Model Checking LTLR Formulas under Localized Fairness

Kyungmin Bae and José Meseguer

Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana IL 61801
{kbae4,meseguer}@cs.uiuc.edu

Abstract. Many temporal logic properties of interest involve both state and action predicates and only hold under suitable fairness assumptions. Temporal logics supporting both state and action predicates such as the Temporal Logic of Rewriting (TLR) can be used to express both the desired properties and the fairness assumptions. However, model checking such properties directly can easily become impossible for two reasons: (i) the exponential blowup in generating the Büchi automaton for the implication formula including the fairness assumptions in its condition easily makes such generation unfeasible; and (ii) often the needed fairness assumptions cannot even be expressed as *propositional* temporal logic formulas because they are *parametric*, that is, they correspond to *universally quantified* temporal logic formulas. Such universal quantification is succinctly captured by the notion of *localized fairness*; for example, fairness localized to the parameter o in object fairness conditions. We summarize the foundations and present the language design and implementation of the new Maude LTLR Model Checker under localized fairness. This is the first tool we are aware of which can model check temporal logic properties under parametric fairness assumptions.

1 Introduction

Many temporal logic properties of interest involve both state and action predicates and only hold under suitable fairness assumptions. For example, the effective transmission of data by a fault-tolerant network protocol can only be proved under the assumption that the receiving node will receive messages infinitely often if it is infinitely often enabled to receive them. Although in principle temporal logics supporting both state and action predicates such as the Temporal Logic of Rewriting (TLR) can be used to express both the desired properties and the fairness assumptions, in practice model checking *directly* such properties can easily become impossible for two reasons. First of all, the exponential blowup in generating the Büchi automaton for the formula $\psi \rightarrow \varphi$, where φ is the desired property and ψ specifies the fairness assumptions, easily makes such generation unfeasible. To address this problem, various techniques to build in the fairness assumptions ψ into the model checking algorithm, so that only φ has to be model checked, have been proposed as the basis of various model checkers [12, 14, 18].

However, a second serious difficulty not addressed by those techniques and tools exists: often the needed fairness assumptions ψ cannot even be expressed as *propositional* temporal logic formulas because they are *parametric*, that is, they correspond to *universally quantified* temporal logic formulas which are outside the scope of current fairness-supporting model checkers.

A good example of parametric fairness is provided by *object fairness* assumptions such as "each object o infinitely often enabled to receive a message of the form $m(o,q)$ will receive it infinitely often," which is universally quantified over all the (possibly dynamically changing) objects o in the system. In rewriting logic such message reception can be expressed by a rewrite rule of the form:

$$[rec] : [o \mid s]\; m(o,k) \;\rightarrow\; [o \mid f(s,k)]$$

where $f(s,k)$ denotes a new state after receiving a message. This object fairness assumption can be described as the universally quantified LTLR formula:

$$(\forall o)\;\; enabled.rec(o) \rightarrow\; rec(o)$$

where $enabled.rec(o)$ is the obvious state predicate holding iff the *rec* rule is enabled for object o. Such universal quantification can be succinctly captured by the notion of *localized fairness* [15]. The idea is that a rewrite rule like the one above is itself universally quantified over a finite set of variables; for example, for the *rec* rule the set $\{o, s, k\}$. Then the above strong object fairness condition corresponds to localizing the strong fairness requirement for rule *rec* to the singleton subset $\{o\}$. In general, fairness assumptions for a given rule can be localized to any chosen *subset* of its variables.

Is it possible at all to model check temporal logic properties under such parametric fairness assumptions? The question is nontrivial, because, even under the finite-state assumption which can make the number of actual instances of the universal quantification finite, it may be impossible to have a priori knowledge about the actual number of such instances. For example, we may be dealing with an object-based system where objects are dynamically created, so that the entire state space may first have to be searched to determine which objects exist. We have recently reported on the automata-theoretic and algorithmic foundations of a novel model checking algorithm which solves the problem of model checking LTL properties under parameterized fairness and has good performance in practice [2]. However, the work in [2] dealt with this problem at an automata-theoretic level and did not present a suitable *property specification language* in which such parameterized fairness assumptions could be naturally expressed.

In this paper we show that the intrinsically parametric nature of rewrite rules and the great flexibility of the Linear Temporal Logic of Rewriting (LTLR) to express parametric action patterns based on such rules makes LTLR an ideal property specification language for a model checker under parameterized fairness assumptions. In this way, the algorithm presented in [2] becomes available for rewriting logic specifications. Furthermore, we present and illustrate with examples a new LTLR model checker under localized fairness assumptions for Maude, which implements the algorithm in [2] at the C++ level as an extension of the Maude system. A nontrivial part of this model checker is its user interface. First of all, a simple way for users to specify localized fairness assumptions as rule annotations is provided. Secondly, since LTLR formulas involve spatial action patterns which in turn involve rule labels, LTLR formulas go beyond the syntax available to Maude from parsing the system module to be model checked, or even such module syntax extended by the state predicates and the temporal logic connectives. We explain how rule annotations can also be used for this purpose. Of course, *reflection* techniques and the Full Maude infrastructure are used in an essential way to obtain this expressive and user-extensible user interface. The new Maude LTLR model checker is the first tool we are aware of which can model check temporal logic properties under parametric fairness assumptions. The tool and a collection of examples can be accessed at `http://www.cs.illinois.edu/~kbae4/tlr`.

2 Preliminaries on Localized Fairness

This section recalls preliminary notions on rewrite theories, the linear temporal logic of rewriting (LTLR), and localized fairness specifications. We also summarize our previous works on the LTLR model checking algorithm [1] and the LTL model checking algorithm under parameterized fairness assumptions [2], which provide the basis for the new LTLR model checker described in this paper.

2.1 Rewrite Theories

A rewrite theory [4] is a triple $\mathcal{R} = (\Sigma, E, R)$ such that:

- (Σ, E) is a theory in *membership equational logic* with Σ a signature, E a set of *conditional* equations and memberships, and
- R is a set of (possibly conditional) *rewrite rules* written $l : q \to r$, where l is a *label*, and q and r are Σ-terms.

The state space with a chosen type k is specified as the k-component of the initial algebra $T_{\Sigma/E,k}$, i.e., each state is an E-equivalence class $[t]_E$ of ground terms with type k. Each rule $l : q \to r$ specifies the system's concurrent transitions. A *one-step rewrite* from a state $[t[\theta q]]_E$ containing a substitution instance θq to the state $[t[\theta r]]_E$ in which θq has been replaced by θr is denoted by:

$$[t[l(\theta)]]_E : [t[\theta q]]_E \to_{\mathcal{R}} [t[\theta r]]_E$$

where $[t[l(\theta)]]_E$ is called a *one-step proof term.* A *computation* (π, γ) of $\mathcal{R}$ is a path $\pi(0) \xrightarrow{\gamma(0)} \pi(1) \xrightarrow{\gamma(1)} \pi(2) \xrightarrow{\gamma(2)} \cdots$ where $\pi(i) = [t_i]_E$ with the state type k, $\pi(i) \xrightarrow{\gamma(i)} \pi(i+1)$ is a one-step rewrite with a one-step proof term $\gamma(i)$ for each $i \in \mathbb{N}$. $(\pi, \alpha)^i$ denotes the suffix of (π, α) beginning at position $i \in \mathbb{N}$, i.e., $(\pi, \alpha)^i = (\pi \circ s^i, \alpha \circ s^i)$ with s the successor function. Any finite computation of $\mathcal{R}$ can be extended into an infinite computation if $\mathcal{R}$ has no deadlock states. Any rewrite theory whose rules have no rewrites in their conditions can be transformed into a semantically an equivalent deadlock-free theory [17]. We will assume from now on that a rewrite theory is deadlock free.

2.2 The Linear Temporal Logic of Rewriting

The *linear temporal logic of rewriting* (LTLR) is a state/event extension of linear temporal logic with *spatial action patterns* that describe properties of one-step rewrites [1]. The only syntactic difference between LTLR and LTL is that an LTLR formula may include spatial action patterns $\delta_1, \ldots, \delta_n$ as well as state propositions $p_1, \ldots, p_m$, and therefore can describe properties involving both states and events, e.g., fairness conditions. Given a set of state propositions Π and a set of spatial action patterns W, the syntax of LTLR formulas is as follows:

$$\varphi ::= p \mid \delta \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \bigcirc\varphi \mid \varphi \mathbf{U} \varphi'$$

where $p \in \Pi$ and $\delta \in W$. Other operators can be defined by equivalences, e.g., $\Diamond\varphi \equiv \mathit{true}\mathbf{U}\varphi$, and $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

The semantics of LTLR over a set of state propositions Π and a set of spatial action patterns W is defined on a rewrite theory $\mathcal{R}$ that contains a subtheory for Π, W, boolean values, and one-step proof terms. A state proposition (resp., a spatial action pattern) is defined by a parametric function symbol of the form $p : s_1 \ldots s_n \to \texttt{Prop}$ (resp., $\delta : s_1 \ldots s_m \to \texttt{Action}$). The satisfaction relations for state propositions and spatial action patterns are defined by means of equations using the following auxiliary operators:

$$\_\models\_ : k\ \texttt{Prop} \to \texttt{Bool} \qquad \_\models\_ : \texttt{ProofTerm Action} \to \texttt{Bool}$$

in which a state proposition p is satisfied on a state $[t]_E$ if and only if $E \vdash (t \models p) = \mathit{true}$, and a spatial action pattern δ is satisfied on a one-step proof term $[\lambda]_E$ if and only if $E \vdash (\lambda \models \delta) = \mathit{true}$. An LTLR formula φ is satisfied on $\mathcal{R}$ from an initial state $[t]_E$, denoted by $\mathcal{R}, [t]_E \models \varphi$, if and only if for each infinite computation (π, γ) starting from $[t]_E$, the path satisfaction relation $\mathcal{R}, (\pi, \gamma) \models \varphi$ holds, which is defined inductively as follows:

- $\mathcal{R}, (\pi, \gamma) \models p$ iff $E \vdash (\pi(0) \models p) = \mathit{true}$
- $\mathcal{R}, (\pi, \gamma) \models \delta$ iff $E \vdash (\gamma(0) \models \delta) = \mathit{true}$
- $\mathcal{R}, (\pi, \gamma) \models \neg\varphi$ iff $\mathcal{R}, (\pi, \gamma) \not\models \varphi$
- $\mathcal{R}, (\pi, \gamma) \models \varphi \wedge \varphi'$ iff $\mathcal{R}, (\pi, \gamma) \models \varphi$ and $\mathcal{R}, (\pi, \gamma) \models \varphi'$
- $\mathcal{R}, (\pi, \gamma) \models \bigcirc\varphi$ iff $\mathcal{R}, (\pi, \gamma)^1 \models \varphi$
- $\mathcal{R}, (\pi, \gamma) \models \varphi \mathcal{U} \varphi'$ iff $\exists j \geq 0.\ \mathcal{R}, (\pi, \gamma)^j \models \varphi'$, $\forall 0 \leq i < j.\ \mathcal{R}, (\pi, \gamma)^i \models \varphi$.

2.3 Localized Fairness

Fairness of a rewrite theory $\mathcal{R}$ is often expressed by patterns of rewrite events, i.e., by spatial action patterns. A one-step rewrite event is usually too specific to describe a general fairness requirement.

Definition 1. *A* basic action pattern *of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is a simple parametric spatial action pattern of the form $l(\overline{y})$ such that l is the label of some rule $l : q \to r \in R$ and $\overline{y} \subseteq vars(q)$. A ground instance $\theta(l(\overline{y}))$ of a basic action pattern $l(\overline{y})$ is satisfied on each one-step proof term of the form $[t[l(\theta)]]_E$.*

A basic action pattern $l(\emptyset)$ with no variables is denoted by just a rule label l. Also, a parametric state proposition $\mathit{enabled}(l(\overline{y}))$ can be defined in $\mathcal{R}$ such that a ground instance $\theta(\mathit{enabled}(l(\overline{y})))$ is satisfied on each state $[t]_E$ from which there exists a one-step rewrite $[t[l(\theta)]]_E$. The satisfaction relations for both basic action patterns and enabled propositions can be defined by equations and automatically generated from $\mathcal{R}$ (see Section 4.3).

A *localized fairness* specification [15] is a pair of finite sets $(\mathcal{J}, \mathcal{F})$ whose elements are basic action patterns of the general form $l(\overline{y})$. The set $\mathcal{J}$ stands for weak fairness conditions[1] and $\mathcal{F}$ stands for strong fairness conditions.[2] This localized fairness specification is quite general so that many different notions of fairness, such as object/process fairness, can be expressed in a unified way [15]. Intuitively, localized fairness given by $l(\overline{y}) \in \mathcal{J} \cup \mathcal{F}$ means that for each ground instance $\vartheta(l(\overline{y}))$ of $l(\overline{y})$, the corresponding one-step rewrite satisfies the desired weak or strong fairness requirements. Each localized fairness pattern can be expressed by an equivalent *universally quantified* LTLR formula [2] of the form $\forall \overline{y}\ \varphi$, where φ is quantifier-free, $\mathit{vars}(\varphi) \subseteq \overline{y}$, and:

$$\mathcal{R}, (\pi, \gamma) \models \forall \overline{y}\ \varphi \quad \Leftrightarrow \quad \mathcal{R}, (\pi, \gamma) \models \theta\varphi \text{ for each ground substitution } \theta$$

A weak (resp. strong) fairness condition with respect to a basic action pattern $l(\overline{y})$ is then expressed by the quantified LTLR formula $\forall \overline{y}\ \Diamond\Box \mathit{enabled}(l(\overline{y})) \rightarrow \Box\Diamond l(\overline{y})$ (resp., $\forall \overline{y}\ \Box\Diamond \mathit{enabled}(l(\overline{y})) \rightarrow \Box\Diamond l(\overline{y})$).

A localized fairness specification $(\mathcal{J}, \mathcal{F})$ defines a set of fair computation of a rewrite theory $\mathcal{R}$. An infinite computation (π, γ) of $\mathcal{R}$ is $\mathcal{J}, \mathcal{F}$–fair if and only if every localized fairness condition in $\mathcal{J} \cup \mathcal{F}$ is satisfied on (π, γ) in $\mathcal{R}$. That is,

– $\mathcal{R}, (\pi, \gamma) \models \forall \overline{y}_j\ \Diamond\Box \mathit{enabled}(l_j(\overline{y}_j)) \rightarrow \Box\Diamond l_j(\overline{y}_j)$, for each $l_j(\overline{y}_j) \in \mathcal{J}$, and
– $\mathcal{R}, (\pi, \gamma) \models \forall \overline{y}_f\ \Box\Diamond \mathit{enabled}(l_f(\overline{y}_f)) \rightarrow \Box\Diamond l_f(\overline{y}_f)$, for each $l_f(\overline{y}_f) \in \mathcal{F}$.

An LTLR formula φ is then *fairly* satisfied on $\mathcal{R}$ from an initial state $[t]_E$ under $(\mathcal{J}, \mathcal{F})$, denoted by $\mathcal{R}, [t]_E \models_{\mathcal{J} \cup \mathcal{F}} \varphi$, if and only if $\mathcal{R}, (\pi, \gamma) \models \varphi$ holds for each $\mathcal{J}, \mathcal{F}$-fair computation (π, γ) starting from the initial state $[t]_E$.

2.4 Model Checking Algorithms

The model checking problem for an LTLR formula φ on a rewrite theory $\mathcal{R}$ can be characterized by automata-theoretic techniques on the associated *labeled Kripke structure* (LKS) using the Büchi automaton $\mathcal{B}_{\neg\varphi}$ [1, 3].

Definition 2. *An LKS $\mathcal{K}$ is a 6-tuple $(S, S_0, \Pi, \mathcal{L}, W, T)$, where S is a set of states, $S_0 \subseteq S$ is a set of initial states, Π is a set of state propositions, $\mathcal{L} : S \rightarrow \mathcal{P}(\Pi)$ is a state-labeling function, W is a set of events (i.e., spatial action patterns), and $T \subseteq S \times \mathcal{P}(W) \times S$ is a labeled transition relation.*

[1] If an event is continuously enabled beyond a certain point, it is taken infinitely often.
[2] If an event is enabled infinitely often, then it is taken infinitely often.

A *path* (π, α) of $\mathcal{K}$ is an infinite sequence $\langle \pi(0), \alpha(0), \pi(1), \alpha(1), \ldots \rangle$ such that $\pi(i) \in S$, $\alpha(i) \subseteq W$, and $\pi(i) \xrightarrow{\alpha(i)} \pi(i+1)$ for each $i \geq 0$. If $\mathcal{R}$ is a *computable* rewrite theory that satisfies additional decidability conditions [16], given an initial state $[t]_E$, a set of state propositions Π, and a set of spatial action patterns W, we can construct the corresponding LKS $\mathcal{K}_{\Pi,W}(\mathcal{R})_t$ such that $\mathcal{R}, [t]_E \models \varphi$ if and only if $\mathcal{K}_{\Pi,W}(\mathcal{R})_t \models \varphi$ for any LTLR formula φ over Π and W [1]. Therefore, a formula φ has no counterexample on $\mathcal{R}$ from an initial state $[t]_E$ if and only if the product automaton $\mathcal{K}_{\Pi,W}(\mathcal{R})_t \times \mathcal{B}_{\neg\varphi}$ has no accepting path, which can be easily checked using the nested depth first search algorithm [10].

On the other hand, the model checking algorithm for a universally quantified LTLR formula $\forall \overline{x}\ \varphi$ on a rewrite theory $\mathcal{R}$ is nontrivial, since in general, such a variable quantification ranges over an infinite set, e.g., a set of tuples of ground terms having specified sorts in $\mathcal{R}$. We cannot directly use the corresponding LKS $\mathcal{K}_{\Pi,W}(\mathcal{R})_t$ for model checking such universally quantified LTLR formulas. However, the satisfaction relation for $\forall \overline{x}\ \varphi$ can be efficiently determined on a finite LKS satisfying *finite instantiation property* (FIP) [2].

Definition 3. *An LKS $\mathcal{K} = (S, S_0, \Pi, \mathcal{L}, W, T)$ satisfies a* finite instantiation property *(FIP) if and only if: (i) for each state $s \in S$, $\mathcal{L}(s)$ is finite, and (ii) for each transition $s \xrightarrow{A} s' \in T$, the set A is finite.*[3]

The *path-realized* set $\mathscr{R}_{(\pi,\alpha),\varphi}$ is a set of substitutions that is guaranteed to be finite if the underlying LKS $\mathcal{K}$ is *finite* and satisfies FIP (see [2] for a detailed definition). Only such a finite path-realized set is necessary to decide the satisfaction of a universally quantified formula $\forall \overline{x}\ \varphi$ on $\mathcal{K}$ as shown in the following lemma [2], in which $\mathcal{K}_\perp = (S, S_0, \Pi \cup \Pi_\perp, \mathcal{L}, W \cup W_\perp, T)$ is an extension of $\mathcal{K}$ such that $\mathcal{K}_\perp$ may have additional *void* state propositions $\Pi_\perp$ and spatial action patterns $W_\perp$ which are never satisfied on $\mathcal{K}_\perp$.

Lemma 1. *Given a finite LKS $\mathcal{K}$ satisfying FIP, a universally quantified LTLR formula $\forall \overline{x}\ \varphi$, and a path (π, α), for each ground substitution θ, there exists $\vartheta \in \mathscr{R}_{(\pi,\alpha),\varphi}$ such that $\mathcal{K}, (\pi, \alpha) \models \theta\varphi$ iff $\mathcal{K}_\perp, (\pi, \alpha) \models \vartheta\varphi$.*

For a finite LKS $\mathcal{K}$ satisfying FIP, we can have an efficient algorithm to model check an LTLR formula φ under fairness assumptions of the form $\forall \overline{y}\ \Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$ or $\forall \overline{y}\ \Box\Diamond\Phi \rightarrow \Box\Diamond\Psi$, where Φ and Ψ are boolean formulas with no temporal operators [2]. The model checking algorithm consists basically in finding a reachable strongly connected component (SCC) $\mathfrak{S}$ from initial states in the product automaton $\mathcal{K} \times \mathcal{B}_{\neg\varphi}$ such that: (i) $\mathfrak{S}$ satisfies an acceptance condition of $\mathcal{B}_{\neg\varphi}$, and (ii) all realized substitution instances of fairness formulas hold in $\mathfrak{S}$. The SCC $\mathfrak{S}$ satisfies the universally quantified fairness formulas, thanks to Lemma 1. Such an SCC $\mathfrak{S}$ exists in $\mathcal{K} \times \mathcal{B}_{\neg\varphi}$ if and only if there is a fair counterexample of φ in $\mathcal{K}$. The complexity of the algorithm is $O(k \cdot f \cdot |\mathcal{K}| \cdot 2^{|\varphi|})$, where k is the number of fairness formulas and f is the number of realized substitutions [2].

[3] More precisely, there are only finitely many ground instances for each parametric proposition. But if the signature is finite as usual, the definitions are equivalent.

3 The Maude LTLR Model Checker under Fairness

We have developed a new Maude LTLR model checker to support model checking under localized fairness specifications using the algorithm in [2]. This tool extends the previous LTLR model checker [1] and the existing LTL model checker [8] in Maude. The new LTLR model checker allows the user to specify localized fairness conditions for each rule in a system module, and provides a simple user interface to perform model checking under localized fairness assumptions. The earlier version of the LTLR model checker [1] did not support localized fairness assumptions, and before this work the model checking algorithm in [2] had not been integrated with a suitable *property specification language* in which such parameterized fairness assumptions could be naturally expressed. This section presents rewrite theories and LTLR as an ideal property specification language for a parameterized fairness assumptions with user-friendly tool support.

Throughout this section, we will use a simple fault-tolerant client-server communication model borrowed from [16] to illustrate the new LTLR model checker under localized fairness specifications. In this model, each client `C` sends a query `N` to a server `S` to receive an answer, and the server returns the answer `f(S,C,N)` of the query using a function `f`. The configuration of the system is a multiset of clients, servers, and messages, with the empty multiset `null`. A client is represented as a term `[C,S,N,W]` with `C` the client's name, `S` a server's name, `N` a number representing a query, and `W` either a number representing an answer or `nil` if the answer has not yet been received. A server is represented as a term `[S]` with the name `S`, and a message is represented as a term `I <- {J,N}` with `I` the receiver's name, `J` the sender's name, and `N` a number. The following rewriting rules define the behavior of the system:

```
rl [req]  : [C,S,N,nil]          =>  [C,S,N,nil] S <- {C,N} .
rl [reply]: S <- {C,N} [S]       =>  [S] C <- {S,f(S,C,N)} .
rl [rec]  : C <- {S,M} [C,S,N,W] =>  [C,S,N,M] .
rl [dupl] : I <- {C,N}           =>  I <- {C,N} I <- {C,N}  .
rl [loss] : I <- {C,N}           =>  null  .
```

This system has an infinite number of states, but we can apply the equational abstraction [17] to collapse the set of states into a finite set by adding the following abstraction equation and coherent completion rule as described in [16]:

```
eq I <- {C,N} I <- {C,N} = I <- {C,N} .
rl [reply]: S <- {C,N} [S]  =>  S <- {C,N} [S] C <- {S,f(S,C,N)} .
```

A liveness property we may wish to verify is the LTLR formula $\Diamond$`rec` with a basic action pattern `rec`, which means that some client will eventually receive an answer; however, $\Diamond$`rec` does *not* hold without fairness. The fairness assumptions needed to prove the formula $\Diamond$`rec` are: (i) weak fairness of the rule *req for each client* `C`, (ii) strong fairness of the rule *reply for each server* `S` *and client* `C`, and (iii) strong fairness of the rule *rec for each client* `C`. A general fairness specification of this system is nontrivial, because the number of fairness conditions depends on the number of servers and clients in initial configurations.

Furthermore, if the number of clients and servers can be changed during execution, the number of fairness conditions depends on the maximum number of clients and servers during execution. However, such fairness conditions can be naturally expressed as the localized fairness specification:

$$\mathcal{J} = \{\texttt{req(C)}\} \qquad \mathcal{F} = \{\texttt{reply(S,C)},\ \texttt{rec(C)}\}$$

no matter how many clients and servers make up the system. It is then easy to show that for any initial state *init* consisting of one or more servers, each with one or more clients connected to it and having `nil` in their fourth component, we have the desired satisfaction $\mathcal{R}, \mathit{init} \models_{\mathcal{J}\cup\mathcal{F}} \Diamond\texttt{rec}$.

3.1 Specification of Localized Fairness

A localized fairness specification $(\mathcal{J}, \mathcal{F})$ of a system module is given by a *metadata* attribute for each rule, which is a list of fairness items separated by the "`;`" symbol. Each fairness item for a rule $l : q \to r$ has one of the following forms:

$$\mathit{just}(x_1, \ldots, x_n) \qquad \mathit{fair}(x_1, \ldots, x_n) \qquad l(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n \in \mathit{vars}(q)$. A fairness item with no variables is expressed by *just*, *fair*, or l. A variable in the left side of a matching condition can also be used in a fairness item. Whenever a fairness item $\mathit{just}(x_1, \ldots, x_n)$ (resp., $\mathit{fair}(x_1, \ldots, x_n)$) is included in a metadata attribute of a rule with label l, the corresponding basic action pattern $l(x_1, \ldots, x_n)$ is included in the weak fairness specification $\mathcal{J}$ (resp., the strong fairness specification $\mathcal{F}$). A fairness item $l(x_1, \ldots, x_n)$ in a metadata rule attribute only declares the signature of the basic action pattern $l(x_1, \ldots, x_n)$, which may be used in model checking formulas,[4] but not in a localized fairness specification $(\mathcal{J}, \mathcal{F})$. Such a signature is required to parse LTLR formulas containing the basic action pattern in model checking commands.

Each fairness item in metadata attributes determines the signature of the corresponding basic action patterns, which is usually not a part of the original rewrite theory. The user needs to define the necessary basic action pattern signature before executing any model checking command under localized fairness assumptions. Although it is possible to automatically generate all the possible basic action patterns from a rewrite theory, it easily causes confusion and ambiguity on the meanings of different basic action patterns. For example, a rewrite rule $l : f(x_1, x_2) \to g(x_1)$ in which the variables x_1 and x_2 have the same sort S has two ambiguous basic action patterns $l(x_1)$ and $l(x_2)$ that cannot be distinguished by their syntax when applied to a concrete instance, e.g., $l(u)$ with some ground term u of sort S. In order to avoid such ambiguities, our tool takes the metadata rule attributes into account so that the user can specify the exact basic action patterns they intended to use for model checking purposes.

[4] $l(x_1, \ldots, x_n)$ gives us a very expressive syntax for basic action patterns, since any subset $\{x_1, \ldots, x_n\} \subseteq X$ contained in the set of variables X of the rule labeled l can then be used as a basic action pattern if $l(x_1, \ldots, x_n)$ has been declared this way.

In the following client-server communication example introduced above, the metadata attributes of the rules define the localized fairness specification ($\mathcal{J} = \{\texttt{req(C)}\}$, $\mathcal{F} = \{\texttt{reply(S,C)}, \texttt{rec(C)}\}$). The fairness item `rec` of the rule *rec* is written in order to declare the basic action pattern, so that the formula $\Diamond$`rec` can be used in the model checking command later. The metadata attributes define the signature of the basic action patterns `req(C)`, `reply(S,C)`, `rec`, and `rec(C)`, and their corresponding enabled propositions such as `enabled(req(C))`.

```
rl [req]  : [C,S,N,nil]          =>  [C,S,N,nil] S <- {C,N}
       [metadata "just(C)"] .
rl [reply]: S <- {C,N} [S]       =>  [S] C <- {S,f(S,C,N)}
       [metadata "fair(S,C)"] .
rl [rec]  : C <- {S,M} [C,S,N,W]  =>  [C,S,N,M]
       [metadata "rec; fair(C)"] .
rl [dupl] : I <- {C,N}           =>  I <- {C,N} I <- {C,N}  .
rl [loss] : I <- {C,N}           =>  null  .
```

Some ambiguous basic action patterns can still be mistakenly introduced, and should be manually resolved by the user. For example, the ambiguity between `req(C)` and `req(S)` could be removed by making `C` and `S` have different kinds.

3.2 The Fair LTLR Model Checker Interface

The interface of the new Maude LTLR model checker under localized fairness specifications is developed as an extension of Full Maude. Contrary to our previous LTLR model checker [1], in which each spatial action pattern should be manually defined in a similar way to the case of state propositions, the new interface automatically generates the necessary declarations for the basic action patterns from rule attributes (see Section 4.3). If the formulas to be verified only contain basic action patterns written in metadata attributes, then no additional declarations are required. As in [1], the tool also supports more general user-defined spatial action patterns (see Section 3.3), for example, a pattern $l(u_1, \ldots, u_n)$ with some of the u_i non-variable terms.

First of all, there is a command `pfmc` t `|=` φ for model checking an LTLR formula φ with an initial state t under a given localized fairness specification $(\mathcal{J}, \mathcal{F})$. For example, the following is the fair model checking result of the formula $\Diamond$`rec` for the client-server communication example:

```
Maude> (pfmc [a] [b,a,1,nil] [c,a,0,nil]  |=  <> rec .)
ltlr model check under localized fairness in CLIENT-SERVER-CHECK :
  [a][b,a,1,nil][c,a,0,nil] |= <> rec
result Bool :
  true
```

The other command `mc` t `|=` φ is a usual LTLR model checking command without localized fairness. However, unlike the previous Maude LTLR model checker, basic action patterns are automatically declared for input formulas if the patterns were declared in the rule attribute. For example, the following `mc` command returns a counterexample, where the server a keeps replying to the client b, but the client b receives no message because rec(b) does not satisfy the fairness assumptions (parts of the counterexample are replaced by . . .):

```
Maude> (mc [a] [b,a,1,nil] [c,a,0,nil]  |=  <> rec .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a][b,a,1,nil][c,a,0,nil] |= <> rec
result ModelCheckResult :
  counterexample(
   {[a][b,a,1,nil][c,a,0,nil], {'req : 'C \ b ; 'S \ a}}
   {[a](a <-b,1)[b,a,1,nil][c,a,0,nil], {'reply : 'C \ b ; 'S \ a}}
   ...,
   {[a](a <-{b,1})(a <-{c,0})(b <-{a,f(a,b,1)})[b,a,1,nil][c,a,0,nil],
     {'reply : 'C \ b ; 'S \ a}})
```

A counterexample of an LTLR formula consists of a finite prefix and an infinite cycle in which each item is a pair of a state and a simplified one-step proof term.

Furthermore, each model checking command allows the user to specify additional *ground fairness conditions*, which can be used when some fairness conditions cannot be expressed by a localized fairness specification.[5] A ground fairness specification is a finite set of ground weak fairness (just : Φ => Ψ) and ground strong fairness (fair : Φ => Ψ), where just : Φ => Ψ (resp., fair : Φ => Ψ) is a shorthand for a fairness formula $\Diamond\Box\Phi \to \Box\Diamond\Psi$ (resp., $\Box\Diamond\Phi \to \Box\Diamond\Psi$). In this case, the formulas Φ and Ψ can be any boolean formulas involving state propositions and spatial action patterns. The following model checking result is for the same example with enough ground fairness conditions to prove $\Diamond$rec:

```
Maude> (mc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec under
          (just : enabled(req(b)) => req(b)) ;
          (fair : enabled(rec(b)) => rec(b)) ;
          (fair : enabled(reply(a,b)) => reply(a,b)) .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a][b,a,1,nil][c,a,0,nil] |= <> rec
under fairness :
  (just : enabled(req(b))=> req(b));
  (fair : enabled(rec(b))=> rec(b));
   fair : enabled(reply(a,b))=> reply(a,b)
result Bool :
  true
```

In contrast, the model checking command with only ground weak fairness conditions gives the following counterexample in which no client receives any message since all of them are taken away by the loss rule:

[5] For example, we may have objects a, b, c, d, and e, but we may only want to specify fairness requirements for a, c, and e, but not for b and d. Or we may have fairness requirements $\Diamond\Box\Phi \to \Box\Diamond\Psi$ or $\Box\Diamond\Phi \to \Box\Diamond\Psi$ where the formulas Φ and Ψ do not correspond to the fairness requirements for a rule application.

```
Maude> (mc [a] [b,a,1,nil] [c,a,0,nil] |= <> rec under
          (just : enabled(req(b)) => req(b));
          (just : enabled(req(c)) => req(c));
          (just : enabled(reply(a,b)) => reply(a,b));
          (just : enabled(reply(a,c)) => reply(a,c));
          (just : enabled(rec(b)) => rec(b));
           just : enabled(rec(c)) => rec(c) .)
ltlr model check in CLIENT-SERVER-CHECK :
  [a][b,a,1,nil][c,a,0,nil] |= <> rec
under fairness :
  (just : enabled(req(b))=> req(b)); (just : enabled(req(c))=> req(c));
  (just : enabled(reply(a,b))=> reply(a,b));
  (just : enabled(reply(a,c))=> reply(a,c));
  (just : enabled(rec(b))=> rec(b)); just : enabled(rec(c))=> rec(c)
result ModelCheckResult :
  counterexample(nil,
{[a][b,a,1,nil][c,a,0,nil], {'req : 'C \ b ; 'S \ a}}
{[a](a <-{b,1})[b,a,1,nil][c,a,0,nil], {'reply : 'C \ b ; 'S \ a}}
{[a](b <-{a,f(a,b,1)})[b,a,1,nil][c,a,0,nil], {'req : 'C \ c ; 'S \ a}}
{[a](a <-{c,0})(b <-{a,f(a,b,1)})[b,a,1,nil][c,a,0,nil], {'loss : 'I \ a}}
{[a](b <-{a,f(a,b,1)})[b,a,1,nil][c,a,0,nil], {'loss : 'I \ b}})
```

Note that, since all the objects in the initial state have been given weak fairness requirements corresponding to rule applications, we can simplify the above complex model checking command using the `pfmc` command and metadata attributes in the style shown above.

3.3 More General Spatial Action Patterns

The Maude LTLR model checker under localized fairness provides capabilities for the user to define spatial action patterns, in a way similar to the equational definition of state propositions, as well as basic action patterns. Recall that the syntax of a spatial action pattern is defined by a parametric function symbol of sort `Action`, and the satisfaction relation of a spatial action pattern is given by equations using the auxiliary operator $\_ \models \_$: `ProofTerm Action` $\rightarrow$ `Bool` involving one-step proof terms and spatial action patterns. As a matter of fact, the syntax and the semantics of the basic action patterns given in metadata attributes are also defined in the exact same way, but such definitions are automatically generated by the tool.

The basic signature for model checking is specified in the system module `LTLR-MODEL-CHECKER`, which is inherited from the earlier version of the LTLR model checker [1] but has been extended to support localized fairness specifications. First, sort `BasicActionPattern` for basic action patterns is introduced as a subsort of a spatial action pattern sort `Action`. A one-step proof term $[t[l(\theta)]]_E$ is represented as a triple of a context term $t[\square]$ that has a hole `[]` inside, a rule label l, and a substitution θ as an assignment set of the form $\mathtt{x}_1\backslash\mathtt{u}_1\mathtt{;}\ldots\mathtt{;x}_n\backslash\mathtt{u}_n$, enclosed by the triple operator:

```
op {_|_:_} : StateContext RuleName Substitution -> ProofTerm [ctor ...] .
```

Each spatial action pattern is then declared using the above constructs. For instance, a basic action pattern $l(x_1, \dots, x_n)$ can be defined by:

```
op $l$ : $S_1$ ... $S_n$ -> BasicActionPattern [ctor] .
eq {CONTEXT | '$l$ : '$x_1$\$x_1$ ; ... ; '$x_n$\$x_n$ ; SUBST} |= $l$($x_1$,...,$x_n$) = true .
```

where $'l$, $'x_1$, ..., $'x_n$ are quoted identifier constants of sort `Qid`, which are used for explicitly expressing variable names.

This mechanism enables the user to define much more general form of spatial action patterns. The following declarations show some predefined spatial action patterns in the module `LTLR-MODEL-CHECKER` whose satisfaction depends on the rewriting positions [1, 16] in addition to the rule labels and the substitutions:

```
op top : BasicActionPattern -> ActionPattern .
op {_|_} : StateContext BasicActionPattern -> ActionPattern .

var C : StateContext . var S : Substitution . var BSP : BasicActionPattern .

eq {[] | R:RuleName : S} |= top(BSP)  = {[] | R:RuleName : S} |= BSP .
eq {C  | R:RuleName : S} |= {C | BSP} = {C  | R:RuleName : S} |= BSP .
```

As defined by the above satisfaction equations, a ground spatial action pattern $top(l(u_1, \dots, u_n))$ holds on a one-step rewrite that happens at the top position and satisfies the basic action pattern $l(u_1, \dots, u_n)$. Similarly, a ground spatial action pattern $\{t[\Box] \mid l(u_1, \dots, u_n)\}$ holds on a one-step rewrite that happens at the position represented by the context term $t[\Box]$ and satisfies the basic action pattern $l(u_1, \dots, u_n)$. Such spatial action patterns with context terms are meaningful only if the signature of context terms is given [1]. In the new Full Maude interface, the signature of context terms can be generated by the module expression `CONTEXT[M]` from a system module `M`.

4 The Maude LTLR Model Checker Implementation

The Maude LTLR model checker has been implemented at both the Core Maude and Full Maude levels for the sake of gaining efficiency while keeping expressiveness and user-friendliness. The new LTLR model checker under localized fairness consists of three components: (i) the graph traversal engine that constructs the corresponding LKS from a rewrite theory, (ii) the model checking algorithms under parameterized fairness assumptions using the LKS, and (iii) the user interface of the model checker. For efficiency reasons, the first and second components are implemented at the C++ level within the Maude system. In particular, the model checking algorithm under parameterized fairness requires that the underlying LKS satisfies FIP. But for basic action patterns, the corresponding LKS of a rewrite theory satisfies FIP for free as we show below. Finally, the user interface of the model checker has been implemented by extending Full Maude, since it involves several theory transformations that automate the user interface.

4.1 Localized Fair Model Checking of Rewrite Theories

Basically, model checking an LTLR formula φ under a localized fairness specification $(\mathcal{J}, \mathcal{F})$ is to find an $\mathcal{J}, \mathcal{F}$-fair counterexample that satisfies $\neg\varphi$. By definition, given a rewrite theory $\mathcal{R}$ and an LTLR formula φ, a $\mathcal{J}, \mathcal{F}$-fair counterexample (π, γ) invalidating φ should satisfy:

- $\mathcal{R}, (\pi, \gamma) \models \neg\varphi$,
- for each $l_j(\overline{y}_j) \in \mathcal{J}$, $\mathcal{R}, (\pi, \gamma) \models \forall \overline{y}_j \ \Diamond\Box \mathit{enabled}(l_j(\overline{y}_j)) \rightarrow \Box\Diamond l_j(\overline{y}_j)$, and
- for each $l_f(\overline{y}_f) \in \mathcal{F}$, $\mathcal{R}, (\pi, \gamma) \models \forall \overline{y}_f \ \Box\Diamond \mathit{enabled}(l_f(\overline{y}_f)) \rightarrow \Box\Diamond l_f(\overline{y}_f)$.

In order to apply the model checking algorithm for parameterized fairness [2] to find such a counterexample, the corresponding LKS $\mathcal{K}$ of $\mathcal{R}$ should be finite and satisfy FIP. Given a computable rewrite theory $\mathcal{R}$ with a finite number of reachable states from an initial state $[t]_E$, we can construct the *finite* LKS $\mathcal{K}_{\Pi,W}(\mathcal{R})_t$ with respect to state propositions Π and spatial action patterns W.

Definition 4. *A rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *is* finite-state *if and only if* E *and* R *are finite, and for each initial state* $[t_0]_E \in T_{\Sigma/E,k}$*, the set of reachable states* $\mathit{Reach}_{\mathcal{R}}([t_0]_E) = \{[t^*]_E \in T_{\Sigma/E,k} \mid [t_0]_E \rightarrow^*_{\mathcal{R}} [t^*]_E\}$ *is always finite.*

The only remaining requirement is that the LKS $\mathcal{K}$ satisfies FIP with respect to the state propositions and the spatial action patterns appearing in $(\mathcal{J}, \mathcal{F})$, which have the form of either $\mathit{enabled}(l(\overline{y}))$ or $l(\overline{y})$. By definition, each ground instance $\theta(\mathit{enabled}(l(\overline{y})))$ is satisfied on a state $[t]_E$ if and only if there exists a one-step rewrite $[t[l(\theta)]]_E$. Since a finite-state rewrite theory has only finitely many one-step rewrites for each state, each state $[t]_E$ of $\mathcal{R}$ satisfies only finitely many ground instances of $\mathit{enabled}(l(\overline{y}))$. Similarly, since each ground instance $[\theta(l(\overline{y}))]_E$ is satisfied on a one-step proof term $[t[l(\theta)]]_E$, each one-step rewrite of $\mathcal{R}$ satisfies only one ground instance of $l(\overline{y})$. Therefore, the associated LKS of a finite-state rewrite theory $\mathcal{R}$ always satisfies FIP with respect to a localized fairness specification $(\mathcal{J}, \mathcal{F})$.

4.2 The Model Checking Algorithm Implementation

The new LTLR model checker under localized fairness implements the parametric generalized fairness algorithm [2] in C++ on top of the previous LTLR model checking algorithm, which constructs state/event-based product automaton between a system LKS and a formula Büchi automaton [1]. For generating a Büchi automaton, it reuses the existing LTL model checker implementation [8]. Besides dealing with fairness, the new model checker also generates shorter counterexamples than the previous model checkers in Maude. When a counterexample is found, we perform a breadth-first search from loop states to the initial states using only *already visited* states to find the shortest prefix in the explored state space. The performance of the new Maude LTLR model checker is comparable to other explicit-state model checkers such as SPIN [11] and PAT [18] as shown in [2], and it is currently the only tool we know supporting *parameterized* fairness.

Different model checking algorithms are used by the tool for handling different cases of input fairness, because the general algorithms are more computationally expensive. For usual LTLR model checking with no fairness requirements, we use the nested depth first search algorithm of [10] as the previous LTLR model checker. If only weak fairness conditions are specified, we use the SCC-based algorithm [5] for generalized Büchi automata, in which weak fairness conditions are directly incorporated as an acceptance condition. In the case of strong fairness conditions, the Streett automata emptiness checking algorithm is employed as explained in [2]. If some of the fairness conditions are given by a localized fairness specification, such a fairness model checking algorithm is combined with the parametric fairness algorithm that computes realized substitutions.

4.3 Theory Extension for Localized Fairness

In order to simplify the user interface, the model checker uses theory transformations to automatically generate each basic action pattern $l(\overline{y})$ and its corresponding state proposition $\mathit{enabled}(l(\overline{y}))$ in the metadata rule attribute. Such theory transformations are incorporated into the Maude LTLR model checker as part of the model checking interface extending Full-Maude.

Given a system module `M`, the module expression `ACTION[M]` builds a module that contains a signature for the basic action patterns $\mathfrak{B} = \{l_1(\overline{y}_1), \ldots, l_n(\overline{y}_n)\}$ given by the metadata attributes of the rules in `M`. For each basic action pattern $l(x_1, \ldots, x_n)$ in $\mathfrak{B}$, where each variable x_i has sort S_i in the corresponding rule in `M`, the module `ACTION[M]` includes the following operators and equations:

- a constructor for the basic action pattern $l(x_1, \ldots, x_n)$

  `op` l `:` S_1 `...` S_n `-> BasicActionPattern [ctor] .`

- assignment operators for each sort S_i of the variable x_i in $l(x_1, \ldots, x_n)$

  `op _\_ : Qid` S_i `-> Assignment [ctor ...] .`

- an equation to define the satisfaction relation with respect to proof terms

  `eq {C | '`l `: '`x_1`\`x_1 `; ... ; '`x_n`\`x_n `; SUBST} |=` $l(x_1,\ldots,x_n)$ `= true .`

Together with the theory transformation `CONTEXT[_]` to generate a context signature, the theory transformation `ACTION[_]` replaces a previous theory transformation [1] that was defined in the old version of the LTLR model checker to generate a signature for context terms and assignment operators.

Next, the module expression `FAIR[M]` creates a declaration of the state proposition $\mathit{enabled}(l(\overline{y}))$ for each $l(\overline{y}) \in \mathfrak{B}$. Basically, such *enabled* propositions are defined by operators `_enables_:` K `Action -> Bool` for each relevant kind K, where $E \vdash t\ \mathit{enables}\ \delta = \mathit{true}$ means that the one-step rewrite associated to the one-step proof term δ can happen *inside* the term t.

```
ceq S:State |= enabled(BSP) = true if S:State enables BSP .
```

For each $l(x_1, \ldots, x_n) \in \mathfrak{B}$ and its associated rule $l : t \to t'\ \mathit{if}\ \mathit{cond}$, where the kind of t is K_l, the basic declarations of *enables* operators are given as follows:

```
 op _enables_ : $K_l$ Action -> Bool .
ceq $t$ enables $l(x_1,\ldots,x_n)$ = true if $cond$ .
```

For each free constructor operator $f : K_1 \ldots K_n \to K$, where *enables* operators are defined for a nonempty set of kinds $\{K_{i_1},\ldots,K_{i_k}\} \subseteq \{K_1,\ldots,K_n\}$, the following declarations of *enables* operators are given in `FAIR[M]`:

```
 op _enables_ : $K$ Action -> Bool .
ceq f($X_1$:$K_1$,...,$X_n$:$K_n$) enables BSP
 if $X_{i_1}$ enables BSP  or-else  ...  or-else  $X_{i_k}$ enables BSP .
```

Finally, for each associative constructor operator $g : K\ K \to K$, and for each equation `ceq` t `enables` $\delta =$ `true` *if cond* for a kind K, the extended declarations of *enables* operators are given in `FAIR[M]` as follows, where $X_1 : K$ and $X_2 : K$ are fresh variables not appearing in the original equation:

```
 op _enables_ : $K$ Action -> Bool .
ceq g($X_1$:$K$,$t$,$X_2$:$K$) enables $\delta$ = true if $cond$ .
ceq g($X_1$:$K$,$t$) enables $\delta$ = true if $cond$ .
ceq g($t$,$X_2$:$K$) enables $\delta$ = true if $cond$ .
```

If the associative constructor satisfies another axiom such as commutative or identity, only some of the above equations will be required. Note that the above extended declarations are essentially the generalization of equations for extension matching modulo equational axioms [4].

The *enables* declarations for free and associative constructors can be computed iteratively until reaching a fixed point. Since the right side of each *enables* equation is `true`, we can easily prove the following proposition by induction on the height of the conditional proof tree.

Proposition 1. *Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, \mathcal{R})$ with signature of constructors Ω that are free modulo the axiom of B,*[6] *for a basic action pattern $l(\overline{y})$ of $\mathcal{R}$, a term $[t]_E$, and a substitution ϑ, if $E \vdash t$ enables $\vartheta(l(\overline{y})) = true$, then there exists a one-step rewrite from $[t]_E$ with a one-step proof term λ such that $E \vdash (\lambda \models l(\vartheta\overline{y})) = true$.*

5 An Example

This section illustrates a rewriting logic specification of the Evolving Dining Philosophers problem [13] with a localized fairness specification. This problem is similar to the famous Dining Philosophers problem, but a philosopher can join or leave the table, so that the number of philosophers can change dynamically. In this example, it is very hard to specify exact fairness conditions even if an initial state of the system is already given, because we cannot know how many philosophers can be in the model without exploring the entire state space, but the fairness conditions of this system depend on *each* philosopher in the system.

[6] That is, $T_{\Sigma/E\cup B}|_{\Omega} \simeq T_{\Omega/B}$.

Each philosopher is represented by a term `ph(I, S, C)`, where `I` is the philosopher's id, `S` is the philosopher's status, and `C` is the number of chopsticks held. Likewise, a chopstick with id `I` is represented by a term `stk(I)`. The configuration of the system is described by a *set* of philosophers and chopsticks, built by an associative-commutative set union operator `_;_`. The signature is defined in the Maude language as follows:

```
sorts Philo Status Chopstick Conf .    op ph : Nat Status Nat -> Philo .
subsort Philo Chopstick < Conf .       ops think hungry : -> Status .
op none : -> Conf .                    op stk : Nat -> Chopstick .
op _;_ : Conf Conf -> Conf [comm assoc id: none] .
```

The state is a triple `< P, N, CF >` with sort `Top`, where `P` is a global counter, `N` is the number of philosophers, and `CF` is a *set* of philosophers and chopsticks. The behavior of philosophers is then described by the following rewrite rules with a necessary localized fairness specification:

```
 rl [wake] : ph(I, think, 0) => ph(I, hungry, 0)
     [metadata "just(I)"] .
crl [grab] : < P, N, ph(I, hungry, C) ; stk(J) ; CF >
          => < P, N, ph(I, hungry, C + 1) ; CF >
  if J == left(I) or J == right(I, N)
     [metadata "fair(I)"] .
 rl [stop] : < P, N, ph(I, hungry, 2) ; CF >
          => < P, N, ph(I, think, 0) ; stk(left(I)) ; stk(right(I, N)) ; CF > .
```

The functions `left(I) = I` and `right(I,N) = (I + 1) rem N` return the chopstick's id on the left (resp., right) of philosopher `I`, where `_rem_: Nat Nat -> Nat` is the reminder operator.

We now specify the dynamic behavior of philosophers in the Evolving Dining Philosopher problem. Although there is no limit to the number of philosophers in the original problem, we can give an unpredictable bound using the Collatz conjecture [6]. There is a global counter `P` that symbolizes a philosophical problem, and philosophers keep thinking the problem by changing the number n to: (i) $3n + 1$ for n odd, or (ii) $n/2$ for n even.

```
crl [solve]: < P, N, ph(I, think, 0) ; CF >  =>  < Q, N, ph(I, think, 0) ; CF >
 if P > 1 /\ Q := collatz(P) .
```

New philosophers can join the group only if the global number is a multiple of the current number of philosophers. No more philosophers can join after the number eventually goes to 1. We assume that only the last philosopher can leave the group for simplicity. To keep consistency, whenever a philosopher joins or leaves the table, the related chopsticks should not be held by another philosopher.

```
crl [join] : < P, N, ph(N, think, 0) ; CF >
          => < P, N + 1, ph(N, think, 0) ; ph(N + 1, think, 0) ; stk(N + 1) ; CF >
 if P rem N == 0 .
crl [leave]: < P, N, CF ; ph(N, think, 0) ; stk(N) >  =>  < P, N - 1, CF >
 if N > 2 .
```

In order to perform model checking, we define the following module after loading the LTLR model checker interface in Full Maude.

```
(mod PHILO-CHECK is
  including PHILO .
  including LTLR-MODEL-CHECKER .

  subsort Top < State .
  op eating : Nat -> Prop [ctor] .
  op init : -> State .

  vars P N : Nat .  var I : NzNat . var CF : Conf .
  eq < P, N, ph(I, hungry, 2) ; CF > |= eating(I) = true .
  eq init = < 12, 2, ph(1,think,0); stk(1); ph(2,think,0); stk(2) > .
endm)
```

The state proposition `eating(I)` is satisfied if the philosopher `I` is eating. The initial state is the case of 2 philosophers with the global counter 12, expressed by `< 12, 2, ph(1, think, 0); stk(1); ph(2, think, 0); stk(2) >`.

We are interested in verifying the liveness property `[]~deadlock -> <> eating(1)`, where `deadlock` is a spatial action pattern satisfied on deadlock states [1]. Without fairness assumptions, the model checker generates the following counterexample for this formula, in which only the philosopher 2 performs actions while the order philosophers keep idle and no new philosopher joins:

```
Maude> (mc init |=  [] ~ deadlock -> <> eating(1) .)
ltlr model check in PHILO-CHECK :
  init |= []~ deadlock -> <> eating(1)
result ModelCheckResult :
  counterexample(
{< 12,2,stk(1); stk(2); ph(1,think,0); ph(2,think,0)>, {'solve : 'I \ 1}}
{< 6,2,stk(1); stk(2); ph(1,think,0); ph(2,think,0)>, {'solve : 'I \ 1}}
 ...,
{< 1,2,stk(1); stk(2); ph(1,hungry,0); ph(2,think,0)>, {'wake : 'I \ 2}}
{< 1,2,stk(1); stk(2); ph(1,hungry,0); ph(2,hungry,0)>,
   {'grab : 'I \ 2 ; 'J \ 1}}
{< 1,2,stk(2); ph(1,hungry,0); ph(2,hungry,1)>, {'grab : 'I \ 2 ; 'J \ 2}}
{< 1,2,ph(1,hungry,0); ph(2,hungry,2)>, {'stop : 'I \ 2} })
```

When we assume localized fairness conditions, the model checker can verify the formula `[]~deadlock -> <> eating(1)` as follows:

```
Maude> (pfmc init |=  [] ~ deadlock -> <> eating(1) .)
ltlr model check under localized fairness in PHILO-CHECK :
  init |= []~ deadlock -> <> eating(1)
result Bool :
  true
```

Note that the reachable state space from the initial state has 12 ground fairness conditions instantiated by realized substitutions. The previous LTL and LTLR model checkers cannot verify the formula with those 12 fairness conditions in a reasonable time. Furthermore, using the previous model checker, we could not know how many ground fairness conditions would be required to prove the formula before exploring the entire state space .

6 Related Work and Conclusions

The usual model checking method to verify a property φ under parameterized fairness assumptions, is to construct the conjunction of corresponding instances of fairness, and to apply either: (i) a standard LTL model checking algorithm for the reformulated property $\mathit{fair} \to \varphi$, or (ii) a specialized model checking algorithm which handles fairness, based on either explicit graph search [7, 9, 14], or a symbolic algorithm [12]. Approach (i) is inadequate for fairness, since the time complexity is exponential in the number of strong fairness conditions, while the other is linear. Furthermore, compiling such a formula, expressing a conjunction of fairness conditions, into Büchi automata is usually not feasible in reasonable time [19]. There are several tools to support the specialized algorithms such as PAT [18] and Maria [14]. Our tool is related to the second approach to deal with fairness, but it does not require pre-translation of parameterized fairness, and can handle *dynamic* fairness instances.

In conclusion, we have addressed the real need of verifying temporal logic properties under parametric fairness assumptions. Such parametric assumptions occur very often in practice, but up to now have not been supported by existing model checking techniques and tools. To address this need three things are required: (i) expressive system specification languages; (ii) expressive temporal logics; and (iii) novel model checking techniques and tools. This paper has argued and demonstrated with examples that rewriting logic answers very well need (i) and that TLR, and in particular LTLR, are very expressive to deal with need (ii). It has also presented a novel Maude LTLR model checker under localized fairness which directly addresses need (iii) in an efficient way.

Acknowledgments. This work has been supported in part by NSF Grants CNS 09-04749 and CCF 09-05584, AFOSR Grant FA8750-11-2-0084, Boeing Grant C8088-557395, and the "Programa de Apoyo a la Investigación y Desarrollo" (PAID-02-11) of the Universitat Politècnica de València.

References

1. Bae, K., Meseguer, J.: The Linear Temporal Logic of Rewriting Maude Model Checker. In: WRLA. LNCS, vol. 6381, pp. 208–225. Springer (2010)
2. Bae, K., Meseguer, J.: State/event-based LTL model checking under parametric generalized fairness. In: Computer Aided Verification. LNCS, vol. 6806, pp. 132–148. Springer (2011)
3. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Proc. 4th Intl. Conf. on Integrated Formal Methods (IFM'04). LNCS, vol. 2999, pp. 128–147. Springer (2004)

4. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
5. Couvreur, J., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. Model Checking Software pp. 169–184 (2005)
6. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems 19, 253–291 (1997)
7. Duret-Lutz, A., Poitrenaud, D., Couvreur, J.M.: On-the-fly emptiness check of transition-based Streett automata. In: ATVA. LNCS, vol. 5799. Springer (2009)
8. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: Model Checking Software: Proc. 10th Intl. SPIN Workshop. LNCS, vol. 2648, pp. 230–234. Springer (2003)
9. Henzinger, M.R., Telle, J.A.: Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In: SWAT. LNCS, vol. 1097, pp. 16–27. Springer (1996)
10. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search (extended abstract). In: The Spin Verification System. pp. 23–32. American Mathematical Society (1996)
11. Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison Wesley Publishing Company (2004)
12. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods in System Design 28(1), 57–84 (2006)
13. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. Software Engineering, IEEE Transactions on 16(11), 1293–1306 (2002)
14. Latvala, T.: Model checking LTL properties of high-level Petri nets with fairness constraints. In: ICATPN. LNCS, vol. 2075, pp. 242–262. Springer (2001)
15. Meseguer, J.: Localized fairness: A rewriting semantics. In: RTA. LNCS, vol. 3467, pp. 250–263. Springer (2005)
16. Meseguer, J.: The temporal logic of rewriting: A gentle introduction. In: Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 354–382. Springer (2008)
17. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403(2-3), 239–264 (2008)
18. Sun, J., Liu, Y., Dong, J., Pang, J.: PAT: Towards flexible verification under fairness. In: Computer Aided Verification. pp. 709–714. Springer (2009)
19. Vardi, M.: Automata-theoretic model checking revisited. In: Verification, Model Checking, and Abstract Interpretation. pp. 137–150. Springer (2007)