Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 113–133

www.elsevier.com/locate/entcs

Proving VLRL Action Properties with the Maude Model Checker*

Miguel Palomino and Isabel Pita

Departamento de Sistemas Informáticos, Universidad Complutense de Madrid

Abstract

The Verification Logic for Rewriting Logic (VLRL) is a modal action logic in which rewrite rules are captured as actions. This paper studies a possible representation of the VLRL action formulae using the *Next* and the *Until* operators of Linear Temporal Logic (LTL). In particular, it studies the use of the Maude model checker to prove VLRL action formulae. Action modalities of VLRL fix the transition that will take place in a state and the context in which it will be applied, while LTL operators do not. Thus, to represent action modalities in LTL it is necessary to transform the initial rewrite theory into a new one in which the states carry the information about the transitions used and the context in which they have taken place. VLRL properties are then studied in the transformed theory by translating VLRL formulae into equivalent LTL formulae.

Keywords: Modal logic, liner temporal logic, Maude, rewriting logic, verification logic for rewriting logic, transformation of theories.

1 Introduction

The Verification Logic for Rewriting Logic (VLRL) [3,10,11] is a modal action logic in which rewrite rules are captured as actions. It supports the verification of properties of systems specified in rewriting logic [6,7]. In order to express the properties of the system, VLRL allows the definition of observations of the behaviour of the system. In this way, it is the user who determines in each case the facts of the system that he wants to emphasize.

A VLRL signature fixes a designated sort *State* that represents the objects of the system relative to which change is captured. Then, VLRL actions

* Research partially supported by the Spanish CICYT Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000.

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2004.06.026

represent those transitions that are "atomic" in the sense that, even if more than one rewrite takes place during such a transition, this happens because the structure of the state allows for such rewrites to be performed concurrently. The VLRL modal language provides four action modalities to express properties of a successor of a given state. Two of the modalities are *existential* in the sense that they require the action to denote an existing transition from the current state. The other two are *universal* and do not impose such requirement.

VLRL allows the verification of a particular sequence of rewrites, in contrast with a Linear Temporal Logic (LTL) where all computation paths are explored. Properties concerning a computation path are expressed in VLRL in a natural way through the use of the action modalities. Actually, VLRL can be used as a flexible interface in which to prove properties expressed in different branching time temporal logics [3]. And besides the action part of the logic, which is the one explored in this paper, VLRL also provides a spatial modality used to formulate properties of parts of the system, and the logic allows the combination of the action and the spatial modalities with great freedom.

This paper studies how action properties can be expressed using the *Next* and the *Until* operators of LTL in order to use the Maude model checker [2] to help proving the VLRL properties. But while the action modalities fix the transition that will take place in a state together with its context, the LTL operators do not. Thus, to represent the action modalities in LTL it is necessary to transform the initial rewrite theory into a new one in which the states carry information about the transitions that are applied and the context in which they take place; this is based on previous work by Meseguer about transformation of rewrite theories to express fairness and justice properties [8]. VLRL properties are then studied in the transformed theory by translating VLRL formulae into LTL ones that make use of that additional information to express the same property. For that we use a state predicate to express when an action α has been applied to obtain a certain state, $taken(\alpha)$, and another one to express when actions have been applied concurrently, *concurrent*. Satisfaction of both state predicates is defined over states of the transformed theory. We illustrate our approach with a specification of a simple protocol, but our construction applies to any system with a commutative and associative structure, and it can also be used with arbitrary systems by a simple extension of the ideas presented here.

Section 2 summarizes the main VLRL concepts used in the paper and Section 3 presents the way a rewrite theory is transformed into a new one with information about actions. Section 4 explains how to obtain atomic

propositions from observations; then the translation of a VLRL action formula to an LTL formula is explained. Section 5 shows the use of the Maude model checker for proving some VLRL formulae. Finally, the appendix contains the complete specification of the example presented in the paper.

This paper assumes familiarity with rewriting logic and Maude and its LTL model checker; detailed accounts about them can be found in [6,1,2]. Our notation follows standard practice: T_Σ denotes the initial algebra of terms over the signature Σ and $T_{\Sigma/E}$ is the initial algebra of equivalence classes of terms $[t]$ of the theory (Σ, E). We use $t[w/x]$ to denote the term obtained from replacing the variable x by w in t; sometimes an overbar is used to abbreviate sequences of expressions.

2 Overview of VLRL

VLRL [3,10,11] is a logic to *talk about change* in an indirect and global manner like other modal and temporal logics [5,4], in contrast with rewriting logic which is a logic *of change*. The idea is to make available attributes for making observations of the state of a system and action symbols to account for its elementary state changes.

2.1 Verification signature

Given a rewrite signature (Σ, E), a *verification signature* $(\Sigma^+, E^+, State, At, L)$ consists of:

- a distinguished sort *State* of Σ;
- additional sorts and operators that define an extension Σ^+ of Σ, together with a set E^+ of equations that axiomatize the extension in a way that protects the original signature, i.e., such that $T_{\Sigma^+/E^+}|_\Sigma \simeq T_{\Sigma/E}$;
- a family At of observation attributes, each of which has an associated sort s of Σ^+;
- a collection L of labels indexed over strings of sorts in Σ. The index corresponds to the sequence of sorts of the variables appearing in the rule that defines the action associated with the label.

2.2 A simple mutual exclusion example

In what follows, we illustrate our approach with a simple specification borrowed from [1]. It describes a system with two processes, `a` and `b`, that share a critical resource. Each process can be either waiting or in the critical section,

and they take turns accesing the critical section by passing to each other a different *token* (either `$` or `#`).

```
mod MUTEX is sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf .
  op __ : Conf Conf -> Conf [assoc comm id: none] .
  ops a b : -> Name .
  ops wait critical : -> Mode .
  op <_,_> : Name Mode -> Proc .
  ops # $ : -> Token .
  rl [a-enter] : $ < a, wait > => < a, critical > .
  rl [b-enter] : # < b, wait > => < b, critical > .
  rl [a-exit] : < a, critical > => < a, wait > # .
  rl [b-exit] : < b, critical > => < b, wait > $ .
endm
```

We can define two observation attributes of sort `Bool`, `crit-a` and `crit-b`, to check if a given process is in its critical section. The verification signature associated to the signature $(\Sigma_{\texttt{MUTEX}}, E_{\texttt{MUTEX}})$ of `MUTEX` that we are going to use in what follows is then

$$(\Sigma^{+}_{\texttt{MUTEX}}, E^{+}_{\texttt{MUTEX}}, \texttt{Conf}, \{\texttt{crit-a}, \texttt{crit-b}\}, L)\,,$$

with $\Sigma^{+}_{\texttt{MUTEX}}$ extending Σ with the sort `Bool`, $E^{+}_{\texttt{MUTEX}}$ extending E with the equations for the Boolean operators, and $L = \{\texttt{a-enter}, \texttt{b-enter}, \texttt{a-exit}, \texttt{b-exit}\}$.

2.3 Actions

We start by defining *pre-actions*, α. These correspond to the quotient of the set of proof terms obtained through the following rules of deduction:

- *Identities*:[1] for each $[t]$

$$\frac{}{[t] : [t] \to [t]}\,,$$

- *Replacement*: for each rewrite rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ and terms $w_1, \ldots, w_n$,

$$\frac{}{r(\overline{w}) : [t(\overline{w}/\overline{x})] \to [t'(\overline{w}/\overline{x})]}\,,$$

  and

- Σ-*structure*: for each $f \in \Sigma$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}\,,$$

modulo the following equations:

- *Identity transitions*: $f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$,

[1] Note that $[t]$ denotes a state as well as the corresponding identity transition for that state.

- *Axioms in E*: $t(\overline{\alpha}) = t'(\overline{\alpha})$, for each equation $t = t'$ in E.

Actions are the pre-actions that rewrite terms of sort *State*.[2]

Intuitively, actions (or more generally, pre-actions) correspond to "atomic" transitions where no sequential composition or nested applications of the replacement rule have taken place. It was proved in [6] that any transition in the initial model of a rewrite theory can be decomposed as an interleaving sequence of pre-actions.

One should be careful not to confuse this notion with that of *one-step rewrite*, that we will also use. The one-step rewriting relation, denoted with $\longrightarrow^1$ to distinguish it from the general rewrite relation $\longrightarrow$, holds between two terms u and v iff there is a one-step proof of $u \longrightarrow v$, that is, a proof of $u \longrightarrow v$ in which only one rewrite rule is applied to a single subterm. In other words, a one-step rewrite is a pre-action in which the replacement rule has been applied exactly once. This relation is used by the Maude model checker for the transition relation of the Kripke structure associated to a rewrite theory [2].

2.4 Action modalities

VLRL defines four action modalities, $[\alpha]$, $[[\alpha]]$, $\langle\alpha\rangle$, and $\langle\langle\alpha\rangle\rangle$ that capture the state transitions performed by the action α. The modal language[3] associated to a verification signature is given by

$$\varphi ::= \mathit{true} \mid t_1 = t_2 \mid \neg\varphi \mid \varphi \supset \varphi \mid [\alpha]\varphi \mid [[\alpha]]\varphi \mid \langle\alpha\rangle\varphi \mid \langle\langle\alpha\rangle\rangle\varphi$$

where t_1 and t_2 are terms over the signature Σ^+ extended with the attributes as constants of the corresponding sort.

Given an interpretation I for the observation attributes, mapping each attribute of sort s to a function from $T_{\Sigma/E,State}$ to $T_{\Sigma^+/E^+,s}$, the value $[\![t']\!]^{I,\sigma}[t]$ of term t' at state $[t]$ with respect to I and ground substitution σ is:

- $[\![x]\!]^{I,\sigma}[t] = \sigma(x)$,
- $[\![at]\!]^{I,\sigma}[t] = I(at)([t])$,
- $[\![f(t_1,\ldots,t_m)]\!]^{I,\sigma}[t] = f([\![t_1]\!]^{I,\sigma}[t],\ldots,[\![t_m]\!]^{I,\sigma}[t])$.

Satisfaction of action modality formulae at a given state $[t]$, observation interpretation I, and ground substitution σ is defined by structural induction:

- $[t], I, \sigma \models_{VLRL} t_1 = t_2$ iff $[\![t_1]\!]^{I,\sigma}[t] = [\![t_2]\!]^{I,\sigma}[t]$.

[2] The above definition of actions assumes that the rules in R are unconditional. The extension to conditional rules is straightforward.

[3] Actually, as remarked in the introduction, this is only the action part of the language: see [3] for the complete definition.

- $[t], I, \sigma \models_{VLRL} [\alpha]\varphi$ iff $[\![\alpha]\!]^\sigma : [t] \to [t']$ implies $[t'], I, \sigma \models_{VLRL} \varphi$.
- $[t], I, \sigma \models_{VLRL} [[\alpha]]\varphi$ iff $[t], I, \sigma \models_{VLRL} [\alpha]\varphi$ and $[t], I, \sigma \models_{VLRL} [\beta(\alpha)]\varphi$, for all actions $\beta(\alpha)$ that put α in context, that is, $\beta(\alpha)$ is constructed only with identities and Σ-structure rules on top of α.
- $[t], I, \sigma \models_{VLRL} \langle\alpha\rangle\varphi$ iff $[\![\alpha]\!]^\sigma : [t] \to [t']$ and $[t'], I, \sigma \models_{VLRL} \varphi$.
- $[t], I, \sigma \models_{VLRL} \langle\langle\alpha\rangle\rangle\varphi$ iff either $[t], I, \sigma \models_{VLRL} \langle\alpha\rangle\varphi$ or there is an action $\beta(\alpha)$ that puts α in context such that $[t], I, \sigma \models_{VLRL} \langle\beta(\alpha)\rangle\varphi$.

and in the expected way in the other cases. Note that $[\alpha]$ and $[[\alpha]]$ are universal modalities that require that the formula holds for all possible successors, as opposed to $\langle\alpha\rangle$ and $\langle\langle\alpha\rangle\rangle$ which are existential.

The actions α are subject to interpretation because they may contain variables. We denote by $[\![\alpha]\!]^\sigma$ the state transition that is given by the ground action obtained by applying the substitution σ to the action α. Then, for a given substitution actions are deterministic but partial, i.e., they do not apply to every state. In fact, once instantiated, an action is only applicable to a single state; this is due to the fact that an action carries within itself all the context information about where rewrite rules are applied.

The difference between the *single* action modalities $[\alpha]$ and $\langle\alpha\rangle$, and the *double* action modalities $[[\alpha]]$ and $\langle\langle\alpha\rangle\rangle$ is that the double modalities allow the formulation of properties about parts of the system in a wider context without complicating the notation by the need to make explicit the contextual identity transitions for those subterms that remain unchanged. That is, the action can happen *anywhere* in the term that represents the state.

For example, the intended interpretation I of the attributes for the `MUTEX` verification signature is such that I(`crit-a`) maps a term of sort `Conf` to `true` if it contains the subterm `< a, crit >` and to `false` otherwise, and similarly for I(`crit-b`). Under this interpretation, the formulae [[`a-enter`]](`crit-a` = `true`) and [[`a-enter b-enter`]](`crit-b` = `true`) hold in the state `$ < a, wait > < b, wait>`, but the formula $\langle\langle$`b-enter`$\rangle\rangle$(`crit-b` = `true`) does not.

3 Adding actions and contexts to a state

In general, for a rewrite theory $\mathcal{R}$ it may not be easy to specify *directly* a predicate of the form $taken(a(\overline{y}))$ holding in those states that arise from applying the action a to some other state, or a predicate *concurrent* stating that two actions can be done concurrently. What we can do is to associate to $\mathcal{R}$ a *semantically equivalent* theory $\mathcal{R}'$ with information about the actions that are applied in the system and the context in which the actions are triggered. We can then reason about the VLRL properties in this new theory and transfer

back to $\mathcal{R}$ the results obtained.

In this section we illustrate our general technique by defining a theory NEW-MUTEX that extends our rewrite theory MUTEX with some new sorts, operators, and rules, but also modifies the original rules in the way explained below. The complete specification can be found in the appendix.

3.1 The new states

States in NEW-MUTEX will carry information about the actions that have created them. They are represented as tuples

```
{ C | R | A | S }
```

where:

- C is a term that represents a state in the original system MUTEX.
- R can be concur or no-concur depending on whether the last action has been executed concurrently with the previous ones. For the initial state its value is indifferent: we arbitrarily choose concur.
- A is the last action that has been carried out.
- S is a *marked state* in which those subterms that have been changed by the last action are marked by enclosing them inside @_@.

To help grasp the underlying idea consider the transition

```
$ < a, wait > # < b, wait > ⟶ < a, critical > < b, critical >
```

in MUTEX. This transition can arise after the sequential application of action a-enter followed by b-enter, the application of b-enter followed by a-enter, or a single application of the action a-enter b-enter. In the first case (and similarly in the second), the corresponding rewrites in the transformed theory are as follows (the operators * and _+_ are explained in the next section).

```
{ $ < a, wait > # < b, wait > | concur | * |
  $ + < a, wait > + # + < b, wait > }
⟶¹
{ < a, critical > # < b, wait > | concur | a-enter |
  @ < a, critical > @ + # + < b, wait > }
⟶¹
{ < a, critical > < b, critical > | no-concur | b-enter |
  < a, critical > + @ < b, critical > @ }
```

Note that the action associated to the last state is only b-enter, and the marked subterm, < b, critical >, corresponds to the subterm changed by the last action. The second argument of the second state is trivially concur since it is the only action that has taken place so far.

On the other hand, the single action `a-enter b-enter` is simulated in the transformed theory by performing the last rewrite in a different manner.

```
{ $ < a, wait > # < b, wait > | concur | * |
  $ + < a, wait > + # + < b, wait > }
⟶¹
{ < a, critical > # < b, wait > | concur | a-enter |
  @ < a, critical > @ + # + < b, wait > }
⟶¹
{ < a, critical > < b, critical > | concur | a-enter b-enter |
  @ < a, critical > @ + @ < b, critical > @ }
```

The third component of the state reflects the fact that both actions have happened concurrently, as indicated by the presence of `concur` in the second argument. And since the action `a-enter b-enter` has changed the two subterms, both of them appear marked now.

It may seem surprising that the concurrent action `a-enter b-enter` actually corresponds to two steps in the transformed theory: this is enforced upon us by the fact that the Maude model-checker works by taking one-step rewrites. Since it is not possible to foresee in advance what concurrent steps will take place, the best we can do is just to translate each original rule into a single one in the transformed theory and to rely on their interleaving to allow for all possible concurrent actions. It is precisely the second argument, when its value is `concur`, which tells us that the interleaving must be interpreted as a concurrent action whose complete name appears as the third argument.

3.2 The new sorts

New sorts are introduced to deal with actions, marked states, and the new states.

```
sorts Action Action+ .
sort Concur .
sorts Conf+ Conf' .
sort NewConf .
subsort Conf < Conf+ .
subsort Conf < Conf' .
subsort Action < Action+ .
```

Two operators are declared to signal when actions happen concurrently.

```
ops concur no-concur : -> Concur .
```

To capture actions arising from the application of the replacement rule of deduction, the idea is to add for each rewrite rule $l : [t(\overline{x})] \to [t'(\overline{x})]$, where we assume a fixed order in $\overline{x} = (x_1 : s_1, \ldots, x_n : s_n)$, an operator

```
op l : s_1 ... s_n -> Action .
```

Since the rules in MUTEX only involve ground terms what we get is

```
ops a-enter b-enter a-exit b-exit : -> Action .
```

Actions obtained by the Σ-structure rule are represented by allowing the operators of the signature to apply to actions as well. In our case,

```
op none : -> Action .
op __ : Action Action -> Action [assoc comm id: none] .
```

We add to the actions a new value, *, different from all actions defined in the system. This value is used in the initial state to represent that no action has occurred yet.

```
op * : -> Action+ .
```

Marked states are represented by terms of sort Conf+; they are constructed from system states and the operators

```
op @_@ : Conf -> Conf+ .
op _+_ : Conf+ Conf+ -> Conf+ [assoc comm] .
```

The first operator is the "marker": its argument is a state in MUTEX. The second operator combines marked states to get another marked state. This last operator is declared with the same properties as the operator __ that constructs the states in MUTEX, except for the identity element which is not declared to facilitate the definition of auxiliary operations.

Conf' is used to determine the context in which a rewrite rule of the original system MUTEX will be applied. It is defined by the following operations:

```
op _._ : Conf+ Action -> Conf' .
op [_] : Conf -> Conf' .
op _+'_ : Conf' Conf' -> Conf' [assoc comm] .
```

with the equation

```
var S : Conf+ .   var A : Action .   var C : Conf .
eq (S . A) +' C = ((S + C) . A) .
```

The first operation adds to a state information about the action used to obtain it. These *extended* states are the result of applying a transformed rewrite rule of the original system (see, e.g., rule a-enter at the end of this section). The second operation is used to single out the subterm to which the rewrite rule will be applied. This operation is combined with the last one and the rule descend below to decompose a state into its components.

Finally, the sort NewConf is used to represent the states in the transformed theory.

```
op {_|_|_|_} : Conf Concur Action+ Conf+ -> NewConf .
```

3.3 The new rules

The original rules in MUTEX have to be transformed so that they only apply to the currently selected terms, as well as to point out which new terms are created. For that, the lefthand side is encapsulated within brackets and the righthand side consists of the resulting processes and tokens, marked, together with the name of the rule (that corresponds to the action executed).

```
rl [a-enter] : [ $ < a , wait > ] =>
               (@ < a , critical > @ . a-enter) .
rl [b-enter] : [ # < b , wait > ] =>
               (@ < b , critical > @ . b-enter) .
rl [a-exit] :  [ < a , critical > ] =>
               ((@ < a , wait > @ + @ # @) . a-exit) .
rl [b-exit] : [ < b , critical > ] =>
              ((@ < b , wait > @ + @ $ @) . b-exit) .
```

Then, NEW-MUTEX includes rules to transform a state { C | R | A | S } into a state { C' | R' | A' | S' } where:

- C' is the resulting state of applying a one-step rewrite B to the state C in the original system,
- R' indicates whether B has been executed concurrently with A,
- A' is the combined action of A and B if they have executed concurrently, or only B if not, and
- S' is the result of updating S by marking the subterm that has been changed by the action B.

The rewrite rules used to specify those steps are as follows:

```
vars S S' : Conf+ .          var C : Conf .
var R : Concur .             var A : Action .
                             var A+ : Action+ .

crl [step1] : { C | R | A+ | S } =>
              { ! S' | concur | (|(A, A+)) | (|(S', S)) }
              if [ C ] => (S' . A) /\ (S' & S) .

crl [step2] : { C | R | A+ | S } =>
              { ! S' | no-concur | A | S' }
              if [ C ] => (S' . A) .
```

The rule step1 can be used only if the last action can be executed concurrently with the previous ones. This is ensured by the S' & S condition, where the & operation

```
op _&_ : Conf+ Conf+ -> Bool .
```

checks if the context in which the action A is applied, represented by S', is disjoint from the context in which the last action was applied, represented by

the marked state S. The basic idea is to look for occurrences of marked states @ C @ in both S and S'.

Remark. If S and S' are marked states, S' & S returns true if S' has been obtained from S by an action that can be executed concurrently with the previous ones.

The new state is obtained with the help of the auxiliary operations

```
op !_ : Conf+ -> Conf .
op | : Conf+ Conf+ -> Conf+ .
op | : Action Action -> Action .
```

The first one eliminates the marks of a marked state. The second one updates the marks of the state given as its second argument with those of the marked state given as its first argument. And the third operation constructs an action out of two actions by adding the first to the second one. We illustrate their behavior with the following reductions, that arise in the last step of the second example in Section 3.1:

```
Maude> red !(< a, critical > + @ < b, critical > @) .
result Conf: < a,critical > < b,critical >

Maude> red |(< a, critical > + @ < b, critical > @,
             @ < a, critical > @ + # + < b, wait >) .
result Conf+: @ < a,critical > @ + @ < b,critical > @

Maude> red |(b-enter, a-enter) .
result Action: a-enter b-enter
```

The rule step2 simply executes a new action, in a non-concurrent manner with respect to the previous ones.

The first term of the condition of the step rules rewrites the state by mimicking the transitions in MUTEX. First we define a rule to obtain the subterm to which the rewrite rules of MUTEX will be applied.

```
crl [descend] : [ C1 C2 ] => C1 +' [ C2 ] if C1 =/= none and C2 =/= none .
```

Remark. The sequence of rewrites $t \stackrel{\alpha_1}{\to} t_1 \stackrel{\alpha_2}{\to} \cdots \stackrel{\alpha_n}{\to} t_n$ corresponding to the actions $\alpha_1,\ldots,\ \alpha_n$ is valid in MUTEX if and only if { t | BA | A | S } $\to$ { t1 | BA1 | alpha1 | S1 } $\to \cdots \to$ { tn | BAn | alphan | Sn } is a valid sequence of rewrites in NEW-MUTEX, where S is the term representing the decomposition of t into its tokens and processes.

4 Defining VLRL formulae in LTL

Our goal for this section is, given an interpretation I of the observations, to define a translation from action formulae to LTL such that a formula holds in a state in VLRL under the given interpretation for a ground substitution of variables if and only if the translated formula holds in the corresponding state of the transformed system. We start by studying how to express the atomic VLRL formulae as state predicates in LTL, and then consider how to translate arbitrary formulae.

4.1 Defining propositions from observations

The idea is to have a state predicate for each possible atomic VLRL formulae, that is, for each possible equation. For that we associate a supersort $s*$ to each sort s, represent each attribute of sort s as a constant of sort $s*$, and define an operator

```
op _=_ : s* s* -> Prop .
```

In the case of `MUTEX` we have

```
mod MUTEX-OBSERVED is
  protecting NEW-MUTEX .

  sort Bool*
  subsort Bool < Bool* .

  ops crit-a crit-b : -> Bool* .
  op _=_ : Bool* Bool* -> Prop .
```

The given interpretation I for the attributes is extended homomorphically to arbitrary terms as explained in Section 2.4 and we can assume, without loss of generality, that this extension can be defined equationally through a family of operators `interp`. The homomorphic extension of our distinguished interpration I^* for `MUTEX` is equationally defined as follows.

```
  op interp : Conf Bool* -> Bool .
  op critical : Conf Name -> Bool .

  var C : Conf .       vars B1 B2 : Bool* .
  vars N M : Name .    var R : Concur .
  var O : Token .      var A : Action .
  var S : Conf+ .

  eq critical(none, N) = false .
  eq critical(< N, critical > C, N) = true .
  eq critical(O C, N) = critical(C, N) .
  ceq critical(< M, critical > C, N) = critical(C, N) if N =/= M .
  eq critical(< M, wait > C, N) = critical(C, N) .
```

```
  eq interp(C, crit-a) = critical(C, a) .
  eq interp(C, crit-b) = critical(C, b) .
  eq interp(C, true) = true .
  eq interp(C, false) = false .
  eq interp(C, not B1) = not interp(C, B1) .
  eq interp(C, B1 or B2) = interp(C, B1) or interp(C, B2) .
  eq interp(C, B1 and B2) = interp(C, B1) and interp(C, B2) .
```

Then, the semantics of the atomic VLRL formulae is defined in LTL by means of

```
  ceq ({ C | R | A | S } |= B1 = B2) = true
      if interp(C, B1) = interp(C, B2) .
  ceq ({ C | R | A | S } |= B1 = B2) = false
      if interp(C, B1) =/= interp(C, B2) .
endm
```

With these definitions we clearly have, for all ground terms `C` of sort `Conf`, and `t` and `t'` ground terms of sort `Bool*`, that

$$\texttt{C}, I \models_{VLRL} \texttt{t} = \texttt{t'} \iff \texttt{\{ C | R | A | S \}} \models_{LTL} \texttt{t = t'}$$

The next step is to extend the translation from atomic to arbitrary VLRL formulae.

4.2 Defining action formulae in LTL

The most important difference between VLRL formulae and LTL formulae is that the former carry within themselves information about the transition that has to be applied. It was for this reason that we had to transform the original system `MUTEX`, so that states store information about the actions taken. In addition, now we define two predicates to recognize when such actions have occurred: `taken_top`(α), that will be true when the action α has taken place at the top of the state, and `taken`(α), that will be true when the action α is applied in some subterm of the term representing the state.

```
  op taken-top : Action -> Prop .
  op taken : Action -> Prop .

  var C : Conf .          var R : Concur .
  var A : Action .        var S : Conf+ .

  eq { C | R | A | S } |= taken(A) = true .
  ceq { C | R | A | S } |= taken-top(A) = true if top(S) .
```

We use an auxiliary operation that checks whether the transition has been applied on top of the state. For this it is enough to check if all processes and tokens of the state are marked.

```
  op top : Conf+ -> Bool .

  eq top(C) = false .
```

```
eq top(@ C @) = true .
eq top(C + S) = false .
eq top(@ C @ + S) = top(S) .
```

Since actions may happen concurrently with other actions we need a state predicate that expresses this fact. We define the `concurrent` predicate as follows:

```
op concurrent : -> Prop .
eq { C | R | A | S } |= concurrent = R == concur .
```

By looking at the rules `step1` and `step2` it is clear that `concurrent` satisfies the following:

Remark. A state of the form `{ C | R | A | S }` satisfies the atomic proposition `concurrent` if and only if the last action used to construct `A` has been executed concurrently with the previous ones.

Now we are ready to express *universal* action formulae in LTL, for which we will use the *Next* ($\bigcirc$) and the *Until* ($\mathcal{U}$) temporal operators. Intuitively, $[\alpha]\varphi$ holds in a given state if after the action α is done the resulting state fulfills φ. Then, for non-concurrent actions the resulting LTL formula is simply $\bigcirc(\texttt{taken\_top}(\alpha) \rightarrow \varphi)$. But arbitrary actions cannot be captured only with the *Next* operator because it only considers one-step rewrites; they can be expressed in LTL by allowing several actions to be executed concurrently before checking that the action of the formula has been taken. Thus, we translate $[\alpha]\varphi$ as

$$\bigcirc\,(\texttt{concurrent}\ \mathcal{U}\ (\texttt{taken\_top}(\alpha) \wedge \varphi)) \vee$$
$$\bigcirc\,((\texttt{concurrent} \wedge \neg\texttt{taken\_top}(\alpha))\ \mathcal{U}\ \neg\texttt{concurrent})\,.$$

The first part states that φ must hold if action α is taken, while the second one considers the case in which α does not take place.[4] The translation of $[[\alpha]]\varphi$ is similar, but replacing `taken_top` by `taken`:

$$\bigcirc\,(\texttt{concurrent}\ \mathcal{U}\ (\texttt{taken}(\alpha) \wedge \varphi))\ \vee$$
$$\bigcirc\,((\texttt{concurrent} \wedge \neg\texttt{taken}(\alpha))\ \mathcal{U}\ \neg\texttt{concurrent})$$

Existential action properties are translated to universal properties by duality. Negation and implication in VLRL correspond to negation and implication in LTL.

[4] This translation is correct if there are no deadlocks in the system (which is the case for `MUTEX`) because then it is not possible to apply actions concurrently in an indefinite manner and $\neg$`concurrent` must eventually be true. In [9] it was proved that any system in rewriting logic can be specified by a deadlock-free theory, so this is not a serious restriction.

5 Proving formulae with the model checker

To prove satisfaction of VLRL formulae in a state using the Maude model checker, we define the following `MUTEX-CHECK` module that imports the transformed theory with all the state predicates from a module `MUTEX-SAT` in which `taken` and `top` are specified (see the appendix), and where we declare the inital state for the model checker. The initial state consists of two processes `a` and `b` which are *waiting*, and a token `$`.

```
mod MUTEX-CHECK is
  protecting MUTEX-SAT .
  including MODEL-CHECKER .
  op init : -> NewConf .

  eq init = { < a , wait > < b , wait > $ | concur | * |
              < a , wait > + < b , wait > + $ } .
endm
```

We can prove satisfaction of the VLRL property

$$[[\texttt{a-enter}]](\texttt{crit-a} = \texttt{true} \vee \texttt{crit-b} = \texttt{true})\,,$$

stating that after performing the action `a-enter` there is a process in the critical section. The VLRL property is expressed in LTL as

$$\bigcirc\,(\texttt{concurrent}\ \mathcal{U}\ \texttt{taken(a-enter)}\ \wedge\ (\texttt{crit-a = true} \vee \texttt{crit-b = true})) \vee$$
$$\bigcirc\,(\texttt{concurrent} \wedge \neg\texttt{taken(a-enter)}\ \mathcal{U}\ \neg\texttt{concurrent})\,,$$

and is reduced by the model checker to:

```
reduce in CHECK-MUTEX : modelCheck(init, O (concurrent U taken(a-enter) /\
    (crit-a = true \/ crit-b = true)) \/ O (concurrent /\ ~ taken(a-enter)
    U ~ concurrent)) .
rewrites: 250 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

The VLRL property

$$\langle\langle \texttt{a-enter} \rangle\rangle \neg(\texttt{crit-a} = \texttt{true} \wedge \texttt{crit-b} = \texttt{true})$$

stating that it is possible to perform action `a-enter` and after performing it there are not two processes in the critical section is expressed in LTL, using duality, the formula

$$\neg\bigcirc\ (\texttt{concurrent}\ \mathcal{U}\ (\texttt{taken(a-enter)} \wedge (\texttt{crit-a = true} \wedge \texttt{crit-b = true}))\,,$$

which can be checked with

```
reduce in CHECK-MUTEX : modelCheck(init, ~ O (concurrent U (crit-a = true
    /\ crit-b = true) /\ taken(a-enter))) .
rewrites: 272 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Notice that it is easy to follow the sequence of rewrites of the first argument of the transformed state to obtain a counterexample for the original VLRL theory.

Finally, the property

$$[\texttt{a-enter b-enter}]\ \mathit{false}$$

stating that process `a` and process `b` cannot enter the critical section concurrently is proved with

```
reduce in CHECK-MUTEX : modelCheck(init, O (concurrent U taken-top(a-enter
    b-enter) /\ true = false) \/ O (concurrent /\ ~ taken-top(a-enter
    b-enter) U ~ concurrent)) .
rewrites: 249 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Note that in this case the action is done at the top of the state.

6 Conclusions

In this paper we have presented a technique to prove VLRL action properties using the Maude model checker; it can be viewed as a case study of the transformation of general theories proposed by Meseguer [8] for studying fairness in the framework of rewriting logic. First we transform the rewrite theory into a new one in which the states carry information about the transitions used and the context in which they have taken place. Then, VLRL properties are studied in the transformed theory by translating VLRL formulae into equivalent LTL formulae. This procedure has allowed us to check mechanically the validity of formulae in VLRL for the first time and it will permit us to explore in a deeper way the benefits of using the VLRL logic in the specification of systems. An alternative that we also intend to explore in the future is the development of a tool to deal directly with VLRL without the intermediate translation to LTL.

The main problem with the use of model checking for proving VLRL formulae is the size of the state space since the study of a single transition usually gives rise to many different computation paths. This fact is reflected in our transformed theory, for example, in the three possible rewrite sequences that correspond to the transition explained in Section 3.1.

The presented procedure is general, in the sense that it can be adapted to any system specified in rewriting logic. For a specification with a commutative and associative structure like that in the `MUTEX` example the changes should be minor, if any. For arbitrary theories the underlying idea is the same but the construction is more involved. In particular, we have been able to use a single sort `Action` (together with a supersort) to represent actions because the rules in the system only involve terms of a single sort, `Conf`. But if there were rules for more sorts then it would be necessary to have a pre-action sort for each of them, and to modify the extension of the operators in the signature to actions in a corresponding manner.

We are now working on the formalization of a general procedure for transforming arbitrary theories. In particular, we have to check if Meseguer's ideas for treating rules with rewrites in their conditions can be applied to our case. Besides, we are also working on translating spatial VLRL formulae to LTL formulae, where spatial formulae are an extension of the formulae considered here that allow to prove properties that happen in a concrete part of the state and, in particular, to define properties of actions that happen concurrently in different parts of it.

Acknowledgments

We would like to warmly thank José Meseguer for suggesting to us the transformation of theories as a way to prove VLRL properties using the Maude model checker, and Narciso Martí-Oliet for interesting discussions on the subject and useful comments to a previous draft. The first author would also like to thank the Computer Science Department of the University of Illinois at Urbana-Champaign for financial support during a visit in which this work was started.

References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual. http://maude.cs.uiuc.edu, June 2003.

[2] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[3] J. L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Chateau de Bonas, France, September 15–18, 1999, Selected Papers*, volume 1827 of *Lecture Notes in Computer Science*, pages 438–458. Springer-Verlag, 2000.

[4] R. Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes 7, Center for the Study of Language and Information, Stanford University, Second edition, 1992.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[6] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[7] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[8] J. Meseguer. Lecture notes for CS376, University of Illinois at Urbana-Champaign. http://www-courses.cs.uiuc.edu/~cs476/.

[9] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer-Verlag, 2003.

[10] I. Pita and N. Martí-Oliet. Proving modal and temporal properties of rewriting logic programs. In L. Moniz Pereira and P. Quaresma, editors, *Proc. APPIA-GULP-PRODE 2001, Joint Conference on Declarative Programming, Évora, Portugal*, Universidade de Évora, pages 277–295, September 2001.

[11] I. Pita. *Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura*. PhD thesis, Facultad de Matemáticas, Universidad Complutense de Madrid, March 2003.

A Maude code of the mutex example

```
mod NEW-MUTEX is
  sorts  Name Mode Proc Token Conf .
  sorts Action Action+ .
  sorts Concur .
  sorts Conf+ Conf' .
  sort NewConf .
  subsort Token Proc < Conf < Conf+ Conf' .
  subsort Action < Action+ .

  op * : -> Action+ .
  op none : -> Action .
  ops a-enter b-enter a-exit b-exit : -> Action .
  op __  : Action Action -> Action [assoc comm id: none] .

  ops concur no-concur : -> Concur .

  op none : -> Conf .
  op __ : Conf Conf -> Conf [assoc comm id: none] .

  ops a b : -> Name .
  ops wait critical : -> Mode .
  op <_,_> : Name Mode -> Proc .
  ops $ # : -> Token .

  op @_@ : Conf -> Conf+ .
  op _+_ : Conf+ Conf+ -> Conf+ [assoc comm] .

  op _._ : Conf+ Action -> Conf' .
  op [_] : Conf -> Conf' .
  op _+'_ : Conf' Conf' -> Conf' [assoc comm] .

  op !_ : Conf+ -> Conf .
  op | : Conf+ Conf+ -> Conf+ .
  op _&_ : Conf+ Conf+ -> Bool .
  op red : Conf -> Conf+ .

  op | : Action Action+ -> Action .

  op {_|_|_|_} : Conf Concur Action+ Conf+ -> NewConf .

  vars S S' : Conf+ .
  vars C C1 C2 C3 : Conf .
```

```
vars A A1 A2 : Action .
var A+ : Action+ .
var R : Concur .
var T : Token .
var P : Proc .

 eq (S . A) +'  C = ((S + red(C)) . A ) .

 eq ! C = C .
 eq ! @ C @ = C .
 eq ! (C + S) = C ! S .
 eq ! (@ C @ + S) = C ! S .

 eq red(none) = none .
 eq red(T) = T .
 eq red(P) = P .
 ceq red(C1 C2) = red(C1) + red(C2) if C1 =/= none  /\ C2 =/= none .

 op _not-marked-in_ : Conf Conf+ -> Bool .

 eq C1 not-marked-in C2 = true .
 eq C not-marked-in @ C @ = false .
 ceq C1 not-marked-in @ C2 @ = true if C1 =/= C2 .
 eq C1 not-marked-in (C2 + S) = C1 not-marked-in S .
 eq C not-marked-in @ C @ + S = false .
 ceq C1 not-marked-in (@ C2 @ + S) = C1 not-marked-in S if C1 =/= C2 .

 eq |(C, C) = C .
 ceq |(C, C + S) = C if C not-marked-in S .
 eq |(C , @ C @) = @ C @ .
 eq |(C, @ C @ + S) = @ C @ .

 eq |(@ C @, S) = @ C @ .

 ceq |(C + S, C + S') = C + |(S, S') if C not-marked-in S' .
 eq |(C + S, @ C @ + S') = @ C @ + |(S, S') .
 eq |(@ C @ + S, S') = @ C @ + |(S, S') .

 eq S & C = true .
 eq S & @ C @ = C not-marked-in S .

 ceq (C + S) & (C + S') =  S & S' if C not-marked-in S' .
 ceq S & (C + S') = S & S' if not (C not-marked-in S) .

 eq (C + S) & @ C @ + S' = S & S' .
 ceq S & @ C @ + S' = false if not (C not-marked-in S) .

 eq |(A, *) = A .
 eq |(A1, A2) = (A1 A2) .

crl [descend] : [ C1 C2 ] => C1 +' [ C2] if C1 =/= none and C2 =/= none .

rl [a-enter] : [ $ < a , wait > ] => (@ < a , critical > @ . a-enter) .
```

```
  rl [b-enter] : [ # < b , wait > ] => (@ < b , critical > @ . b-enter) .
  rl [a-exit] :  [ < a , critical > ] =>
                 ((@ < a , wait > @ + @ # @) . a-exit) .
  rl [b-exit] : [ < b , critical > ] =>
                ((@ < b , wait > @ + @ $ @) . b-exit) .
  crl [step1] : { C | R | A+ | S } =>
          {! S' | concur | (|(A, A+)) | (|(S', S)) }
          if [ C ] => (S' . A) /\ (S' & S) .
  crl [step2] : { C | R | A+ | S } =>
      {! S' | no-concur | A | S' }
      if [ C ] => (S' . A) .
endm

mod MUTEX-OBSERVED is
  including SATISFACTION .
  protecting NEW-MUTEX .

  sort Bool* .
  subsort Bool < Bool* .

  subsort NewConf < State .

  ops crit-a crit-b : -> Bool* .
  op critical? : Conf Name -> Bool .
  op interp : Conf Bool* -> Bool .
  op _=_ : Bool* Bool* -> Prop .

  var C : Conf .
  vars N M : Name .
  var O : Token .
  var S : Conf+ .
  vars B1 B2 : Bool* .
  var R : Concur .
  var A : Action .

  eq critical?(none, N) = false .
  eq critical?(< N, critical > C, N) = true .
  eq critical?(O C, N) = critical?(C, N) .
  ceq critical?(< M, critical > C, N) = critical?(C, N) if N =/= M .
  eq critical?(< M, wait > C, N) = critical?(C, N) .

  eq interp(C, crit-a) = critical?(C, a) .
  eq interp(C, crit-b) = critical?(C, b) .
  eq interp(C, not B1) = not interp(C, B1) .
  eq interp(C, B1 or B2) = interp(C, B1) or interp(C, B2) .
  eq interp(C, B1 and B2) = interp(C, B1) and interp(C, B2) .
  eq interp(C, true) = true .
  eq interp(C, false) = false .

  ceq ({C | R | A | S } |= B1 = B2) = true if interp(C, B1) = interp(C, B2) .
  ceq ({C | R | A | S } |= B1 = B2) = false if interp(C, B1) =/= interp(C, B2) .
endm
```

```
mod MUTEX-SAT is
  protecting MUTEX-OBSERVED .


  op taken : Action -> Prop [ctor] .
  op taken-top : Action -> Prop [ctor] .
  op concurrent : -> Prop .

  op top : Conf+ -> Bool .

  var C : Conf .
  vars A A' : Action .
  var S : Conf+ .
  var N : Name .
  var R : Concur .

  eq {C | R | A | S } |= taken(A') = A == A' .
  ceq {C | R | A | S } |= taken-top(A) = true if top(S) .
  eq {C | R | A | S } |= concurrent = R == concur .

  eq top(C) = false .
  eq top(@ C @) = true .
  eq top(C + S) = false .
  eq top(@ C @ + S) = top(S) .
endm

mod CHECK-MUTEX is
  protecting MUTEX-SAT .
  including MODEL-CHECKER .
  op init : -> NewConf .

  eq init = { < a , wait > < b , wait > $ | concur | * |
              < a , wait > + < b , wait > + $ } .
endm
```