Notes on Model Checking and Abstraction in Rewriting Logic*

José Meseguer Miguel Palomino Narciso Martí-Oliet

Contents

1 Introduction **2**

2 The Underlying Framework **3**
2.1 Temporal Logic 3
2.2 Simulation Maps 4
2.3 Rewriting Logic 6

3 Intuitions **6**
3.1 Readers and Writers 6
3.2 A Mutual Exclusion Protocol 9

4 LTL Properties of Rewrite Theories and Model Checking **11**

5 Calculating Minimal Structures **12**
5.1 Example: The Bakery Protocol 13

6 Minimal Structures as Quotients **17**
6.1 Example 1: A Communication Protocol 21
6.2 Example 2: The Alternating Bit Protocol 24
6.3 Example 1 Revisited 29
6.4 Readers and Writers Revisited 30
6.5 The Bakery Protocol Revisited 31

7 The Deadlock Difficulty **34**

8 All Together: The Bounded Retransmission Protocol **36**

9 Model Checking with Fairness **41**
9.1 Example: Mutual Exclusion by Semaphores 42

*DRAFT.

1 Introduction

Model checking [7] is a popular method for the verification of hardware and software systems that was introduced in [4, 29]. Its two more salient characteristics are the fact that it is fully automated and that, in case the property under consideration is actually disproved, it comes with a concrete counterexample that can suggest ways in which the specification can be modified so that it meets the property, and also help in a better understanding of the problem. Its main drawback, on the other hand, is its inability to handle infinite state systems, or, more practically, finite systems big enough.

One approach for verifying big systems is *abstraction* [6, 22]. The idea is to consider a finite system which abstracts those characteristics of the infinite one we are interested in, to translate the property we want to prove to this new system, and to apply model checking to it. The abstract system should be fine enough so that the property actually holds in it, but at the same time it must be of restricted size to be tractable by model checking.

In [6], the approximation of a system M by another M' through a function h is defined, and an optimal approximation $M^h_{\min}$ is identified. However, the labelling function associated to Kripke structures is not taken into account. This optimal approximation, though, is discarded in favor of less precise approximations because calculating it is "computationally expensive." Actually, what happens is that, in general, $M^h_{\min}$ is not computable. An important drawback of their presentation is that, instead of starting with a concrete property ψ to be proved for the concrete system, and then abstracting it into an appropriate property for the abstract system, they start with an abstract property and show how to translate (concretize) it. This situation is remedied in [7, 5], either by considering abstraction functions that preserve the atomic propositions, or by restricting the formulas for which the result can be proved. In [18, 19] the authors directly work with $M^h_{\min}$ and, given a concrete property ψ, they translate it into an abstract property $\alpha(\psi)$ such that, if it is proved for the abstract system, then ψ must hold for the concrete one. However, no method for actually constructing $M^h_{\min}$ is given. The example they consider is the bakery protocol, whose optimal abstraction is "straightforward to compute." And, in addition, the abstract formula $\alpha(\psi)$ contains quantified variables that range over sets in the concrete model, which seems not to be amenable to model checking. In none of these works [6, 5, 19, 18] theorem proving is considered.

In [30], a different approach, *predicate abstraction*, is presented. The semantic relation $M \models \psi$ is replaced by a syntactic one by embedding the model M in the formula ψ as a binary relation. As opposed to the previous approaches, the logic used here is the μ-calculus. The abstraction substitutes predicates φ_i for boolean variables B_i, and the concretization function γ replaces these variables in an abstract predicate by the corresponding φ_i. The abstraction $[P]$ of a predicate P is defined as the conjunction of all expressions b satisfying $P \rightarrow \gamma(b)$: theorem proving is used here. Then it is proved that $\vdash [P]$ implies $\vdash P$.

In the methodology of abstraction two different parts can be distinguished: (1) definition of the function giving the abstraction; (2) extraction of an abstract

model out of the concrete one and the abstraction function. For the second point there is, as it has been already noted, an optimal solution. These notes deal with how to obtain and represent it in the framework of rewriting logic, and present some case studies. In addition, we also treat explicitly the problems that deadlocks can raise, and show how to impose fairness constraints in the Maude model checker.

2 The Underlying Framework

2.1 Temporal Logic

To specify the properties of interest about our systems we will use *linear temporal logic* (LTL), which is interpreted in a standard way in Kripke structures. In what follows, we assume a fixed non-empty set of atomic propositions AP.

Definition 1 *A* Kripke structure *is a triple $M = (S, \rightarrow, L)$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow \mathcal{P}(AP)$ is a labelling function associating to each state the set of atomic predicates that hold in it.*

Note that the transition relation must be *total*, and that we will usually employ the notation $a \rightarrow b$ to say that $(a, b) \in \rightarrow$. A *path* in a Kripke structure M is a function $\pi : \mathbb{N} \longrightarrow S$ such that, for every $i \in \mathbb{N}$, $\pi(i) \rightarrow \pi(i+1)$. We use π^i to refer to the suffix of π starting in $\pi(i)$; explicitly, $\pi^i(n) = \pi(i+n)$.

The syntax of LTL, or LTL(AP) if we want to make explicit the set of atomic propositions, is given by the following grammar:

$$\varphi = p \in AP \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \,.$$

The semantics of the logic is defined by structural induction. Given a Kripke structure $M = (S, \rightarrow, L)$, and an element $a \in S$,

$$M, a \models \varphi \iff M, a, \pi \models \varphi \quad \text{for all paths } \pi \text{ such that } \pi(0) = a \,,$$

where the satisfaction relation $M, a, \pi \models \varphi$ is defined as

$$\begin{array}{l}
M, a, \pi \models p \iff p \in L(a) \\
M, a, \pi \models \varphi \vee \psi \iff M, a, \pi \models \varphi \text{ or } M, a, \pi \models \psi \\
M, a, \pi \models \neg\varphi \iff M, a, \pi \not\models \varphi \\
M, a, \pi \models \bigcirc \varphi \iff M, \pi(1), \pi^1 \models \varphi \\
M, a, \pi \models \varphi \,\mathcal{U}\, \psi \iff \text{there exists } n \in \mathbb{N} \text{ such that } M, \pi(n), \pi^n \models \psi \text{ and,} \\
\qquad\qquad\qquad\qquad\quad \text{for all } m < n,\ M, \pi(m), \pi^m \models \varphi
\end{array}$$

Other boolean and temporal operators can be defined as syntactic sugar. The most common ones are: true: $\top = p \vee \neg p$, for $p \in AP$; false: $\bot = \neg\top$; conjunction: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$; implication: $\varphi \rightarrow \psi = \neg\varphi \vee \psi$; eventually: $\Diamond \varphi = \top \,\mathcal{U}\, \varphi$; henceforth: $\Box \varphi = \neg \Diamond \neg\varphi$.

We will be interested in a restriction of LTL without negation that will be denoted by LTL^-. In this logic, since we can no longer define the previous extra operators by syntactic sugar, we will consider all of them, except for $\neg$ and $\rightarrow$ (which could be used together with $\bot$ to define $\neg$), to be basic ones. From a practical point of view, this restriction will not limit the expressiveness of our logic at all since, by using duality, all negations occurring in a formula in LTL can always be pushed inside it, so that they only apply to atomic propositions, and these negated atoms can be replaced by fresh ones to give a formula in LTL^- as a result.

2.2 Simulation Maps

Definition 2 *Let $M = (S_M, \rightarrow_M, L_M)$ and $N = (S_N, \rightarrow_N, L_N)$ be Kripke structures. A* simulation map *$(h, \Gamma) : M \longrightarrow N$ consists of a function $h : S_M \longrightarrow S_N$ mapping states in M to states in N, and a set of atomic propositions $\Gamma \subseteq AP$ such that: (1) whenever $a \rightarrow_M b$, then $h(a) \rightarrow_N h(b)$, and (2) for all $a \in S_N$, $\Gamma \cap L_N(a) \subseteq \Gamma \cap \bigcap_{h(x)=a} L_M(x)$.*

In the last condition, we use the convention that $\bigcap_{x \in \emptyset} L_M(x) = AP$. The intuition behind requirement (1) is that the target system N must be able to simulate every possible run in M; condition (2) is imposed to avoid that a state in N satisfies a proposition that does not hold in all the states it simulates.

It is easy to check that the composition of simulation maps $(f, \Gamma) : M \longrightarrow N$ and $(g, \Delta) : N \longrightarrow P$ as $(g \circ f, \Gamma \cap \Delta) : M \longrightarrow P$ is well-defined, associative, and with identities given by $(1, AP)$. Therefore, Kripke structures together with simulation maps define a category **KSim**.

The usefulness of the definition of simulation is justified by the next theorem.

Theorem 1 *Let $(h, \Gamma) : M \longrightarrow N$ be a simulation, a a state in M, and φ a formula in $LTL^-(\Gamma)$. If $N, h(a) \models \varphi$, then $M, a \models \varphi$. If the simulation preserves the atomic propositions, in the sense that for all a, b in M, $h(a) = h(b)$ implies that $\Gamma \cap L_M(a) = \Gamma \cap L_M(b)$, then the result holds for arbitrary formulas in $LTL(\Gamma)$.*

Proof The theorem is an immediate consequence of the following two results, that can be proved by induction on the length of the path and by structural induction, respectively:

1. if π is a path in M starting at a, then $h(\pi)$ is a path in N starting at $h(a)$, and
2. for each path π in M starting at a, $N, h(a), h(\pi) \models \varphi$ implies $M, a, \pi \models \varphi$.

□

This theorem is the key point behind the whole method of model checking by abstraction: Given an infinite system M, find a finite system N that simulates it, and use model checking to prove that φ holds in N; then, by Theorem 1, φ also holds in M.

In general, however, we will only have our concrete system M and a function $h : S_M \longrightarrow A$ mapping concrete states to a simplified (usually finite) domain. In these cases, there is a canonical way of constructing a Kripke structure out of h, in such a way that h becomes a simulation.

Definition 3 *The* minimal system $M^h_{\min}$ *corresponding to* M *and* $h : S_M \longrightarrow A$ *is given by the triple* $(A, h(\rightarrow_M), L_{M^h_{\min}})$*, where* $L_{M^h_{\min}}(a) = \bigcap_{h(x)=a} L_M(x)$*.*

The following proposition is an immediate consequence of the definitions.

Proposition 1 *For all* M *and* h*,* $(h, AP) : M \longrightarrow M^h_{\min}$ *is a simulation.*

The use of the adjective minimal is correct since, as pointed out in [6], $M^h_{\min}$ is the most accurate approximation to M that is consistent with h. This can be recast in a more precise categorical setting as follows.

Proposition 2 *The forgetful functor* U : **KSim** $\longrightarrow$ **Set***, mapping a Kripke structure* $M = (S, \rightarrow, L)$ *to its underlying set* S*, and a simulation map* (h, Γ) *to* h*, is an opfibration.*

Proof Let $M = (S_M, \rightarrow_M, L_M)$ be an object in **KSim**, and $h : S_M \longrightarrow A$ an arrow in **Set**. Let us define $h^*(M) = M^h_{\min}$ and check that $(h, AP) : M \longrightarrow h^*(M)$ is an opcartesian map.

Given $(f, \Gamma) : M \longrightarrow N$ in **KSim** such that it can be factorized in **Set** as $f = g \circ h$ for some function $g : A \longrightarrow S_N$, we have to find a unique Δ such that (g, Δ) is also an arrow $(g, \Delta) : h^*(M) \longrightarrow N$ in **KSim**, and $(f, \Gamma) = (g, \Delta) \circ (h, AP)$.

By definition of composition in **KSim**, Δ must be equal to Γ. Now, if $x \rightarrow y$ in $h^*(M)$, there exists a and b in M such that $h(a) = x$, $h(b) = y$, and $a \rightarrow_M b$. Hence, since (f, Γ) is a simulation, $g(x) = g(h(a)) = f(a) \rightarrow_N f(b) = g(h(b)) = g(y)$. On the other hand, using again the fact that (f, Γ) is a simulation, if $p \in \Gamma \cap L_N(s)$ then $p \in L_M(a)$ for all a in M such that $f(a) = s$. Let then $x \in A$ such that $g(x) = s$: for all b in M such that $h(b) = x$, since $f(b) = g(h(b)) = s$, it is the case that $p \in L_M(b)$. Therefore, $p \in L_{h^*(M)}(x)$, and since this holds for all x with $g(x) = s$, we have $\Gamma \cap L_N(s) \subseteq \Gamma \cap \bigcap_{g(x)=s} L_{h^*(M)}(x)$. □

However, it is not always possible to calculate $M^h_{\min}$. The definition of $\rightarrow_{M^h_{\min}}$ can be rephrased as $x \rightarrow_{M^h_{\min}} y$ if and only if $\exists a \exists b.(h(a) = x \wedge h(b) = y \wedge a \rightarrow_M b)$, and this relation is, in general, recursively enumerable but not recursive, even if $\rightarrow_M$ is. Our goal, then, will be to try to calculate abstract models as similar to $M^h_{\min}$ as possible.

One could also think of finding a function τ over formulas such that, for a simulation $h : A \longrightarrow B$ and $h(a) = b$, if $B, b \models \tau(\varphi)$ then $A, a \models \varphi$, and which would allow us to prove more things than just by using the identity. But the previous implication should hold, in particular, when h is the identity simulation so that $A, a \models \tau(\varphi)$ implies $A, a \models \varphi$ and it turns out that the identity over formulas was already optimal.

2.3 Rewriting Logic

We summarize here the main ideas about rewriting logic and refer the reader to the appropriate references for a full account.

Rewriting logic [25] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which we will use membership equational logic [26]. Roughly, membership equational logic is like many-sorted equational logic, but now the sorts are called *kinds* and have an associated set of unary predicates that are called sorts, and in addition to equations there are also membership assertions of the form $t : s$ with t a term and s a sort. Given a sort s, we write $[s]$ for the kind it is associated to. As usual, given a signature Ω and a set of sentences E, we denote by T_Ω and $T_{\Omega/E}$ the corresponding initial algebras, with an extra subscript k to refer to the terms of kind k.

Then, a rewrite theory $\mathcal{R} = (\Omega, E, \phi, R)$ consists of a membership equational theory (Ω, E), a set of rewrite rules R of the form $t \longrightarrow t'$ if C (where C is a condition that can contain equations, membership assertions, and rewrites), and ϕ is a function specifying the arguments *frozen* for each operator, under which rewrites are not allowed. We will denote by $\rightarrow^1_{\mathcal{R},k}$ the one-step rewrite relation between terms of kind k. Intuitively, $t \rightarrow^1_{\mathcal{R}} t'$ if t can be reduced to t' by a single application of one of the rewrite rules in R (but possibly using also some other rules of deduction, like congruence).

Rewriting logic has been implemented in a highly efficient language called Maude [8] whose syntax we will use to introduce our rewrite theories.

3 Intuitions

We introduce in this section two examples to illustrate our techniques and show the basic intuitions behind them. The presentation is not completely rigorous in that some issues are skipped (in particular, those dealing with the labelling function), but nonetheless we think that they manage to convey the main ideas.

3.1 Readers and Writers

Consider the following rewrite theory specifying a readers-writers system [3]:

```
mod R&W is
  sorts Nat State .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op <_,_> : Nat Nat -> State [ctor] .

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
```

```
  rl < s(R), W > => < R, W > .
endfm
```

R represents the number of readers in the system, and W that of the writers. If the critical section is empty, either a reader or a writer can enter it, and a reader can also enter it whenever another reader is there already. A reader or a writer can decide to abandon the system at any time. We are interested in verifying that the following property holds in the system: starting in < 0, 0 >, it is always the case that either the number of readers or the number of writers is 0.

Since the number of reachable states is infinite, model checking is not directly applicable to this system. What we can do is to *abstract* the system to get a finite one in which the property can be model checked. For that, let us consider the following abstraction given by the function abs.

```
  abs(< 0, 0 >) = sinit .
  abs(< s(R), 0 >) = srea .
  abs(< 0, s(W) >) = swri .
  abs(< s(R), s(W) >) = srw .
```

Intuitively, this abstraction identifies all concrete states having only readers (srea), only writers (swri), none (sinit), or both (srw). Note that, if two states are identified, then either both satisfy our required property, or none does.

The question now is: How can the transitions of the abstract system be obtained? We already know that this set is just the image by abs of the set of concrete transitions. In the following, we show an appealing procedure to calculate it, and leave for Section 5 the proof of its correctness.

Let us start with the first rule of the module R&W. We can trivially unify the left-hand side term of this rule with the argument of abs in the first equation of the abstraction. Similarly, the right-hand side term of the rule can be unified with the argument of abs in the third equation by means of the most general unifier (m.g.u) $\sigma = \{\mathtt{R} \rightarrow \mathtt{0}\}$. Replacing the concrete states by their abstract counterparts, the concrete rule is transformed into

```
  rl sinit => swr .
```

We cannot unify < 0, 0 > with any other argument of abs, so we are done.

Let us move to the second rule in R&W. Its left-hand side term can be unified with the argument of the third equation defining abs by means of the m.g.u $\sigma = \{\mathtt{R} \rightarrow \mathtt{0}\}$. The term < 0, W >, resulting from applying σ to the right-hand side term in the rule, can be unified with the argument in the first equation by means of $\{\mathtt{W} \rightarrow \mathtt{0}\}$ to produce the abstract rule

```
  rl swr => sinit .
```

and with the argument of the third one to produce

```
  rl swr => swr .
```

Note that `< R, s(W) >` can also be unified with a fresh copy of the argument in the fourth equation for `abs` in which `R` and `W` have been renamed as `R'` and `W'`, by means of the m.g.u $\sigma = \{\texttt{R} \rightarrow \texttt{s(R')}, \texttt{W} \rightarrow \texttt{W'}\}$. The result of applying σ to the right-hand term in the rule is `< s(R'), W' >`, which can be unified with the arguments of the second and the fourth equations. As a result, the concrete rule gives rise to the following two abstract rules, too.

```
  rl srw => sre .
  rl srw => srw .
```

Repeating the process with the remaining two rules, we finally get the following module specifying the abstract system.

```
mod R&W-ABS is
  sort AbsState .

  ops sinit sre swr srw : -> AbsState [ctor] .

  rl sinit => swr .
  rl sinit => sre .
  rl sre => sinit .
  rl sre => sre .
  rl swr => sinit .
  rl swr => swr .
  rl srw => srw .
  rl srw => sre .
  rl srw => swr .
endm
```

Our required property that readers and writers are never simultaneously in the concrete system is reduced to showing that the abstract state `srw` (corresponding to those concrete states with both readers and writers) is not reachable, which can be proved by model checking (or by direct inspection of the rules in this simple example).

Our abstraction has proved to be very useful to reduce our problem to a simpler one. Note, however, that for certain properties of the original system we have actually abstracted too much. Consider for example the following requirement: the number of writers in the system is always equal either to `0` or to `s(0)`. This property holds in the original system but not in the abstract one, since we have identified all the states with no readers in the abstract state `swr`. To cope with this difficulty we could define the following abstraction, obtained following a method proposed in [5], in which states with zero or one writers are no longer identified with states with a larger number of writers.

```
  abs(< 0, 0 >) = sinit .
  abs(< 0, s(0) >) = swr1 .
  abs(< 0, s(s(W)) >) = swr2 .
```

```
  abs(< s(R), 0 >) = sre .
  abs(< s(R), s(0) >) = srw1 .
  abs(< s(R), s(s(W)) >) = srw2 .
```

The same procedure can now be applied with this new function to get the corresponding abstract system, in which the desired property holds and can be proved using model checking.

3.2 A Mutual Exclusion Protocol

Consider now the following protocol adapted from [12].

```
mod DAMS is
  protecting MACHINE-INT .
  sorts State Condition .
  ops think eat : -> Condition [ctor] .
  op <_,_,_> : Condition Condition MachineInt -> State [ctor] .

  vars L0 L1 : Condition .
  var N : MachineInt .

  crl < think, L1, N > => < eat, L1, N > if (N % 2) == 1 .
  rl  < eat, L1, N > => < think, L1, 3 * N + 1 > .
  crl < L0, think, N > => < L0, eat, N > if (N % 2) == 0 .
  crl < L0, eat, N > => < L0, think, N / 2 > if (N % 2) == 0 .
endm
```

Following [12], this specification can be thought of as a protocol controlling the mutually exclusive access to a common resource of two concurrent processes, modelling the behavior of two mathematicians, corresponding to the first two components in a state. They alternate phases of "thinking" and "eating," regulated by the current value `N` of the third component of the state: if `N` is even, then the first mathematician has the right to enjoy the meal, otherwise, the turn corresponds to the second one. After finishing the eating phase, each mathematician leaves the dining room and modifies the value of `N` in his own fashion.

Some properties that we could be interested in checking for this system are: (1) it is always the case that at least one of the mathematicians is thinking; (2) every eating phase of the first mathematician is eventually followed by an eating phase of the second one; (3) the same as (2), swapping the roles of the mathematicians.

Again, since the system is infinite, model checking is not immediately applicable; the abstraction proposed in [12] consists of using the parity of `N` instead of its actual value.

```
  abs(< L0, L1, N >) = < L0, L1, e > if even(N) .
  abs(< L0, L1, N >) = < L0, L1, o > if odd(N) .
```

We can now try to apply the same technique we used with the readers-writers system. However, this turns out to be not possible due to the presence of terms

with non-constructor operations in the rules, which makes our procedure based on syntactic unification useless. In these cases it is easier just to add some equations to the original specification making the suitable identifications, and to keep the original rules. In this example, it is only necessary to add

```
ceq < L0, L1, N > = < L0, L1, N % 2 > if N > 1 .
```

Note that, in this way, all states are reduced either to `< L0, L1, 0 >`, or to `< L0, L1, 1 >`, which correspond to the abstract states `< L0, L1, e >` and `< L0, L1, o >`, respectively.

A price for this simplification has to be paid, however. We must now check that the resulting system is coherent, in the sense that no transition that could be taken before applying the reduction is no longer available once the state has been reduced. For our current example this property is actually false. A state of the form `< L0, eat, 2 >` can make a transition step to the state `< L0, think, 1 >` in the concrete system. However, in the abstract one, it would first be reduced to `< L0, think, 0 >`, from which `< L0, think, 1 >` is not reachable. Fortunately, this problem can be solved by simply adding the rule `< L0, eat, 0 > => < L0, think, 1 >`. Overall, then, the specification of our abstract system is as follows.

```
mod DAMS-ABS is
  protecting MACHINE-INT .
  sort State Condition .
  ops think eat : -> Condition [ctor] .
  op <_,_,_> : Condition Condition MachineInt -> State [ctor] .

  vars L0 L1 : Condition .
  vars N : MachineInt .

  ceq < L0, L1, N > = < L0, L1, N % 2) if N > 1 .

  crl < think, L1, N > => < eat, L1, N > if (N % 2) == 1 .
  rl  < eat, L1, N > => < think, L1, 3 * N + 1 > .
  crl < L0, think, N > => < L0, eat, N > if (N % 2) == 0 .
  crl < L0, eat, N > => < L0, think, N / 2 > if (N % 2) == 0 .
  rl  < L0, eat, 0 > => < L0, think, 1 > .
endm
```

This system is finite, and model checking can be used to show that properties (1) and (2) actually hold. Property (3), however, fails to hold in the abstract system. In fact, as pointed out in [12], property (3) is not amenable to be proved in this way, even by refining the abstraction, since for that it would be necessary to identify all the states with `N` divisible by 4, all those with `N` divisible by 8, by, 16, ..., and that would produce another infinite system.

The work presented in [12] relies on the use of ordered sets and Galois connections. In particular, both abstract domains $\{\texttt{think}, \texttt{eat}\}$ and $\{\texttt{e}, \texttt{o}\}$ are extended with a top element $\top$, and suitable abstract interpretations for all

concrete operations (addition, multiplication, ...) are defined. We strongly believe that our approach is simpler.

4 LTL Properties of Rewrite Theories and Model Checking

To associate LTL properties to a rewrite theory $\mathcal{R} = (\Omega, E, \phi, R)$ we need to make explicit two things: (1) the intended kind k of states and (2) the relevant *state* predicates. In general, the state predicates need not be part of the system specification $\mathcal{R}$; they are typically part of the property specification. We assume that they have been defined by means of equations D in an equational theory $(\Omega', E \cup D)$ extending (Ω, E) in a conservative way; specifically, the unique Ω-homomorphism $T_{\Omega/E} \to T_{\Omega'/E \cup D}$ should be bijective at each sort s in Ω. The syntax defining the state predicates consists of a subsignature $\Pi \subseteq \Omega'$ of function symbols p of the general form $p : s_1 \dots s_n \longrightarrow \mathit{Prop}$, reflecting the fact that state predicates can be parametric. Then the semantics of the state predicates Π is defined with the help of an operator $\_ \models \_ : k \ [\mathit{Prop}] \longrightarrow [\mathit{Bool}]$ in Ω' and, for ground terms $u_1, \dots, u_n$, we say that the state predicate $p(u_1, \dots, u_n)$ holds in the state $[t]$ if

$$E \cup D \vdash (\forall \emptyset)\, t \models p(u_1, \dots, u_n) = \mathit{true}\,.$$

We are now ready to associate to $\mathcal{R}$ a Kripke structure whose atomic predicates are specified by the set $AP_\Pi = \{\theta(p) \mid p \in \Pi, \theta \text{ canonical ground substitution}\}$. It is given by $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Omega/E,k}, (\rightarrow^1_{\mathcal{R}})^\bullet, L_\Pi)$, where

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid \theta(p) \text{ holds in } [t]\}.$$

In practice we typically want the equality $t \models p(u_1, \dots, u_n) = \mathit{true}$ to be decidable. This can be achieved by assuming that $D \cup E$ is a set of confluent, sort-decreasing, and terminating equations and memberships, and that the semantics of each state predicate p is defined by a finite set of conditional equations of the form $t \models p(u_1, \dots, u_n) = \mathit{true} \Leftarrow C$, where the terms t, $u_1, \dots, u_n$ are all patterns.

The Maude system has been recently extended with an on-the-fly LTL model checker [15]. Given a rewrite theory specified in Maude by a system module `M`, and an initial state `init` of sort $\texttt{State}_\texttt{M}$, we can model check different LTL properties beginning at this state. For that, a new module `CHECK-M` must be defined importing `M` and the predefined module `MODEL-CHECKER`, and a subsort declaration $\texttt{State}_\texttt{M}$ `< State` must be added. Then the syntax of the state predicates must be declared by means of operations of sort `Prop`, and their semantics must be given by equations involving the satisfaction operator

```
op _|=_ : State Prop -> Result .
```

Once the semantics of the state predicates has been defined, we can model check any LTL `formula` by giving the command

```
reduce init |= formula .
```

Of course, the set of states reachable from `init` should be finite.

Let us illustrate the use of the Maude model checker with the abstract version of our mutual exclusion system of Section 3.2. The first thing we have to do is to create a new module `DAMS-CHECK` including `MODEL-CHECKER` and `DAMS-ABS`; since `State` is already our working sort, it is not necessary to add any subsort declarations. Then, since we are interested in verifying properties involving the current phase the mathematicians are in, we can think of declaring the following operations

```
ops status1 status2 : Condition -> Prop [ctor] .
```

with their semantics given by

```
eq (< L0, L1, N > |= status1(L0)) = true .
eq (< L0, L1, N > |= status2(L1)) = true .
```

Then, the mutual exclusion property (1) would be expressed by the LTL formula $\Box$(`status1(think)` $\vee$ `status2(think)`), which can be checked with the command

```
red < think, think, 0 > |= [] (status1(think) \/ status2(think)) .
```

that returns

```
Result Bool: true
```

as answer. Similarly, property (2) can be checked by means of

```
red < think, think, 0 > |= [] (status1(eat) -> <> status2(eat)) .
```

5 Calculating Minimal Structures

Suppose we are given a Kripke structure specified by means of a rewrite theory (Ω, E, ϕ, R). (Ω, E) would define the data types under consideration, with an operation `<_,...,_>` to represent the states of the system, and R would be a set of rules of the form

$$\texttt{<}\ t_1\texttt{,...,}t_n\ \texttt{>}\ \texttt{=>}\ \texttt{<}\ t'_1\texttt{,...,}t'_n\ \texttt{>}\ \texttt{if}\ \varphi$$

specifying the transitions. The abstraction function `abs` is defined equationally in a theory extension (Ω', E') of (Ω, E) by

$$\begin{array}{l}\texttt{ceq abs(}p_1\texttt{) = }v_1\ \texttt{if}\ \psi_1\ \texttt{.}\\ \vdots\\ \texttt{ceq abs(}p_m\texttt{) = }v_m\ \texttt{if}\ \psi_m\ \texttt{.}\end{array}$$

We present a procedure to obtain the corresponding minimal system.

Suppose that the terms involved in the equations defining `abs`, as well as the left-hand sides of the rewrite rules, are all patterns, that is, terms that only contain constructor operations. Then, for each rewrite rule l `=>` r `if` φ, we try to unify l with each p_i in turn. If σ_i is a most general unifier of l and p_i, we try to unify $\sigma_i(r)$ with each p_j in turn to get a m.g.u δ_j. Assume that the right-hand side r of the rewrite rule is a pattern, too. Then, if $(\delta_j \circ \sigma_i)(\varphi \wedge \psi_j) \wedge \delta(\psi_j)$ is satisfiable, we add the rewrite rule

`rl` $(\delta_j \circ \sigma_i)(v_i)$ `=>` $\delta_j(v_j)$ `.`

to the abstract system. To show satisfiability, narrowing or theorem proving could be tried.

The semantics of the atomic propositions should be such that $L(\texttt{abs}(S)) = \bigcap_{\texttt{abs}(S)=\texttt{abs}(X)} L(X)$; our construction of the labelling function, however, will only guarantee that the first set is included in the second one, which is still enough to have a simulation map. Furthermore, we assume that the terms v_i in the abstraction are "disjoint", in the sense that there is no substitution σ such that $E' \vdash v_i\sigma = v_j\sigma$ for any i, j. Then, for each equation

`ceq (`q_j `|=` f_j`) = true if` φ_j `.`

giving the semantics of the atomic propositions, and each

`ceq abs(`p_i`) =` v_i `if` ψ_i `.`

with (renaming, if necessary) a different set of variables, we try to unify p_i and q_j. In case we can find an m.g.u σ such that it does not instantiate the variables in $vars(p_i) \setminus vars(v_i)$, then we add the equation

`ceq (`$v_i\sigma$ `|=` $f_j\sigma$`) = true if` $\varphi_j\sigma$ `.`

Note, however, that there could be variables in the condition that do not appear in $v_i\sigma$ or $f_j\sigma$. In those cases, since we want our specification to be executable, we will only add the equation if we can show that the implication $\psi_i\sigma \Rightarrow \varphi_j\sigma$ holds, meaning that the condition holds for all the concrete states that are mapped to those abstract ones.

If such an m.g.u σ does not exist, it means that either there are no concrete terms satisfying the atomic proposition, or that some of them satisfy it but some others do not: in both cases, no equation has to be added.

We will refer to the transformation described here as procedure **(A1)**.

5.1 Example: The Bakery Protocol

We will use the bakery protocol [20, 21], a two-process mutual exclusion protocol, to illustrate this method. The specification in Maude of the protocol is the following.

```
fmod NAT is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _<_ : Nat Nat -> Bool .

  vars N M : Nat .

  eq N < 0 = false .
  eq 0 < s N = true .
  eq s N < s M = N < M .
endfm

mod BAKERY is
  protecting NAT .

  sorts PC State .

  ops sleep wait crit : -> PC [ctor] .
  op <_,_,_,_> : PC Nat PC Nat -> State [ctor] .

  vars p q : PC .
  vars x y : Nat .

  rl [p1_sleep] : < sleep, x, q, y > => < wait, s y, q, y > .
  rl [p1_wait] : < wait, x, q, 0 > => < crit, x, q, 0 > .
  crl [p1_wait] : < wait, x, q, y > => < crit, x, q, y > if not (y < x) .
  rl [p1_crit] : < crit, x, q, y > => < sleep, 0, q, y > .

  rl [p2_sleep] : < p, x, sleep, y > => < P, x, wait, s X > .
  rl [p2_wait] : < p, 0, wait, y > => < P, 0, crit, y > .
  crl [p2_wait] : < p, x, wait, y > => < P, x, crit, y > if y < x .
  rl [p2_crit] : < p, x, crit, y > => < P, x, sleep, 0 > .
endm
```

The property we want to verify is mutual exclusion. The labelling function associated to the original system, using the notation of the Maude model checker, is given by

```
  op excl : -> Prop [ctor] .

  ceq (< P, X, Q, Y > |= excl) = true if (P =/= crit) or (Q =/= crit) .
```

and mutual exclusion would be expressed by `[] excl`. Since the original system is infinite, the model checker cannot prove that this property holds.

To try to simplify the system and make it amenable to model checking, the following abstraction function is defined:

```
abs(< P, X, Q, Y >) = < P, Q, X = 0, Y = 0, Y < X >
```

pattern	most general unifier with `< sleep, x, q, y >`
$p_1 =$ `< P, 0, Q, 0 >`	$\sigma_1 = \{$`P` $\rightarrow$ `sleep`, `x` $\rightarrow$ `0`, `Q` $\rightarrow$ `q'`, `q` $\rightarrow$ `q'`, `y` $\rightarrow$ `0`$\}$
$p_2 =$ `< P, 0, Q, s Y >`	$\sigma_2 = \{$`P` $\rightarrow$ `sleep`, `x` $\rightarrow$ `0`, `Q` $\rightarrow$ `q'`, `q` $\rightarrow$ `q'`, `y` $\rightarrow$ `s y'`, `Y` $\rightarrow$ `s y'`$\}$
$p_3 =$ `< P, s X, Q, 0 >`	$\sigma_3 = \{$`P` $\rightarrow$ `sleep`, `x` $\rightarrow$ `s x'` `q` $\rightarrow$ `q'`, `Q` $\rightarrow$ `q'`, `y` $\rightarrow$ `0`$\}$
$p_4 =$ `< P, s X, Q, s Y >`	$\sigma_4 = \{$`P` $\rightarrow$ `sleep`, `x` $\rightarrow$ `s x'`, `q` $\rightarrow$ `q'`, `Q` $\rightarrow$ `q'`, `y` $\rightarrow$ `s y'`, `Y` $\rightarrow$ `y'`$\}$
$p_5 =$ `< P, s X, Q, s Y >`	$\sigma_5 = \{$`P` $\rightarrow$ `sleep`, `x` $\rightarrow$ `s x'`, `q` $\rightarrow$ `q'`, `Q` $\rightarrow$ `q'`, `y` $\rightarrow$ `s y'`, `Y` $\rightarrow$ `y'`$\}$

Table 1: Table

Intuitively, we do not care about the actual values of the variables, but only about which one is greater, and whether they are equal to zero. In Maude, it is specified as follows.

```
op <_,_,_,_,_> : PC PC Bool Bool Bool -> State_A [ctor] .
op abs : State -> State_A .

eq abs(< P, 0, Q, 0 >) = < P, Q, true, true, false > .
eq abs(< P, 0, Q, s Y >) = < P, Q, true, false, false > .
eq abs(< P, s X, Q, 0 >) = < P, Q, false, true, true > .
ceq abs(< P, s X, Q, s Y >) = < P, Q, false, false, true > if Y < X .
ceq abs(< P, s X, Q, s Y >) = < P, Q, false, false, false > if not(Y < X) .
```

We will now describe the construction of the "abstract" rewrite rules in the minimal system corresponding to the rule `[p1_sleep]`.

Table 1 shows the m.g.u of the right-hand side of the rewrite rule and the patterns in the definition of `abs`. Note that the construction requires the use of fresh variables.

In the following, let t' denote the right-hand side of the rewrite rule, i.e., $t' =$ `< wait, s y, q, y >`.

1. $\sigma_1(t') =$ `< wait, s 0, q', 0 >`. The only pattern defining `abs` unifiable with $\sigma_1(t')$ is p_3, by means of the m.g.u $\sigma_1' = \{$`P` $\rightarrow$ `wait`, `X` $\rightarrow$ `0`, `Q` $\rightarrow$ `q'`$\}$. By construction, we obtain the rewrite rule

   ```
   rl < sleep, q', true, true, false > => < wait, q', false, true, true > .
   ```

2. $\sigma_2(t') =$ `< wait, s s y', q', s y' >`. Only the patterns p_4 and p_5 are unifiable with $\sigma_2(t')$ via the m.g.u $\sigma_2' = \{$`P` $\rightarrow$ `wait`, `X` $\rightarrow$ `s y'`, `Q` $\rightarrow$ `q'`, `Y` $\rightarrow$ `y'`$\}$. We now have to check whether $\sigma_2'($`Y < X`$) =$ `y' < s y'`, and

σ_2'(`not (Y < X)`) are satisfiable. Since `y' < s y'` is satisfiable we obtain the rewrite rule

```
rl < sleep, q, true, false, false > => < wait, q, false, false, true > .
```

and since `not (y' < s y')` is unsatisfiable, we do *not* add the rewrite rule

```
rl < sleep, q, true, false, false > => < wait, q, false, false, false > .
```

3. $\sigma_3(t') =$ `< wait, s 0, q', 0 >`. Analogously to (1) we obtain the rewrite rule

```
rl < sleep, q', false, true, true > => < wait, q', false, true, true > .
```

4. $\sigma_4(t') =$ `< wait, s s y', q', s y' >`. Only the patterns p_4 and p_5 are unifiable with $\sigma_4(t')$ via the m.g.u $\sigma_4' = \{$`P` $\rightarrow$ `wait`, `X` $\rightarrow$ `s y'`, `Q` $\rightarrow$ `q'`, `Y` $\rightarrow$ `y'`$\}$. We now have to check whether

$$(\sigma_4' \circ \sigma_4)(\texttt{Y < X}) \wedge \sigma_4'(\texttt{Y < X})$$

and

$$(\sigma_4' \circ \sigma_4)(\texttt{Y < X}) \wedge \sigma_4'(\texttt{not (Y < X)})$$

are satisfiable. By construction, since only the first conjunction is satisfiable, we add the rewrite rule

```
rl < sleep, q', false, false, true > => < wait, q', false, false, true > .
```

5. Analogously to (4) we obtain the rewrite rule

```
rl < sleep, q', false, false, false > => < wait, q', false, false, true > .
```

The equations giving the semantics of the labelling function are also easily computed. Consider, for example, the fourth equation defining `abs`. We discard its condition and try to unify its argument `< P, s X, Q, s Y >` with `< P', X', Q', Y' >`, a renaming of the state term appearing in the definition of `excl`. Since the unification is possible and does not instantiate `X` or `Y` (which do not appear in the right-hand side of that fourth equation), we have that

```
ceq (< P, Q, false, false, true > |= excl) = true if (P =/= crit) or (Q =/= crit) .
```

Repeating the same process with the other four equations, we get

```
eq (< P, Q, true, true, false > |= excl) = true .
eq (< P, Q, true, false, false > |= excl) = true if (P =/= crit) or (Q =/= crit) .
eq (< P, Q, false, true, true > |= excl) = true if (P =/= crit) or (Q =/= crit) .
ceq (< P, Q, false, false, false > |= excl) = true if (P =/= crit) or (Q =/= crit) .
```

The resulting system is finite and can be model checked to show that `excl` holds.

6 Minimal Structures as Quotients

Now, let us take a closer look at the second method of calculating the minimal system, as illustrated by the mutual exclusion protocol in Section 3.2.

Notation. Given a function $h : A \longrightarrow B$, we will denote by $\equiv_h$ the equivalence relation on A defined by $a \equiv_h a'$ if $h(a) = h(a')$, and by $[a]_h$ the corresponding equivalence classes. We will drop the subscript if h can be inferred from the context.

Then, in case the function h is surjective, an equivalent presentation of the minimal system is the following.

Definition 4 *Given a Kripke structure $M = (S, \rightarrow, L)$, and a function $h : S \longrightarrow A$, we define the quotient Kripke structure $M/h = (S_h, \rightarrow_h, L_h)$, where*

1. $S_h = (S/\equiv) = \{[a] \mid a \in S\}$;
2. $[a] \rightarrow_h [b]$ *if and only if* $\exists a' \in [a].\exists b' \in [b].a \rightarrow b$;
3. $L_h([a]) = \bigcap_{x \in [a]} L(x)$.

Theorem 2 *Let $M = (S, \rightarrow, L)$ be a Kripke structure and $h : S \longrightarrow A$ a surjective function. Then, the Kripke structures M_h and M/h are isomorphic in the category* **KSim**.

Proof Define the functions $f : A \longrightarrow S_h$ by $f(h(s)) = [s]$, and $g : S_h \longrightarrow A$ by $g([s]) = h(s)$. By definition of $\equiv$, and since h is surjective, both functions are indeed well-defined.

If $x \rightarrow_{M^h_{\min}} y$, then there exists a and b in M such that $h(a) = x$, $h(b) = y$, and $a \rightarrow b$, and therefore $f(x) = [a] \rightarrow_{M/h} [b] = f(y)$. Similarly, if $[a] \rightarrow_{M/h} [b]$, then there exists a' such that $h(a) = h(a')$, and b' such that $h(b) = h(b')$, with $a' \rightarrow b'$, and hence $g([a]) = h(a') \rightarrow_{M^h_{\min}} h(b') = g([b])$.

Finally,

$$\begin{aligned} L_{M/h}([a]) &= \textstyle\bigcap_{h(x)=h(a)} L_M(x) \\ &= \textstyle\bigcap_{f(h(b))=f(h(a))} \left(\bigcap_{h(x)=h(b)} L_M(x) \right) \\ &= \textstyle\bigcap_{f(h(b))=[a]} L_{M^h_{\min}}(h(b)) \,, \end{aligned}$$

and

$$\begin{aligned} L_{M^h_{\min}}(h(a)) &= \textstyle\bigcap_{h(x)=h(a)} L_M(x) \\ &= \textstyle\bigcap_{g([x])=h(a)} \left(\bigcap_{y\in[x]} L_M(y)\right) \\ &= \textstyle\bigcap_{g([x])=h(a)} L_{M/h}([x])\,. \end{aligned}$$

Thus, $f : M^h_{\min} \longrightarrow M/h$ and $g : M/h \longrightarrow M^h_{\min}$ are simulations, and it is clear that $f \circ g = 1_{M/h}$ and $g \circ f = 1_{M^h_{\min}}$. □

That is, we can perform the abstraction either by mapping the concrete states to an abstract domain, or by identifying some states and thereafter working with the corresponding equivalence classes.

The second case, in rewriting logic, corresponds just to the addition of some equations E' to the rewrite theory making the required identifications. The resulting theory, in case only states satisfying the same atomic propositions are identified, is actually the rewrite theory corresponding to the minimal system. However, from a practical point of view, in which we are also interested in being able to execute the theory, this may not be enough because of lack of *coherence* [32]. Due to the internal strategy that Maude follows, it could happen that a transition that applies to a concrete state cannot be applied to the abstract state to which is reduced. Consider for example the transition system given by

```
eq b = a .
rl a => c .
rl b => d .
```

The strategy followed by Maude to reduce a term consists of first applying the equations as rewrite rules from left to right until a canonical form is reached, and then applying the rules. In this case, even though the equivalence class of `a` should have a transition to that of `d`, with this strategy both `a` and `b` are only reduced to `c`. But the problem can be solved by including one additional rule.

```
rl a => d .
```

As this example shows, even if the resulting abstract system is not coherent, we can still hope to find a finite set of additional rules that "complete" it. Checking coherence and, if necessary, completion of the system, can be done by hand, but there are also tools available in Maude to help the user in this task (see [14]). Note, however, that there is no guarantee that the completion procedure terminates.

The only remaining point to comment on concerns the labelling function in the minimal system. In general, the labelling function of the system modulo the new equations is different from that of the original system. By construction, given an equivalence class $[s]$ in the quotient system, and an atomic proposition p, $p \in L_h([s])$ if and only if $p \in L_M(s')$ for all states $s' \in [s]$. In the mutual exclusion example, if two states are equivalent then they satisfy the same set of atomic propositions because the abstraction does not modify either `think` or `eat`. Therefore, there is no need to modify the labelling function. Again, this

can be proved by hand (for this example it is immediate), or with the help of the inductive theorem prover ITP [10, 9] distributed with Maude.

Note that a particularly easy case is that of *k-topmost* rewrite theories, in which the kind k only appears as the codomain of an operation $f : k_1 \dots k_n \longrightarrow k$. This is not a very restrictive condition, since any rewrite theory $\mathcal{R}$ can be transformed into a semantically equivalent k'-topmost one as made precise by the following lemma.

Lemma 1 *Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a kind $k \in \Sigma$, define the rewrite theory $\mathcal{R}' = (\Sigma', E, R)$ with Σ' extending Σ with a new kind k' and an operation $\{\_\} : k \longrightarrow k'$. $\mathcal{R}'$ so defined is k'-topmost. Furthermore, if Π is a set of state predicates for $\mathcal{R}$ defined by a set of equations D, define state predicates Π for $\mathcal{R}'$ by transforming each equation $t \models p = b$ if C in D into $\{t\} \models p = b$ if C. Then, the signature morphism $H : \Sigma' \longrightarrow \Sigma$ that is the identity on Σ, maps k' to k, and the operation $\{\_\}$ to the term x, with x a variable of kind k, induces a bijective bisimulation $\mathcal{K}(\mathcal{R}, k)_\Pi \equiv \mathcal{K}(\mathcal{R}', k')_\Pi$.*

Proof Since no new rules or equations are added to $\mathcal{R}'$ it is immediate that $\{t\} \rightarrow^1_{\mathcal{R}',k'} t'$ iff $t \rightarrow^1_{\mathcal{R},k} t'$. But then, since H relates the term $\{t\}$ to t, we have that the transition relation is preserved in both directions. As for the state predicates, by the tranformation applied to the equations in D and, again, since no new equations have been added to $\mathcal{R}'$, we have $L_\Pi(\{t\}) = L_\Pi(t)$, and the result follows. □

Proposition 3 *Suppose a k-topmost rewrite theory in which all (possibly conditional) equations in E' are of the form*

$$t = t' \quad \textit{if} \quad C$$

with t and t' of kind k, and that the equations $E \cup E' \cup D$ are ground confluent, sort-decreasing, and terminating. Furthermore, suppose that no equations between terms of kind k appear in the conditions of any equation. If for each such equation and each state predicate $p \in \Pi$ we can prove the inductive property

$$E \cup D \vdash_{ind} (\forall \vec{x}\ \forall \vec{y})\ C \Rightarrow (t(\vec{x}) \models p(\vec{y}) = \mathit{true} \Leftrightarrow t'(\vec{x}) \models p(\vec{y}) = \mathit{true})$$

then we have established the preservation of the state predicates Π by the equations E'.

Proof To show that, for x and y of kind k, $[x]_{E \cup E'} = [y]_{E \cup E'}$ implies $L_\Pi([x]_E) = L_\Pi([y]_E)$, we proceed by structural induction on the derivation of $E \cup E' \vdash x = y$:

- Reflexivity, symmetry, and transitivity are immediate.
- Congruence. By Lemma 2 below, if we have

$$\frac{E \cup E' \vdash t_1 = t'_1 \ \dots \ E \cup E' \vdash t_n = t'_n}{E \cup E' \vdash f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)}$$

  then $E \vdash t_i = t'_i$ and therefore $E \vdash f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)$, whence the result follows.

- Replacement. Suppose

$$\frac{E \cup E' \vdash C\theta}{E \cup E' \vdash (t = t')\theta}$$

for some equation $t = t' \Leftarrow C$ in $E \cup E'$. By our assumptions and Lemma 2 we have $E \vdash C\theta$. Then, if the equation belongs to E, it follows that $E \vdash (t = t')\theta$ and the result holds. Otherwise, by combining it with our hypothesis

$$E \cup D \vdash_{ind} (\forall \vec{x}\ \forall \vec{y})\ C \Rightarrow (t(\vec{x}) \models p(\vec{y}) = true \Leftrightarrow t'(\vec{x}) \models p(\vec{y}) = true)$$

we have, for any substitution θ' extending θ,

$$E \cup D \vdash (t \models p = true \Leftrightarrow t' \models p = true)\theta'$$

so that $L_\Pi([t\theta]_E) = L_\Pi([t'\theta]_E)$, as required.

□

Lemma 2 *Under the conditions of the previous theorem, if $E \cup E' \vdash t = t'$ and the kind of t, t' is different from k, it holds that $E \vdash t = t'$.*

Proof By structural induction on the proof:

- Reflexivity, symmetry, and transitivity. Trivial.
- Congruence. If

$$\frac{E \cup E' \vdash t_1 = t'_1 \ \ldots \ E \cup E' \vdash t_n = t'_n}{E \cup E' \vdash f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$$

then, since the assumptions preclude any t_i or t'_i from being of kind k, we can apply the induction hypothesis to get $E \vdash t_i = t'_i$ whence the result follows.

- Replacement. If

$$\frac{E \cup E' \vdash C\theta}{E \cup E' \vdash (t = t')\theta}$$

for some equation $t = t' \Leftarrow C$ in E (note that by hypothesis it cannot belong to E'), then we can apply the induction hypothesis to $C\theta$ since it cannot contain equations between terms of kind k, and the result follows.

□

In general, there will be some atomic propositions which will fail to hold in all equivalent states. This means, according to the definition of the quotient Kripke structure, that we must modify the corresponding equations defining the semantics of the atoms so that none of the states in the equivalence class satisfy it. This way the preservation results still hold, even if some of those atomic propositions appear in the formula we are trying to prove; note, however, that if this is actually the case, then it may be necessary to refine the abstraction in order to prove the property in the minimal system.

So we can sum up our proposed methodology as the following procedure **(A2)**:

1. Specify the concrete system as a rewrite theory $\mathcal{R} = (\Omega, E, \phi, R)$.
2. Define a set of equations E' identifying some states.
3. Modify (or, perhaps, remove) those atomic predicates which cannot be proved to be preserved by all the equations in E' using the ITP.
4. Check coherence of the system $(\Omega, E \cup E', \phi, R)$ and, if necessary, complete it, with the help of the automatic coherence checker.
5. Use the Maude model checker to verify the property.

Since the resulting system $\mathcal{R}'$ is coherent, every one-step rewrite $t \rightarrow^1_{\mathcal{R}} t'$ in the original theory has a corresponding one-step rewrite $can(t) \rightarrow^1_{\mathcal{R}'} can(t')$ in the new one. Also, if $((S \,|\!=\varphi) = \texttt{true}) \in L(t)$, then, by the third point above, it must be the case that $((S \,|\!=\varphi) = \texttt{true}) \in L(t')$ for all t' with $E \cup E' \vdash t = t'$. All together, this means that, even though $\mathcal{R}'$ may not be the minimal system corresponding to the identifications made by E' (coherence completion may introduce new rules, and some atomic propositions may be lost during step 3), it is still the case that the mapping $[t]_E \rightarrow [t]_{E \cup E'}$ is a simulation and that $\mathcal{R}'$ is sound to infer results about $\mathcal{R}$.

6.1 Example 1: A Communication Protocol

Our first example is a protocol for in-order communication of messages between a sender and a receiver in an asynchronous communication medium [27]. To guarantee that the messages are received in the correct order, both sender and receiver keep a counter that refers to the message they are currently working with. The sender can, at any moment, nondeterministically choose a message (in the set {`a, b, c`} in this presentation) which is then paired with the sender's counter and released to the medium; the message itself is also added to a list of messages owned by the sender. The receiver has a corresponding list: the purpose of these lists is basically to allow us to state the property we are interested in proving for the system. When the receiver "sees" a message paired with a number equal to its current counter, it removes it from the medium and adds it to its list of received messages.

The following is the specification in Maude of the protocol, where there are only three different types of messages. States are represented as triples `< S, R, PS >`, where `S` represents the status of the sender, `R` that of the receiver, and `PS` the medium.

```
mod PROTOCOL is protecting NAT .
  sorts Message MList Content Pair State PSoup .
  subsort Message < MList .
  subsort Pair < PSoup .

  ops a b c : -> Message [ctor] .
  op nil : -> MList [ctor] .
```

```
  op _:_ : MList MList -> MList [ctor assoc id: nil] .
  op ct : Nat MList -> Content [ctor] .

  op pair : Nat Message -> Pair [ctor] .
  op null : -> PSoup [ctor] .
  op _;_ : PSoup PSoup -> PSoup [ctor assoc comm id: null] .

  op <_,_,_> : Content Content PSoup -> State [ctor] .

  vars N M : Nat .
  var X : Message .
  vars L L1 L2 : MList .
  var PS : PSoup .
  vars R S : Content .

  rl < ct(N, L), R, PS > => < ct(s(N), L : a), R, PS ; pair(N, a) > .
  rl < ct(N, L), R, PS > => < ct(s(N), L : b), R, PS ; pair(N, b) > .
  rl < ct(N, L), R, PS > => < ct(s(N), L : c), R, PS ; pair(N, c) > .
  rl < S, ct(N, L), pair(N, X) ; PS > => < S, ct(s(N), L : X), PS > .
endm
```

The property we would like our system to have is that messages are delivered in the correct order. Thanks to the sender's and receiver's lists, this can be formally expressed by the formula $\Box$ `pref`, where `pref` is an atomic proposition that holds in those states in which the receiver's list is a prefix of the sender's list. In Maude, this would be expressed as

```
  op pref : -> Prop [ctor] .

  eq (< ct(N, L1 : L2), ct(M, L1), PS > |= pref) = true .
```

There are two different sources of infiniteness in this example. The first one corresponds to the counters, that are natural numbers that can reach arbitrarily large numbers. The second one is the communication medium, which is unbounded and can contain an arbitrary number of messages. To deal with it and to able to apply model checking, we define the following abstraction.

First of all, a state whose corresponding sender's and receiver's lists have the same message as the first element can be identified with the state resulting from removing that message from both lists. This can be expressed by means of the equation:

```
  eq < ct(N, X : L1), ct(M, X : L2), PS > = < ct(N, L1), ct(M, L2), PS > .
```

Secondly, if at a certain time both counters are equal and there are no messages in the medium, then the counters can be reset to zero.

```
  ceq < ct(N, L1), ct(N, L2), null > = < ct(0, L1), ct(0, L2), null >
      if N =/= 0 .
```

Finally, if in the medium of the current state there is a message `pair(N, X)` and the receiver's counter is `N`, we can reduce it (that is, identify it) to that in which the message has been read by the receiver.

```
eq < ct(M, L1 : X : L2), ct(N, L1), pair(N, X) ; PS > =
   < ct(M, L1 : X : L2), ct(s(N), L1 : X), PS > .
```

(The conditions on the sender are imposed so that either both states satisfy `pref`, or none does.)

It is clear, by inspection of the equations, that no state satisfying `pref` is identified with one which does not. Therefore, there is no need to modify the equations giving the semantics of `pref`. However, the resulting rewrite theory is not coherent. On the one hand, note that the last equation in the abstraction is actually a particular case of the last rewrite rule. The term

```
< ct(5, a : b : c), ct(3, a), pair(3, b) >
```

for example, can be reduced to

```
< ct(5, a : b : c), ct(3, a : b), null >
```

by applying any of those, but this term, in turn, cannot be rewritten by any rule to a term to which is provably equal, as should be the case to have coherence. To solve it, it is enough to add the following "lazy" rule:

```
rl < ct(M, L1 : X : L2), ct(s(N), L1 : X), PS > =>
   < ct(M, L1 : X : L2), ct(s(N), L1 : X), PS > .
```

On the other hand, the second equation can also raise a coherence problem. For example, a term of the form `< ct(5, L1), ct(5, L2), null >` can be rewritten to

t = `< ct(6, L1 : a), ct(5, L2), pair(5, a) >`

or reduced by the equations defining the abstraction to `< ct(0, L1), ct(0, L2), null >`. This last term itself can be rewritten to

t' = `< ct(1, L1 : a), ct(0, L2), pair(0, a) >`

Suppose now that `L1` and `L2` are equal: then, both t and t' can be reduced to

```
< ct(0, nil), ct(0, nil), null >
```

and we have coherence. This, however, is not true in general, but we can enforce it by requiring `L1` and `L2` to be `nil` in the corresponding equation.

```
ceq < ct(N, nil), ct(N, nil), null > =
    < ct(0, nil), ct(0, nil), null > if N =/= 0 .
```

This way we obtain a coherent rewrite theory which can be model checked to show that $\Box$ `pref` holds.

It is worth commenting on the previous lines. The reason why we have coherence there is because the abstraction collapses almost everything! In particular, every reachable state is reduced to

```
< ct(0, nil), ct(0, nil), null >
```

Such a general identification does not always happen, however, as shown in our next example.

6.2 Example 2: The Alternating Bit Protocol

In this section we study the correctness of the Alternating Bit Protocol [2]. The task of the Alternating Bit Protocol is to ensure that messages, sent from a sender to a receiver, are delivered in order in the presence of unreliable channels that can lose or duplicate messages. Our presentation here is adapted from [28].

The state of the system at any specific time is given by a 9-tuple

```
< next, MS, HS, MR, HR, QS, QR, LMR, LMS >
```

where

- `MS` and `MR` are the current messages the sender and the receiver, respectively, are dealing with;
- `HS` and `HR` are boolean variables used by the sender and the receiver for synchronization purposes;
- `QS` is the channel used by the sender to send the messages (together with an alternating bit);
- `QR` is the channel used by the receiver to acknowledge the reception of a message;
- the boolean variable `next` models the assumption that the environment will only send a message if requested to do so (see [28]);
- `LMR` and `LMS` keep, respectively, the lists of messages already sent and received, and are introduced only for verification purposes.

Note that infinity creeps into the protocol in two different ways. On the one hand, the message alphabet may be infinite; on the other, the channels are unbounded. As it is done in [28, 3], thanks to Wolper's data-independence results [33] we can discard the first source of infiniteness and assume a finite message alphabet (actually, with only two elements). Now, the specification in Maude of the data types involved in the protocol would be as follows.

```
fmod QBOOL is
   sort QBool .
   subsort Bool < QBool .

   op nil : -> QBool [ctor] .
   op _;_ : QBool QBool -> QBool [ctor assoc id: nil] .
endfm

fmod MESSAGE is
   sorts Message NonEMessage LMessage .
   subsort NonEMessage < Message .
   subsort NonEMessage < LMessage .

   op none : -> Message [ctor] .
   ops a b : -> NonEMessage [ctor] .
   op nil : -> LMessage [ctor] .
   op _;_ : LMessage LMessage -> LMessage [ctor assoc id: nil] .
endfm

fmod QPAIR is
   protecting MESSAGE .
   sorts Pair QPair .
   subsort Pair < QPair .

   op [_,_] : Bool Message -> Pair [ctor] .

   op nil : -> QPair [ctor] .
   op _;_ : QPair QPair -> QPair [ctor assoc id: nil] .
endfm

mod ABP is
   protecting MESSAGE .
   protecting QBOOL .
   protecting QPAIR .

   op <_,_,_,_,_,_,_,_,_> : Bool Message Bool Message Bool QPair
                            QBool LMessage LMessage -> State [ctor] .
```

The sender sends packages consisting of a boolean value and a message (a `QPair`), and the receiver acknowledgment is just a boolean. The behavior of the system is specified by twelve rules in rewriting logic. For example, if the sender buffer is currently empty, then the boolean variable `next` is set to `true` to allow the environment to output a new message.

```
   rl [3] : < NEXT, none, HS, MR, HR, QS, QR, LMS, LMR > =>
            < true, none, HS, MR, HR, QS, QR, LMS, LMR > .
```

Then, the sender can choose between inputting an `a` or a `b`.

```
   rl [1] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
```

```
          < false, a, HS, MR, HR, QS, QR, LMS, LMR > .

   rl [2] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
          < false, b, HS, MR, HR, QS, QR, LMS, LMR > .
```

The complete set of rules is as follows. Note that rule `[4]` allows for duplication of messages, and rule `[7]` for deletion.

```
   rl [1] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
          < false, a, HS, MR, HR, QS, QR, LMS, LMR > .

   rl [2] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
          < false, b, HS, MR, HR, QS, QR, LMS, LMR > .

   rl [3] : < NEXT, none, HS, MR, HR, QS, QR, LMS, LMR > =>
          < true, none, HS, MR, HR, QS, QR, LMS, LMR > .

   rl [4] : < NEXT, M, HS, MR, HR, QS, QR, LMS, LMR > =>
          < NEXT, M, HS, MR, HR, QS ; [HS, M], QR, LMS, LMR > .

   crl [5] : < NEXT, MS, HS, none, HR, [B, M] ; QS, QR, LMS, LMR > =>
          < NEXT, MS, HS, M, not(HR), [B, M] ; QS, QR, LMS, LMR >
          if (HR =/= B) .

   crl [6] : < NEXT, MS, HS, none, HR, [B, M] ; QS, QR, LMS, LMR > =>
          < NEXT, MS, HS, M, not(HR), QS, QR, LMS, LMR >
          if (HR =/= B) .

   crl [7] : < NEXT, MS, HS, MR, HR, [B, M] ; QS, QR, LMS, LMR > =>
          < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR >
          if (HR == B) or (MS =/= none) .

   rl [8] : < NEXT, MS, HS, M, HR, QS, QR, LMS, LMR > =>
          < NEXT, MS, HS, none, HR, QS, QR, LMS, LMR ; M > .

   rl [9] : < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
          < NEXT, MS, HS, MR, HR, QS, QR ; HR, LMS, LMR > .

   crl [10] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
          < NEXT, none, not(HS), MR, HR, QS, B ; QR, LMS ; MS, LMR >
          if (HS == B) .

   crl [11] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
           < NEXT, none, not(HS), MR, HR, QS, QR, MS ; LMS, LMR >
           if (HS == B) .

   crl [12] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
           < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR >
           if (HS =/= B) .
endm
```

The property we are interested in is that the messages are delivered in the correct order. For that, what we are going to check is that, at any time, either the list of sent messages is a prefix of the list of received messages, or vice versa. It is not always the case that the list of received messages is a prefix of the list of sent messages because a message is not included in this second list until the sender has received the acknowledgment. Similarly, since the receiver can send the acknowledgment before consuming the message, the other possibility does not always hold either. In the Maude model checker, this property is specified as follows.

```
mod CHECK is
   inc ABP .
   inc LTL-SIMPLIFIER .

   op init : -> State .
   op pref : -> Prop [ctor] .

   vars NEXT HS HR : Bool .
   vars MS MR : Message .
   vars LM LMR LMS : LMessage .
   var QS : QPair .
   var QR : QBool .

   eq init = < false, none, true, none, false, nil, nil, nil, nil > .

   eq (< NEXT, MS, HS, MR, HR, QS, QR, LMS ; LM, LMS > |= pref) =
      true .
   eq (< NEXT, MS, HS, MR, HR, QS, QR, LMR, LMR ; LM > |= pref) =
      true .
endm
```

This completes the specification of the protocol. We can now run the model checker on it but, since the set of reachable states is infinite (and the property is true!), it does not provide any answer.

The way out of the problem consists of simplifying the system by using a clever abstraction defined in [28]. The idea is the following: even though the channels are unbounded, their contents, as can be observed by inspection of some simple runs starting in `init`, are always of the form $m_1^* m_2^*$, with $m_1, m_2 \in \{\mathtt{a}, \mathtt{b}\}$. Hence, we can obtain an abstract system by merging adjacent equal messages. Note that we need not prove that the contents of the channels are actually of that form: this is only used as an intuition to obtain a simpler system. Also, this abstraction does not produce a finite state system: in principle, it is still possible to have infinite chains of messages of the form $m_1 m_2 m_1 m_2 \ldots$, that cannot be further reduced (abstracted). However, it turns out that the set of reachable states is finite, and that is all that the model checker needs.

We still need to abstract the lists of already sent and received messages introduced for verification purposes, and what we will do is to remove, from the front part of both lists, their common messages. The abstraction function is

then defined as follows. (Note that we are using the same sort `State` for both the concrete and the abstract systems.)

```
eq abs(< next, MS, HS, MR, HR, QS, QR, LMR, LMS >) =
       < next, MS, HS, MR, HR, reduce1(QS), reduce2(QR),
               diff(LMR, LMS), diff(LMS, LMR) > .
```

where `reduce1` is recursively defined as

```
eq reduce1(nil) = nil .
eq reduce1([B, M]) = [B, M] .
eq reduce1(P ; P ; QS) = P ; QS .
ceq reduce1(P ; Q ; QS) = P ; reduce1(Q ; QS) if P =/= Q .
```

and similarly for `reduce2`.

The definition of `diff` is

```
eq diff(nil, LMS) = nil .
eq diff(M ; LMS, nil) = M ; LMS .
eq diff(M ; LMS, M ; LMS') = diff(LMS, LMS') .
ceq diff(M ; LMS, M' ; LMS') = M ; LMS if M =/= M' .
```

If we try now to apply procedure **(A1)**, we are faced with the difficulty of not having constructor terms in the right-hand side of the definition of the abstraction function. In a situation like this, we can still squeeze some results out of the specification, at the cost of being less precise. Consider for example rule `[3]`. Applying the method of the previous section it would be transformed into

```
< NEXT, none, HS, MR, HR, reduce1(QS), reduce2(QR), diff(LMS), diff(LMR) >
=>
< true, none, HS, MR, HR, reduce1(QS), reduce2(QR), diff(LMS), diff(LMR) > .
```

which is not directly executable. However, if we are willing to lose some information, we can in turn simplify it to

```
< NEXT, none, HS, MR, HR, QS, QR, LMS, LMR >
=>
< true, none, HS, MR, HR, QS, QR, LMS, LMR > .
```

which, in this case, is actually the original rule.

A similar procedure can be applied to the other rules, and the resulting system is still sound for inferring properties of the concrete one. However, this is a gross simplification which is of no use here since, not only the number of possible transitions is much larger than should be in the minimal system, but in fact, in this case the set of reachable states remains infinite.

Therefore, we will use our method **(A2)**. In our present case, the identification imposed by the abstraction can be defined by means of the following three equations:

```
eq < NEXT, MS, HS, MR, HR, QS, QR ; B ; B ; QR', LMS, LMR > =
   < NEXT, MS, HS, MR, HR, QS, QR ; B ; QR', LMS, LMR > .
eq < NEXT, MS, HS, MR, HR, QS ; [B, M] ; [B, M] ; QS', QR, LMS, LMR > =
   < NEXT, MS, HS, MR, HR, QS ; [B, M] ; QS', QR, LMS, LMR  > .
eq < NEXT, MS, HS, MR, HR, QS, QR, M ; LMS, M ; LMR > =
   < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR > .
```

Note that the first two equations correspond to the merging of adjacent equal messages in the queues, while the last one corresponds to the removal of common already sent and received messages. Note also that, since we only have to specify those cases in which the reduction actually applies, these equations are simpler than the definitions of `reduce1`, `reduce2`, and `diff`.

It is immediate to check that if two states are identified by the above equations then either both of them satisfy the proposition `pref`, or none does, so the semantics of `pref` needs not be changed. Since the resulting system is also coherent, we can run the system in the Maude model checker and verify that the property $\Box$ `pref` indeed holds.

```
Maude> red init |= [] pref .
Result Bool: true
```

And that's all.

We can compare our treatment of this example with that of other authors. Müller and Nipkow [28] first define an abstract system and then use the abstraction function to prove that it is indeed an abstraction of ABP. In [3], like we have done, the abstract system is computed by making use of the abstraction function: there is nothing to be proved here. However, we believe that our treatment is much simpler; in particular, we do not have to consider "... a suitable proof strategy that handles the involved lists and the recursively defined functions *reduce* and *reverse*." This example is also reported (although not discussed in detail) in [11, 31], in the context of predicate abstraction, as an illustration of an automatic way of obtaining abstraction functions.

In [24], an equivalence relation between the concrete states is defined satisfying certain properties, and a representation function satisfying some others is used to extract the corresponding finite system. Compared to our approach, we find it slightly less general because equivalent states are required to satisfy the same atomic propositions, and we also think that the theorem proving needed to show that the equivalence relation and the representation function satisfy the required properties is heavier than our use of the ITP and the coherence checker.

6.3 Example 1 Revisited

The communication protocol as presented in [27] adopted a slightly different form from our presentation in Section 6.1. In [27], the sender, instead of nondeterministically choosing a message from a given set, owns a list of messages to be sent and, at any moment, it can decide to send to the medium the head

of the current list. The rules specifying the transitions in this system are (we use the same signature as in Section 6.1):

```
rl < ct(N, X : L), R, PS > => < ct(s(N), L), R, PS ; pair(N, X) > .
rl < S, ct(N, L), pair(N, X) ; PS > => < S, ct(s(N), L : X), PS > .
```

The initial state is of the form `< ct(0, L), ct(0, nil), null >`, with L the list of messages to be sent, and the property to verify, `pref(`L`)`, is that the receiver's list is at any moment a prefix of the initial list. Therefore, we have a class of systems parameterized with respect to the initial list L, each of which is finite and can be model checked after appropriate instantiation of the parameter L.

To prove that the property holds for all systems at the same time, we can show that each of these systems can be simulated by the one in Section 6.1. For that, if a state is of the form

S `= < ct(N+p+1, LS), ct(N, LR), pair(N, M0) ; ... ; pair(N+p, Mp) >`

then it is mapped to

$h(S)$ `= < ct(N+p+1, LR : M0 : ... : Mp), ct(N, LR), pair(N, M0) ; ... ; pair(N+p, Mp) >`

Otherwise, $h(S) = chaos$, where *chaos* is a new state with a transition $chaos \rightarrow chaos$, and $L(chaos) = \emptyset$.

It is not difficult to check that h so defined is a simulation.

Another possibility: add to the rules in the original system a condition imposing the requirement that the state S to be rewritten is actually a "valid" state. This way, there would be no need of introducing *chaos* (no problems of coherence appear).

6.4 Readers and Writers Revisited

To illustrate the generality of the quotient approach, let us consider again the readers-writers system presented in Section 3.1. The specification of the system is still the same, where the predicates are defined by:

```
mod R&W-CHECK is
  inc LTL-SIMPLIFIER .
  inc R&W .

  op excl : -> Prop .
  op onew : -> Prop .

  vars R W : Nat .

  eq < 0, W > |= excl = true .
  eq < R, 0 > |= excl = true .
  ceq < R, W > |= onew = true if (W < s s 0) .
endm
```

Now, in order to get the second abstraction discussed in Section 3.1, it is enough to add the following equations, which trivially preserve the atomic propositions.

```
  eq < 0, s s s W > = < 0, s s 0 > .
  eq < s s R, 0 > = < s 0, 0 > .
  eq < s s R, s 0 > = < s 0, s 0 > .
  eq < s s R, s s 0 > = < s 0, s s 0 > .
```

6.5 The Bakery Protocol Revisited

Similarly, the bakery protocol in Section 5.1 can be handled with our method **(A2)**. Recall how the protocol was specified there

```
mod BAKERY is
  inc MODEL-CHECKER .
  protecting NAT .

  sorts PC .

  ops sleep wait crit : -> PC [ctor] .
  op <_,_,_,_> : PC Nat PC Nat -> State [ctor] .
  op initial : -> State .

  vars P Q : PC .
  vars X Y : Nat .

  eq initial = < sleep, 0, sleep, 0 > .

  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y > if not (Y < X) .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .

  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y > if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm

mod BAKERY-CHECK is
  inc LTL-SIMPLIFIER .
  inc BAKERY .

  op excl : -> Prop [ctor] .

  vars P Q : PC .
  vars X Y : Nat .

  ceq (< P, X, Q, Y > |= excl) = true if (P =/= crit) or (Q =/= crit) .
```

```
endm
```

and the abstraction which was used:

$$abs(\langle P, X, Q, Y\rangle) = \langle P, Q, X = 0, Y = 0, Y < X\rangle)\,.$$

Let us now characterize all the states that are identified by this function.

First of all, a state in which both `X` and `Y` are equal to `0` is identified with no other state. However, all states in which only `X` or `Y` is equal to `0` are mapped to the same abstract state. In the first case we can consider `< P, 0, Q, s(0) >` to be a representative of that class, and for the second we can choose `< P, s 0, Q, 0 >`. Therefore, we can add the following two equations to our module:

```
  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
```

Finally, when none of `X` and `Y` is `0`, states become identified depending on which one of those values is greater.

```
  ceq < P, s X, Q, s Y > = < P, s s 0, Q, s 0 >
      if (Y < X) /\ not(Y == 0 and X == s 0) .
  ceq < P, s X, Q, s Y > = < P, s 0 , Q, s 0 >
      if not (Y < X) /\ not (Y == 0 and X == 0) .
```

It is clear that only states satisfying the same atomic propositions are identified, so there is no need to modify the labelling function. This can be mechanically proved with the ITP. For example, to show that the third equation preserves `1wait` the following commands can be used:

```
|-ind {P ; Q ; X ; Y}( ((Y < X) = true) =>
                        (((< P, (s (s 0)), Q, (s 0) > |= 1wait) = true) =>
                         ((< P, (s X), Q, (s Y) > |= 1wait) = true)) ) .)

(ctor-split (1) on P .)

--- case P = split
(all (1 . 1) .)
(cns (1 . 1) .)
(imp (1 . 1) .)
(imp (1 . 1) .)
--- --- (show (1 . 1) .)
(simp 9 in (1 . 1) .)
--- --- (show (1 . 1) .)
(cnt 10 in (1 . 1) .)

--- case P = crit
(all (1 . 3) .)
(cns (1 . 3) .)
(imp (1 . 3) .)
(imp (1 . 3) .)
```

```
--- --- (show (1 . 3) .)
(simp 9 in (1 . 3) .)
--- --- (show (1 . 3) .)
(cnt 10 in (1 . 3) .)

--- case P = wait
(all (1 . 2) .)
(cns (1 . 2) .)
(imp (1 . 2) .)
(imp (1 . 2) .)
(rwr (1 . 2) .)
(idt (1 . 2) .)
```

What about coherence? Consider the first equation and the rule `[p1_sleep]`. Only terms of the form `< sleep, 0, Q, s s Y >` can be reduced by both of them, to get $t_1 =$ `< sleep, 0, Q, s 0 >` in the first case, and $t_2 =$ `< wait, s s s Y, Q, s s Y >` in the second. Now, `[p1_sleep]` can be applied to t_1 to get $t_3 =$ `< wait, s s 0, Q, s 0 >`, that is the same term to which t_2 reduces applying the equations of the quotient. Therefore, no incoherence arises here. The same process can be repeated for all pairs of equations and rules to show that the specification is indeed coherent. (Note that, actually, there is no need to consider the `[wait]` rules because the values of `X` and `Y`, which are the ones affected by the abstraction, are not modified by them.)

We are left with checking that there are no deadlocks in the system. For that we specify an *enabled* predicate as explained in Section 7 below, that returns *true* when applied to a term if and only if that term represents a non-deadlocked state. In our case, we add the following equations:

```
  eq enabled(< sleep, X, Q, Y >) = true .
  eq enabled(< wait, X, Q, 0 >) = true .
  ceq enabled(< wait, X, Q, Y >) = true if not (Y < X) .
  eq enabled(< crit, X, Q, Y >) = true .
  eq enabled(< P, X, sleep, Y >) = true .
  eq enabled(< P, 0, wait, Y >) = true .
  ceq enabled(< P, X, wait, Y >) = true if Y < X .
  eq enabled(< P, X, crit, Y >) = true .
```

and the equation we have to prove is

```
|-ind {S}(enabled(S) = true) .
```

The proof is by induction with no particular difficulties. Alternatively, we could also prove the result in a more automated way by using a completeness checker recently written by Joe Hendrix [17]. This tool is given a module and it checks whether it is complete, in the intuitive sense that enough equations are given so that every term can be reduced to a canonical form in which only constructor operators are used. In our case the tool returns that the module is complete which means, in particular, that all terms of the form *enabled*(t) can be reduced

to a canonical term in the sort *Bool* and, due to the equations used, this term must be *true* as required.

Finally, there is nothing else to worry about and we can model check the desired property in the quotient theory.

```
Maude> red initial |= [] excl .
reduce in BAKERY-CHECK : initial |= []excl .
rewrites: 190 in 10ms cpu (11ms real) (19000 rewrites/second)
result Bool: true
```

7 The Deadlock Difficulty

The reason why deadlock can pose a problem is a subtle point in the semantics of LTL that, so far, has not been taken into account. As emphasized in its definition, the transition relation of a Kripke structure is total, and this is a requirement that is also imposed in the Kripke structures arising from rewrite theories by means of idle transitions of the form $[t] \rightarrow [t]$ when no other rewrite is possible starting in $[t]$. Consider then the following specification of a rewrite theory, together with the declaration of two state predicates.

```
mod FOO is
  inc MODEL-CHECKER .
  ops a b c : -> State [ctor] .
  ops p1 p2 : -> Prop [ctor] .

  eq (a |= p1) = true .
  eq (b |= p2) = true .
  eq (c |= p1) = true .

  rl a => b .
  rl b => c .
endm
```

The transition relation of the Kripke structure corresponding to this specification has *three* elements: `a` $\rightarrow$ `b`, `b` $\rightarrow$ `c`, and `c` $\rightarrow$ `c`, the last one consistently added by the model checker according to the semantics given to LTL.

Suppose then that we wanted to abstract this system (although, of course, in this case it would not be necessary since the original system is already finite, and very small), and that we decided to identify `a` and `c` by means of a simulation h. For that, according to the previous sections, it would be enough to add the equation

```
  eq c = a .
```

to the specification. The resulting system is coherent, and `a` and `c` satisfy the same state predicates.

Note that the resulting Kripke structure has only two elements in its transition relation: one from the equivalence class of `a` to that of `b`, and another in the

opposite direction. Now, since no deadlock can occur in any of the states, the model checker does not add any additional transition steps. In particular, there is no transition from `a` to itself, but that means that the resulting specification does *not* correspond to the minimal system associated to h, in which such a transition exists. Is the lack of this idle transition a serious issue? Yes, because now we can prove properties about the "abstract" system that are actually false in the original one, e.g., $\Box \Diamond$ `p2`. (Note that this formula does not hold in the real minimal system, either.)

A first, simple way, to deal with this difficulty would be just to add to the specifications resulting from applying either of our two techniques idle transitions for each of the states in the resulting specification by means of a rule of the form `x => x`. This means that the resulting system, in addition to all the rules that the minimal system should contain, may in fact have some extra "junk" rules that are not part of it. Therefore, we end up with a system that can be soundly used to infer properties of the original system (it is immediate to see that we have a simulation map) but that, in general, will be coarser than the minimal system. Continuing with our previous example, the formula $\Box \Diamond$ `p1` holds both in the concrete and in the minimal system. It does not hold, however, for the system with transition relation $\{\mathtt{a} \to \mathtt{b}, \mathtt{b} \to \mathtt{a}, \mathtt{a} \to \mathtt{a}, \mathtt{b} \to \mathtt{b}\}$ that results from explicitly adding an idle transition to every state. Nonetheless, as far as all our previous examples are concerned, all the properties we checked can still be proved after the addition of the identity rule.

A better way of addressing the problem would be to characterize the set of deadlock states. For that, given a rewrite theory $\mathcal{R}$ we declare an operation $\mathit{enabled} : k \longrightarrow [\mathit{Bool}]$ for each kind k in $\mathcal{R}$. Then we add, for each rule $t \to t'$ if C, the equation $\mathit{enabled}(t) = \mathit{true}$ if C and, for each operation $f : k_1 \dots k_n \longrightarrow k$, n equations of the form $\mathit{enabled}(f(x_1, \dots, x_n)) = \mathit{true}$ if $\mathit{enabled}(x_i) = \mathit{true}$, so that $(\exists t')\ t \to^1_{\mathcal{R}} t' \iff \mathit{enabled}(t) = \mathit{true}$. This *enabled* predicate is the key point in the proof of the following proposition, which allows us to transform an executable rewrite theory into a semantically equivalent one that is deadlock-free and executable.

Proposition 4 *Let $\mathcal{R} = (\Omega, E, \phi, R)$ be a rewrite theory whose equations are ground confluent, sort-decreasing, and terminating, and whose rules contain only equational conditions and are ground coherent relative to E. Given a chosen kind of states k, a theory extension $\mathcal{R} \subseteq \mathcal{R}^k_{d.f.} = (\Omega', E', \phi', R')$ satisfying the same conditions can be constructed such that:*

- *$\mathcal{R}^k_{d.f.}$ is k'-deadlock free for a certain kind k';*
- *there is a function $h : T_{\Omega', k'} \longrightarrow T_{\Omega, k}$ inducing a bijection $h : T_{\Omega'/E', k'} \longrightarrow T_{\Omega/E, k}$ such that for each $t, t' \in T_{\Omega', k'}$ we have*

$$h(t)(\to^1_{\mathcal{R}})^\bullet h(t') \iff t \to^1_{\mathcal{R}^k_{d.f.}} t'.$$

Furthermore, if Π are state predicates for $\mathcal{R}$ on the kind k defined by equations D, then one can define state predicates Π for $\mathcal{R}^k_{d.f.}$ on the kind k' by equations

D' such that the above map h becomes a bijective bisimulation

$$h : \mathcal{K}(\mathcal{R}^k_{d.f.}, k')_\Pi \longrightarrow \mathcal{K}(\mathcal{R}, k)_\Pi \,.$$

Proof Let us prove the first two points. For that, declare a new kind k', a new operation $\{\_\} : k \longrightarrow k'$, and add the rule

```
crl {X} => {X} if enabled(X) =/= true .
```

to R: the resulting rewrite theory $\mathcal{R}^k_{d.f.}$ is k'-deadlock free. Given a term $\{t\}$ with t of kind k, if there is t' in $\mathcal{R}$ such that $t \rightarrow^1_{\mathcal{R}} t'$ then $\{t\} \rightarrow^1_{\mathcal{R}^k_{d.f.}} \{t'\}$; otherwise, $\mathit{enabled}(t) = /= \mathit{true}$ and, by the rule we have just added, $\{t\} \rightarrow^1_{\mathcal{R}^k_{d.f.}} \{t\}$. We can then define $h(\{t\}) = t$ which, since no equations have been introduced, induces a bijection, and clearly satisfies the equivalence in the second point.

Regarding the state predicates, transform each equation $t \models p(u_1, \ldots, u_n) = \mathit{true} \Leftarrow C$ into $\{t\} \models p(u_1, \ldots, u_n) = \mathit{true} \Leftarrow C$. This implies that $L_\Pi(\{t\}) = L_\Pi(t)$ and, together with the previous results, that h is a strict bisimulation. □

8 All Together: The Bounded Retransmission Protocol

The Bounded Retransmission Protocol (BRP) [16, 13] is an extension of the ABP where a limit is placed on the number of transmissions of the messages. The following description is borrowed from [1].

At the sender side, the protocol requests a sequence of data $s = d_1, \ldots, d_n$ (action REQ) and communicates a confirmation which can be either SOK, SNOK, or SDNK. The confirmation SOK means that the file has been transferred successfully, SNOK means that the file has not been transferred completely, and SDNK means that the file may not have been transferred completely. This occurs when the last datum d_n is sent but not acknowledged.

Now, at the receiver side, the protocol delivers each correctly received datum with an indication which can be RFST, RINC, ROK, or RNOK. The indication RFST means that the delivered datum is the first one and more data will follow, RINC means that the datum is an intermediate one, and ROK means that this was the last datum and the file is completed. However, when the connection with the sender is broken, an indication RNOK is delivered (without datum).

In Maude, the corresponding declarations are as follows (ignore for the moment the sort `Label` and its operations).

```
fmod DATA is
  sorts Sender Receiver .
  sort Label .
  sorts Msg MsgL .
  subsort Msg < MsgL .
```

```
  ops 0s 1s 2s 3s 4s 5s 6s 7s : -> Sender [ctor] .
  ops 0r 1r 2r 3r 4r : -> Receiver [ctor] .

  ops none req snok sok sdnk rfst rnok rinc rok : -> Label [ctor] .

  ops 0 1 fst last : -> Msg [ctor] .
  op nil : -> MsgL [ctor] .
  op _;_ : MsgL MsgL -> MsgL [ctor assoc id: nil] .
endfm
```

The properties the service should satisfy are the following:

1. a request REQ must be followed by a confirmation (SOK, SNOK, or SDNK) before the next request;
2. an RFST indication must be followed by one of the two indications ROK or RNOK before the beginning of a new transmission (new request of a sender);
3. an SOK confirmation must be preceded by an ROK indication;
4. an RNOK indication must be preceded by an SNOK or SDNK confirmation (abortion).

The BRP is modelled in [1], after some simplifications to make the system *untimed*, as a lossy channel system. Our following Maude specification is adapted from theirs. There is however, a minor addition that has to do with the new sort `Label`: we add a new component to the constructor representing the state of the system, of sort `Label`, to keep track of the name of the last transition used to reach the current state (hence the name of the operations `req`, `snok`, `sok`, ...). We only make explicit the name of these transitions for the cases we are interested in (namely, those required by the properties); in the rest of the cases, `none` is used.

```
mod BRP is
   protecting DATA .
   inc MODEL-CHECKER .

   op <_,_,_,_,_,_,_> : Sender Receiver Bool Bool MsgL MsgL Label
                        -> State [ctor] .

   var S : Sender .
   var R : Receiver .
   var M : Msg .
   vars K L KL : MsgL .
   vars A RT : Bool .
   var LA : Label .

   rl [REQ] : < 0s, R, A, false, nil, nil, LA > =>
              < 1s, R, false, false, nil, nil, req > .
```

```
rl [K!fst] : < 1s, R, A, RT, K, L, LA > =>
             < 2s, R, A, RT, K ; fst, L, none > .
rl [K!fst] : < 2s, R, A, RT, K, L, LA > =>
             < 2s, R, A, RT, K ; fst, L, none > .
rl [L?fst] : < 2s, R, A, RT, K, fst ; L, LA > =>
             < 3s, R, A, RT, K, L, none > .
crl [L?-fst] : < 2s, R, A, RT, K, M ; L, LA > =>
               < 2s, R, A, RT, K, L, none > if M =/= fst .
rl [K!0] : < 3s, R, A, RT, K, L, LA > =>
           < 4s, R, A, RT, K ; 0, L, none > .
rl [K!last] : < 3s, R, A, RT, K, L, LA > =>
              < 7s, R, A, RT, K ; last, L, none > .
rl [K!0] : < 4s, R, A, RT, K, L, LA > =>
           < 4s, R, A, RT, K ; 0, L, none > .
crl [L?-0] : < 4s, R, A, RT, K, M ; L, LA > =>
             < 4s, R, A, RT, K, L, none > if M =/= 0 .
rl [L?0] : < 4s, R, A, RT, K, 0 ; L, LA > =>
           < 5s, R, A, RT, K, L, none > .
rl [SNOK] : < 4s, R, A, RT, K, nil, LA > =>
            < 0s, R, true, RT, K, nil, snok > .
rl [K!1] : < 5s, R, A, RT, K, L, LA > =>
           < 6s, R, A, RT, K ; 1, L, none > .
rl [K!last] : < 5s, R, A, RT, K, L, LA > =>
              < 7s, R, A, RT, K ; last, L, none > .
rl [K!1] : < 6s, R, A, RT, K, L, LA > =>
           < 6s, R, A, RT, K ; 1, L, none > .
crl [L?-1] : < 6s, R, A, RT, K, M ; L, LA > =>
             < 6s, R, A, RT, K, L, none > if M =/= 1 .
rl [SNOK] : < 6s, R, A, RT, K, nil, LA > =>
            < 0s, R, true, RT, K, nil, snok > .
rl [K!last] : < 7s, R, A, RT, K, L, LA > =>
              < 7s, R, A, RT, K ; last, L, none > .
crl [L?-last] : < 7s, R, A, RT, K, M ; L, LA > =>
                < 7s, R, A, RT, K, L, none > if M =/= last .
rl [SOK] : < 7s, R, A, RT, K, last ; L, LA > =>
           < 0s, R, A, RT, K, L, sok > .
rl [SDNK] : < 7s, R, A, RT, K, nil, LA > =>
            < 0s, R, true, RT, K, nil, sdnk > .

rl [RFST] : < S, 0r, false, RT, fst ; K, L, LA > =>
            < S, 1r, false, true, K, L ; fst, rfst > .
rl [K?fstL!fst] : < S, 1r, A, RT, fst ; K, L, LA > =>
                  < S, 1r, A, RT, K, L ; fst, none > .
rl [RNOK] : < S, 1r, true, RT, nil, L, LA > =>
            < S, 1r, true, false, nil, L, rnok > .
rl [RINC] : < S, 1r, false, RT, 0 ; K, L, LA > =>
            < S, 2r, false, RT, K, L ; 0, rinc > .
rl [ROK] : < S, 1r, false, RT, last ; K, L, LA > =>
           < S, 4r, false, RT, K, L ; last, rok > .
rl [K?0L!0] : < S, 2r, A, RT, 0 ; K, L, LA > =>
```

```
                < S, 2r, A, RT, K, L ; 0, none > .
  rl [RINC] : < S, 2r, false, RT, 1 ; K, L, LA > =>
              < S, 3r, false, true, K, L ; 1, rinc > .
  rl [RNOK] : < S, 2r, true, RT, nil, L, LA > =>
              < S, 0r, true, false, nil, L, rnok > .
  rl [ROK] : < S, 2r, false, RT, last ; K, L, LA > =>
             < S, 4r, false, RT, K, L ; last, rok > .
  rl [RINC] : < S, 3r, false, RT, 0 ; K, L, LA > =>
              < S, 2r, false, RT, K, L ; 0, rinc > .
  rl [K?1L!1] : < S, 3r, A, RT, 1 ; K, L, LA > =>
                < S, 3r, A, RT, K, L ; 1, none > .
  rl [ROK] : < S, 3r, false, RT, last ; K, L, LA > =>
             < S, 4r, false, RT, K, L ; last, rok > .
  rl [RNOK] : < S, 3r, true, RT, nil, L, LA > =>
              < S, 0r, true, false, nil, L, rnok > .
  rl [K?lastL!last] : < S, 4r, A, RT, last ; K, L, LA > =>
                      < S, 4r, A, RT, K, L ; last, none > .
  rl [empty] : < S, 4r, A, RT, last ; K, L, LA > =>
               < S, 0r, A, false, nil, L, none > .
endm
```

All the properties the system should satisfy impose requirements of the form that certain transitions happen before certain other transitions. To formulate them, we declare a parametric atomic proposition, `tr(`L`)`, that is true in those states resulting from the application of a transition named L.

```
mod BRP-CHECK is
  inc BRP .
  inc LTL-SIMPLIFIER .

  op tr : Label -> Prop [ctor] .

  var S : Sender .   var R : Receiver .
  var M : Msg .      vars K L : MsgL .
  vars A RT : Bool . vars LA : Label .

  eq (< S, R, A, RT, K, L, LA > |= tr(LA)) = true .
endm
```

The required properties can then be expressed in Maude as

1. `[](tr(req) -> o (~ tr(req) W (tr(sok) \/ tr(snok) \/ tr(sdnk))));`
2. `[](tr(rfst) -> (~ tr(req) W (tr(rok) \/ tr(rnok))));`
3. `[](tr(req) -> (~ tr(sok) W tr(rok)));`
4. `[](tr(req) -> (~ tr(rnok) W (tr(snok) \/ tr(sdnk)))).`

Note that both negations and implications appear in these formulas. Therefore, for Theorem 1 to apply, we must ensure that the abstraction we define preserves the atomic propositions.

As in the case of the ABP, the system is infinite but, exactly in the same way as before, the contents of the channels are of the form $m_1^* m_2^*$, where m_1, m_2 now range over $\{\texttt{first}, \texttt{last}, \texttt{0}, \texttt{1}\}$. Therefore, we can use the same idea of merging adjacent equal messages, which can be specified by the following two equations.

```
eq < S, R, A, RT, KL ; M ; M ; K, L, LA > =
   < S, R, A, RT, KL ; M ; K, L, LA > .
eq < S, R, A, RT, K, KL ; M ; M ; L, LA > =
   < S, R, A, RT, K, KL ; M ; L, LA > .
```

Again, it is immediate to check, since the abstraction does not affect the label of a state, that only states satisfying the same atomic propositions are identified. We therefore meet the requirements of Theorem 1.

What about the deadlock difficulty? By inspection of the left-hand sides of the rules, it is easy to see that the equation

```
enabled(< S, R, A, RT, KL ; M ; K, L, LA >) = true
```

does not hold (consider the case in which `S` is `0s`), so that the rule

```
rl < S, R, A, RT, KL ; M ; K, L, LA > =>
   < S, R, A, RT, KL ; M ; K, L, LA > .
```

should be added; similarly for the second equation defining the abstraction. Notice that this is not the best we can do. By direct inspection of the rules, it is easy to check that, except for the case in which `S` is `0s`, all terms of those forms are enabled. Hence, it is only necessary to add the rules

```
rl [deadlock] : < 0s, R, A, RT, KL ; M ; K, L, LA > =>
                < 0s, R, A, RT, KL ; M ; K , L, LA > .
rl [deadlock] : < 0s, R, A, RT, K, KL ; M ; L, LA > =>
                < 0s, R, A, RT, K , KL ; M ; L, LA > .
```

Finally, the last proof obligation to check is that of coherence, and this, too, happens to fail. Consider for example the term

```
< 2s, 0r, true, true, nil, fst ; fst, none >
```

This term can be rewritten using the first of the `[L?fst]` rules to

t = `< 3s, 0r, true, true, nil, fst, none >`

However, if we had first reduced it using the equations, we would have got

```
< 2s, 0r, true, true, nil, fst, none >
```

which can no longer be rewritten to t, or any term provably equal to it (an extra message `fst` has been consumed following this way). To fix this problem, the following rule must be added:

```
rl [L?fst'] : < 2s, R, A, RT, K, fst ; L, LA > =>
              < 3s, R, A, RT, K, fst ; L, none > .
```

Note that this rule does not raise a new coherence problem.

The same situation occurs with all those other rules in which a message is removed from one of the lists: the solution is the same in all cases.

We can then model check the abstract system and see that all the properties hold in it; since all of our proof obligations are fulfilled, we can soundly infer that they hold in the concrete system, too.

9 Model Checking with Fairness

In this section we explain how we can recycle some of the ideas presented in the previous sections to perform model checking with fairness constraints. First, we recall some basic definitions [23].

Definition 5 *Given a Kripke structure $M = (S, \rightarrow, L)$, a transition $(a, b) \in \rightarrow$ is enabled in a path π at position k if $\pi(k) = a$.*

Definition 6 *A path π is* just *with respect to a certain transition if it is not the case that the transition is continually enabled beyond some point $k \in \mathbb{N}$ without being taken beyond k.*

Definition 7 *A path π is* compassionate *with respect to a certain transition if it is not the case that the transition is enabled infinitely many times but not taken beyond a certain $k \in \mathbb{N}$.*

There exist algorithms for model checking of formulas with fairness constraints, but the current implementation of the Maude model checker does not allow this. However, thanks to the expressiveness of rewriting logic, it is very easy, by slightly modifying the specifications, to take those constraints into account.

The first idea was illustrated in the BRP example: Add an extra component to the constructor `<...>` of the sort `State` to keep track of the name of the transition that gave rise to the current state.

The second one (in a certain sense, already presented in Section 7) consists of introducing a parameterized atomic proposition `t_enabled`(L) that is true in those states for which the transition L is enabled, and another one `taken`(L), which should be true in those states resulting from the application of transition L. Then, assuming that the label component has been added as the last argument of the operation `<...>`, the semantics of this proposition would be specified by an equation of the form

```
eq (< ..., L > |= taken(L)) = true .
```

Then, to check a property φ under a requirement of justice with respect to a transition L, it is enough to check $\psi_1 \Rightarrow \varphi$, where

$$\psi_1 = \Diamond \Box \, \texttt{t\_enabled}(L) \Rightarrow \Box \Diamond \, \texttt{taken}(L) \, .$$

Similarly, to check a property φ under a compassion constraint, it is enough to check $\psi_2 \Rightarrow \varphi$, with

$$\psi_2 = \Box \Diamond \, \texttt{t\_enabled}(L) \Rightarrow \Box \Diamond \, \texttt{taken}(L) \, .$$

9.1 Example: Mutual Exclusion by Semaphores

We illustrate the use of fairness constraints in the Maude model checker with a simple example taken from [18]. Consider the program $\|_{i=1}^{n} P[i]$, where each $P[i]$ is of the form

loop forever do
N_i : **NonCritical**
T_i : **request** y
C_i : **Critical; release** y

with y a variable global to all the processes. The semantics, allowing for idle transitions, is the intuitive one, and in [18] the compassionate requirement that if the transition from T_i to C_i is infinitely enabled, then it is eventually taken, together with the justice requirement that no process can remain stuck forever at a location, are also imposed. As done there, we will be interested in showing the following liveness property for the first process: $\Box(T_1 \rightarrow \Diamond C_1)$.

In Maude, we can specify the system as follows:

```
mod MUTEX is inc MODEL-CHECKER .
  sorts PC Soup .
  subsort PC < Soup .

  ops n t c : -> PC [ctor] .
  op null : -> Soup [ctor] .
  op __ : Soup Soup -> Soup [ctor assoc comm id: null] .

  op <_,_|_> : Bool PC Soup -> State [ctor] .
  op initial : -> State .

  var B : Bool .    var P : PC .
  var S : Soup .    var ST : State .

  rl < B, P | n S > => < B, P | t S > .
  rl < true, P | t S > => < false, P | c S > .
  rl < false, P | c S > => < true, P | n S > .
  rl < B, n | S > => < B, t | S > .
  rl < true, t | S > => < false, c | S > .
  rl < false, c | S > => < true, n | S > .
  rl ST => ST .
```

```
endm

mod MUTEX-CHECK is inc MUTEX .
  inc LTL-SIMPLIFIER .

  op pc : PC -> Prop [ctor] .
  op critR : -> Prop [ctor] .

  var B : Bool .  var P : PC .
  var S : Soup .

  eq (< B, P | S > |= pc(P)) = true .
  eq (< B, P | c S > |= critR) = true .
endm
```

Note that we have assigned a distinguished role to the first process.

These modules specify, in a certain sense, the program $\|_{i=1}^{n} P[i]$ for all possible values of n. The initial state has been left unspecified; it will be of the form `< true, n | n ... n >`, with the number of occurrences of `n` determining the number of total processes in the system.

According to the discussion in the previous section, we should have added an additional component of sort `Label` to the constructor of the sort `State` if we want to deal with fairness constraints. This is true, but in these modules the role of the sort `Label` in the general case can be played by the sort `PC` of the second component, and `Soup` of the third. For that, the following lines should be added to the module `MUTEX-CHECK`:

```
  ops enabled taken : -> Prop [ctor] .
  ops compassion justice : -> Formula .

  eq compassion = []<> enabled -> []<> taken .
  eq justice = ~ <> [] pc(c) /\ ~ <> [] critR .

  eq (< true, t | S > |= enabled) = true .
  eq (< B, c | S > |= taken) = true .
```

Now, the property can be proved, for each particular value of `initial` with the Maude model checker.

```
Maude> red initial |= (compassion /\ justice) -> [](pc(t) -> pc(c)) .
result Bool: true
```

Actually, we could think of proving the result for all different values of `initial` in just one blow. For that, note that the only thing that matters to the first process is whether any of the other processes is in the critical section or not. Therefore, we can define the following abstraction

```
  eq < B, P | c Q S > = < B, P | c > .
  ceq < B, P | Q Q' S > = < B, P | n > if not(c in (Q Q' S)) .
```

where `_in_` is an auxiliary function that checks for membership of an element to a multiset.

It is clear that the abstraction preserves the atomic predicates and, because of the rule `ST => ST` in the original module, there is no need to worry about the deadlock difficulty. The resulting specification is not coherent, however, and to make it so the following two rules must be added:

```
rl < B, P | c > => < B, P | c > .
rl < B, P | n > => < B, P | c > .
```

Now, by noticing that any initial state is transformed by the reduction to the state `< true, n | n >`, the parameterized system can be model checked once and for all with

```
Maude> red < true, n | n > |= (compassion /\ justice) -> [](pc(t) -> pc(c)) .
result Bool: true
```

Acknowledgments. We warmly thank Saddek Bensalem, Yassine Lakhnech, David Basin, Felix Klaedtke, Natarajan Shankar, Hassen Saidi, and Tomás Uribe for many useful discussions that have influenced the ideas presented here.

References

[1] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction of Analysis of Systems, 5th International Conference, TACAS'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[2] K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261, 1969.

[3] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 319–331. Springer-Verlag, 1998.

[4] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification. 12th International Conference, CAV 2000 Chicago, IL, USA, July 15-19, 2000 Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[6] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[8] M. Clavel, F. Durán, S. Ecker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[9] Manuel Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[10] Manuel Clavel. The ITP tool. In A. Nepomuceno, J. F. Quesada, and F. J. Salguero, editors, *Logic, Language, and Information. Proceedings of the First Workshop on Logic and Language*, pages 55–62. Kronos, 2001.

[11] Michael A. Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.

[12] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291, 1997.

[13] P. R. D'Argenio, J. P. Katoen, T. Ruys, and G. T. Tretmans. The bounded retransmission protocol must be on time. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems Third International Workshop, TACAS'97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–432. Springer-Verlag, 1997.

[14] Francisco Durán. Coherence checker and completion tools for Maude specifications. `http://maude.cs.uiuc.edu/tools`, 2000.

[15] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In Fabio Gadducci and Ugo Montanari, editors, *Fourth International Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[16] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In M.-C. Gaudel and J. Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods. Third International Symposium of Formal Methods Europe Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18 - 22, 1996. Proceedings.*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, March 1996.

[17] Joe Hendrix. A completeness checker for maude. Manuscript, University of Illinois at Urbana-Champaign, 2003.

[18] Yonit Kesten and Amir Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 4(2):328–342, 2000.

[19] Yonit Kesten and Amir Pnueli. Verification by augmentary finitary abstraction. *Information and Computation*, 163:203–243, 2000.

[20] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[21] Leslie Lamport. The synchronization of independent processes. *Acta Informatica*, 7(1):15–34, 1976.

[22] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–36, 1995.

[23] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification.* Springer-Verlag, 1992.

[24] Panagiotis Manolios. *Mechanical Verification of Reactive Systems.* PhD thesis, University of Texas at Austin, August 2001.

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

[26] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[27] José Meseguer. Lecture notes for CS376. Computer Science Department, University of Illinois at Urbana-Champaign, `http://www-courses.cs.uiuc.edu/~cs376/`, 2002.

[28] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O-automata. In Ed Brinksma, W. Rance Cleaveland, Kim G. Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *Tools and Algorithms for*

the Construction and Analysis of Systems. First International Workshop, TACAS '95, Aarhus, Denmark, May 19 – 20, 1995. Selected Papers, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1995.

[29] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS)*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.

[30] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification. 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, 1999.

[31] Tomás E. Uribe Restrepo. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Department of Computer Science, Stanford University, December 1998.

[32] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2), August 2002.

[33] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193. ACM Press, 1986.