Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 133–150
www.elsevier.com/locate/entcs

Proving Modal Properties of Rewrite Theories Using Maude's Metalevel [⋆]

Isabel Pita[1] , Miguel Palomino[2]

Departamento de Sistemas Informáticos
Universidad Complutense de Madrid, Spain

Abstract

Rewriting logic is a very expressive formalism for the specification of concurrent and distributed systems; more generally, it is a logic of change. In contrast, VLRL is a modal logic built on top of rewriting logic to reason precisely about that change. Here we present a technique to mechanically prove VLRL properties of rewrite theories using the reflective capability of rewriting logic through its Maude implementation.

Keywords: Rewriting logic, Maude, modal logic, VLRL, reflection, mechanical verification.

1 Introduction

Rewriting logic [9] provides a formal framework for modelling concurrent systems in terms of states and state transitions, which is efficiently implemented in the Maude system [5]. It is a logic *of* change in which deduction directly corresponds to the change. In contrast, the Verification Logic for Rewriting Logic (VLRL) [7] is an action modal logic to talk *about* change in a more indirect and global manner, like other modal and temporal logics. VLRL was developed to prove abstract properties of systems specified in rewriting logic.

⋆ Research supported by the Spanish Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000.
[1] Email: ipandreu@sip.ucm.es
[2] Email: miguelpt@sip.ucm.es

doi:10.1016/j.entcs.2005.01.042

In VLRL, rewrite rules are captured as actions, transitions are represented by action modalities, and the structure of the state is represented by spatial modalities. In this way, action modalities allow the definition of properties of states after a particular sequence of rewrites, and the spatial modality allows the definition of properties of components of the state.

This kind of formulae can be proved at the object level with the help of Maude's LTL model checker if we are able to define a predicate *taken(a)*, holding in those states that arise from applying the action a to some other state, and a predicate *concurrent* stating that two actions can be executed concurrently. But in general, for a rewrite theory $\mathcal{R}$ it may be very difficult (or even impossible) to specify such predicates. In [10] we solved this problem by showing how to obtain a *semantically equivalent* theory $\mathcal{R}'$ in which those predicates can be straightforwardly defined.

In this paper we develop an alternative way of proving that a VLRL formula holds in a given state, that relies heavily on the use of *reflection* [6]. Reflection allows a system to access its own metalevel and manipulate theories as ordinary objects, thus providing a powerful mechanism for controlling the rewriting process. Maude's metalevel provides convenient operations that permit us to control the execution of rewrite rules in the object system and we will use them to obtain the result from applying an action to a state as well as to look at the states' inner structure. This approach is simpler than the one explored before, as it requires neither a theory transformation nor translating the VLRL formulae to linear temporal logic and the use of Maude's model checker. For the sake of concreteness we illustrate our technique with a concrete example, whose complete code can be found at `maude.sip.ucm.es/vlrl/metalevelprover/`, but the method is general and can be easily adapted to any system.

2 Overview on rewriting logic

A distributed system is axiomatized in rewriting logic [9] by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E). The system's *transitions* are axiomatized by the *conditional rewrite rules* R which are of the form $l : t \longrightarrow t'$ **if** *cond*, with l a label, t and t' Σ-terms, possibly with variables, and *cond* a condition involving equations and rewrites. Under reasonable assumptions about E and R, rewrite theories are *executable*, and there are several rewriting logic language implementations, including ELAN [1], CafeOBJ [8], and Maude [4,5]. In particular, Maude offers support for multiple sorts, subsort relations, operator overloading, and, unique among the other implementations, reflection.

We illustrate rewriting logic specifications by borrowing a distributed banking system example from the Maude manual [5]. Such a system is a "soup" of objects, which represent bank accounts, and messages, that represent actions to perform over those accounts.

```
mod BANK-ACCOUNT is
  protecting INT .
  including CONFIGURATION .
  op Account : -> Cid .
  op bal :_ : Int -> Attribute .
  ops credit debit : Oid Nat -> Msg .
  op from_to_transfer_ : Oid Oid Nat -> Msg .
  vars A B : Oid .
  vars M N N' : Nat .

  rl [credit] : < A : Account | bal : N > credit(A,M) =>
                < A : Account | bal : N + M > .
  crl [debit] : < A : Account | bal : N > debit(A,M) =>
                < A : Account | bal : N - M > if N >= M .
  crl [transfer] : (from A to B transfer M) < A : Account | bal : N >
                   < B : Account | bal : N' > =>
           < A : Account | bal : N - M > < B : Account | bal : N' + M >
      if N >= M .
endm
```

Integers are imported in protected form in the second line, while the syntax for objects is imported from module CONFIGURATION. An object has the form < A : C | Atts >, where A is the object's name, C its class, and Atts the list of the object's attributes. In the module BANK-ACCOUNT only one class, Account, was declared with an attribute bal. We also have three messages for credit, debit, and transfer of some money, each with an associated rewrite rule (the last two are conditional) that axiomatize the behaviour of the system when a message is received. Note that objects and messages are combined with an operator __ that is declared in CONFIGURATION to be associative, commutative, and with identity none, and that returns a term of sort Configuration. Finally, we can declare a new module that extends the previous one with a new class Manager whose objects will be in charge of creating new accounts.

```
mod BANK-MANAGER is
  inc BANK-ACCOUNT .
  op Manager : -> Cid .
  op new-account : Oid Oid Nat -> Msg [ctor] .
  vars O C : Oid .
  var N : Nat .
  rl [new] : < O : Manager | none > new-account(O, C, N) =>
             < O : Manager | none > < C : Account | bal : N > .
endm
```

3 The VLRL logic

VLRL permits the definition of observations over a system after choosing a distinguished sort *State*. In this way, the user defines the means by which he wants to talk about the behaviour of the system; see [7] for a complete presentation.

For example, in the bank account system we can choose `Configuration` as the distinguished sort. Then, we can observe if there is an account in the system with the help of an `account?` observation, and the current balance of an account with an observation `balance?`.

```
op account? : Oid -> VLRLBool .
op balance? : Oid -> VLRLInt .
```

This defines, for each object's name, two observations. Their actual meaning is given by means of interpretations I that take a term of sort *State* and an observation of sort s and return a term of sort s. For the "standard" interpretation one would expect, for example,

$$I(\texttt{< A-002 : Account | bal: 300 >}, \texttt{balance?(A-002)}) = \texttt{300}\,,$$

where `A-002` is an account name.

Actions

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

- *Identities*:[3] for each $[t]$,

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

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

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

  and

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

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

modulo the following equations:

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

[3] We use $[t]$ to denote both the equivalence class that represents the state and the identity transition for that state.

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

Actions are the preactions that rewrite terms of sort *State*. [4]

Intuitively, actions (or more generally, preactions) correspond to transitions where no sequential composition or nested applications of the replacement rule have taken place. It was proved in [9] that any transition in the initial model of a rewrite theory can be decomposed as an interleaving sequence of preactions.

The modal language

The formulae in the modal language are given by

$$\varphi ::= \mathit{true} \mid t_1 = t_2 \mid \neg\varphi \mid \varphi \supset \varphi \mid \langle\alpha\rangle\varphi \mid \langle\langle\alpha\rangle\rangle\varphi \mid f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$$

where t_1 and t_2 are terms that may contain observations, α is an action, $f : s_1 \ldots s_m \longrightarrow \mathit{State} \in \Sigma$, $\overline{d}$ is a sequence of data terms corresponding to the arguments of f that are not of sort *State*, and the φ_i are in one-to-one correspondence with the arguments of f that are of sort *State*. We also have the dual action operators $[\_]$ and $[[\_]]$, and the dual spatial operator $f_{\overline{d}}[\varphi_1, \ldots, \varphi_n]$.

Satisfaction of VLRL formulae at a given state $[t]$, observation interpretation I, and ground substitution σ is defined by structural induction in the usual way. An equation $t_1 = t_2$ holds in $[t]$ if the interpretations of t_1 and t_2 are the same; $\langle\alpha\rangle\varphi$, resp. $\langle\langle\alpha\rangle\rangle\varphi$, is true in $[t]$ if there exists a state $[t']$ that satisfies φ that can be reached from $[t]$ by applying action α at the top of $[t]$, resp. anywhere in $[t]$; and $f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$ is satisfied if there is a term of the form $f_{\overline{w}}(t_1, \ldots, t_n)$ in the equivalence class $[t]$, where $\overline{w}$ is the value of $\overline{d}$ at state $[t]$ for the observation interpretation I and ground substitution σ, such that each $[t_i]$ satisfies φ_i. We refer to [7] for a formal definition.

4 VLRL in Maude

We now embark ourselves on specifying VLRL inside Maude, aiming at using it to automatically check whether a given VLRL formula holds in a given state.

Syntax

The first step in that direction consists in defining VLRL's syntax in a Maude module.

```
fmod VLRL-FORMULAE is
```

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

```
  sorts Action VLRLFormula .

  ops True False : -> VLRLFormula .
  op _->_ : VLRLFormula VLRLFormula -> VLRLFormula .
  op ~_ : VLRLFormula -> VLRLFormula .
  op <_>_ : Action VLRLFormula -> VLRLFormula .
  op <<_>>_ : Action VLRLFormula -> VLRLFormula .

  op [_]_ : Action VLRLFormula -> VLRLFormula .
  op [[_]]_ : Action VLRLFormula -> VLRLFormula .
  op _/\_ : VLRLFormula VLRLFormula -> VLRLFormula .
  op _\/_ : VLRLFormula VLRLFormula -> VLRLFormula .

  vars X Y : VLRLFormula .
  var A : Action .

  eq [ A ] X = ~ (< A > ~ X) .
  eq [[ A ]] X = ~ (<< A >> ~ X) .
  eq X \/ Y = ~ X -> Y .
  eq X /\ Y = ~ (X -> ~ Y) .
endfm
```

The module VLRL-FORMULAE above defines the syntax of the propositional and modal action formulae, which is system-independent. The concrete syntax for basic formulae $t_1 = t_2$, spatial formulae $f_{\overline{d}}\langle\varphi_1, \ldots, \varphi_n\rangle$, and actions depends on the particular system at hand, and is defined in a supermodule. For the banking system we declare, in a module VLRL-SAT that imports BANK-MANAGER and VLRL-FORMULAE, the operators

```
  op _=_ : VLRLInt VLRLInt -> VLRLFormula .
  op _=_ : VLRLBool VLRLBool -> VLRLFormula .
```

for atomic formulae (one for each observation sort).

Then we define the spatial formulae. For each $f_{\overline{d}} : \mathit{State} \ldots \mathit{State} \longrightarrow \mathit{State}$, we have to declare an operator

$$f_{\overline{d}}\langle\_\,.\,.\,.\,\_\rangle \;:\; \mathit{VLRLFormula} \ldots \mathit{VLRLFormula} \rightarrow \mathit{VLRLFormula}\,,$$

and its dual

$$f_{\overline{d}}[\_\,.\,.\,.\,\_] \;:\; \mathit{VLRLFormula} \ldots \mathit{VLRLFormula} \rightarrow \mathit{VLRLFormula}\,.$$

In our example, the only such operator is __ and we get:

```
  op <__> : VLRLFormula VLRLFormula -> VLRLFormula .
  op [__] : VLRLFormula VLRLFormula -> VLRLFormula .

  vars X Y : VLRLFormula .
  eq [ X Y ] = ~ < (~ X) (~ Y) > .
```

Finally, to capture preactions we declare a sort *PreAction-s* for each sort s in the original signature. Note that in the absence of conditional rules we only need to declare preaction sorts for those sorts used in the definition of state constructors. Then the preaction corresponding to the sort selected as state of the system is made a subsort of `Action`.

```
sort PreActionConfiguration .
subsort PreActionConfiguration < Action .
```

For each sort s, actions arising from the identity rule are represented with an operator $[\_] : s \longrightarrow$ *PreAction-s*. To capture actions resulting from the application of the replacement rule we add, for each rewrite rule $l : t(\overline{x}) \longrightarrow t'(\overline{x})$, where we assume a fixed order in $\overline{x} = (x_1 : s_1, \ldots, x_n : s_n)$, the operator $l : s_1 \ldots s_n \longrightarrow$ *RPreAction-s*. *RPreAction-s* is a subsort of *PreAction-s* used to identify precisely those preactions obtained through the replacement rule. Finally, actions obtained with the Σ-structure rule are represented by allowing the operators of the signature to apply to actions as well: for each operator $f : s_1 \ldots s_n \longrightarrow s$, we declare $f : \mathit{PreAction}\text{-}s_1 \ldots \mathit{PreAction}\text{-}s_n \longrightarrow \mathit{PreAction}\text{-}s$. In addition, in order to take into account the quotient described on page 4, all equations in the specification are duplicated so that now they also apply to preactions and the corresponding equations for the identity transitions are included. (Note that the new operator __ is declared with the same attributes as __ over `Configuration`.)

```
sort RPreActionConfiguration .
subsort RPreActionConfiguration < PreActionConfiguration .
op [_] : Configuration -> PreActionConfiguration .
op none : -> PreActionConfiguration .
op __ : PreActionConfiguration PreActionConfiguration ->
                         PreActionConfiguration [assoc comm id: none] .
```

The `RPreActionConfiguration` operators depend on the rewriting rules of the particular system we are observing; in the banking system:

```
ops credit debit : Oid Int Int -> RPreActionConfiguration .
op transfer : Oid Oid Int Int Int -> RPreActionConfiguration .
op new : Oid Oid Int -> RPreActionConfiguration .
```

Formulae satisfaction

To study satisfiability of VLRL formulae we first have to define a valid interpretation for the observations and then to extend it to arbitrary terms. For example, for the `balance?` observation and the “standard” interpretation we would declare:

```
op interp : Configuration VLRLInt -> Int .
op balance? : Oid -> VLRLInt .
op balance-aux : Configuration Oid -> Int .

var C : Configuration .          vars O O1 O2 : Oid .
vars N N1 N2 : Int .             var M : Msg .

eq balance-aux(none, O) = 0 .
```

```
  eq balance-aux(< O : Account | bal : N > C, O) = N .
  ceq balance-aux(< O1 : Account | bal : N > C, O2) = balance-aux(C, O2)
      if O1 =/= O2 .
  eq balance-aux(M C, O) = balance-aux(C, O) .

  eq interp(C,balance?(O)) = balance-aux(C,O) .
  eq interp(C,0) = 0 .
  eq interp(C,s(N)) = s(interp(C,N)) .
```

Let us now make two assumptions. The first one is that we have at our disposal two operations

```
  op nextState : State Action -> State .
  op nextStateinContext : State Action -> StateList .
```

that given a state and an action return, respectively, the unique (if any) succesor of the state that follows from applying that action *at the top* and the set of all succesors that can be obtained from applying the action *in context*. The second assumption is that for each $f_{\overline{d}} : State \ldots State \longrightarrow State$ we have an operation $decomposable? : State\ VLRLFormula \ldots VLRLFormula \longrightarrow Bool$ such that $decomposable?(S, F_1, \ldots, F_n)$ is true if S is of the form $f_{\overline{d}}(t_1, \ldots, t_n)$, with each t_i satisfying F_i for $i = 1 \ldots n$; for the bank system:

```
  op decomposable? : State VLRLFormula VLRLFormula -> Bool .
```

Now, satisfiability of arbitrary VLRL formulae can be straightforwardly defined by structural induction. The distinguished sort, `Configuration`, is declared as a subsort of `State`, to which an `error` constant is also added to signal when an action cannot be applied to a given state.

```
  subsort Configuration < State .
  op error : -> State .
  op _|=_ : State VLRLFormula -> Bool .

  var S : State .             vars I1 I2 : VLRLInt .
  vars B1 B2 : VLRLBool .     vars F1 F2 : VLRLFormula .
  var A : Action .

  ceq (S |= I1 = I2) = true if interp(S, I1) == interp(S, I2) .
  ceq (S |= I1 = N2) = false if interp(S, I1) =/= interp(S, I2) .
  ceq (S |= B1 = B2) = true if interp(S, B1) == interp(S, B2) .
  ceq (S |= B1 = B2) = false if interp(S, B1) =/= interp(S, B2) .
  eq (S |= True) = true .
  eq (S |= False) = false .
  eq (S |= (F1 -> F2)) = (S |= F2) or not (S |= F1) .
  eq (S |= (~ F1)) = not (S |= F1) .
--- Actions
  ceq (S |= < A > F1) = nextState(S,A) |= F1 if nextState(S,A) =/= error .
  eq (S |= < A > F1) = false [owise] .
  ceq (S |= << A >> F1) = nextStateinContext(S,A) |= F1
      if nextStateinContext(S,A) =/= nil .
  eq (S |= << A >> F1) = false [owise] .
```

```
--- Spatial
  ceq (S |= < F1 F2 >) = true if decomposable?(S, F1, F2) .
  eq (S |= < F1 F2 >) = false [owise] .
```

In Maude, an `owise` equation is tried only if no other applies; its semantics can be defined purely in terms of equations [5, Chapter 4].

It only remains to specify the three operations that we took for granted, and it is precisely here where the main innovation of this paper lies. Note that it is in their nature that they should use as *input* the module that specifies the banking system itself, for instance to select a particular rule to apply to the state in the case of `nextState`. To define them, then, we require the use of Maude's metalevel, that we explain in the next section.

5 The module META-LEVEL

Maude's metalevel is specified in a predefined module called `META-LEVEL`. This module contains operators to (meta)represent terms and modules respectively as terms of sort `Term` and `Module`. A variable `X` and a constant `C` of sort `s` are represented by the quoted identifiers `'X:s` and `'C.s`, respectively; the representation of a term `f(t1,...,tn)` is `'f[`$\overline{\texttt{t1}}$`,...,`$\overline{\texttt{tn}}$`]`, where $\overline{\texttt{t1}}$,...,$\overline{\texttt{tn}}$ are the representations of `t1` and `tn`. Similar conventions apply to modules. In addition, Maude's metalevel supplies the user with efficient descent operations to reduce metalevel computations to object-level ones. We presently give a brief summary of the ones we need; for a complete and more accurate description we refer the reader to [5, Chapter 10].

- The operation `metaApply` takes five arguments: the metarepresentation of a module M, a term's metarepresentation, the name of a rule in M, a substitution's metarepresentation, and a natural number n. It tries to match the term with the lefthand side of the rule using the substitution, discards the first n matches, and applies the rule at the top of term with the $n+1$ match. It returns a triple formed by the metarepresentation of the reduced term, its corresponding sort or kind, and the substitution used; we can obtain the first component of the triple with the operation `getTerm`.
- The operation `metaXapply` is analogous to `metaApply`, but the rule can be applied anywhere in the term.
- The operation `metaMatch(R, t, t', Cond, n)` tries to match at the top the terms `t` and `t'` in the module `R` in such a way that the resulting substitution satisfies the condition `Cond`. The last argument is used to enumerate possible matches. It returns a term of sort `Substitution` if it succeeds, and `noMatch` otherwise. There is a corresponding generalization called `metaXmatch`.

- `upModule` takes as a first argument the name of a module already in Maude's database, and returns its representation as a term of sort `Module`. There is also a second argument that we can safely assume it is always `true`.
- The function `upTerm` takes a term and returns its metarepresentation; the operation `downTerm` works as its inverse.

6 The functions `nextState`, `nextStateinContext`, and `decomposable?`

The functionality required by `decomposable?` is supplied by `metaMatch` for free. Therefore, the semantics of spatial formulae can be defined by:

```
ceq (S |= < F1 F2 >) = true
    if metaMatch(upModule('VLRL-SAT+,true),
                 '__['S1:State,'S2:State],
                 upTerm(S),
                 '_|=_['S1:State,upTerm(F1)] = 'true.Bool /\
                 '_|=_['S2:State,upTerm(F2)] = 'true.Bool,
                 0) =/=  noMatch .
ceq (S |= < F1 F2 >) = false [owise] .
```

We simply decompose the state into two substates, `S1` and `S2`, and require that each of them satisfies the corresponding property `F1` or `F2` by means of the metacondition

```
'_|=_['S1:State,upTerm(F1)] = 'true.Bool /\
'_|=_['S2:State,upTerm(F2)] = 'true.Bool
```

The specification of the functions `nextState` and `nextStateinContext` is however much more involved, although the idea is very simple: extract the labels from the actions and apply the corresponding rules with `metaApply` and `metaXapply`.

For that, we need two auxiliary operations over `RPreActionConfiguration`: `upAction`, to obtain the rule's label associated to the action, and `getSubs`, that constructs the substitution to be used when applying the rule.

```
op upAction : RPreActionConfiguration -> Qid .
op getSubs : RPreActionConfiguration -> Substitution .

vars O O1 O2 : Oid .
vars N M N1 N2 N3 : Int .

eq upAction(credit(O,N,M)) = 'credit .
eq upAction(debit(O,N,M)) = 'debit .
eq upAction(transfer(O1,O2,N1,N2,N3)) = 'transfer .
eq upAction(new(O1,O2,N)) = 'new .

eq getSubs(credit(O,N,M)) =
   'A:Oid <- upTerm(O) ; 'N:Int <- upTerm(N) ; 'M:Int <- upTerm(M) .
```

```
eq getSubs(debit(O,N,M)) =
   'A:Oid <- upTerm(O) ; 'N:Int <- upTerm(N) ; 'M:Int <- upTerm(M) .
eq getSubs(transfer(O1,O2,N1,N2,N3)) =
   'A:Oid <- upTerm(O1) ; 'B:Oid <- upTerm(O2) ; 'M:Int <- upTerm(N1) ;
   'N:Int <- upTerm(N2) ; 'N':Int <- upTerm(N3) .
eq getSubs(new(O1,O2,N)) =
   'O:Oid <- upTerm(O1) ; 'C:Oid <- upTerm(O2) ; 'N:Int <- upTerm(N) .
```

Then, an identity action can be executed at the top of a state only if it corresponds to the whole state. The state does not change if an identity action is applied.

```
eq nextState(S, [ S ]) = S .
eq nextState(S, [ S1 ]) = error [owise] .
```

The state that results from applying a replacement action α to a state $[t]$ is obtained with the metalevel operation metaApply and the help of the auxiliary operations just defined.

```
ceq nextState(S, RA) = downTerm(getTerm(R:ResultTriple?),error)
    if R:ResultTriple? := metaApply(upModule('VLRL-SAT+,true),upTerm(S),
                                    upAction(RA),getSubs(RA),0) /\
       R:ResultTriple? =/= failure .
eq nextState(S, RA) = error [owise] .
```

For actions $f(\alpha_1, \ldots, \alpha_n)$ obtained with the Σ-structure rule, we define an operation nextState-aux that tries all possible decompositions $f(t_1, \ldots, t_n)$ of the state by using the metalevel operation metaMatch, and chooses the one, if any, such that each action α_i can by applied to the state t_i. If there is no such a state decomposition the error state is generated. We first consider the operator __; the functions getFirstTerm and getSecondTerm are used to extract the two terms in the substitution returned by metaMatch.

```
ceq nextState(S, A1 A2) = nextState-aux(S, A1 A2, 0)
    if A1 =/= none /\ A2 =/= none .

op nextState-aux : State Action Nat -> State .

ceq nextState-aux(S, A1 A2, i) =
           nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
           nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
     if A1 =/= none /\ A2 =/= none /\
        R:Substitution? := metaMatch(upModule('VLRL-SAT+,true),
                                     '__['M1:State, 'M2:State],
                                     upTerm(S), nil, i) /\
        R:Substitution? =/= noMatch /\
        nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
                 =/= error /\
        nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
                 =/= error .

ceq nextState-aux(S, A1 A2, i) = error
```

```
    if A1 =/= none /\ A2 =/= none /\
       metaMatch(upModule('VLRL-SAT+,true),
                 '__['M1:State, 'M2:State],
                 upTerm(S),
                 nil,i) == noMatch .

ceq nextState-aux(S, A1 A2, i) = nextState-aux(S, A1 A2, s(i))
    if A1 =/= none /\ A2 =/= none /\
       R:Substitution? := metaMatch(upModule('VLRL-SAT+,true),
                                    '__['M1:State, 'M2:State],
                                    upTerm(S),
                                    nil,i) /\
       R:Substitution? =/= noMatch /\
       (nextState(downTerm(getFirstTerm(R:Substitution?),error),A1)
                  == error or
       nextState(downTerm(getSecondTerm(R:Substitution?),error),A2)
                  == error) .
```

And finally, we also have the constant action `none`:

```
eq nextState(none,none) = none  .
eq nextState(S,none) = error [owise] .
```

The idea behind the specification of `nextState` also applies to the operation `nextStateinContext`. Now, since the action can happen anywhere in the state and not only at the top, we use `metaXmatch` and `metaXapply` instead of `metaMatch` and `metaApply`. Becase the same action can rewrite different parts of the state, the result of applying `nextStateinContext` won't be a single state in general, but a set of states (actually, a list in our implementation). In order not to get bogged dow with too many technicals details, we defer the specification to the appendix.

7 Some examples of properties

We illustrate some properties of the bank system for an initial bank configuration `S` with three accounts, `A-001`, `A-002` and `A-003`, three debit messages, a transfer order from `A-003` to `A-002`, and a new account message that we define in a module `VLRL-SAT+`. We want to decide, for a given formula `F`, whether `F` holds in `S` or not.

```
ops A-001 A-002 A-003 A-004 manager : -> Oid .
op bankConf : -> Configuration .
eq bankConf = < manager : Manager | none >
              < A-001 : Account | bal : 300 > debit(A-001, 150)
              debit(A-001, 150) < A-002 : Account | bal : 250 >
              debit(A-002, 400) < A-003 : Account | bal : 1250 >
              (from A-003 to A-002 transfer 300)
              new-account(manager, A-004, 1000).
```

Let us prove that a debit operation does not affect the existence of an account and that the balance of the account decreases in the debit amount. Notice that the action is done in context, since it only refers to the A-001 account and we have more terms in the state.

```
reduce in VLRL-SAT+ :
      bankConf |= << debit(A-001, 150, 300) >> (account?(A-001) = true /\
                                                balance?(A-001) = 150) .
result Bool: true
```

We can concurrently execute a debit operation on the A-001 account and create the new account A-004,

```
reduce in VLRL-SAT+ :
    bankConf |= << debit(A-001,150,300) new(manager,A-004,1000) >> True .
result Bool: true
```

but we cannot concurrently create a new account and make an operation on it:

```
reduce in VLRL-SAT+ :
    bankConf |= << debit(A-004,150,300) new(manager,A-004,1000) >> True .
result Bool: false
```

We cannot debit an account with more money than its current balance

```
reduce in VLRL-SAT+ : bankConf |= << debit(A-002, 400, 250) >> True .
result Bool: false
```

unless we first transfer some money to it:

```
reduce in VLRL-SAT+ :
    bankConf |= << transfer(A-003, A-002, 300, 1250, 250) >>
                << debit(A-002, 400, 250) >> True .
rewrites: 71 in 120ms cpu (120ms real) (591 rewrites/second)
result Bool: true
```

But note that both actions cannot be executed concurrently:

```
reduce in VLRL-SAT+ :
    bankConf |= << debit(A-002, 400, 250)
                   transfer(A-003, A-002, 300, 1250, 250) >> True .
result Bool: false
```

We can also express concurrent actions by defining the part of the state in which they take place by means of the spatial operator. For example we can express a debit to the A-001 account and a transfer from account A-003 to the account A-002 by:

```
reduce in VLRL-SAT+ :
    bankConf |= < (< debit(A-001, 150, 300) > balance?(A-001) = 150)
                  (<< transfer(A-003, A-002, 300, 1250, 250) >>
                   balance?(A-002) = 550) > .
result Bool: true
```

The first action is done at the top while the second one is done in context. If we want to express all the actions at the top we can use the identity actions to fix the rest of the state. For example, in the following property we divide the state in three substates:

```
reduce in VLRL-SAT+ :
     bankConf |= < (< debit(A-001, 150, 300) > True)
                  < (< new(manager, A-004, 1000) > True)
                    (< [((((< A-003 : Account | bal : 1250 >
                             from A-003 to A-002 transfer 300)
                             debit(A-002, 400))
                             < A-002 : Account | bal : 250 >)
                             debit(A-001, 150)] > True) > > .
result Bool: true
```

8 Final remarks

The VLRL logic is a powerful logic for proving modal and temporal properties of systems specified in rewriting logic. It permits the specification of action and spatial properties that explore the structure of a system similar to the ones defined by the modal logic for mobile ambients of Cardelli and Gordon [3].

In this paper we have presented a method to check if a given VLRL formula holds in a given state that uses Maude's metalevel to decompose the state into its components and to execute the actions in a system. Our technique can be applied to arbitrary systems by slightly adapting to the particular operators at hand the description in this paper. Nevertheless, this description should already be widely applicable because distributed systems tend to be specified in rewriting logic with an associative and commutative operator as here; examples of a different kind can be found at `maude.sip.ucm.es/vlrl/metalevelprover/`.

Actually, and though not presented for the sake of generality, the definition of the operations `nextState` and `nextStateinContext` can be simplified for associative and commutative structure; in this case it is enough to distinguish the case in which the Σ-structure rule applies to an identity action from the case in which the rule applies to actions obtained by replacement. This definition is more efficient since it does not obtain all possible state decompositions, but it proceeds by matching some part of the state with some action and proving that the rest of the state fulfills the rest of the action.

Performance of the implementation could be improved by programming directly at the metalevel. That is, instead of continually changing levels by using the operations `upTerm` and `downTerm`, we could write equations that would directly deal with the metarepresentations of the terms representing the corresponding states. The drawback of this alternative is that it produces a less readable specification and, since efficiency was not our main concern, we have chosen to keep the specification as clear as possible.

Compared with the technique for proving VLRL formulae at the object level with the help of the LTL Maude model checker [10], this method does not require the transformation of the original theory but simply the definition of the `nextState` and `nextStateinContext` operations for each function of the original signature. From the computational point of view, the use of the metalevel reduces drastically the number of rewrites needed to prove the formulae because the rewrite steps needed to simulate the execution of actions in the transformed theory are avoided. Nevertheless, in the examples we have run the time it takes to prove a formula is almost the same in both cases, due to the fact that computation at the metalevel is less efficient than computation at the object level, and the high performance of the LTL Maude model checker. Also, the LTL model checker returns a counterexample when the property is not fulfilled, whereas the current implementation of the method explained in this paper does not; nevertheless, we plan to support it in a future version.

Acknowledgement

We thank Narciso Martí-Oliet for his helpful comments on a previous draft and Francisco Durán for his interesting remarks during the presentation of the paper.

References

[1] Borovanský, P., C. Kirchner, H. Kirchner and P.-E. Moreau, *ELAN from a rewriting logic point of view*, Theoretical Computer Science **285** (2002), pp. 155–185.

[2] Bruni, R. and J. Meseguer, *Generalized rewrite theories*, in: J. C. M. Baeten, J. K. Lenstra, J. Parrow and G. J. Woeginger, editors, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, Lecture Notes in Computer Science **2719** (2003), pp. 252–266.

[3] Cardelli, L. and A. D. Gordon, *Anytime, anywhere: Modal logics for mobile ambients*, in: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2000), pp. 365–377.

[4] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 187–243.

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *Maude manual (version 2.1)* (2004), `http://maude.cs.uiuc.edu/manual/`.

[6] Clavel, M. and J. Meseguer, *Reflection in conditional rewriting logic*, Theoretical Computer Science **285** (2002), pp. 245–288.

[7] Fiadeiro, J. L., 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, Château de Bonas, France, September 15–18, 1999, Selected Papers*, Lecture Notes in Computer Science **1827** (2000), pp. 438–458.

[8] Futatsugi, K. and R. Diaconescu, "CafeOBJ Report," World Scientific, AMAST Series, 1998.

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

[10] Palomino, M. and I. Pita, *Proving VLRL action properties with the Maude model checker*, in: N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA'04, Barcelona, Spain, March 27–28, 2004*, Electronic Notes in Theoretical Computer Science (2004).

A The function nextStateinContext

The successor states after applying an action *in context* are obtained with the operation nextStateinContext. In this case the result of the operation is a set (a list in the implementation) of states, since the context in which the action is applied is not fixed. Satisfaction is defined for lists of states as follows:

```
sort StateList .
subsort State < StateList .
op nil : -> StateList .
op _;_ : StateList StateList -> StateList [assoc id: nil].
op nextStateinContext : State Action -> StateList .
op _|=_ : StateList VLRLFormula -> Bool .

eq nil |= F1 = false .
ceq S ; L |= F1 = (S |= F1) or (L |= F1) if L =/= nil .
```

We define an operation that helps us in forming the set of successor states. Intuitively, || is used to compose a state with a set of states, such that the state is added to each set.

```
op _||_ : StateList StateList -> StateList .

eq S || nil = error .
eq S1 || S2 = S1 S2 .
eq S1 || (S2 ; nil) = S1 S2 .
ceq S1 || (S2 ; L) = (S1 S2) ; (S1 || L) if L =/= nil .
ceq (S1 ; L) || L' = (S1 || L') ; (L || L') if L =/= nil .
```

As we have done in Section 6 with the nextState operation, we define nextStateinContext by distinguishing cases for each possible kind of action.

When executing an identity action in context the obtained state does not change, but the action can only be executed if it is a substate of the state. We check it using the metaXmatch operation of the metalevel, with no condition, and no upper nor lower bounds on the subterm searching. It is enough to find a match.

```
ceq nextStateinContext(S, [ S1 ]) = S
    if R:MatchPair? := metaXmatch(upModule('VLRL-SAT+,true),
                                  '__[upTerm(S1), 'M:State],
                                  upTerm(S),nil,0,unbounded,0) /\
       R:MatchPair? =/= noMatch .

eq nextStateinContext(S, [ S1 ]) = nil [owise] .
```

For actions obtained with the replacement rule we use the auxiliary operation `nextStateinContext-aux` that gives us all possible successor states by means of the metalevel operation `metaXapply`. We find all possible successor states and combine them with the ; operation.

```
eq nextStateinContext(S, RA) = nextStateinContext-aux(S, RA, 0) .
ceq nextStateinContext-aux(S, RA, i) =
       downTerm(getTerm( R:Result4Tuple?), error) ;
       nextStateinContext-aux(S, RA, s(i))
    if R:Result4Tuple? := metaXapply(upModule('VLRL-SAT+, true),
                                     upTerm(S),
                                     upAction(RA),
                                     getSubs(RA), 0, unbounded,i) /\
     R:Result4Tuple? =/= failure .

eq nextStateinContext-aux(S, RA, i) = nil [owise] .
```

For actions obtained with the Σ-structure rule we use the same auxiliary operation as before to get all possible decompositions of the state. For each state decomposition we verify if the corresponding actions can be executed in the corresponding substates. If they can, we add the state to the list of successor states, otherwise we look for the next decomposition. When there are no more state decompositions, `nil` is returned. Notice that the actions executed in the substates are also executed in context, and then the lists of states returned by the execution of the actions in each substate are merged appropriately with the `||` operation.

```
ceq nextStateinContext(S,A1 A2) = nextStateinContext-aux(S,A1 A2,0)
    if A1 =/= none /\ A2 =/= none .

ceq nextStateinContext-aux(S, A1 A2, i) =
      nextStateinContext-aux(
       downTerm(getFirstTerm(getSubstitution(R:MatchPair?)),error),A1,0)
      ||
      nextStateinContext-aux(
       downTerm(getSecondTerm(getSubstitution(R:MatchPair?)),error),A2,0)
      ;
      nextStateinContext-aux(S, A1 A2, s(i))
 if A1 =/= none /\ A2 =/= none /\
    R:MatchPair? := metaXmatch(upModule('VLRL-SAT+,true),
                               '__['M1:State, 'M2:State],
                               upTerm(S),
                               nil,0,unbounded,i) /\
    R:MatchPair? =/= noMatch /\
    nextStateinContext-aux(
      downTerm(getFirstTerm(getSubstitution(R:MatchPair?)),error),A1,0)
     =/= nil /\
    nextStateinContext-aux(
      downTerm(getSecondTerm(getSubstitution(R:MatchPair?)),error),A2,0)
     =/= nil .

ceq nextStateinContext-aux(S, A1 A2, i) = nil
    if A1 =/= none /\ A2 =/= none /\
       metaXmatch(upModule('VLRL-SAT+,true),
                  '__['M1:State, 'M2:State],
                  upTerm(S),
                  nil,0,unbounded,i) == noMatch .

ceq nextStateinContext-aux(S, A1 A2, i) =
    nextStateinContext-aux(S, A1 A2, s(i))
    if A1 =/= none /\ A2 =/= none /\
       R:MatchPair? := metaXmatch(upModule('VLRL-SAT+,true),
                                  '__['M1:State, 'M2:State],
                                  upTerm(S),
                                  nil,0,unbounded,i) /\
       R:MatchPair? =/= noMatch /\
    (nextStateinContext-aux(
       downTerm(getFirstTerm(getSubstitution(R:MatchPair?)),error),A1,0)
      == nil or
    nextStateinContext-aux(
       downTerm(getSecondTerm(getSubstitution(R:MatchPair?)),error),A2,0)
      == nil) .
```

Finally, and unlike what happend for `nextState`, since `none` is the identity element for both configurations an actions, we have

```
  eq nextStateinContext(S, none) = S .
```