Model Checking TLR* Guarantee Formulas on Infinite Systems*

Óscar Martín, Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
{omartins,jalberto,narciso}@ucm.es

Abstract. We present the implementation of a model checker for systems with a potentially infinite number of reachable states. It has been developed in the rewriting-logic language Maude. The model checker is explicit-state, that is, not symbolic. In infinite systems, we cannot expect it to finish in every case: it provides a semi-decision algorithm to validate guarantee formulas (or, equivalently, to falsify safety ones). To avoid getting lost in infinite paths, search is always performed within bounded depth. The properties to be checked are written in the Temporal Logic of Rewriting, TLR*, a generalization of CTL* that uses atomic propositions both on states and on transitions, providing, in this way, a richer expressive power. As an intermediate step, a strategy language is used. Guarantee formulas are first translated into strategy expressions and, then, the system and the strategy *evolve* in parallel searching for computations that satisfy the strategy and, thus, the formula. An example on verifying cache coherence protocols is presented, showing the usefulness of the tool.

Keywords: Infinite-state system, rewriting logic, Maude, model checking, strategy, temporal logic, TLR*, guarantee formula, cache coherence.

1 Introduction

Rewriting logic is a language for the specification of concurrent systems [19]. It is also an executable logic, which makes it a very useful formalism. Maude is a language and development system that incorporates both equational logic and rewriting logic [11]. Parallelism and nondeterminism are natural features of rewriting logic and Maude.

We now describe a very simple system that we use to introduce some important concepts in this paper. There are a number of counting devices in the system. Each time some external event happens, one and only one device detects it and increases its own counter. In order to be able to share data, the devices are organized as a ring, so that each device knows to whom it must send its messages, that are then resent until they have visited the whole ring. As we do

* Research supported by MINECO Spanish project StrongSoft (TIN2012–39391–C04–04) and Comunidad de Madrid program PROMETIDOS (S2009/TIC-1465).

S. Iida, J. Meseguer, and K. Ogata (Eds.): Futatsugi Festschrift, LNCS 8373, pp. 129–150, 2014.

not care about the nature of the events being counted, in our model each device is able by itself to increase its counter by one and immediately send a message to its *next* device. Here is the complete Maude specification:

```
mod COUNTING is
  protecting NAT .
  sort Id .
  subsort Nat < Id .
  sort Device .
  op [_,_,_] : Id Id Nat -> Device [ctor] .
  sort Message .
  op _|>_ : Id Id -> Message [ctor] .
  sort State .
  subsorts Device Message < State .
  op nullState : -> State [ctor] .
  op __ : State State -> State [ctor comm assoc id: nullState] .
  vars I J N : Id .  var A : Nat .
  rl [change] : [I, N, A] => [I, N, s(A)] (I |> N) .
  crl [resend] : [I, N, A] (J |> I) => [I, N, s(A)] (J |> N) if I =/= J .
  rl [remove] : (I |> I) => nullState .
endm
```

Types are introduced in Maude by the keyword `sort`. A `Device` is built by enclosing between square brackets three natural numbers: the device's `Id`, the `Id` of the next device in the ring, and the counter. A `Message` is given by two `Id`s: the first argument is the sender's `Id`, and the second the addressee's `Id`. The last sort we need is the `State` of the system. Maude uses an order-sorted type system. Thus, we declare that any `Device` or `Message`, by itself, constitutes a `State`. We also provide an operator with empty syntax, `__`, that allows to juxtapose any number of `State`s to get a new one. Note that the `comm` and `assoc` attributes given to the operator allow commutative and associative matching. This way of defining states is a usual idiom in Maude. We have also declared a `nullState` to be used as identity element for states.

The three rewrite rules represent the different ways the system can evolve. Rule `change` represents the counting of an event and the sending of the associated message. Rule `resend` states that when device `I` sees a message addressed to itself, it updates its counter and resends the message to the next device. It is a conditional rule, because this should only happen for devices other than the original one. Rule `remove` just drops a message whose sender and addressee coincide—that is, the message has already visited the whole ring. Notice that the variable `I` is used twice: this rule should only be applied when both arguments of a message are equal.

A nice property of rewriting logic is that both states and transitions between states can be represented by terms. In the example, states are represented by terms of sort `State`, like `[0, 1, 5] [1, 0, 5]`. Transitions are represented by terms on a larger signature, so-called *proof terms* [19]. For instance, the transition

`[0, 1, 5] [1, 0, 5]` $\longrightarrow$ `[0, 1, 5] [1, 0, 6] (1 |> 0)`

is represented by this proof term:

```
{[0, 1, 5] [] | 'change : ('I \ 1) ; ('N \ 0) ; ('A \ 5)}
```

This is a triple of a context (a term with a hole symbol `[]` showing where the rewrite took place), the name of the rule that has been applied, and the substitution used. (The leading quotes are a syntactic requirement of Maude.)

Atomic propositions on states and on transitions can be declared, and their satisfaction relations be defined, based on the shape of the term representing them. Once defined, they can be used to formally express temporal properties by means of temporal-logic formulas. For the example system, a proposition `selfMsg` that asserts that some message has completed its trip around the ring, and another parametric proposition `rule` that asserts that the transition is executing the rule whose label is given in its argument can be defined like this:

```
var I : Id .   var S : State .   var Cn : Context$State .
var L : Qid .  var Sb : Subst .  var T : Trans .

op selfMsg : -> StateProp [ctor] .
eq (I |> I) S |= selfMsg = true .
eq S |= selfMsg = false [owise] .

op rule : Qid -> TransProp [ctor] .
eq {Cn | L : Sb} |= rule(L) = true .
eq T |= rule(L) = false [owise] .
```

Thus, a `State` satisfies `selfMsg` iff it matches the pattern `(I |> I) S`.

The Temporal Logic of Rewriting TLR* [20] has been designed to take profit of this strength of rewriting logic. The logic CTL* allows only propositions on states; TLR* extends CTL* by allowing also propositions on transitions. Some interesting properties of systems are only naturally expressible using both state and transition propositions. For instance, the TLR* formula $\mathbf{G}(\texttt{selfMsg} \rightarrow \texttt{rule('remove)})$ asserts that each time a message has completed its trip around the ring it must be immediately removed.

The model checker we have implemented accepts guarantee formulas of TLR*. Guarantee formulas assert that some property is going to hold in the future. For instance, $\mathbf{F}\,\texttt{selfMsg}$ asserts that, at some future time, the system will arrive to a state satisfying `selfMsg`. The model checker explores all possible evolutions of the given system in search for that future time in which the property holds. If the formula happens to be false for the given system, the algorithm may not terminate: thus, for infinite systems, it only provides a semi-decision algorithm (but a complete decision one for finite systems). Bounded-depth search is necessary to avoid getting lost into an infinite branch when, perhaps, the answer is on another. Our implementation provides a way to specify the maximum depth to be explored. Also, it provides a command to ask the system to explore some more levels based on the open branches left by a previous model-checking command. Note that any tool that can verify guarantee formulas can also be used to falsify safety ones through the duality trick of verifying their (guarantee) negations.

Internally, the model checker uses strategies. Strategies [22,8,17] applied to system specifications are a means of guiding their evolution and restricting their nondeterminism. The strategy "any^+ . `selfMsg`", for instance, accepts only

executions that, after some positive number of steps, land on a state that satisfies `selfMsg`. And the strategy "(`rule('change)` ; `rule('resend)`$^+$; `rule('remove)`)$^+$" guides the system in such a way that once a message is added to the system, it is processed by all the devices and removed before a new event can be counted.

We will describe below a strategy language and show how TLR* guarantee formulas can be translated into it. We internally implement the strategies and use this implementation to model check TLR* guarantee formulas.

In the rest of the paper we first review rewrite systems, proof terms, TLR* and its semantics, and the strategy language and its semantics, following [20], and then we show how all of them are used to implement the model checker. Then we present an example on verifying the MSI cache coherence protocol. We finish with some related work and conclusions. An extended version of this paper can be found in [18]. Also, the complete Maude specifications for the model checker as well as for some examples, including the MSI cache coherence protocol, are available for download at `http://maude.sip.ucm.es/ismc`.

2 Rewrite Systems

Formally, a rewrite system [19] is a triple $\mathcal{R} = (\Sigma, E, R)$, where Σ is an order-sorted signature, E a set of equations, and R a set of rewrite rules of the form $l : q \to q'$, with l a label, and q, q' terms of the same sort and such that all variables in q' appear also in q. Such a triple specifies a concurrent, nondeterministic system in which the states of the system are E-equivalence classes of ground terms $[t]_E$; that is, the initial algebra $T_{\Sigma/E}$ constitutes the state space. The dynamics of the system are given by the rewrite rules in R. As states are equivalence classes of terms, rewriting happens also at this level. Thus, a transition from state $[t]_E$ to state $[t']_E$, denoted by $[t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$, is possible in $\mathcal{R}$ iff there exist $u \in [t]_E$ and $u' \in [t']_E$ such that u can be rewritten to u' using some rule $l : q \to q'$ in R.

For arbitrary E and R, whether $[t]_E \longrightarrow^*_{\mathcal{R}} [t']_E$ holds is undecidable in general.

Definition 1 (computable rewrite system [20]). *A rewrite system $\mathcal{R} = (\Sigma, E \cup A, R)$ (where the set of equations has been split into two disjoint subsets) is* computable *if E, A and R are finite and the following conditions hold:*

1. *Equality modulo A is decidable, and there exists a matching algorithm modulo A, producing a finite number of A-matching substitutions or failing otherwise, that can implement rewriting in A-equivalence classes.*
2. *$(\Sigma, E \cup A)$ is* ground terminating *and* confluent *modulo A. That is: (i) there are no infinite sequences of reductions with E modulo A; and (ii) for each $[t]_A \in T_{\Sigma/A}$ there is a unique A-equivalence class $[\mathrm{can}_{E/A}(t)]_A \in T_{\Sigma/A}$, called the* E-canonical form of $[t]_A$ modulo A*, such that the last term, which cannot be further reduced with E modulo A, of any terminating sequence beginning at $[t]_A$ is necessarily $[\mathrm{can}_{E/A}(t)]_A$.*

3. *The rules R are* ground coherent *relative to the equations E modulo A. That is, if $[t]_A$ is rewritten to $[t']_A$ by a rule $l \in R$, then $[\mathrm{can}_{E/A}(t)]_A$ is also rewritten by l to some $[t'']_A$ such that $[\mathrm{can}_{E/A}(t')]_A = [\mathrm{can}_{E/A}(t'')]_A$.*

These three conditions imply that for each sort $S \in \Sigma$ the relation $\longrightarrow^1_{\mathcal{R},S}$ is computable: one can decide $[t]_{E\cup A} \longrightarrow^1_{\mathcal{R}} [t']_{E\cup A}$ by generating the finite set of all one-step R-rewrites modulo A of $\mathrm{can}_{E/A}(t)$ and testing if any of them has the same E-canonical form modulo A as $[\mathrm{can}_{E/A}(t')]_A$.

The three conditions are quite natural and are typically met in practical rewriting-logic specifications. In Maude, the set of equations A is given by *operator attributes* like `comm` and `assoc` used in the example of Section 1, for which Maude knows specific matching algorithms.

3 Proof Terms and Computations

In rewriting logic computation and proof are equivalent. Given a system $\mathcal{R} = (\Sigma, E\cup A, R)$, the state $[u]_{E\cup A}$ can be rewritten to $[u']_{E\cup A}$ if and only if the inference rules of rewriting logic [19,9] allow to prove $\mathcal{R} \vdash [u]_{E\cup A} \rightarrow^+ [u']_{E\cup A}$. Single rewritings are witnessed by so-called *one-step proof terms*. One-step proof terms can be characterized in an algebraic fashion. We define the signature $\mathrm{Trans}(\Sigma)$ (Trans for "transition"), on which one-step proof terms are built, extending Σ in this way:

- For each sort $S \in \Sigma$, we add a new sort $\mathrm{Trans}(S)$ to $\mathrm{Trans}(\Sigma)$, and state that $S < \mathrm{Trans}(S)$, that is, S is a subsort of $\mathrm{Trans}(S)$. Terms of sort $\mathrm{Trans}(S)$ represent one-step rewrites between terms of sort S.
- Given a rule $l : q \rightarrow q'$ in R, let S be the sort of q and q', and let the variables appearing in q, taken in their textual order of appearance, have sorts $S_1, \ldots, S_n$. Then, for each such rule $l \in R$, we add to $\mathrm{Trans}(\Sigma)$ a new function symbol $l : S_1 \times \cdots \times S_n \rightarrow \mathrm{Trans}(S)$.
- For each function symbol $f : S_1 \times \cdots \times S_n \rightarrow S$ in Σ and each $i = 1, \ldots, n$, we add to $\mathrm{Trans}(\Sigma)$ an overloaded function symbol (with the same attributes as the original f) $f : S_1 \times \cdots \times \mathrm{Trans}(S_i) \times \cdots \times S_n \rightarrow \mathrm{Trans}(S)$.

In this paper, we assume that the sort of the terms that represent states of the system is called `State`. In that case, $\mathrm{Trans}(\texttt{State})$ is the sort of one-step proof terms. We declare `Trans` as a convenient synonym for $\mathrm{Trans}(\texttt{State})$.

Thus, a proof term has the form $v[l(\bar{u})]_p$: a term v of sort `State` whose subterm at position p has been replaced by $l(\bar{u})$. In such a proof term, $v[\ ]_p$ shows the *context* in which the rewrite is taking place, l is the rule being executed, and the substitution $\bar{x} \mapsto \bar{u}$ is being used, where $\bar{x}$ is the tuple of all the variables in q in the textual order in which they appear.

Now consider the rewrite system $\mathrm{Trans}(\mathcal{R}) = (\mathrm{Trans}(\Sigma), E \cup A, R)$. There are no new equations and no new rules. Thus, if $\mathcal{R}$ is computable (as defined in Section 2), so is $\mathrm{Trans}(\mathcal{R})$. In particular, every term has a unique E-canonical form modulo A. For each sort S, let $(\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_S$ denote the set of all

A-equivalence classes of the form $[\mathrm{can}_{E/A}(t)]_A$, where t is a ground-term of sort S. Thus, $(\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{Trans}}$ describes the set of all transitions between `State`s in the system specified by $\mathcal{R}$ in their canonical form representation. And $(\mathrm{Can}_{\Sigma/E\cup A})_{\mathtt{State}} = (\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{State}}$ describes the set of all `State`s.

Definition 2. *[20] A* computation (s,t) *in* $\mathcal{R}$ *is a pair of functions*

$$s : \mathbb{N} \to (\mathrm{Can}_{\Sigma/E\cup A})_{\mathtt{State}} \qquad \textit{and} \qquad t : \mathbb{N} \to (\mathrm{Can}_{\mathrm{Trans}(\Sigma)/E\cup A})_{\mathtt{Trans}}$$

such that for all $n \in \mathbb{N}$ *we have* $s(n) \xrightarrow{t(n)} s(n+1)$*. Usually, we write* $s_i = s(i)$ *and* $t_i = t(i)$*, and we consider* s *and* t *as sequences. Thus, the computation* $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots$ *is represented as* $(s,t) = (s_0 s_1 s_2 \ldots, t_0 t_1 t_2 \ldots)$*.*

Note that we only consider infinite computations. This allows an easier definition of the semantics, and it is not at all a strong restriction. See [20] for details.

Computations are the semantic entities on which the truth of TLR* formulas is evaluated.

4 Temporal Logics and TLR* Guarantee Formulas

Temporal logics, in their different flavors, are a usual formalism to express the properties we expect a system to satisfy as it evolves in time. Usually, temporal logics fall in one of two classes according to the kind of atomic propositions they use: state-based or action-based. State-based logics, like LTL, CTL and CTL*, can only talk directly about states [10]. Action-based logics, like A-CTL* [12] and Hennessy-Milner logic [16], can only talk directly about actions (that is, transitions). Rewriting logic provides algebraic structure both to states and to actions and TLR* was designed to form a good tandem with it [20].

The formulas our model checker understands are guarantee formulas with a leading path quantifier, either universal or existential. They constitute a sublogic of TLR*. We define its syntax now, with σ denoting an atomic state proposition and τ an atomic transition proposition:

$$\phi ::= \top \mid \bot \mid \sigma \mid \neg\sigma \mid \tau \mid \neg\tau \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\,\phi \mid \phi_1\,\mathbf{U}\,\phi_2 \mid \mathbf{F}\,\phi$$
$$\varphi ::= \mathbf{A}\,\phi \mid \mathbf{E}\,\phi$$

The definition of the semantics, following [20], is given below. Remember that s_0 and t_0 always denote the first state and transition in the sequences s and t, respectively. Also, for a computation (s,t), its suffix resulting by removing the first k elements from s and from t is denoted by $(s,t)^k$.

- $\mathcal{R}, (s,t) \models \top$;
- $\mathcal{R}, (s,t) \not\models \bot$;
- $\mathcal{R}, (s,t) \models \sigma \Leftrightarrow \mathcal{R}, s_0 \models \sigma$;
- $\mathcal{R}, (s,t) \models \tau \Leftrightarrow \mathcal{R}, t_0 \models \tau$;

- $\mathcal{R}, (s,t) \models \neg\phi \Leftrightarrow \mathcal{R}, (s,t) \not\models \phi$;
- $\mathcal{R}, (s,t) \models \phi_1 \vee \phi_2 \Leftrightarrow \mathcal{R}, (s,t) \models \phi_1$ or $\mathcal{R}, (s,t) \models \phi_2$;
- $\mathcal{R}, (s,t) \models \mathbf{X}\,\phi \Leftrightarrow \mathcal{R}, (s,t)^1 \models \phi$;
- $\mathcal{R}, (s,t) \models \phi_1\,\mathbf{U}\,\phi_2 \Leftrightarrow \exists k$ s.t. $\mathcal{R}, (s,t)^k \models \phi_2$ and $\forall i \in [0,k), \mathcal{R}, (s,t)^i \models \phi_1$;
- $\mathcal{R}, (s,t) \models \mathbf{F}\,\phi \Leftrightarrow \exists k$ s.t. $\mathcal{R}, (s,t)^k \models \phi$;
- $\mathcal{R}, s_0 \models \mathbf{A}\,\phi \Leftrightarrow$ for all computations (s,t) we have $\mathcal{R}, (s,t) \models \phi$;
- $\mathcal{R}, s_0 \models \mathbf{E}\,\phi \Leftrightarrow$ for some computation (s,t) we have $\mathcal{R}, (s,t) \models \phi$.

Thus, an existentially quantified formula, $\mathbf{E}\,\phi$, represents a reachability predicate, while a universally quantified one, $\mathbf{A}\,\phi$, has the semantics of a linear temporal formula.

5 A Strategy Language

There exists a rich strategy language for Maude [17]. Now we describe another such strategy language, proposed in [20], designed with the construction of the model checker in sight. As above, we denote by σ and τ generic atomic propositions on states and transitions. The syntax contains three syntactic categories: *Test* (tests on states), *Strat* (strategy expressions), and *StratForm* (strategy formulas). Here, e, e_1, e_2 are strategy expressions, and b, b_1, b_2 are tests.

- *Test*: $b ::= \top \mid \bot \mid \sigma \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$
- *Strat*: $e ::=$ idle $\mid \tau \mid \neg\tau \mid$ any $\mid e_1 \wedge e_2 \mid (e_1 \,|\, e_2) \mid e_1\,;\, e_2 \mid e^+ \mid e_1\,\mathcal{U}\,e_2 \mid e\,.\,b$
- *StratForm*: $f ::= \mathbf{A}\,e \mid \mathbf{E}\,e$

Formal semantics are given below. A few informal explanations follow on the operators that may not be trivial:

- The strategy idle does nothing, is always satisfied, and leaves the system in the same state it was.
- The expression $e_1\,;\,e_2$ means sequential composition, that is, the system is first guided by e_1 and then, when e_1 has finished its job, by e_2.
- The strategy $e_1\,\mathcal{U}\,e_2$ is an *until* operator: e_1 holds for subcomputations beginning at the first state, at the second, and so on, until a subcomputation beginning at state $n \geq 0$, and then e_2 holds from state $n+1$.
- The strategy $e\,.\,b$ combines e with a test b. It holds iff e holds and the test b succeeds for the last state reached.

Our model checker internally works with strategies, but we want the user to introduce TLR* guarantee formulas. So a semantically appropriate translation from the former to the latter is needed. It is given by the function [20]

$$\beta : \text{TLR* guarantee formulas} \to \textit{Strat}$$

defined by

$\beta(\phi) = \text{idle} \,.\, \phi$ for $\phi = \top, \bot, \sigma, \neg\sigma$

$\beta(\phi) = \phi$ for $\phi = \tau, \neg\tau$

$\beta(\phi_1 \vee \phi_2) = \beta(\phi_1) \,|\, \beta(\phi_2)$

$\beta(\phi_1 \wedge \phi_2) = \beta(\phi_1) \wedge \beta(\phi_2)$

$\beta(\mathbf{X}\,\phi) = \text{any} \,;\, \beta(\phi)$

$\beta(\phi_1 \,\mathbf{U}\, \phi_2) = \beta(\phi_1)\,\mathcal{U}\,\beta(\phi_2)$

$\beta(\mathbf{F}\,\phi) = (\text{idle} \,|\, \text{any}^+) \,;\, \beta(\phi)$

In the next section we define a semantics for strategy expressions with the aim of making ϕ and $\beta(\phi)$ semantically equivalent for each formula ϕ.

6 Strategy Semantics

Our semantics is different, but equivalent, to the one defined in [20] (see also [18]). Two nice features of our semantics are:

1. That it is bounded by definition. We are defining the relation $\mathcal{R}, (s,t) \models_k e$ with the intuitive meaning that the computation (s,t) needs to perform at most k steps before satisfying e. (This is in the same spirit as the bounded semantics defined in [7].)
2. That it allows a step-by-step implementation. That is, we are able to check whether $\mathcal{R}, (s,t) \models_k e$ by, first, checking whether (s_0, t_0) is a step *compatible* with e, and, second, checking whether the rest of the computation $(s,t)^1$ satisfies *the rest* of e in at most $k-1$ steps.

The semantics uses two auxiliary functions with these intuitive meanings:

$\text{fail}(e) =$ does e always fails, for any computation and with no need to take any step? *A posteriori*, $\text{fail}(e)$ means that e is semantically equivalent to $\text{idle} \,.\, \bot$.

$\text{tick}(e) =$ is e always satisfied, for any computation and with no need to take any step? *A posteriori*, $\text{tick}(e)$ means that e is semantically equivalent either to idle or to $\text{idle} \,|\, e'$ for some e'.

Definition 3 (fail **and** tick**).** *The functions* $\text{fail}, \text{tick} : \mathit{Strat} \to \mathit{Bool}$ *are defined as shown in this table:*

e	$\text{fail}(e)$	$\text{tick}(e)$
idle	*false*	*true*
τ	*false*	*false*
$\neg\tau$	*false*	*false*
any	*false*	*false*
$e_1 \wedge e_2$	$\text{fail}(e_1) \vee \text{fail}(e_2)$	$\text{tick}(e_1) \wedge \text{tick}(e_2)$
$e_1 \,\|\, e_2$	$\text{fail}(e_1) \wedge \text{fail}(e_2)$	$\text{tick}(e_1) \vee \text{tick}(e_2)$
$e_1 \,;\, e_2$	$\text{fail}(e_1) \vee \text{fail}(e_2)$	$\text{tick}(e_1) \wedge \text{tick}(e_2)$
${e_1}^+$	$\text{fail}(e_1)$	$\text{tick}(e_1)$
$e_1 \,\mathcal{U}\, e_2$	$\text{fail}(e_1) \wedge \text{fail}(e_2)$	$\text{tick}(e_2)$
$e_1 \,.\, b$	$\text{fail}(e_1) \vee b \equiv \bot$	$\text{tick}(e_1) \wedge b \equiv \top$

The most important function in the definition of the semantics is $\text{rest}_{s_0,t_0}(e)$. It answers the question: what strategy, derived from e, remains to be satisfied after step (s_0, t_0)? For instance, $\text{rest}_{s_0,t_0}(\text{idle} \,|\, (\tau \,;\, \text{any})) = \text{any}$ if t_0 satisfies τ. We define this function below, based on the structure of the formula.

In some cases, the result of $\text{rest}_{s_0,t_0}(e)$ is a disjunction, showing the different ways in which a step can be taken. For instance, $e_1 \,;\, e_2$ can take a step in two nonexclusive ways: (i) e_1 takes a step to become $\text{rest}_{s_0,t_0}(e_1)$, with e_2 still pending behind, or (ii) e_1 is already satisfied and then e_2 takes a step to become $\text{rest}_{s_0,t_0}(e_2)$. In cases like this, we use a convenient shorthand notation, showing each possible first step on a different line.

Definition 4. *The notation*

$$\begin{array}{l} e_1 \\ |\, e_2 \ \textit{if}\ B \end{array}$$

where e_1 and e_2 are strategies and B is a Boolean expression, is equal to just (e_1) if B evaluates to false, and is equal to $((e_1)\,|\,(e_2))$ if B evaluates to true.

Definition 5 (rest_{s_0,t_0})**.** *Given s_0 and t_0 (a state and a transition from it), the function $\text{rest}_{s_0,t_0} : \textit{Strat} \to \textit{Strat}$ is defined by:*

e	$\text{rest}_{s_0,t_0}(e)$
idle	$\text{idle} \,.\, \bot$
τ	$\textit{if}\ \mathcal{R}, t_0 \models \tau\ \textit{then}\ \text{idle}\ \textit{else}\ \text{idle} \,.\, \bot$
$\neg\tau$	$\textit{if}\ \mathcal{R}, t_0 \models \tau\ \textit{then}\ \text{idle} \,.\, \bot\ \textit{else}\ \text{idle}$
any	idle
$e_1 \wedge e_2$	$\text{rest}_{s_0,t_0}(e_1) \wedge \text{rest}_{s_0,t_0}(e_2)$ $\,\vert\, \text{rest}_{s_0,t_0}(e_1)\ \textit{if}\ \text{tick}(e_2)$ $\,\vert\, \text{rest}_{s_0,t_0}(e_2)\ \textit{if}\ \text{tick}(e_1)$
$e_1 \,\vert\, e_2$	$\textit{if}\ \text{fail}(e_1)\ \textit{then}\ \text{rest}_{s_0,t_0}(e_2)$ $\textit{if}\ \text{fail}(e_2)\ \textit{then}\ \text{rest}_{s_0,t_0}(e_1)$ $\textit{otherwise}\ \text{rest}_{s_0,t_0}(e_1) \,\vert\, \text{rest}_{s_0,t_0}(e_2)$
$e_1 \,;\, e_2$	$\text{rest}_{s_0,t_0}(e_1) \,;\, e_2$ $\,\vert\, \text{rest}_{s_0,t_0}(e_2)\ \textit{if}\ \text{tick}(e_1)$
${e_1}^+$	$\text{rest}_{s_0,t_0}(e_1) \,;\, (\text{idle} \,\vert\, {e_1}^+)$ $\,\vert\, \text{idle}\ \textit{if}\ \text{tick}(e_1)$
$e_1 \,\mathcal{U}\, e_2$	$\text{rest}_{s_0,t_0}(e_2)$ $\,\vert\, \text{idle}\ \textit{if}\ \text{tick}(e_2)$ $\,\vert\, e_1 \,\mathcal{U}\, e_2\ \textit{if}\ \text{tick}(e_1)$ $\,\vert\, \text{rest}_{s_0,t_0}(e_1) \wedge (e_1 \,\mathcal{U}\, e_2)\ \textit{if}\ \neg\,\text{fail}(e_1)$
$e_1 \,.\, b$	$\text{rest}_{s_0,t_0}(e_1) \,.\, b$ $\,\vert\, \text{idle}\ \textit{if}\ \text{tick}(e_1) \wedge \mathcal{R}, s_0 \models b$

Note that $\text{rest}_{s_0,t_0}(e)$ is never called with an e such that $\text{fail}(e)$, and thus we avoid some cases in the definition.

The last case in this definition uses the semantics for *Test*, that follows the usual definition for Boolean expressions, as shown next.

Definition 6 (*Test* semantics). *Given a rewrite system $\mathcal{R}$, a state s_0, and a Test on states b:*

$\mathcal{R}, s_0 \models \top$
$\mathcal{R}, s_0 \not\models \bot$
$\mathcal{R}, s_0 \models \sigma$ *according to the definition of* σ*, i.e., iff* $E \cup A \vdash (s_0 \,\texttt{|=}\, \sigma) = \texttt{true}$
$\mathcal{R}, s_0 \models \neg b \Leftrightarrow \mathcal{R}, s_0 \not\models b$
$\mathcal{R}, s_0 \models b_1 \vee b_2 \Leftrightarrow \mathcal{R}, s_0 \models b_1$ *or* $\mathcal{R}, s_0 \models b_2$
$\mathcal{R}, s_0 \models b_1 \wedge b_2 \Leftrightarrow \mathcal{R}, s_0 \models b_1$ *and* $\mathcal{R}, s_0 \models b_2$

The semantics for strategies and strategy formulas are given in the following two definitions:

Definition 7 (*Strat* semantics). *Given a rewrite system $\mathcal{R}$, a strategy e, a computation (s,t), and an integer k, we define the bounded semantic relation $\models_k$, whose value can be true, false or uncertain, by case distinction:*

If $\mathrm{fail}(e)$ *then*	$\mathcal{R}, (s,t) \models_k e$ *is false for every* k
else if $\mathrm{tick}(e)$ *then*	$\mathcal{R}, (s,t) \models_k e$ *is true for every* k
else if $k = 0$ *then*	$\mathcal{R}, (s,t) \models_0 e$ *is uncertain*
else	$\mathcal{R}, (s,t) \models_k e \;=\; \mathcal{R}, (s,t)^1 \models_{k-1} \mathrm{rest}_{s_0,t_0}(e)$

Definition 8 (*StratForm* semantics). *Given a rewrite system $\mathcal{R}$, a strategy e, a computation (s,t) (with s_0 always denoting the first state in s), and an integer k, we define the bounded semantic relation $\models_k$, whose value can be true, false or uncertain, by case distinction:*

$\mathcal{R}, s_0 \models_k \mathbf{A}\, e$ *is true* $\Leftrightarrow$ $\mathcal{R}, (s,t) \models_k e$ *is true for all* (s,t)
$\mathcal{R}, s_0 \models_k \mathbf{A}\, e$ *is false* $\Leftrightarrow$ $\mathcal{R}, (s,t) \models_k e$ *is false for some* (s,t)
$\mathcal{R}, s_0 \models_k \mathbf{A}\, e$ *is uncertain otherwise*

$\mathcal{R}, s_0 \models_k \mathbf{E}\, e$ *is true* $\Leftrightarrow$ $\mathcal{R}, (s,t) \models_k e$ *is true for some* (s,t)
$\mathcal{R}, s_0 \models_k \mathbf{E}\, e$ *is false* $\Leftrightarrow$ $\mathcal{R}, (s,t) \models_k e$ *is false for all* (s,t)
$\mathcal{R}, s_0 \models_k \mathbf{E}\, e$ *is uncertain otherwise*

Theorem 1. *[20,18] Given a computable rewrite system $\mathcal{R}$, a TLR* guarantee formula ϕ and a computation (s,t) in $\mathcal{R}$, we have*

$\mathcal{R}, (s,t) \models \phi \Leftrightarrow \exists k \in \mathbb{N}$ *such that* $\mathcal{R}, (s,t) \models_k \beta(\phi)$ *is true*
$\mathcal{R}, s_0 \models \mathbf{A}\, \phi \Leftrightarrow \exists k \in \mathbb{N}$ *such that* $\mathcal{R}, s_0 \models_k \mathbf{A}\, \beta(\phi)$ *is true*
$\mathcal{R}, s_0 \models \mathbf{E}\, \phi \Leftrightarrow \exists k \in \mathbb{N}$ *such that* $\mathcal{R}, s_0 \models_k \mathbf{E}\, \beta(\phi)$ *is true*

Proof (sketch). Based on the definition, semantic equivalences $e \equiv e'$ between strategy expressions e and e' can be proved just by proving that $\mathrm{fail}(e) = \mathrm{fail}(e')$,

$\text{tick}(e) = \text{tick}(e')$, and $\text{rest}_{s_0,t_0}(e) \equiv \text{rest}_{s_0,t_0}(e')$. In this way, we get some intuitive and useful equivalences. For instance:

$$e \,|\, e' \equiv e' \text{ if } \text{fail}(e)$$
$$\text{idle} \,;\, e \equiv e$$

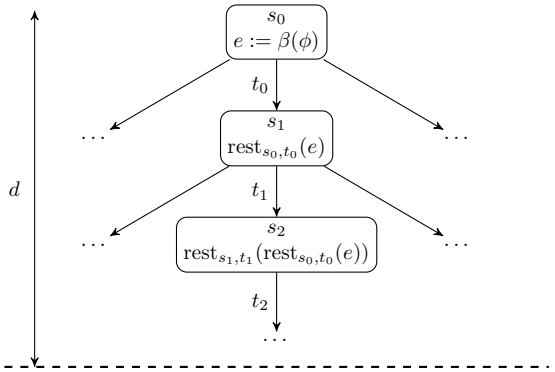
With equivalences like these, a proof of the theorem by structural induction is straightforward. We review just the case for the **X** operator:

$\mathcal{R}, (s,t) \models \mathbf{X}\,\phi$	$\Leftrightarrow$ (TLR* semantics)
$\mathcal{R}, (s,t)^1 \models \phi$	$\Leftrightarrow$ (induction hypothesis)
$\exists k$ s.t. $\mathcal{R}, (s,t)^1 \models_k \beta(\phi)$ is true	$\Leftrightarrow$ (semantic equivalence)
$\exists k$ s.t. $\mathcal{R}, (s,t)^1 \models_k \text{idle} \,;\, \beta(\phi)$ is true	$\Leftrightarrow$ (definition of rest_{s_0,t_0})
$\exists k$ s.t. $\mathcal{R}, (s,t)^1 \models_k \text{rest}_{s_0,t_0}(\text{any} \,;\, \beta(\phi))$ is true	$\Leftrightarrow$ (strategy semantics)
$\exists k$ s.t. $\mathcal{R}, (s,t) \models_{k+1} \text{any} \,;\, \beta(\phi)$ is true	$\Leftrightarrow$ (definition of β)
$\exists k$ s.t. $\mathcal{R}, (s,t) \models_{k+1} \beta(\mathbf{X}(\phi))$ is true	

7 The Implementation

Maude has reflective capabilities through its *metalevel* [11]. That means, for instance, that using the Maude language we can ask Maude itself about the possible rewrites from some given state, so that we can manipulate them in our code. We use the metalevel to compute and explore all evolutions of the system.

Functions equivalent to tick, fail, and rest_{s_0,t_0} have been coded into Maude. With this we make the strategy evolve at the same time as the system. To avoid getting lost in infinite branches, we use a bounded depth-first search. The depth is a parameter the user provides. The tool includes a command to search some additional levels when the previous search has not been conclusive. This shows a scheme of the algorithm:



At each node, the algorithm checks whether the state definitively satisfies the strategy, or it definitively does not, or more states have to be explored. It has to take care of a few more points not mentioned yet:

– Whether the quantifier is `forall` or `exists` to stop or continue the search when a satisfying or falsifying node is found.

- Storing the path from the initial state to the current one for loop detection and witness reporting. This is achieved by means of an ordered list of (state, strategy) pairs. At present, we do not store the whole set of visited nodes, so that our tests for repetitions are limited to the current path.
- Storing the open branches of the computation tree, in case they are later needed for a deeper search. It is not enough to store the set of leaves open at the depth bound, but whole paths to all of the open leaves are needed, so that a new, deeper search will be able to do loop detection and witness reporting properly from the original initial state.

The model checker, as it is, does not show an industrial-level performance and, although we show its usefulness below, it must be rather seen as a prototype.

The tool's user interface has two components: some commands related to model checking (described later), and an operator on modules, called `EXTENDED`. Given a system module—let us call it `UserMod`—our tool is able to generate the module `EXTENDED[UserMod]` and put it at the user's disposal. The operator `EXTENDED` assumes and needs that `UserMod` has a sort named `State`. The module `EXTENDED[UserMod]` adds to `UserMod` the following (syntax largely borrowed from [3,5]):

- New sorts `Trans`, `StateProp`, and `TransProp`, and satisfaction operators
  `op |= : State StateProp -> Bool .` `op |= : Trans TransProp -> Bool .`
- For each sort `S` in `UserMod`, a new sort `Context$S`, a subsort declaration `S < Context$S`, and an overloaded constant `op [] : -> Context$S`.
- For each constructor operator, say `op f : S1 S2 ... Sn -> S`, new operators
  ```
  op f : Context$S1 S2 ... Sn -> Context$S .
  ...
  op f : S1 S2 ... Context$Sn -> Context$S .
  ```
- The constructor for proof terms
  `op {_|_:_} : Context$State Qid Subst -> Trans .`

  Note that this is not the algebraic syntax proposed in Section 3, but is equivalent to it and more convenient for implementation.
- New sorts `Assign` and `Subst`, with the subsort relation `Assign < Subst`.
- For each sort `S` in `UserMod`, an operator `op _\_ : Qid S -> Assign`.
- New operators to build substitutions:
  ```
  op noSubst : -> Subst .
  op _;_ : Subst Subst -> Subst [comm assoc id: noSubst] .
  ```
- An operator `op _instanceOf_ : Subst Subst -> Bool` and equations to define the `instanceOf` relation as true when the first argument, taken as a set of assignments, is a subset of the second.

Users need to write at least two modules:

```
(mod UserMod is
   sort State .
   ...
 endm)
```

```
(mod UserMod-FULL is
   extending EXTENDED[UserMod] .
   op foo : -> StateProp .
   ...
 endm)
```

The first one, `UserMod`, has the whole specification of the system in the usual Maude way. Of course, it can import other modules as needed. The other module has to include the instruction `extending EXTENDED[UserMod]`. In this second module the users—having at their disposal all the infrastructure on proof terms, satisfaction, etc.—can define their own atomic propositions to be used in TLR* formulas, as we did in the example of Section 1. The definition of initial states is usually included in this second module as well.

The main model-checking command the user can issue is

`(ismc [`d`]` s `|=` q ϕ `.)`

Here, d is a natural number that specifies the maximum depth in the system's state space to which the search has to be performed; s is a term of sort `State`; q is a quantifier, either the literal `exists` or `forall`; and ϕ is a TLR* guarantee formula.

The concrete syntax for TLR* guarantee formulas is:

$$\phi ::= \texttt{TRUE} \mid \texttt{FALSE} \mid \sigma \mid \texttt{NOT}\ \sigma \mid \tau \mid \texttt{NOT}\ \tau \mid \phi_1\ \texttt{AND}\ \phi_2 \mid \phi_1\ \texttt{OR}\ \phi_2 \mid \texttt{X}\ \phi \mid \texttt{F}\ \phi \mid \phi_1\ \texttt{U}\ \phi_2$$

The possible answers to a model-checking command are `Yes` or `No`, with a witness computation when appropriate, or `DontKnow` when the search was not conclusive, followed in this case by the number of open tasks left. Witnesses and counterexamples can be, in particular, looping computations. In the `DontKnow` case, the tool keeps in its memory all the open tasks, so that this new command can be issued:

`(ismc deeper [`d`] .)`

This asks the model checker to search d more levels for each open task remaining after the latest `ismc` or `deeper` command.

There are also two `set` commands:

- `(ismc set loops on / off .)`
- `(ismc set contexts on / off .)`

The first instructs the tool to look (or not) for possible looping computations as it searches. It must be noted that in a loop not only states have to repeat, but also the strategies coupled with them. Detecting loops is not for free, as it involves the storing of information and the comparison of terms. Some model-checking tasks in which loops happen to be rare benefit notably from disabling loop detection. However, in many cases the effort is worthwhile—it may even be the only way to reach a final answer.

The command about contexts instructs the tool to generate (or not) contexts for proof terms. Contexts tend to be seldom used in practice; when they are not

used, one gains some performance by setting contexts off (about 10% to 20% according to our measures).

8 Verification of the MSI Cache Coherence Protocol

As an example to test our tool, we have chosen cache coherence protocols, a problem that does not seem to have been previously modeled in rewriting logic. In multiprocessor computers it is frequently the case that a small cache memory is attached to each processor. This cache memory, or just *cache*, holds a copy of a part of the main memory. A processor only reads from and writes to its cache, improving in this way the overall computer's performance. The cache coherence problem arises because several caches may hold copies of the same main-memory address, and the respective processors may write different data on them.

To avoid this problem, cache coherence protocols have been devised, that dictate the actions a cache must perform according to the orders that arrive to it. Here we consider one of the best-known protocols: MSI. In this protocol, like in most others, each cache line is marked with a *mode* that determines the validity of the information it stores. A cache line is the smallest chunk of memory that can be moved between the main memory and a cache. In MSI a cache line can be in one of three modes:

Modified: the line has been modified in this cache, so other copies of the same memory address, both in caches and in main memory, are unreliable;
Shared: the line is valid and so is every other copy of the line stored in any cache and in main memory;
Invalid: the line is not valid, presumably because it has been modified elsewhere.

The initials of these three modes give its name to the protocol. We abbreviate the three modes as mdf, shr, and inv.

There is usually a bus through which all communication between caches and main memory happens. MSI was designed for buses that do not allow direct communication of data from cache to cache. However, the caches are able to *snoop* the bus, that is, to monitor it to detect when another cache is trying to access a certain main-memory address. By snooping, a cache cannot see the data being read or written by another cache, but only its address in main memory.

To simplify the model, operations on the bus are taken to be atomic, that is, they are fully dealt with before the system does anything else. Reality is usually not that simple, but considering technicalities on the bus side would make the model unnecessarily complex. Also, we assume that each cache only contains one line of information. This turns out to be an unrealistic but sensible simplification: each operation happens to a particular line, all other lines in the cache being irrelevant to this operation.

A computer is a finite object. Although the number of processors and the size of the memory are not limited in principle, once a computer is built and running, these numbers are fixed. Or this was so until the advent of virtual

machines. Virtualization software allows running operating systems on virtual hardware that is not mapped in a one-to-one fashion to actual hardware. The online manual for VMware vSphere 5.1 [23] states: "When the virtual machine is turned on [...] you can hot add virtual CPUs to the running virtual machine." Thus, we include in our model the possibility of adding new processors *on the fly*, which turns the number of states reachable from a given one into infinite.

First, we need the data structures. A `Line` consists of two natural numbers representing the address and the data stored. Caches and processors are independent entities, identified and coupled by its `ChipId` (a `Nat`). A CPU, or processor, contains just its `ChipId` and a Boolean that indicates whether it sent a message to its cache and is waiting for the answer. A cache registers its `ChipId`, its mode and its only line of information.

```
sorts Mode Line CPU Cache .
ops mdf shr inv : -> Mode [ctor] .
op line : Address Data -> Line [ctor] .
op cpu : ChipId Bool -> CPU [ctor] .
op cache : ChipId Mode Line -> Cache [ctor] .
```

Main memory is declared as a set of lines enclosed in double curly brackets:

```
sort MemContents .  subsort Line < MemContents .
op mtMemContents : -> MemContents [ctor] .
op __ : MemContents MemContents -> MemContents
        [ctor comm assoc id: mtMemContents] .
eq MC:MemContents MC:MemContents = MC:MemContents .
sort Memory .
op {{_}} : MemContents -> Memory [ctor] .
```

The bus is not a distinct entity in our model: there are `BusMessage`s loose in the state. There are also `LocalMessage`s, that is, messages between a processor and its cache, whose means of transmission is of no concern to us either.

```
sorts BusMessage LocalMessage .
op bus-read : ChipId Address -> BusMessage [ctor] .
op bus-hereur : ChipId Line -> BusMessage [ctor] .
op read : ChipId Address -> LocalMessage [ctor] .
op hereur : ChipId Line -> LocalMessage [ctor] .
op write : ChipId Line -> LocalMessage [ctor] .
```

We want to control the amount of memory addresses and of possible data values available, so that they can be kept to the minimum we need in each moment. We do so by defining a sort of sets on natural numbers, `NatSet`, and these two sorts:

```
sorts AddressRange DataRange .
op aRange : NatSet -> AddressRange [ctor] .
op dRange : NatSet -> DataRange [ctor] .
```

A `State` is defined as a *soup* of elements, enclosed in curly brackets:

```
sort StateContents .
subsorts CPU Cache Memory BusMessage LocalMessage
         AddressRange DataRange ChipId < StateContents .
```

```
op mtStateContents : -> StateContents [ctor] .
op __ : StateContents StateContents -> StateContents
        [ctor comm assoc id: mtStateContents] .
sort State .
op {_} : StateContents -> State [ctor] .
```

We also store in the state the maximum `ChipId` currently used, so that adding new caches is easier. Some provisos are missing that are important. For instance, one and only one `Memory`, `AddressRange`, and `DataRange` should exist in a given `State`, and there should be as many `Caches` as `CPUs`, coupled by id. These and others will be enforced in the initial states we use and in the rules that govern the system.

Let us consider now the dynamics of the system. A processor starts sending a read or write order. Its cache, and maybe others, reacts in a number of ways, depending on where the information is stored. There are ten possible cases a cache must be ready to react to: from its processor it can receive a read or a write order; from the bus it can snoop a read or a write being performed on another cache, or also an invalidate signal (to be explained soon). Each of these five cases unfolds into two, as we need to separately consider a *hit*, that happens when the order the cache receives refers to the memory address that the cache is already storing, and a *miss*, which is the opposite.

First, this is the way a new processor and cache may come into existence:

```
var Id : ChipId .  var SC : StateContents .
crl [add] : { Id SC }
         => { s(Id) cpu(s(Id), false) cache(s(Id), inv, line(0, 0)) SC }
         if allCpusBusy(SC) .
```

The function `allCpusBusy` checks that each existing processor is waiting for an answer. So, we can only add a processor when all others are busy. These are the ways a processor sends an order to its cache:

```
vars N N' : Nat .      vars NS NS' : NatSet .  var Md : Mode .
vars A A' : Address .  var  MM : Memory .
crl [read] : { cpu(Id, false) aRange(N ; NS) SC }
          => { cpu(Id, true) read(Id, N) aRange(N ; NS) SC }
          if not hasBusMsg(SC) .
crl [write] : { cpu(Id, false) aRange(N ; NS) dRange(N' ; NS') SC }
           => { cpu(Id, true) write(Id, line(N, N'))
                aRange(N ; NS) dRange(N' ; NS') SC }
           if not hasBusMsg(SC) .
```

The condition in the rules ensures that bus messages are dealt with before any other action takes place.

We review next some of the ways in which the system can react to an order. Specially simple are the cases for bus misses. On snooping these, a cache would think: "Someone is reading from or writing to or invalidating an address I don't have stored, so I have nothing to do." Therefore, we include no rule for this case. For a more complex case, consider the "processor write hit." On detecting this, a cache in mode `shr` would think: "My processor needs to write to a line I already have stored. I will just write the new data. But, as this is the first time I modify

this line, I will ask the bus to send an *invalidate* signal, so that other caches are aware that some change has happened. And I will change to mdf."

```
crl [write-hit] :
    { cpu(Id, true) cache(Id, Md, line(A, D)) write(Id, line(A, D')) SC }
 => { cpu(Id, false) cache(Id, mdf, line(A, D'))
      (if Md == shr then invalidate(Id, A, SC) else SC fi) }
 if not hasBusMsg(SC) .
```

The `invalidate` function simulates the workings of the bus and the snooping caches, running through the state to find caches that need to be invalidated:

```
op invalidate : ChipId Address StateContents -> StateContents .
ceq invalidate(Id, A, cache(Id', Md, line(A, D)) SC) =
    cache(Id', inv, line(0, 0)) invalidate(Id, A, SC) if Id =/= Id' .
eq invalidate (Id, A, SC) = SC [owise] .
```

For a "processor read miss" the cache must react like this:

```
crl [read-miss] : { MM cache(Id, Md, line(A, D)) read(Id, A') SC }
               => { (if Md == mdf then update(MM, line(A, D)) else MM fi)
                    cache(Id, Md, line(A, D)) bus-read(Id, A') SC }
               if A =/= A' /\ not hasBusMsg(SC) .
```

The processor needs to read an address `A'` that is not the one stored now in its cache. The cache puts in the system a message `bus-read`, asking for the needed data. When the answer is finally received, cache contents are going to be overwritten, so if cache was in mode mdf, that is, if it had the only valid copy of its data, it has to copy its line to main memory (*eviction* is the technical term for this action). That is what the `update` function is for.

Now, when a mdf cache sees a `bus-read` order for the address it is storing, it copies its data to main memory and produces the answer.

```
crl [bus-read-hit] : { MM cache(Id', mdf, line(A, D)) bus-read(Id, A) SC }
                  => { update(MM, line(A, D)) cache(Id', shr, line(A, D))
                       bus-hereur(Id, line(A, D)) SC }
                  if Id =/= Id' .
```

In case no mdf cache has the data, it is the main memory who must answer to the `bus-read`, through a rule not shown here. By the way, the previous rule is a simplification of the standard MSI, as caches cannot usually communicate directly data to each other. The reading cycle ends with these two rules:

```
rl [bus-read-done] : { cache(Id, Md, L) bus-hereur(Id, L') SC }
                  => { cache(Id, shr, L') hereur(Id, L') SC } .
rl [read-done] : { cpu(Id, true) hereur(Id, L) SC }
              => { cpu(Id, false) SC } .
```

To begin with model-checking tasks, we wonder first whether invalidating is really needed. Namely: if we remove invalidation from our system, is coherence still guaranteed? This is more intuitive if viewed as a question about the safety formula **A G** `coherent`, but we use its negation, as our model checker only accepts guarantee formulas. We redefine `invalidate` to an identity in its third argument, and define a `coherent` proposition (after importing `EXTENDED[MSI]`):

```
op coherent : -> StateProp [ctor] .
```

```
ceq { cache(Id, shr, line(A, D)) cache(Id', shr, line(A, D')) SC }
    |= coherent = false if D =/= D' .
ceq { cache(Id, shr, line(A, D)) {{line(A, D')}} SC }
    |= coherent = false if D =/= D' .
eq { SC } |= coherent = true [owise] .
```

We ask our brand-new model checker whether coherence can be violated

```
(ismc [8] init |= exists F NOT coherent .)
```

from this initial state:

```
op init : -> State .
eq init = { cpu(1, false) cache(1, mdf, line(1, 2))
            cpu(2, false) cache(2, inv, line(0, 0))
            2 {{line(1, 1)}} aRange(1 ; 2) dRange(1 ; 3) } .
```

We get a `Yes` and a witness computation ending in this state:

```
{ {{line(1, 1)}} aRange(1 ; 2) dRange(1 ; 3) 2
  cpu(1, true) cpu(2, true) cache(1, mdf, line(1, 1))
  cache(2, shr, line(1, 2)) bus-read(1, 2) hereur(2, line(1, 2)) }
```

So, yes, invalidating is necessary, and we restore it to go on. The question in this example is a reachability one and does not use propositions on transitions, so its result can be achieved as well with Maude's `search` command.

Next, we consider this initial state:

```
op init2 : -> State .
eq init2 = { cpu(1, false) cache(1, mdf, line(1, 3))
             cpu(2, true) cache(2, inv, line(0, 0)) read(2, 1)
             2 {{line(1, 1)}} aRange(1) dRange(4) } .
```

Processor `2` wants to read the contents of memory address `1`. That information is stored in the main memory, but it is cache `1` who has the only valid value. We want to check that, eventually, cache `2` receives `line(1, 3)`. With `aRange(1)` and `dRange(4)`, we include the possibility for processors to initiate new reads or writes to address `1` with a different value `4`; this is not a big range of possibilities, but it is all we need to try to interfere with the reading.

We model check this:

```
(ismc [10] init2 |= forall F readdone(2, line(1, 3)) .)
```

for this parametric proposition on transitions:

```
var Ln : Line .  var C : Context$State .  var L : Qid .  var Sb : Subst .
op readdone : ChipId Line -> TransProp [ctor] .
eq {Cn | 'read-done : ('Id \ Id) ; ('L \ Ln) ; Sb}
   |= readdone(Id, Ln) = true .
eq {C | L : Sb} |= readdone(Id, Ln) = false [owise] .
```

Unfortunately, it produces a `No` after finding a looping computation in just four steps. The problem is clearly unfairness: the system is only paying attention to processor `1`, or creating new processors. This example shows model checking with an infinity of reachable states and with a proposition on transitions, which puts it out of the scope of other existing tools. Some other model-checking tasks at different levels of abstraction are shown in [18].

Unfortunately, slight increases in the number of caches tend to cause large increases in the time needed for the checking to complete. On the other hand, it is known that most design flaws can often be found using small systems.

9 Related Work

Model Checking TLR*. The papers [6,3,4,5] are all related to model checking Maude modules with LTLR formulas. The logic LTLR is the linear sublogic of TLR*, that is, formulas with no path quantifiers taken to be universally quantified on paths. In [6] the authors implement a translation, already described in [20], that allows the use of Maude's LTL model checker (see [11] for explanations on this model checker). The idea is the following: we are given a rewrite system $\mathcal{R}$, with an initial state on it, and a LTLR formula as parameters to perform model checking on them. From $\mathcal{R}$ we produce a new system, equivalent to $\mathcal{R}$ in an appropriate way, whose states store, in addition to its own information, also data on the transition that took the system to them. In parallel, we translate the given LTLR formula to a LTL formula with equivalent semantics. The produced system satisfies the produced formula iff the given system satisfies the given formula [20]. Looking for better performance, [3] implements a different algorithm for LTLR model checking in C++, by modifying the implementation of Maude's LTL model checker.

The papers [4,5] show how fairness constraints can be included in the system specification, and how to model check LTL or LTLR formulas taking these constraints into account in the very algorithm. Moreover, these papers show how to use *parameterized* fairness properties, that allow the user to specify which entities of the system have to be treated with fairness and which others we do not care about.

None of these model checkers works for systems with an infinity of reachable states.

Infinite Systems. Model checking on infinite systems has been the subject of many studies. Most of them look for either an abstraction that turns the system finite, or a way to finitely represent the elements that compose the system.

Abstraction is a well-known mechanism to make the size of a system smaller, where *smaller* can even mean finite from infinite. See [21] for an approach within rewriting logic. The idea of abstraction is grouping together states that, though different, are indistinguishable to the formula we are model checking. In this respect, model checking on timed systems often uses *time regions* with the same idea: instead of using time *instants*, use well-chosen time *intervals*, taking care that the given temporal formula is not able to tell apart two instants on the same interval.

In the way of finite representability, the method of *well-structured transition systems* has proved useful [1,15]. A well-structured transition system is one in whose infinite set of states a well-quasi-ordering has been defined. A well-quasi-ordering is a reflexive and transitive relation such that no infinite strictly

decreasing sequence exists. In a well-structured system certain sets (so called *upward-closed sets*) of states are finitely representable. These sets are enough to provide algorithms to solve some model-checking problems. The reference [15] lists a collection of natural examples for which a well-quasi-ordering exists.

The papers [13,2] describe a narrowing-based approach to model checking rewrite systems in which terms with variables are used as patterns to represent and let evolve whole sets of states. Also, abstraction and *folding relations* (similar to the quasi-orderings of well-structured transition systems) are used.

As explained below, the advantage of our own method is that it does not need to find relations and prove they are appropriate, but uses the raw system as it is given.

Strategies. Strategies do not seem to have been used as a means to model checking before. However, they are present in several languages. In Maude, there is a rich strategy language; see [17], for instance. In some sense, that is a more powerful strategy language than the one presented in this paper, although none of them contains the other. In [22] strategies are used in the framework of program transformation (like for refactoring, compiling, optimization). In particular, they use Stratego, a language for program transformation based on rewriting and strategies. ELAN, described for instance in [8], is a rewriting-logic language. Both ELAN and Stratego have strategies included in the language, while in Maude system modules and strategy modules are separated syntactic entities.

10 Conclusions and Future Work

Several subjects related to system specification and verification have got roles in this work: rewriting logic (and Maude) as a specification formalism, rewriting logic (and Maude) as a software development tool, state-based and action-based temporal logics and TLR*, guarantee and safety properties, strategies applied to nondeterministic systems, and model checking on infinite systems. We have introduced all of them. We have implemented a strategy language and shown how it can be used to model check TLR* guarantee formulas on possibly infinite systems by first translating them into strategy expressions. Finally, we have proved the usefulness of the tool verifying the MSI cache coherence protocol.

Our model checker has a unique combination of three ingredients: it admits propositions on transitions (and states), bounded search on finite or infinite systems (even with an infinity of reachable states), and existential or universal quantification on paths. Maude's `search` command works in a bounded way, but lacks the other ingredients; Maude's built-in model checker for LTL does not have any of the three; and the LTLR model checkers cited above allow propositions on transitions, but not the other two.

An explicit-state model-checking procedure on infinite systems cannot be expected to produce a definitive answer in all cases, and cannot be expected either to provide the best performance. However, the point is that our model-checking procedure is available almost for free as soon as the system is specified. Quoting

Meseguer, talking about an example presented in [20], "all such efforts to obtain a tractable finite-state abstraction, and the associated theorem proving work to check confluence, coherence and preservation of state predicates for the abstraction, are not even worth it; since this simpler analysis of the system specification has already uncovered a key flaw." Thus, we think explicit-state model checking deserves a place in an infinite-system verification toolbox.

Several improvements and lines for additional work are possible. A C++ implementation in search for better performance is an obvious thing to do. In a different line, we already have loop detection, that is, detection of repeated (state, strategy) pairs on the same path. But, when repetition occurs in different branches, we are not ready to detect it. For some systems, this would provide a drastically improved performance. Also some other tools can be offered to the user: abstraction, well-structured transition systems, and partial order reduction. To this end, means should be implemented to allow the user specify, respectively, when two different states can be considered equivalent to the current model-checking task, or when they are related by the well-quasi-ordering, or when two transitions are independent, so that only one way to order and perform them has to be taken into account. The reference [14] has proposals on how to implement partial order reduction in rewrite systems.

Acknowledgments. We thank José Meseguer for motivating this research and showing us its initial foundations; Fernando Rosa for answering our questions about model checking and suggesting us readings and examples; and the anonymous referees for their helpful suggestions to improve this paper.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321. IEEE Computer Society Press (1996)
2. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) RTA. LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
3. Bae, K., Meseguer, J.: The linear temporal logic of rewriting Maude model checker. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 208–225. Springer, Heidelberg (2010)
4. Bae, K., Meseguer, J.: State/event-based LTL model checking under parametric generalized fairness. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 132–148. Springer, Heidelberg (2011)
5. Bae, K., Meseguer, J.: Model checking LTLR formulas under localized fairness. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 99–117. Springer, Heidelberg (2012)
6. Bae, K., Meseguer, J.: A rewriting-based model checker for the linear temporal logic of rewriting. Electr. Notes Theor. Comput. Sci. 290, 19–36 (2012)
7. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)

8. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E.: ELAN from a rewriting logic point of view. Theoretical Computer Science 285(2), 155–185 (2002)
9. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1-3), 386–414 (2006)
10. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007)
12. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
13. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 153–168. Springer, Heidelberg (2007)
14. Farzan, A.: Static and Dynamic Formal Analysis of Concurrent Systems and Languages: A Semantics-Based Approach. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (2007)
15. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
16. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
17. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. In: Martí-Oliet, N. (ed.) Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 417–441. Elsevier (2004)
18. Martín, Ó.: Model checking TLR* guarantee formulas on infinite systems. Master's thesis, Facultad de Informática, Universidad Complutense de Madrid (July 2013), `http://maude.sip.ucm.es/ismc`
19. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
20. Meseguer, J.: The temporal logic of rewriting. Technical Report UIUCDCS-R-2007-2815, Department of Computer Science, University of Illinois at Urbana-Champaign (2007)
21. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403(2-3), 239–264 (2008)
22. Visser, E.: A survey of strategies in program transformation systems. Electr. Notes Theor. Comput. Sci. 57, 109–143 (2001)
23. VMware. vSphere Virtual Machine Administration (update 1, ESXi 5.1, vCenter Server 5.1), `http://pubs.vmware.com/vsphere-51/topic/com.vmware.ICbase/PDF/vsphere-esxi-vcenter-server-511-virtual-machine-admin-guide.pdf`