considered to be in the critical section. Indeed, the very question of whether a state belongs to the critical section is dubious: mutual exclusion is required when *doing* something, not while standing still. The alternative is to consider the critical section as an unrefined composed instruction:




However, mutual exclusion algorithms specified in this way are rare. In rewriting logic, specifications like the one on the left are way more usual than the one on the right:

```
rl [enter] : rem => crit .          rl [crit] : entering => exiting .
rl [exit] : crit => rem .           rl [rem] : exiting => entering .
```

The reason, or one of them, is that the formula MUTEX involves atomic propositions, and these are usually only available on states. If propositions on transitions were available, we could define `in-crit` to hold true on both writing instructions and then use the same formula MUTEX.

Another desirable property of such a program is that the shared file is not used unless access to it has been granted previously. In a state-based setting this could be expressed by the formula

$$\Box(\texttt{in-crit} \rightarrow \diamondsuit\texttt{gotFileH})$$

with $\diamondsuit$ meaning "at some past time". We would declare `in-crit` to hold on $s_3$, and `gotFileH` to hold on $s_2$. However, an unexpected way to go from $s_1$ to $s_2$ is discovered: through the action `hackH`, that gets a handle in a non-standard way, without asking or letting the system know. So, the question is not whether the program got access to the file, but how it did so. We need to refer to transitions in our formula. If we have action names available as basic formulas, we could use

$$\text{ASKB4USE} := \Box(\texttt{write1} \rightarrow \diamondsuit\texttt{askH})$$

as a more fitting formula. But note the difference between MUTEX and ASKB4USE as written above: while the former reflects mutual exclusion by itself, and is valid for any programs in which `in-crit` can be defined, the latter is only meaningful for file-sharing programs whose actions are named exactly as they appear in the formula. The way to go is by defining propositions on transitions `using-res` and `asking-for-res`, making them hold, for our example system, on the transitions labeled `write1` and `askH`, respectively, and then using the formula

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

This is a meaningful and general formula.

Consider now strategy languages. If, instead of verifying, we want to control the programs so as to ensure mutual exclusion, we can impose the following regular-expression strategy:

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

We have added process indices to instructions, and `other` is a shorthand for any instruction different from `askH` and `relH`. This expression ensures that after $\texttt{askH}_1$ no other instruction related to file handling is possible until $\texttt{relH}_1$ is performed; in particular, $\texttt{askH}_2$ is forbidden in between. And vice versa. Again, this expression can only be applied to systems that use these same labels, and often this is not possible or reasonable.

Specifications should be written just thinking of the behavior they model. Later, atomic propositions are defined and formulas or strategies are built on them. This is usual in state-oriented systems. When such a system is refined or otherwise modified, propositions are redefined if needed, but their names do not need to change, let alone the formulas.

The approach we propose to improve the strategy is to define `enter` and `exit` as propositions on transitions, common to both programs, representing the entrance to and exit from the critical section, respectively. These are true, for our example system, of `askH` and `relH`, respectively, and false otherwise. In Maude-like syntax, we would define, for both programs:

```
var I : Instruction .
eq askH |= other = false .
eq relH |= other = false .
eq I |= other = true [owise] .
```

```
var P : Prop .
eq askH |= enter = true .
eq relH |= exit = true .
eq I |= P = false [owise] .
```

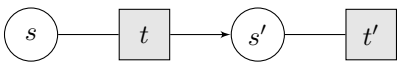
We can thus build the strategy:

```
(enter ; other* ; exit)*
```

Then, if there is the need to specify a system with exclusive access to a communication channel, using instructions, say, `get-channel` and `release-channel`, the same proposition names and the same strategy expression can be used.

In a word, the advantage of propositions on transitions (and on states) is *decoupling*: writing the system specification and writing the temporal property are independent tasks. The definition of propositions provides an interface. Changes in the identifiers used in the system need not be accompanied by changes in temporal formulas, but just in the definition of propositions. Temporal formulas and strategy expressions gain in generality.

Our proposal of giving transitions first-class citizenship is visually represented by making a box appear in the middle of every arrow representing a transition, with states in rounded shapes and transitions in rectangles:




In this way, every element—state or transition—is explicit and can be treated the same. The same as only states were treated before. There remains an only source of discrimination: while a state can have several arrows going in and out, a transition only has one of each. Monogamy for her, polygamy for him. (But see section on future work.)

### 1.1 Related work

Several temporal logics have been proposed that make joint use of actions and propositions on states: ACTL* [6], RLTL [17], SE-LTL [4], TLR* [15], ESTL [11]. There are also definitions of structures with mixed ingredients: LKS [4], $L^2TS$ [5], Petri nets [16].

The best moves towards fairness we know of are the *temporal logic of rewriting*, TLR*, and the *event-and-state-based temporal logic*, ESTL, the former designed for rewriting logic and the latter for Petri nets. The explanations and examples in [15] and [11] are good arguments for an egalitarian view. In both cases, the point is that some properties of systems can only be directly specified if we can talk about states and transitions within the same logic. Our formula ASKB4USE above was inspired by an example in [11]. In another example, this time from [15], fairness for a rule $\ell$ is expressed by the formula:

$$\Box\Diamond\texttt{enabled-}\ell \rightarrow \Box\Diamond\texttt{taken-}\ell$$

The proposition `enabled-`$\ell$ is on states: it means that the current state of the system has the form needed to apply rule $\ell$ to it. But `taken-`$\ell$ is on transitions: it tells that the transition being executed is according to rule $\ell$. Propositions on transitions are unavoidable. Or, rather, they are avoidable at the price of *cooking* the system (in Meseguer's terminology), making it artificially complex, so that some information about transitions is kept in states.

In ESTL, formulas are evaluated on *cuts* composed of places and transitions mixed together. A basic ESTL formula is a name of a place or of a transition. This is indeed an egalitarian view. What ESTL does not achieve is decoupling, as it uses literally names from the Petri net. We are not discussing Petri nets, although at least part of our work can probably be adapted to them through the implementation on rewriting logic proposed in [14].

Rewriting logic is an appropriate formalism to be egalitarian, because, as pointed out in [14], transitions are represented by proof terms, in the same way as states are represented by state terms. We expand on this below. But TLR* stays a step away from our aim, because, while it uses atomic propositions on states, it uses proof-term patterns (called *spatial actions*) to express properties of transitions. These patterns are less powerful than general propositions (for an example, a pattern cannot represent the set of proof terms in which a given variable has been instantiated with an even integer). But the real drawback is that a TLR* formula is only meaningful for algebraically specified systems, and for a particular algebraic specification. They use literally elements from the text of the specification, so that no decoupling is achieved. In contrast, formulas using propositions—on states, like CTL*, or on transitions, as we advocate—are meaningful for any system where the atomic propositions can be defined, irrespective of the formalism used to specify it. Notably, we know of three implementations of model checkers for (the linear-time subset of) TLR*, and all of them propose some kind of propositions on transitions [2,1,13].

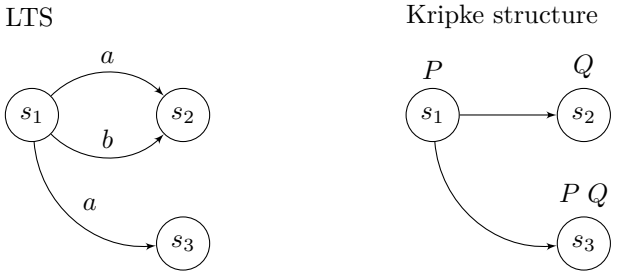
### 1.2 Our contributions in this paper

In Section 2, we propose *egalitarian structures*, and show how they encompass typical state-based and event-based structures. In Section 3, we show how systems (especially rewrite ones) can be given egalitarian semantics. In Section 4, we describe translations from egalitarian structures to state-based ones. Correspondingly, we describe a way to split each rule of a rewrite system into two *halves*, so that new states arise that represent the transitions of the original system. In Section 5, we show how also temporal logics can be translated, and how all this allows performing verification on the resulting state-based systems to draw conclusions about the original, egalitarian systems. In Section 6, we describe our implementation, that allows the specification and model checking of egalitarian structures in Maude. Sections on future work and conclusions complete the paper.

This technical report is an extended version of a paper accepted for WRLA 2016. The Maude code for our implementation and some examples can be found at `http://maude.sip.ucm.es/syncprod`. The latest version of this paper can also be downloaded there.

## 2 Egalitarian structures

Let us recall the usual definitions of labeled transition system (LTS) and Kripke structure. An LTS is given by a tuple $(S, \Lambda, \delta)$, where $S$ is the set of states, $\Lambda$ the alphabet of actions, and $\delta : S \times \Lambda \to 2^S$ the non-deterministic transition function. A Kripke structure is given by a tuple $(S, R, \mathrm{AP}, L)$, where $S$ is again the set of states, $R \subseteq S^2$ the transition relation, AP the set of atomic propositions, and $L : S \to 2^{\mathrm{AP}}$ the labeling function, that assigns to each state the set of propositions that hold true on it. Graphically:




In some cases, both action names and atomic propositions on states are used in a mixed structure. We propose egalitarian structures as a generalization of all these cases. An *egalitarian structure* is given by a tuple $(S, T, R, \mathrm{AP}, L)$, where:

- $S$ is the set of states;
- $T$ is the set of transitions;
- $R \subseteq (S \times T) \cup (T \times S)$ is the bipartite accessibility relation that is functional on $T$, that is, for each $t \in T$ there are exactly one $s \in S$ and exactly one $s' \in S$ such that $(s, t) \in R$ and $(t, s') \in R$;

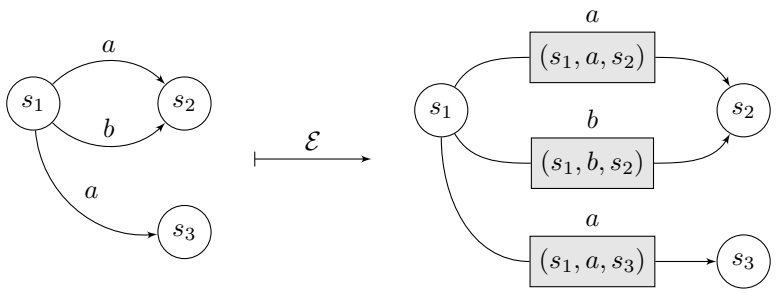
– AP is the set of atomic propositions on both states and transitions;
– $L : S \cup T \to 2^{\mathrm{AP}}$ is the labeling function for both states and transitions.

The same atomic proposition can be defined on states and on transitions in the same structure. Indeed, it is plausible that a proposition that is satisfied on several consecutive states also holds on the transitions between them. As pointed out in the introduction, the functionality of $R$ on $T$ is the only discriminatory requirement we allow.

Egalitarian structures generalize LTSs and Kripke structures. An LTS $\mathcal{L} = (S_{\mathcal{L}}, \Lambda_{\mathcal{L}}, \delta_{\mathcal{L}})$ can readily be made into an equivalent egalitarian structure $\mathcal{E}(\mathcal{L}) = (S_{\mathcal{E}}, T_{\mathcal{E}}, R_{\mathcal{E}}, \mathrm{AP}_{\mathcal{E}}, L_{\mathcal{E}})$ by defining:

– $S_{\mathcal{E}} := S_{\mathcal{L}}$;
– $T_{\mathcal{E}} := \{(s, \lambda, s') \in S_{\mathcal{L}} \times \Lambda_{\mathcal{L}} \times S_{\mathcal{L}} : s' \in \delta_{\mathcal{L}}(s, \lambda)\}$;
– $R_{\mathcal{E}}$ is given by $s \, R_{\mathcal{E}} \, (s, \lambda, s')$ and $(s, \lambda, s') \, R_{\mathcal{E}} \, s'$;
– $\mathrm{AP}_{\mathcal{E}} := \Lambda_{\mathcal{L}}$;
– $L_{\mathcal{E}}((s, \lambda, s')) := \{\lambda\}$, and $L_{\mathcal{E}}(s) := \emptyset$.

Graphically:



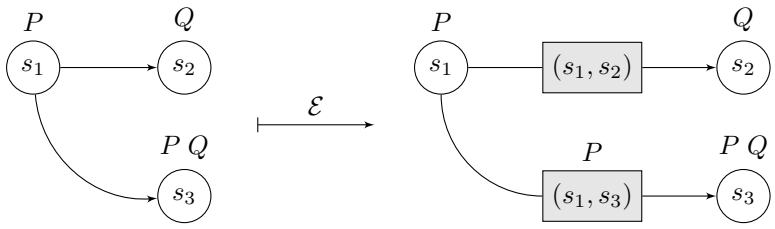

A translation quite similar to this one is described in [6], although there the destination is a Kripke structure, instead of an egalitarian one.

Atomic propositions in $\mathcal{E}(\mathcal{L})$ represent actions in $\mathcal{L}$, so it seems fitting that states are assigned no label. The equivalence between these two structures is intuitively clear and we do not care to make it formal in this paper.

A Kripke structure $\mathcal{K} = (S_{\mathcal{K}}, R_{\mathcal{K}}, \mathrm{AP}_{\mathcal{K}}, L_{\mathcal{K}})$ can be made into an equivalent egalitarian structure $\mathcal{E}(\mathcal{K}) = (S_{\mathcal{E}}, T_{\mathcal{E}}, R_{\mathcal{E}}, \mathrm{AP}_{\mathcal{E}}, L_{\mathcal{E}})$ by defining:

– $S_{\mathcal{E}} := S_{\mathcal{K}}$;
– $T_{\mathcal{E}} := R_{\mathcal{K}}$ (considered as a subset of $S_{\mathcal{K}}^2$);
– $R_{\mathcal{E}}$ is given by $s \, R_{\mathcal{E}} \, (s, s')$ and $(s, s') \, R_{\mathcal{E}} \, s'$;
– $\mathrm{AP}_{\mathcal{E}} := \mathrm{AP}_{\mathcal{K}}$;
– $L_{\mathcal{E}}(s) := L_{\mathcal{K}}(s)$, and $L_{\mathcal{E}}((s, s')) := L_{\mathcal{K}}(s) \cap L_{\mathcal{K}}(s')$.

Graphically:




The choice $L_{\mathcal{E}}((s, s')) := L_{\mathcal{K}}(s) \cap L_{\mathcal{K}}(s')$ allows the continuity of satisfaction, that is, that a proposition true on two consecutive states is also true while traveling between them. Whether this is appropriate depends on the precise concept of equivalence between $\mathcal{K}$ and $\mathcal{E}(\mathcal{K})$, but, again, we do not care to make it formal. We will have something more to say on this below when dealing with rewriting logic.

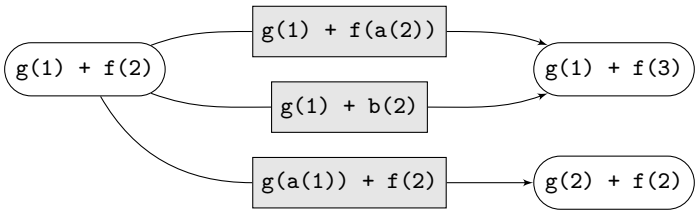
## 3 Egalitarian semantics for rewrite systems

The embedding of Kripke structures in egalitarian ones given above implies that any specification that is interpretable on the former can readily use the latter instead. The definition of transitions as pairs of states can be improved in some cases, because we can produce *objects* (read *terms*, if you prefer) that properly identify transitions without explicitly relying on the states around. In [3], for instance, it is shown how transitions in CCS can be represented by *proof terms* derived from a system of rules implementing the semantics of the language. These proof terms, however, are not CCS terms—they are built according to a different syntax.

Rewriting logic provides a better example. A transition in a rewrite system is represented by a proof term [14], just as a state is represented by a term of appropriate sort. Proof terms need some extra symbols in the signature, but they are still terms, and structural information can be drawn from them. Consider this toy example system:

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

From the initial state `g(1) + f(2)` the three possible transitions are shown here, with their respective proof terms:



Each proof term includes the rule label in the context in which it is being applied, and with the values that instantiate the variables in the rule. Seen in this way, rewrite systems are naturally egalitarian, and are easily interpretable on egalitarian structures. (Indeed, as rewriting logic is well suited for implementing language syntax and semantics, proof terms become available for any language, if only in this indirect way.)

We propose a definition of rewrite system slightly different from the usual one, in order to make its egalitarian nature clearer, and also so that our ensuing exposition gets easier. Namely, we include in its signature the declaration of rule labels. It is not the definition of a more egalitarian kind of rewrite system, but a more egalitarian definition of the same concept.

In the setting of rewriting logic, the standard definition of a rewrite system (or rewrite theory) is given by a tuple $(S, O, E \cup Ax, R)$, where $S$ is a set of declarations of sorts (sometimes with a subsort relation among them), $O$ is a set of declarations of function symbols (operators), $E$ is a set of equations, $Ax$ is a set of equational attributes, and $R$ is a set of rewrite rules. Sometimes, $S$ and $O$ are denoted together by $\Sigma$ and called the *signature*.

The egalitarian definition of a rewrite system is given by a tuple $(S, O, L, E \cup Ax, R)$, where $L$, the only novelty, is a set of declarations of rule labels. As rule labels are used to identify transitions (by building proof terms), it is fair that they are declared, as operators are. Each rule-label declaration has the same form as an operator declaration, that is, it contains argument sorts and a result sort. For the example above, the rule label declarations would be:

```
lb a : Nat -> Nat .
lb b : Nat -> SomeSort .
```

The argument sorts are the ones of the variables that appear in the rule with that label (in their textual order if there are several variables, or the empty list of sorts if there are none). It is a requirement for any valid rewrite rule that both sides are terms of the same kind. We add to this that the result sort of the rule label has to be of the same kind as well. In a simple but typical case, both sides of the rule would be terms of the same sort `State`, and so will be the result sort of its label.

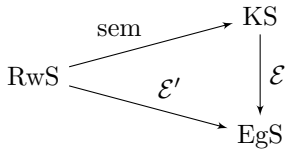
For verification and other purposes, one often assumes, in the standard setting, that $S$ includes declarations for sorts `State` and `Prop`, and that $O$ includes the infix symbol $\models : \texttt{State} \times \texttt{Prop} \to \texttt{Bool}$. For the egalitarian setting we assume the sort `Elem` to represent both states and proof terms. We also still need `Prop`. Thus, the operator $\models$ is declared as $\models : \texttt{Elem} \times \texttt{Prop} \to \texttt{Bool}$. Any sort can include terms built using symbols from $O$ and from $L$, but with either one or no symbol from $L$. (This reflects the remaining discrimination pointed out in the introduction.) States are represented by terms of sort `Elem` with no symbol from $L$; transitions are terms of sort `Elem` with exactly one occurrence of a symbol from $L$ (so-called one-step proof terms). When needed, we assume the existence of sorts `State` and `Trans`, defined as subsorts of `Elem` as described; or, equivalently, we assume the existence and definition of predicates $\texttt{isState}, \texttt{isTrans} : \texttt{Elem} \to \texttt{Bool}$. No particular sort is needed for other proof terms, that is, `f(2)` and `b(2)` are both

of sort `SomeSort`. (More precise, though slightly different, algebraic definitions are given in [13].)

The semantic function, that we denote as $\mathcal{E}'$, is now easy. For a rewrite system $\mathcal{R} = (S_\mathcal{R}, O_\mathcal{R}, L_\mathcal{R}, E_\mathcal{R} \cup Ax_\mathcal{R}, R_\mathcal{R})$, its semantics are given by $\mathcal{E}'(\mathcal{R}) = (S_\mathcal{E}, T_\mathcal{E}, R_\mathcal{E}, \mathrm{AP}_\mathcal{E}, L'_\mathcal{E})$, where:

- $S_\mathcal{E} := T_{S_\mathcal{R} \cup O_\mathcal{R} \,/\, E_\mathcal{R} \cup Ax_\mathcal{R}, \texttt{State}}$ (terms of sort `State` modulo equations);
- $T_\mathcal{E} := T_{S_\mathcal{R} \cup O_\mathcal{R} \cup L_\mathcal{R} \,/\, E_\mathcal{R} \cup Ax_\mathcal{R}, \texttt{Trans}}$ (terms of sort `Trans` modulo equations);
- $R_\mathcal{E}$ is given by $s \, R_\mathcal{E} \, t$ and $t \, R_\mathcal{E} \, s'$ for each $t$ that is a proof term for a one-step derivation from $s$ to $s'$;
- $\mathrm{AP}_\mathcal{E} := T_{S_\mathcal{R} \cup O_\mathcal{R} \,/\, E_\mathcal{R} \cup Ax_\mathcal{R}, \texttt{Prop}}$ (terms of sort `Prop` modulo equations);
- $L'_\mathcal{E}(s) := \{p \in \mathrm{AP}_\mathcal{E} : s \models p = \texttt{true} \text{ modulo } E_\mathcal{R} \cup Ax_\mathcal{R}\}$, for $s \in S_\mathcal{E}$;
- $L'_\mathcal{E}(t) := L'_\mathcal{E}(s) \cap L'_\mathcal{E}(s')$ for $t \in T_\mathcal{E}$, and $s$, $s'$ such that $s \, R_\mathcal{E} \, t$ and $t \, R_\mathcal{E} \, s'$.

The definition of the labeling, in particular, reflects the one for the embedding of Kripke structures in egalitarian ones given in the previous section to guarantee continuity of satisfaction. Thus, we have that this diagram commutes:




In it, we denote as RwS the class of rewrite systems, as KS the class of Kripke structures, and as EgS the class of egalitarian structures. Also, "sem" is the semantics based on term algebras described in [7], $\mathcal{E}$ is the embedding of Kripke structures in egalitarian ones from Section 2, and $\mathcal{E}'$ is the semantics just defined.

The labeling deserves a deeper thought. In a rewrite system, seen in an egalitarian way, atomic propositions and the equations defining them apply equally to states and to transitions. Often, a state and a neighboring transition have similar algebraic shapes, and that eases a continuous definition of satisfaction for them. In the simple example above, consider this proposition `has-g1`:

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

This equational definition makes at once the proposition `true` for the transition `g(1) + b(2)` and for the state `g(1) + f(2)`, and `false` for the transition `g(a(1)) + f(2)` and for the state `g(2) + f(2)`.

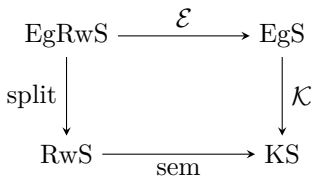
Considering this, a better, more egalitarian, and more flexible definition of the labeling for any `Elem` $e$ is:

- $L_\mathcal{E}(e) := \{p \in \mathrm{AP}_\mathcal{E} : e \models p = \texttt{true} \text{ modulo } E_\mathcal{R} \cup Ax_\mathcal{R}\}$.

The semantics of rewrite systems as egalitarian structures according to this labeling is denoted as $\mathcal{E}$ (instead of the previous $\mathcal{E}'$) from now on.

## 4 Translation to familiar grounds

We define now functions $\mathcal{K}$ and "split" so that the following diagram commutes:




EgRwS is the class of rewrite systems defined in the egalitarian way. (More precisely, the diagram commutes only when monogamy of transitions is guaranteed; more on this below.)

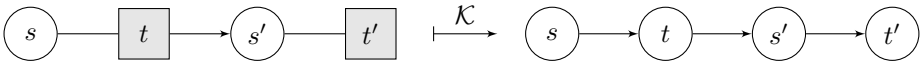
Our aim is to benefit from tools and concepts available for the lower half of the diagram, and use them in the egalitarian systems and structures on the upper half. For instance, in Section 5 we define the satisfaction of a temporal formula on an egalitarian structure based on the standard satisfaction on a Kripke structure, and in Section 6 we use existing model checkers with a new mission.

Sometimes we call *split* systems or structures the ones that result from applying $\mathcal{K}$ or "split". Also, the states of a split system or structure that were originally a transition are called *t-states*; the others are *s-states*.

Transforming an egalitarian structure into a Kripke one is accomplished in this simple way:

$$(S, T, R, \mathrm{AP}, L) \quad \xrightarrow{\mathcal{K}} \quad (S \cup T, R, \mathrm{AP}, L).$$

That is, we make old states and transitions into new states. Visually, this is reflected by changing square shapes into rounded ones:




(Note that this is not the inverse of the embedding $\mathcal{E} : \mathrm{KS} \to \mathrm{RwS}$ given above.)

The transformation "split" on rewrite systems, designed to reflect $\mathcal{K}$, is more involved. The idea is removing each rule

| `rl [`$\ell$`] :` $l(\overline{x}) \to r(\overline{x})$

and adding in its place

| `rl [`$\ell_1$`] :` $l(\overline{x}) \to \ell(\overline{x})$ and `rl [`$\ell_2$`] :` $\ell(\overline{x}) \to r(\overline{x})$

When a term is rewritten using rule $\ell_1$, the result is the proof term for the transition using rule $\ell$, as we need. In the example above, the rule

```
| rl [b] : f(N) => f(3) .
```

gets split as

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

The term `g(1) + f(2)` is rewritten by rule `b1` to the proof term `g(1) + b(2)`. This splitting of a rule into two produces new states in the split rewrite system that correspond to the new states produced by $\mathcal{K}$.

For conditional rules, the condition applies to the firing of the rule, not to its continuation. Each conditional rule

| `crl [`$\ell$`] :` $l(\overline{x}) \to r(\overline{x})$ `if` $C(\overline{y})$

is removed and two rules are added:

| `crl [`$\ell_1$`] :` $l(\overline{x}) \to \ell(\overline{x},\overline{y})$ `if` $C(\overline{y})$ and `rl [`$\ell_2$`] :` $\ell(\overline{x},\overline{y}) \to r(\overline{x})$

The variables in the tuple $\overline{y}$ are the new ones, not in $\overline{x}$, that appear in matching and rewriting conditions in $C$.

More formally now, given $\mathcal{R} = (S_1, O_1, L_1, E_1 \cup Ax_1, R_1)$, we build $\text{split}(\mathcal{R}) = (S_2, O_2, E_2 \cup Ax_2, R_2)$ by this series of steps:

1. $S_2$ is produced by renaming sorts `State` to `SState`, `Trans` to `TState` and `Elem` to `State` in $S_1$ (so `SState` and `TState` are subsorts of `State`); this renaming must be propagated all through the specification;
2. letting $O_2 := O_1 \cup L_1$ (that is, rule labels are transformed into operators);
3. letting $E_2 := E_1$, and $Ax_2 := Ax_1$;
4. splitting rules in $R_1$ to produce the ones in $R_2$, as explained above.

There is still a difficulty with the resulting system. For it to be equivalent to the original one, we need to ensure that half-rule $\ell_1$ is always immediately followed by $\ell_2$, for each original rule $\ell$. Otherwise, another half-rule $\ell_1'$ could take place in between, a behavior not possible in the original system. Again, this reflects the discrimination of monogamy for transitions, polygamy for states.

A solution is restricting our attention to *topmost* rewrite systems. A topmost rewrite system is one in which all rewrites happen on the whole state term—not on its subterms. Formally, this is guaranteed by requiring that all rule labels have result sort `Elem` (or its subsort `Trans`), and that the sort `Elem` does not appear as argument in any constructor or rule label, so that no term of sort `Elem` can be subterm of another term of the same sort. In particular, this prevents that the left-hand side of a rule is a variable (of sort `State`). If this happened, the resulting proof term would have shape $\ell(\mathtt{S})$, for `S` of sort `State`, so the split system would not be topmost even though the original one was. Many interesting rewrite systems are topmost or can be easily transformed into an equivalent one that is topmost and formally similar [9].

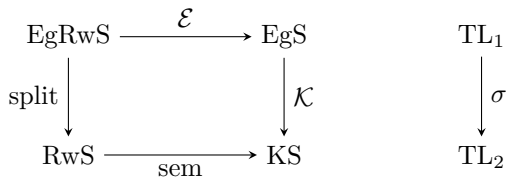
In a topmost system, the term $\ell(t_1, \ldots, t_n)$, resulting from applying the half-rule $\ell_1$, can only be rewritten using half-rule $\ell_2$, as we need. Or, this is so if there are no two rules with the same label and the same argument sorts. We assume that our systems fulfill this mild requirement. This is the same reasonable requirement made to overloaded function symbols.

We assume from now on that transition monogamy is guaranteed in split systems in some way. Thus, the diagram at the beginning of this section is commutative. (The proof is straightforward with all the definitions given up to now in the paper.)

## 5 Temporal logics on egalitarian structures

As LTSs and Kripke structures can be seen as particular cases of egalitarian structures, any temporal logic designed for the former ones can also be interpreted on the latter. This includes HML [10] and the $\mu$-calculus [12], and all the CTL* family [8]. More interestingly, mixed logics like ACTL* [6] that use at the same time action identifiers and atomic propositions on states, are interpretable on egalitarian structures.

As introduced above, we would like to define and verify the satisfaction of temporal formulas on egalitarian structures by translating the problem to well-known non-egalitarian settings. That is, we want temporal logics $\mathrm{TL}_1$ and $\mathrm{TL}_2$ and a translation $\sigma$ to complete the previous diagram to this one:




$\mathrm{TL}_2$ is any state-based logic, like LTL. In Meseguer's terminology [15], the maps

$$(\mathcal{K}, \sigma) : \mathrm{EgS} \times \mathrm{TL}_1 \to \mathrm{KS} \times \mathrm{TL}_2$$
$$(\mathrm{split}, \sigma) : \mathrm{EgRwS} \times \mathrm{TL}_1 \to \mathrm{RwS} \times \mathrm{TL}_2$$

must be *faithful maps of tandems*, that is, the satisfaction relation must be preserved. Indeed, instead of giving a new, independent definition for semantics, we consider they are given by $\sigma$ and define satisfaction on the upper half of the diagram as satisfaction of translations on the lower half:

$$\mathcal{R}, e \models_{eg} \varphi \quad \text{iff} \quad \mathrm{split}(\mathcal{R}), e \models \sigma(\varphi).$$

Here, $\models_{eg}$ is the egalitarian satisfaction relation, $\mathcal{R}$ is an egalitarian rewrite system, $e$ a state or transition, and $\varphi$ a temporal formula. Remember that, for this definition of $\models_{eg}$ to work as expected, we need monogamous transitions in $\mathrm{split}(\mathcal{R})$.

**Raw LTL.** The perfect temporal logic to play the role of $\mathrm{TL}_1$ would also be egalitarian, to exploit the full potential of egalitarian structures. By that we mean a logic able to use propositions both on states and on transitions as its basic formulas, and to evaluate formulas on transitions. In a different way: we want $\sigma$ to be onto. To the best of our knowledge, no such logic has been proposed, although TLR* [15] and SE-LTL [4] come close.

The obvious onto transformation is the identity: $\mathrm{TL}_1 = \mathrm{TL}_2 = \mathrm{LTL}$ and $\sigma = \mathrm{id}$. Thus, for example, the *next* operator $\bigcirc$ has to be interpreted on an egalitarian structure as "in all outgoing transitions" when on a state, and "in the landing state" when on a transition. That gives the specifier full power. This

is equally valid for state-based temporal logics other than LTL. The moral is: instead of (or in addition to) looking for new state-and-transition-based temporal logics, use well-known state-based logics on split systems.

**From LTL$_{eg}$ to LTL.** Consider a Kripke structure $\mathcal{K}$ and an LTL formula $\varphi$ interpreted on $\mathcal{K}$. We can interpret $\varphi$ on $\mathcal{E}(\mathcal{K})$, the embedding of $\mathcal{K}$ as egalitarian structure, by pretending that transitions are not present and jumping from state to state. Let us refer to LTL with these semantics on egalitarian structures as LTL$_{eg}$. We want to find the $\sigma$ that makes faithful the map of tandems $(\mathcal{K}, \sigma) : \mathrm{EgS} \times \mathrm{LTL}_{eg} \to \mathrm{KS} \times \mathrm{LTL}$. From a practical point of view this is pointless, as it amounts to translating a problem $(\mathcal{K}, \varphi)$ on $\mathrm{KS} \times \mathrm{LTL}$ to the more complex one $(\mathcal{K}(\mathcal{E}(\mathcal{K})), \sigma(\varphi))$ on the same setting; but it is an interesting exercise.

The *next* operator $\bigcirc$ is originally only interested in states, so it must skip t-states. The translation $\sigma$ must duplicate this operator: $\sigma(\bigcirc\varphi) := \bigcirc \bigcirc \sigma(\varphi)$. The *at all future times* operator $\Box$, being an LTL operator, must rather be understood as *on all future states*. The translation $\sigma$ must make it skip every second state, which is known to be non-doable in LTL [18]. We have to use the atomic proposition `isTrans`, true for t-states and false otherwise, and define $\sigma(\Box\varphi) := \Box(\texttt{isTrans} \vee \sigma(\varphi))$.

However, intuitively, something is wrong in the specification of a system if a property that is supposed to hold at all future times does not hold while transitions are being executed. If I am feeling sleepy until lunch and also after lunch, so would I be while having lunch. Remember the discussion at the end of Section 3 about the proposition `has-g1`. For another example, think of a system whose states are given as *soups* of objects, that is, independent objects tied by a commutative and associative operator (often represented by empty syntax). This could be such a state:

```
<client1, waiting, info1> <client2, running, info2> <server, client2>
```

We are interested in knowing whether some client is waiting. The proposition `some-waiting` can be defined like this:

```
eq <C, waiting, I> Rest |= some-waiting = true .
eq Conf |= some-waiting = false [owise] .
```

When `client2` finishes its communication with the server, the system executes the rule

```
rl [finish] : <C, running, I> <server, C>
           => <C, finished, I> <server, noclient> .
```

and goes to state

```
<client1, waiting, info1> <client2, finished, info2> <server, noclient>
```

by means of the transition

```
<client1, waiting, info1> finish(client2, info2)
```

The point to note is that `some-waiting` is true, as defined, in both states and in the transition.

**From TLR* to CTL*.** In a way, the egalitarian logic we are looking for is close to TLR* without the R: while TLR* is designed to be used on rewrite systems, we aim at a logic of more general usability. The approach to propositions on transitions in TLR* is through the use of so-called *spatial actions*, that is, patterns for proof terms. A single rule label $\ell$, for instance, is a valid spatial action. Variable instantiations and contexts for rewriting can also be specified. For each of these patterns, an equivalent atomic proposition on t-states can be defined. Equivalent in the sense that a t-state satisfies the proposition iff it matches the pattern. This was implemented in [1] and in [13]. We assume this equivalence, so that any spatial action appearing in a TLR* formula can appear as a proposition in a CTL* formula. (Defining a proposition equivalent to a single rule-label pattern involves exploring the proof term to search for the label at any nesting level. However, rules tend to be applied to particular spots in the term. Compare to operators: it is rare that we are interested in whether a particular operator appears at any nesting level on a state term.)

In TLR*, propositions on states and spatial actions are clearly separate entities: the former are only tested on states, the latter only on transitions. But when interpreted on $\mathcal{K}(\mathcal{R})$ both are propositions on states. In order to be able to define $\sigma$, we need that $\mathcal{K}(\mathcal{R})$ includes two subsorts of `Prop`: `SProp` and `TProp`.

From TLR*'s point of view, a transition is tied to its origin state. Thus, if $P_t$ is a `TProp` and $P_s$ is an `SProp`, the formula $\bigcirc(P_t \wedge P_s)$ means "$P_s$ must hold on the next state, and $P_t$ must hold on the transition going out from that next state". Likewise, the formula $\Box P_t$ means "$P_t$ must hold in all future transitions".

This deserves formalization. Taking as primitive constructs for TLR* negation, disjunction, *next*, *until*, and existential quantification on paths, we define $\sigma : \text{TLR*} \to \text{CTL*}$ by:

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

The proposition `isState`, as explained in Section 3, needs to be defined in $\mathcal{K}(\mathcal{E})$ as `true` on s-states and `false` on t-states.

The previous two examples defined new semantics for raw LTL and $\text{LTL}_{eg}$ through the translations $\sigma$. This case is different, because TLR* already has semantics—the ones given in [15]. The following result is in order.

**Proposition 1.** *The map*

$$(\mathcal{K}, \sigma) : \text{EgS} \times \text{TLR*} \to \text{KS} \times \text{CTL*}$$

*is a faithful map of tandems, that is,*

$$\mathcal{E}, s \models \varphi \iff \mathcal{K}(\mathcal{E}), s \models \sigma(\varphi).$$

*Proof.* The satisfaction symbol on the right is the usual one for Kripke structures and CTL*. The one on the left is defined in [15]. But these semantics of Meseguer are defined on a structure that is not a pure Kripke structure and not an egalitarian one, but something in between. For a proper statement and a proof to be possible we should formally define satisfaction of TLR* formulas on egalitarian systems. We do it now, adapting [15].

First, a *run* on an egalitarian structure is a sequence of states and transitions $s_0, t_1, s_2, t_3 \dots$ such that $(s_i, t_{i+1}) \in R$ for all even $i$ and $(t_i, s_{i+1}) \in R$ for all odd $i$. We are particularly interested in infinite runs. We denote by $\mathrm{Run}(\mathcal{E})_s$ the set of infinite runs of $\mathcal{E}$ starting at state $s$. (We are already being non-egalitarian: it cannot be avoided, as the statement mentions satisfaction on states, not on transitions, and also because, as already mentioned, the semantics in [15] ties transitions to their origin states.)

We use the following syntax for TLR*, slightly adapted from [15]. We use as primitive symbols negation, disjunction, *next*, *until*, and existential quantification on paths, as above. The two categories defined are formulas on states, $\varphi_s$, and formulas on paths (starting at states), $\varphi_p$. The symbol $p_s$ represents an atomic proposition on states, and $p_t$ on transitions. (Remember that here we do not follow Meseguer, who uses spatial actions instead of propositions on transitions.)

- $\varphi_s ::= \top \mid \neg\varphi_s \mid \varphi_s \vee \varphi'_s \mid \mathbf{E}\,\varphi_p$,
- $\varphi_p ::= \top \mid p_s \mid p_t \mid \neg\varphi_p \mid \varphi_p \vee \varphi'_p \mid \mathbf{X}\,\varphi_p \mid \varphi_p\,\mathbf{U}\,\varphi'_p$.

Our choice of syntax entails that a proposition on states $p_s$ is *not* a formula on states. Instead, it is $\mathbf{E}\,p_s$ that expresses the property that $p_s$ holds at the initial state. We admit this is odd, but it is more symmetrical and clean. It is always possible to introduce shortcuts or syntactic sugar afterwards. Anyway, usual definitions of syntax for TLR* or CTL* entail that $\mathbf{E}\,\varphi_p$ is a formula on paths, which is equally odd.

We need to define two satisfaction relations. Some notation is needed: if $\rho$ is an infinite run, $\rho(i)$ denotes the $i$-th element of the run (starting at 0) and $\rho(i, \infty)$ the infinite run that results from $\rho$ by removing the $i$ first elements. The cases for negation, disjunction, and constant truth are defined as usual and we omit them.

- $\mathcal{E}, s \models \mathbf{E}\,\varphi_p \iff$ there exists $\rho \in \mathrm{Run}(\mathcal{E})_s$ such that $\mathcal{E}, \rho \models \varphi_p$;
- $\mathcal{E}, \rho \models p_s \iff p_s \in L(\rho(0))$;
- $\mathcal{E}, \rho \models p_t \iff p_t \in L(\rho(1))$;
- $\mathcal{E}, \rho \models \mathbf{X}\,\varphi_p \iff \mathcal{E}, \rho(2, \infty) \models \varphi_p$;
- $\mathcal{E}, \rho \models \varphi_p\,\mathbf{U}\,\varphi'_p \iff$ there exists $k \in \mathbb{N}$ such that $\mathcal{E}, \rho(2k, \infty) \models \varphi'_p$ and for all $i \in \{0, \dots, k-1\}$ we have $\mathcal{E}, \rho(2i, \infty) \models \varphi_p$.

Now, checking that these semantics agree with the ones given in [15] and completing the real proof of the proposition are both straightforward tasks. □

## 6 Our implementation

We have implemented in Maude the translation just defined, but restricted to the linear-time subset of TLR*, called LTLR, and to topmost systems. That is:

$$(\text{split}, \sigma) : \text{EgRwS} \times \text{LTLR} \to \text{RwS} \times \text{LTL}.$$

The implementation and some examples are available for download from our website `http://maude.sip.ucm.es/syncprod`. Appendix A contains detailed usage instructions.

The function "split" is implemented by a module operator `SPLIT[ModName]`. It produces a module with each original rule split into two, and with the original rule labels added as operators to the signature. Users, after coding a system module, say `Orig`, can import the split module by using `protecting SPLIT[Orig]`. Then, they have available sorts `StateOrig` (a renaming of the original `State`) and `TransOrig`, and also a new sort `State`, which is a supersort of the other two.

In the exposition in Section 3 we proposed the name `Elem` to include states and transitions. However, we want to be ready for our future developments in which we anticipate that nested module operators will be used. That is why we always assume that the input system has a sort named `State`, and we guarantee that the same is true for the produced system. Module operators observing this convention can be combined. For instance, `SPLIT[SPLIT[Orig]]` is a valid module expression.

If model checking is the aim, atomic propositions on s-states and t-states can be declared and their satisfaction defined by the usual means. The model checking function, `modelCheck`, expects an LTL formula, that it interprets in the split module (without any consideration to the fact that it is a split module). We have implemented the syntax of LTLR and the translation $\sigma$ described at the end of the previous section (except that we do not need quantification on paths, as we restrict to linear-time). The function that performs the translation is called `LTLR`. We have not included in this implementation spatial actions, so our flavor of LTLR uses propositions on transitions and no spatial actions. The syntax for LTLR formulas has been defined with a symbol "`@`" attached to each logical symbol, to avoid clashes with LTL syntax: `@True`, `@->`, and so on. Not a beautiful choice, but acceptable for a prototype. The formula `LTLR(@~ P @-> Q)`, for example, can be used in the model checker, assuming propositions `P` and `Q` have been properly declared and defined, each one either of sort `SProp` or `TProp`.

We have coded and model checked some examples, including a couple of them borrowed from the ones made available online by Bae and Meseguer relating their paper [2]. It is interesting to note that one of the examples, on Dekker's algorithm, is not directly specified in Maude, but in a C-like language whose simple semantics are included within the same example. Thus, proof terms and everything about transitions are available for any language implemented in Maude. To test the performance of our tool, we have found useful an example system about a communication channel described in [15]. The system contains a parameter, `maxFaults`, that limits the number of duplications and losses of messages the

communication can suffer. This single number allows tuning the size of the state space and drawing some conclusions on the performance of the model checker.

For a quick reminder, these are the rules of the system:

```
crl [tick] : {X | T} => {next(X) | s(T)} if not other-rule-enabled(X) .
rl [req] : {X [C, S, Q, N, true, noAnswer] | T}
        => {X [C, S, Q, N, false, noAnswer] (N copies S <| C, Q) | T} .
rl [reply] : {X [S] (S <| C, Q) | T}
          => {X [S] (C <| S, f(S, C, Q)) | T} .
rl [rec] : {X [C, S, Q, N, B, A'] (C <| S, A) | T}
        => {X [C, S, Q, N, B, A] | T} .
rl [dupl] : {X Msg dupl(s(N)) | T} => {X Msg Msg dupl(N) | T} .
rl [dupl-quit] : {X dupl(s(N)) | T} => {X dupl(0) | T} .
rl [loss] : {X Msg loss(s(N)) | T} => {X loss(N) | T} .
rl [loss-quit] : {X loss(s(N)) | T} => {X loss(0) | T} .
```

We have chosen a pure LTL formula and have model checked it in the standard way. Then, we have split the system specification and translated the formula (considering it is in LTLR) and have model checked again. In short, we are performing an equivalent model checking in a more involved and costly way. The aim is to get an idea of how much is lost in performance to pay for being egalitarian. The data is in this table:

| | standard LTL | | split | |
|---|---|---|---|---|
| maxFaults | states | secs | states | secs |
| 1 | 287 | 0 | 1,321 | 0 |
| 2 | 1,007 | 0 | 5,489 | 0 |
| 3 | 2,455 | 0 | 14,870 | 0 |
| 4 | 5,025 | 0 | 32,988 | 0 |
| 5 | 9,283 | 0 | 65,088 | 0 |
| 6 | 15,978 | 1 | 118,420 | 1 |
| 7 | 26,077 | 1 | 202,686 | 2 |
| 8 | 40,803 | 1 | 330,534 | 3 |
| 9 | 61,676 | 1 | 518,097 | 4 |
| 10 | 90,557 | 2 | 785,577 | 6 |
| 11 | 129,695 | 3 | 1,157,874 | 10 |
| 12 | 181,777 | 5 | 1,665,260 | 18 |
| 13 | 249,981 | 8 | 2,344,098 | 30 |
| 14 | 338,032 | 17 | 3,237,606 | 63 |
| 15 | 450,261 | 33 | 4,396,666 | 105 |

It must be noted that this system has an infinite number of reachable states. It is a surprise that Maude's model checker behaves gracefully on it. The reason must be that the system is finitely branching and that the property we try to verify is indeed satisfied in finite time in every computation.

According to the table, the transitions in the original system seem to outnumber the states, and this results in large split systems. Each state on the split system needs less mean time to be processed than each state on the original, presumably because t-states have unique in and out arrows.

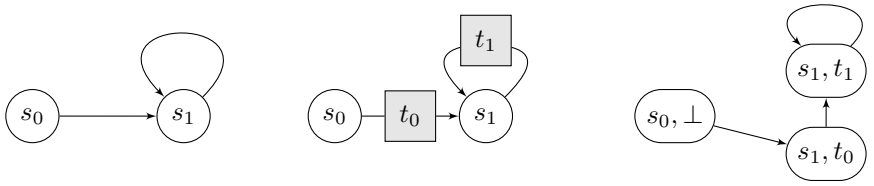
Note that this extra complexity is not introduced by our splitting translation, but by our egalitarian view. Transitions have to be explored, either as such transitions or as new states after the translation.

### 6.1 An alternative translation

Already in [15], Meseguer proposed a different translation from TLR* to CTL*, with a corresponding translation of rewrite systems. It was later implemented in Maude by Bae for LTLR in [2], with the explicit aim of using Maude's LTL model checker. Both their translation and their implementation differ from ours.

They include in each state the information about the transition that took to it. If a given state has several transitions leading to it, the resulting system has a copy of the state for each such transition. The translation of temporal formulas is based on the replacement of each occurrence of a proposition on transitions $P$ by $\bigcirc P$: what was a property of a transition associated to the current state, becomes a property of the *transition part* of the next state.

Their implementation does not require the system to be topmost. Our own translation, on the other hand, seems more intuitive. Consider this simple system (on the left), our translation (in the middle), and theirs (on the right):




Our translation just adds new states in each arrow, without really changing the structure. Also, our approach tries to be more general and give room to further research.

The implementation in [2] works at the metalevel, as does the theoretical description in [15]. Basically, they emulate the original rewrite system with a single rule `rl` $t \to$ `next`$(t)$, with the function `next` doing all the work and the term $t$ including all the information about the original module, the current state, and some bookkeeping needed for the emulation. In contrast, we use the metalevel to produce a new object-level module, which we then model check. Obtaining modules at the object level is important for us because, as noted above, we foresee we will be using nested module operators in the future.

Our method's performance is better on the systems we have tested, even though the number of states is larger in our case; this is most probably due to the fact that we work at the object level, with simpler terms being rewritten. The following data comes from model checking the same communication-channel system cited above, this time with a formula containing a proposition on transitions (resp., a spatial action):

| | our method | | [2]'s method | |
|---|---|---|---|---|
| `maxFaults` | states | secs | states | secs |
| 2 | 3,886 | 0 | 2,406 | 2 |
| 3 | 35,459 | 1 | 20,281 | 25 |
| 4 | 77,804 | 2 | 42,937 | 57 |
| 5 | 332,256 | 12 | 172,510 | 278 |
| 6 | 568,066 | 30 | not run | |

It has to be noted that Bae and Meseguer took one more step and modified Maude's model checker at the C++ level to allow for model checking LTLR directly. Not surprisingly, the performance of this model checker is quite better that the other two mentioned here.

## 7 Future work

We want to point out three directions in which our proposals could be profitable. We intend to pursue some of them in the near future.

**Concurrent proof terms.** The persistence of a discrimination in the definition of egalitarian systems is a hint that we are midway to somewhere. Remember the discrimination: transitions have unique in and out arrows. Bipartite alternating automata—or any bipartite structures—can be seen as extensions of egalitarian structures where the discrimination has been dropped in a particular way.

Removing the *topmostness* requirement seems more interesting, thus allowing several rules to be *executing* simultaneously. For instance, the toy system we used above, with the rule `rl [a] : N => N + 1`, allows the derivation

$$\texttt{f(1) + g(1)} \quad \longrightarrow \quad \texttt{f(a(1)) + g(1)} \quad \longrightarrow \quad \texttt{f(a(1)) + g(a(1))}$$

This last term—a multi-step proof term—represents two concurrent executions of rule `a`. Suppose `f` and `g` represent two components of a software system, and the argument is their version number. Rule `a` is version updating for each component. It happens, however, that version 2 of each component is only compatible with version 2 of the other. A sequential, interleaved execution (like the one performed by Maude's engine) necessarily visits a state with incompatibility. This same engine will find the right way on a split system.

In a different spirit, Petri nets also represent a generalization of our egalitarian structures, allowing several arrows in and out of a transition.

**Synchronization and strategies.** In a paper to come, we study the possibility of synchronized execution of several Maude systems. In principle, the synchronization happens on states by agreement on their propositions, but on transitions only by identity of rule labels. This is often not enough, and having propositions on transitions opens interesting possibilities.

A use of such synchronized execution will be the implementation of strategic control. Strategy languages for rewrite systems usually include rule labels to

denote actions, but more general tests (or propositions) on states. The use of propositions on transitions, as already pointed, would allow decoupling the tasks of system specification and strategy design. If, for example, the system is refined, or modified in some way, the definition of the propositions can also be modified correspondingly, with no change in the name or meaning of the propositions, let alone in the formulas to be verified.

**Shrinking the size of systems.** Several methods for shrinking the size of systems, especially with model checking in mind, are in common use: invisible-transition collapse, partial order reduction, equational abstraction, folding abstractions, well-structured transition systems. Some of them focus only on states and others only on transitions. An egalitarian view could result in new insights.

## 8 Conclusion

We have tried to convince the reader that granting to transitions all the privileges enjoyed by states can help in specification and verification tasks. In particular, we advocate for the free use of atomic propositions on transitions, as well as on states. Mixing both kinds of propositions helps make specifications, both of systems and of their temporal properties, more powerful and intuitive. Indeed, it allows to definitely decouple specification tasks from verification ones: entities from the system specification are not used literally in formulas, because propositions provide an interface. For strategy languages, propositions on transitions make it possible to give general meaning to strategies, independent from the particular formalization of the system to be controlled.

Structures that allow general propositions on transitions are not common. Egalitarian structures are designed to play this role, and labeled transition systems and Kripke structures can be embedded in them. Rewriting logic is particularly well suited for an egalitarian view, that is, there are natural semantics from rewrite systems as egalitarian structures.

There are ways to apply existing tools and concepts to egalitarian structures. Faithful maps of tandems can be given from egalitarian structures and logics to better-known settings. This paper presents a prototype implementation in Maude, allowing the specification of egalitarian systems and their verification using the available LTL model checker.

**Acknowledgements.** It is comforting to realize how much a paper can improve with the help of capable referees. We are most grateful to ours.

# A Instructions on the use of our implementation

This appendix contains instructions on how to specify and verify egalitarian systems in Maude using our implementation of the `SPLIT` module operator. Remember that our implementation is available for download from http://maude.sip.ucm.es/syncprod. It consists of the following files:

- `split.maude`: Code for the meta-level function `split : Module -> Module`.
- `ltlr.maude`: Syntax for LTLR and translation to LTL.
- `mod-expr.maude`: Implementation of module operator `SPLIT`.
- `fm-extension.maude`: Extends Full Maude to make it understand splits.
- `loads.maude`: Loads all needed to work in Full Maude with split systems.

The user only needs to issue the command

```
load loads.maude
```

to the Maude interpreter to get it ready to work in an egalitarian way.

## A.1 The example

The rest of the appendix contains an example of how to specify and model check a very simple system. The first step is coding the system the same well-known way it is always done in Maude:

```
(mod EXAMPLE is
   pr NAT .
   sort State .
   op a : Nat -> State .
   var N : Nat .
   crl [one] : a(N) => a(s N) if N < 10 .
   rl  [two] : a(8) => a(5) .
endm)
```

As we are working in Full Maude, we enclose the whole module in parentheses.

Below, we need to use and extend the splitting of this system. It is produced by the module expression `SPLIT[EXAMPLE]`. We don't need to know how it looks like in the inside, but just make things clearer, the split system is this one (rather, it is equivalent and formally quite similar to this one):

```
(mod SPLIT[EXAMPLE] is
   pr NAT .
   sorts StateEXAMPLE TransEXAMPLE State .
   subsorts StateEXAMPLE TransEXAMPLE < State .
   op a : Nat -> StateEXAMPLE .
   op one : Nat -> TransEXAMPLE .
   op two : -> TransEXAMPLE .
   var N : Nat .
   crl [one1] : a(N) => one(N) if N < 10 .
   rl  [one2] : one(N) => a(s N) .
   rl  [two1] : a(8) => two .
   rl  [two2] : two => a(5) .
endm)
```

We are interested in finding out whether, for any computation starting at state `a(0)`, the only way to arrive to states `a(N)` with `N` $< 5$ is through the repeated use of rule `one`, with no rule `two` in between. This is clearly true, but we want to ask the model checker anyway. Using future-time operators, we can express this property by the LTLR formula

$$\Box(\texttt{running-two} \rightarrow \Box\neg\texttt{at-Nlt5}).$$

The formula mixes a proposition on states (`at-Nlt5`: the system is at a state `a(N)` with `N` $< 5$) with one on transitions (`running-two`: the system is running rule `two`). We propose next two different ways to accomplish our aim. We declare our preference for the second one.

### A.2 First way: LTLR

We give meaning to the propositions by extending `SPLIT[EXAMPLE]`. This is the code (remarks below):

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

   op init : -> StateEXAMPLE .
   eq init = a(0) .

   var N : Nat .
   op at-Nlt5 : -> SProp .
   eq a(N) |= at-Nlt5 = (N < 5) .
   eq S:StateEXAMPLE |= at-Nlt5 = false [owise] .

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

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

The temporal logic LTLR considers propositions on states and on transitions as different objects. So, we have defined `running-two` as a proposition on transitions (`TProp`) true only of `two`. And `at-Nlt5` is a proposition on states (`SProp`). The proposition `isState` is declared and used in file `ltlr.maude`, but must be defined in the module where it is used, as we have done. It must be defined to be true on proper states and false on states that were transitions in the original system. It is needed to translate formulas from LTLR to LTL, as explained below.

The formula we want to model check, written in our syntax for LTLR is

```
@[](running-two @-> @[] @~ at-Nlt5)
```

To avoid clashes with LTL's syntax, we have prefixed every operator with the symbol `@`. It is not a beautiful choice, but it is acceptable for a prototype. Then,

we have the function `LTLR` that translates from LTLR to LTL. This translation needs to use `isState`. For instance, the translation for operator `@U` is defined in this way:

```
eq LTLR(F @U G) = (isState -> LTLR(F)) U (isState /\ LTLR(G)) .
```

That is why we need to include the definition for `isState` in our module.

After loading the module `EXAMPLE-PROPS-LTLR` into Full Maude, the only thing left is running the model checker:

```
red modelCheck(init, LTLR(@[](running-two @-> @[] @~ at-Nlt5))) .
```

To which it answers on the affirmative.

### A.3 Second way: LTL

In many cases, propositions on states can also be given a natural meaning on transitions, and vice versa. For instance, `running-two` is trivially false when nothing is running, that is, on any state. In the same way, `at-Nlt5` can be considered to be true when running rule `one` with a value for `N` less than 5. Thus, we can extend `SPLIT[EXAMPLE]` in this way:

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

   op init : -> State .
   eq init = a(0) .

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

   var N : Nat .
   op at-Nlt5 : -> Prop .
   eq a(N) |= at-Nlt5 = (N < 5) .
   eq one(N) |= at-Nlt5 = (N < 5) .
   eq S:State |= at-Nlt5 = false [owise] .
endm)
```

Note that the sort `State` appearing here is the one from `SPLIT[EXAMPLE]`, that includes as subsorts `StateEXAMPLE` and `TransEXAMPLE`. Also, the sort `Prop` includes both propositions on transitions and on proper states. This allows us to write and use the LTL formula directly when calling the model checker. So, after loading `EXAMPLE-PROPS-LTL` into Full Maude, we issue the command:

```
red modelCheck(init, [](running-two -> [] ~ at-Nlt5)) .
```

The answer is again a boring but reassuring `true`.

## References


1. Bae, K., Meseguer, J.: The linear temporal logic of rewriting Maude model checker. In: Ölveczky, P.C. (ed.) Rewriting Logic and its Applications. 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6381, pp. 208–225. Springer (2010), http://dx.doi.org/10.1007/978-3-642-16310-4_14
2. Bae, K., Meseguer, J.: A rewriting-based model checker for the linear temporal logic of rewriting. Electronic Notes in Theoretical Computer Science 290, 19 – 36 (2012), http://www.sciencedirect.com/science/article/pii/S1571066112000795, ninth International Workshop on Rule-Based Programming (Rule 2008)
3. Boudol, G., Castellani, I.: A non-interleaving semantics for CCS based on proved transitions. Fundamenta Informaticae 11, 433–452 (1988)
4. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G. (eds.) IFM. Lecture Notes in Computer Science, vol. 2999, pp. 128–147. Springer (2004)
5. De Nicola, R., Vaandrager, F.: Three logics for branching bisimulation. J. ACM 42(2), 458–487 (Mar 1995), http://doi.acm.org/10.1145/201019.201032
6. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes. Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer (1990)
7. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002. Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187. Elsevier (2004), http://dx.doi.org/10.1016/S1571-0661(05)82534-4
8. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. J. ACM 33(1), 151–178 (Jan 1986), http://doi.acm.org/10.1145/4904.4999
9. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4533, pp. 153–168. Springer (2007), http://dx.doi.org/10.1007/978-3-540-73449-9_13
10. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
11. Kindler, E., Vesper, T.: Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98 Lisbon, Portugal, June 22–26, 1998 Proceedings, chap. ESTL: A Temporal Logic for Events and States, pp. 365–384. Springer Berlin Heidelberg, Berlin, Heidelberg (1998), http://dx.doi.org/10.1007/3-540-69108-1_20
12. Kozen, D.: Results on the propositional $\mu$-calculus. Theoretical Computer Science 27(3), 333 – 354 (1983), http://www.sciencedirect.com/science/article/pii/0304397582901256, special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982
13. Martín, Ó., Verdejo, A., Martí-Oliet, N.: Model checking TLR* guarantee formulas on infinite systems. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software, Lecture Notes in Computer Science, vol. 8373,

pp. 129–150. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54624-2_7
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992), http://dx.doi.org/10.1016/0304-3975(92)90182-F
15. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 5065, pp. 354–382. Springer (2008), http://dx.doi.org/10.1007/978-3-540-68679-8_22
16. Reisig, W.: Petri Nets: An Introduction, EATCS Monographs on Theoretical Computer Science, vol. 4. Springer (1985), http://dx.doi.org/10.1007/978-3-642-69968-9
17. Sánchez, C., Samborski-Forlese, J.: Efficient regular linear temporal logic using dualization and stratification. In: Proceedings of the 19th International Symposium on Temporal Representation and Reasoning (TIME 2012). pp. 13–20. IEEE Computer Society (2012)
18. Wolper, P.: Temporal logic can be more expressive. Information and Control 56(1-2), 72–99 (1983), http://www.sciencedirect.com/science/article/pii/S0019995883800515