Equational Abstractions*

José Meseguer[1], Miguel Palomino[2], and Narciso Martí-Oliet[2]

[1] Computer Science Department, University of Illinois at Urbana-Champaign
[2] Departamento de Sistemas Informáticos, Universidad Complutense de Madrid
meseguer@cs.uiuc.edu {miguelpt,narciso}@sip.ucm.es

Abstract. Abstraction reduces the problem of whether an infinite state system satisfies a temporal logic property to model checking that property on a finite state abstract version. The most common abstractions are quotients of the original system. We present a simple method of defining quotient abstractions by means of equations collapsing the set of states. Our method yields the minimal quotient system together with a set of proof obligations that guarantee its executability and can be discharged with tools such as those in the Maude formal environment.

1 Introduction

Abstraction techniques (see for example [1, 2, 8–10, 14, 16, 24, 26, 27, 29, 30, 36, 38, 39]) allow reducing the problem of whether an infinite state system, or a finite but too large one, satisfies a temporal logic property to model checking that property on a finite state abstract version. The most common way of defining such abstractions is by defining a *quotient* of the original system's set of states, together with abstract versions of the transitions and the predicates. Many methods differ in their details but agree on their general use of a quotient map. There is always a minimal system (Kripke structure) making this quotient map a simulation.

We present a simple method to build minimal quotient abstractions in an equational way. The method assumes that the concurrent system has been specified by means of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with (Σ, E) an equational theory specifying the set of states as an algebraic data type, and R specifying the system's transitions as a set of rewrite rules. The method consists in adding more equations, say E', to get a quotient system specified by the rewrite theory $\mathcal{R}/E' = (\Sigma, E \cup E', R)$. We call such a system an *equational abstraction* of $\mathcal{R}$. This equational abstraction is useful for model checking purposes if:

1. $\mathcal{R}/E'$ is an *executable* rewrite theory in an appropriate sense; and
2. the state predicates are *preserved* by the quotient simulation.

Requirements 1 and 2 are *proof obligations* that can be discharged by theorem proving methods.

Our approach can be mechanized using the rewriting logic language Maude [11, 12] and its associated LTL model checker [22], inductive theorem prover [13], Church-Rosser checker [19], termination tool [21], coherence checker [20], and sufficient completeness checker [25]. Our present experience with case studies, involving different abstractions discussed in the literature, suggests a fairly wide applicability for this method.

After summarizing prerequisites on Kripke structures and linear temporal logic (LTL) in Section 2 and discussing simulations in Section 3, we explain in Section 4 how a concurrent

* This work is an extended and revised version of a paper presented at CADE-19. Research supported by ONR Grant N00014-02-1-0715, NSF Grant CCR-0234524, and by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130; by the Spanish projects MIDAS TIC 2003-0100 and DESAFIOS TIN2006-15660-C02-01, and by Comunidad de Madrid program PROMESAS S-0505/TIC/0407.

system specified by a rewrite theory $\mathcal{R}$ has an associated Kripke structure giving semantics to its LTL properties; we also explain how Maude can model check such LTL properties for initial states from which finitely many states are reachable. Equational abstractions and their associated proof methods are discussed in Sections 5 and 6. Section 7 presents some case studies, and Section 8 discusses related work and future research. A more complex example is presented in Appendix A; more details about a collection of case studies using this method can be found in [35].

2 Prerequisites on Kripke Structures and LTL

To specify the properties of interest about our systems we will use *linear temporal logic*,[3] which is interpreted in a standard way in Kripke structures. In what follows, we assume a fixed set of atomic propositions AP.

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

We will use the notation $a \rightarrow_{\mathcal{A}} b$ to say that $(a, b) \in \rightarrow_{\mathcal{A}}$. Note that the transition relation must be *total*, that is, for each $a \in A$ there is a $b \in A$ such that $a \rightarrow_{\mathcal{A}} b$. Given an arbitrary relation $\rightarrow$, we write $\rightarrow^{\bullet}$ for the total relation that extends $\rightarrow$ by adding a pair $a \rightarrow^{\bullet} a$ for each a such that there is no b with $a \rightarrow b$. A *path* in a Kripke structure $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ such that, for each $i \in \mathbb{N}$, $\pi(i) \rightarrow_{\mathcal{A}} \pi(i+1)$. We use π^i to refer to the suffix of π starting at $\pi(i)$; explicitly, $\pi^i(n) = \pi(i+n)$.

The syntax of LTL(AP) is given by the following grammar:

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

The semantics of LTL(AP) is defined as follows. Given a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and an element $a \in A$,

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

where the satisfaction relation $\mathcal{A}, \pi \models \varphi$ is defined by structural induction on φ:

$$\begin{array}{lll}
\mathcal{A}, \pi \models p & \iff & p \in L(\pi(0)) \\
\mathcal{A}, \pi \models \top & \iff & \mathit{true} \\
\mathcal{A}, \pi \models \varphi \vee \psi & \iff & \mathcal{A}, \pi \models \varphi \text{ or } \mathcal{A}, \pi \models \psi \\
\mathcal{A}, \pi \models \neg\varphi & \iff & \mathcal{A}, \pi \not\models \varphi \\
\mathcal{A}, \pi \models \bigcirc \varphi & \iff & \mathcal{A}, \pi^1 \models \varphi \\
\mathcal{A}, \pi \models \varphi \,\mathcal{U}\, \psi & \iff & \text{there exists } n \in \mathbb{N} \text{ such that } \mathcal{A}, \pi^n \models \psi \text{ and,} \\
 & & \text{for all } m < n,\ \mathcal{A}, \pi^m \models \varphi
\end{array}$$

Other Boolean and temporal operators (e.g., $\bot$, $\wedge$, $\rightarrow$, $\Box$, $\Diamond$, $\mathcal{R}$, and $\rightsquigarrow$) can be defined as syntactic sugar.

It is sometimes useful to restrict ourselves to the *negation-free fragment* LTL$^-$(AP) of LTL(AP), defined as follows:

$$\varphi = p \in AP \mid \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi \mid \varphi \,\mathcal{R}\, \varphi \,.$$

[3] The choice of LTL is not essential: our main results and techniques apply also to the universal fragment ACTL* of CTL* [10]; we use LTL as a core logic for the exposition because it is the logic supported by the Maude system used in our case studies.

Negation is no longer available in LTL$^-$, and therefore the duals of the basic operators must be considered as basic ones, too. Since LTL$^-$ is a sublogic of LTL, its semantics is the same. Furthermore, in a very practical sense there is no real loss of generality by restricting ourselves to formulas in LTL$^-$, because we can always transform any LTL formula φ into a semantically equivalent LTL$^-$ formula $\hat{\varphi}$. For that, we consider the extended set of atomic propositions $\widehat{AP} = AP \cup \overline{AP}$, where $\overline{AP} = \{\bar{p} \mid p \in AP\}$, and construct $\hat{\varphi}$ by first forming the negation normal form of φ (i.e., all negations are pushed to the atoms), and then replacing each negated atom $\neg p$ by $\bar{p}$. Given $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, we define $\widehat{\mathcal{A}} = (A, \rightarrow_{\mathcal{A}}, L_{\widehat{\mathcal{A}}})$ where $L_{\widehat{\mathcal{A}}}(a) = L_{\mathcal{A}}(a) \cup \{\bar{p} \in \overline{AP} \mid p \notin L_{\mathcal{A}}(a)\}$. Then we have, $\mathcal{A}, a \models \varphi \iff \widehat{\mathcal{A}}, a \models \hat{\varphi}$.

3 Simulations

We present a notion of simulation similar to that in [10], but somewhat more general (simulations in [10] essentially correspond to our *strict* simulations).

Definition 2. *Given Kripke structures $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$, both having the same set AP of atomic propositions, an AP-*simulation *$H : \mathcal{A} \longrightarrow \mathcal{B}$ of $\mathcal{A}$ by $\mathcal{B}$ is given by a total binary relation $H \subseteq A \times B$ such that:*

- *if $a \rightarrow_{\mathcal{A}} a'$ and aHb, then there is $b' \in B$ such that $b \rightarrow_{\mathcal{B}} b'$ and $a'Hb'$, and*
- *if aHb, then $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.*

*If the relation H is a function, then we call H an AP-*simulation map*. If both H and H^{-1} are AP-simulations, then we call H an AP-*bisimulation*. Also we call H* strict *if aHb implies $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$. Note that all AP-bisimulations are strict.*

The first condition guarantees that for each concrete path in $\mathcal{A}$ there is a corresponding abstract path in $\mathcal{B}$; the second condition guarantees that an abstract state in $\mathcal{B}$ can satisfy only those atomic propositions that hold in all the concrete states in $\mathcal{A}$ that it simulates.

Definition 3. *An AP-simulation $H : \mathcal{A} \longrightarrow \mathcal{B}$* reflects *the satisfaction of a formula $\varphi \in$ LTL(AP) iff $\mathcal{B}, b \models \varphi$ and aHb imply $\mathcal{A}, a \models \varphi$.*

Theorem 1. *AP-simulations always reflect satisfaction of* LTL$^-(AP)$ *formulas. In addition, strict AP-simulations also reflect satisfaction of* LTL(AP) *formulas.*

Proof. Let $H : \mathcal{A} \longrightarrow \mathcal{B}$ be a simulation. We extend H to paths by defining $\pi H \rho$ if $\pi(i) H \rho(i)$ for every $i \in \mathbb{N}$. The theorem is then an easy consequence of the following two results, which can be proved by induction on the length of the path and by structural induction on formulas, respectively:

1. if aHb and π is a path in $\mathcal{A}$ starting at a, then there is a path ρ in $\mathcal{B}$ starting at b and such that $\pi H \rho$, and
2. if aHb, π starts at a, ρ starts at b, and $\pi H \rho$, then $\mathcal{B}, \rho \models \varphi$ implies $\mathcal{A}, \pi \models \varphi$; furthermore, this implication becomes an equivalence for strict simulations. □

Theorem 1 also holds for ACTL* formulas and, in that more general formulation, slightly generalizes Theorem 16 in [10].

This theorem is the key basis for the method of model checking by abstraction: given an infinite (or too large) system $\mathcal{M}$, find a system $\mathcal{A}$ with a finite set of reachable states that simulates it and use model checking to try to prove that φ holds in $\mathcal{A}$; then, by Theorem 1, φ also holds in $\mathcal{M}$. In general, however, we typically only have our concrete system $\mathcal{M}$ and a *surjective* function $h : M \longrightarrow A$ mapping concrete states to a simplified (usually with a finite set of reachable states) abstract domain A. 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 4. *The* minimal system $\mathcal{M}^h_{\min}$ *corresponding to* $\mathcal{M}$ *and the surjective function* $h : M \longrightarrow A$ *is given by the triple* $(A, (h \times h)(\rightarrow_{\mathcal{M}}), L_{\mathcal{M}^h_{\min}})$, *where* $(h \times h)(\rightarrow_{\mathcal{M}})$ *is the image of* $\rightarrow_{\mathcal{M}}$ *through* h *and* $L_{\mathcal{M}^h_{\min}}(a) = \bigcap_{x \in h^{-1}(a)} L_{\mathcal{M}}(x)$.

The following proposition is an immediate consequence of the definitions.

Proposition 1. *For all such* $\mathcal{M}$ *and* h, $h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$ *is a simulation map.*

Minimal systems can also be seen as quotients. Let $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ be a Kripke structure on AP, and let $\equiv$ be an arbitrary equivalence relation on A. We can use $\equiv$ to define a new Kripke structure, $\mathcal{A}/\equiv = (A/\equiv, \rightarrow_{\mathcal{A}/\equiv}, L_{\mathcal{A}/\equiv})$, where:

- $[a_1] \rightarrow_{\mathcal{A}/\equiv} [a_2]$ iff there exist $a'_1 \in [a_1]$ and $a'_2 \in [a_2]$ such that $a'_1 \rightarrow_{\mathcal{A}} a'_2$;
- $L_{\mathcal{A}/\equiv}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$.

It is then trivial to check that the projection map to equivalence classes $q_{\equiv} : a \mapsto [a]$ is an AP-simulation map $q_{\equiv} : \mathcal{A} \longrightarrow \mathcal{A}/\equiv$, which we call the *quotient abstraction* defined by $\equiv$. Hence, an equivalent presentation of the minimal system is expressed by the following.

Proposition 2. *Let* $\mathcal{M} = (M, \rightarrow_{\mathcal{M}}, L_{\mathcal{M}})$ *be a Kripke structure and* $h : M \longrightarrow A$ *a surjective function. Then, there exists a bijective bisimulation map between the Kripke structures* $\mathcal{M}^h_{\min}$ *and* $\mathcal{M}/\equiv_h$, *where by definition* $x \equiv_h y$ *iff* $h(x) = h(y)$.

Proof. Define $f : \mathcal{M}^h_{\min} \longrightarrow \mathcal{M}/\equiv_h$ by $f(h(x)) = [x]$; by definition of $\equiv_h$, and since h is surjective, f is a well-defined bijective function. We need to check that both f and $f^{-1}([x]) = h(x)$ are strict simulations.

If $a \rightarrow_{\mathcal{M}^h_{\min}} b$, then there exist x and y in $\mathcal{M}$ such that $h(x) = a$, $h(y) = b$, and $x \rightarrow_{\mathcal{M}} y$, and therefore $f(a) = [x] \rightarrow_{\mathcal{M}/\equiv_h} [y] = f(b)$. Similarly, if $[x] \rightarrow_{\mathcal{M}/\equiv_h} [y]$, then there exists x' such that $h(x) = h(x')$, and y' such that $h(y) = h(y')$, with $x' \rightarrow_{\mathcal{M}} y'$, and hence $f^{-1}([x]) = h(x') \rightarrow_{\mathcal{M}^h_{\min}} h(y') = f^{-1}([y])$.

Finally, $p \in L_{\mathcal{M}/\equiv_h}([x])$ iff $p \in L_{\mathcal{M}}(x')$ for all x' with $h(x') = h(x)$, iff $p \in L_{\mathcal{M}^h_{\min}}(h(x))$, and therefore f and f^{-1} are strict. □

That is, we can perform the abstraction either by mapping the concrete states to an abstract domain or, as we will do in Section 5, by identifying some states and thereafter working with the corresponding equivalence classes.

The use of the adjective "minimal" is appropriate since, as pointed out in [9], $\mathcal{M}^h_{\min}$ is the most accurate approximation to $\mathcal{M}$ that is consistent with h. However, it is not always possible to have a computable description of $\mathcal{M}^h_{\min}$. The definition of $\rightarrow_{\mathcal{M}^h_{\min}}$ can be rephrased as $x \rightarrow_{\mathcal{M}^h_{\min}} y$ iff there exist a and b such that $h(a) = x$, $h(b) = y$, and $a \rightarrow_{\mathcal{M}} b$. This relation, even if $\rightarrow_{\mathcal{M}}$ is recursive, is in general only recursively enumerable. However, Section 5 develops equational methods that, when successful, yield a computable description of $\mathcal{M}^h_{\min}$.

4 Rewriting Logic Specifications and Model Checking

One can distinguish two specification levels: a *system specification* level, in which the computational system of interest is specified, and a *property specification* level, in which the relevant properties are specified. The main interest of rewriting logic [33] is that it provides a very flexible framework for the system-level specification of concurrent systems. Rewriting logic is parameterized by an underlying equational logic. In this paper we will use membership equational logic, whose main characteristics we now review.

4.1 Membership Equational Logic

Membership equational logic is an expressive version of equational logic. A full account of its syntax and semantics can be found in [5, 34]; here we define the basic notions needed in this paper. The logic's expressiveness is due to its rich type structure, that supports sorts, subsorts, and operator overloading, and also errors and partiality through kinds and conditional membership axioms.

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(\vec{x})$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in $\vec{x}$, where $\vec{x} = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Sometimes we use the notation $t(\vec{x})$ to make explicit the set of variables that appear in the term t.

The atomic formulas of membership equational logic are either *equations* $t = t'$, where t and t' are terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. Sentences are Horn clauses on these atomic formulas, i.e., sentences of the form

$$(\forall \vec{x})\, A_0 \textbf{ if } A_1 \wedge \ldots \wedge A_n\,,$$

where each A_i is either an equation or a membership assertion, and $\vec{x}$ is a set of K-kinded variables. In membership equational logic, subsort relations and operator overloading are just a convenient way of writing corresponding Horn clauses. For example, assuming that *Nat* and *Int* are sorts of the same kind and that we have an operator $\_+\_ : [Int]\ [Int] \longrightarrow [Int]$ in Σ, then the subsort relation $Nat < Int$ is convenient notation for the conditional membership $(\forall x : [Int])\, x : Int$ **if** $x : Nat$, and the overloaded operator declarations

$$\_ + \_ : Nat\ Nat \longrightarrow Nat \qquad \_ + \_ : Int\ Int \longrightarrow Int$$

are logically equivalent to

$$(\forall x : [Int], y : [Int])\, x + y : Nat \textbf{ if } x : Nat \wedge y : Nat$$
$$(\forall x : [Int], y : [Int])\, x + y : Int \textbf{ if } x : Int \wedge y : Int\,.$$

A theory in membership equational logic is a pair (Σ, E), where E is a finite set of sentences in membership equational logic over the signature Σ. We write $(\Sigma, E) \vdash \phi$, or just $E \vdash \phi$ if Σ is clear from the context, to denote that (Σ, E) entails the sentence ϕ using the rules in Figure 1. The basic intuition is that correct or well-behaved terms are those that can be proved to have a sort, whereas error or undefined terms are terms that have a kind but do not have a sort. For example, assuming difference $\_ - \_$ and integer division $\_/\_$ operators with the appropriate declarations, $3 + 2 : Nat$ and $3 - 4 : Int$, but $7/0$ is a term of kind $[Int]$ with no sort.

A *Σ-algebra* A consists of a set A_k for each $k \in K$, a function $A_f : A_{k_1} \times \ldots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. An algebra A and a valuation σ, assigning to each variable $x : k$ in $\vec{x}$ a value in A_k, satisfy an equation $(\forall \vec{x})\, t = t'$ iff $\sigma(t) = \sigma(t')$, where we use the same notation σ for the valuation and its homomorphic extension to terms. We write $A, \sigma \models (\forall \vec{x})\, t = t'$ to denote such a satisfaction. Similarly, $A, \sigma \models (\forall \vec{x})\, t : s$ holds iff $\sigma(t) \in A_s$. We write $A \models \phi$ when the formula ϕ is satisfied for all valuations σ, and then say that A is a model of ϕ. As usual, we write $(\Sigma, E) \models \phi$ when all the models of the set E of sentences are also models of ϕ. The rules in Figure 1 specify a sound and complete calculus [34], that is, we have the equivalence $(\Sigma, E) \vdash \phi \iff (\Sigma, E) \models \phi$.

A theory (Σ, E) in membership equational logic has an initial model [34], denoted by $T_{\Sigma/E}$, whose elements are equivalence classes $[t]_E$ of ground terms. In the initial model, sorts are

$$\frac{t \in T_\Sigma(\vec{x})}{(\forall \vec{x})\, t = t}\ \textbf{Reflexivity} \qquad \frac{(\forall \vec{x})\, t' : s \qquad (\forall \vec{x})\, t = t'}{(\forall \vec{x})\, t : s}\ \textbf{Membership}$$

$$\frac{(\forall \vec{x})\, t' = t}{(\forall \vec{x})\, t = t'}\ \textbf{Symmetry} \qquad \frac{(\forall \vec{x})\, t_1 = t_2 \qquad (\forall \vec{x})\, t_2 = t_3}{(\forall \vec{x})\, t_1 = t_3}\ \textbf{Transitivity}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n, k} \qquad (\forall \vec{x})\, t_i = t_i' \;\; t_i, t_i' \in T_{\Sigma, k_i}(\vec{x}) \quad 1 \le i \le n}{(\forall \vec{x})\, f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \textbf{Congruence}$$

$$\frac{\begin{array}{c}(\forall \vec{x})\, A_0 \ \textbf{if}\ A_1 \wedge \ldots \wedge A_n \in E \\ \theta : \vec{x} \to T_\Sigma(\vec{y}) \qquad (\forall \vec{y})\, \theta(A_i) \;\; 1 \le i \le n\end{array}}{(\forall \vec{y})\, \theta(A_0)}\ \textbf{Replacement}$$

Fig. 1. Deduction rules for membership equational logic.

interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms. We write $E \vdash_{ind} \phi$ when ϕ holds in the initial model of E.

4.2 Rewriting Logic

Concurrent systems are axiomatized in rewriting logic by means of *rewrite theories* [33] of the form $\mathcal{R} = (\Sigma, E, R)$. The set of states is described by a membership equational theory (Σ, E) as the algebraic data type $T_{\Sigma/E,k}$ associated to the initial algebra $T_{\Sigma/E}$ of (Σ, E) by the choice of a kind k of states in Σ. The system's *transitions* are axiomatized by the *conditional rewrite rules* R which are of the form

$$\lambda : (\forall \vec{x})\, t \longrightarrow t' \ \textbf{if} \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t_l' \,,$$

with λ a label, $p_i = q_i$ and $w_j : s_j$ atomic formulas in membership equational logic for $i \in I$ and $j \in J$, and for appropriate kinds k and k_l, $t, t' \in T_{\Sigma,k}(\vec{x})$, and $t_l, t_l' \in T_{\Sigma,k_l}(\vec{x})$ for $l \in L$. Throughout this paper we assume that $vars(t') \cup vars(cond) \subseteq vars(t)$; this could be relaxed to allow extra variables in the condition and in t', provided they are added incrementally by "matching equations" in *cond* as explained in [11, 12]. Under reasonable assumptions about E and R, rewrite theories are *executable* (more on this below). Indeed, there are several rewriting logic language implementations, including CafeOBJ [23], ELAN [4], and Maude [11, 12].

We can illustrate rewriting logic specifications by means of an example, namely a simplified version of Lamport's bakery protocol [28]. This is an infinite state protocol that achieves mutual exclusion between processes by dispensing a number to each process and serving them in sequential order according to the number they hold. A simple Maude specification for the case of two processes and atomic transitions is as follows:

```
mod BAKERY is
  protecting NAT .
  sorts Mode BState .
  ops sleep wait crit : -> Mode [ctor] .
  op <_,_,_,_> : Mode Nat Mode Nat -> BState [ctor] .
  op initial : -> BState .
  vars P Q : Mode .
  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
```

This specification corresponds to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) imports the equational theory `NAT` of the natural numbers and where Σ has additional sorts `Mode` and `BState`, with `Mode` consisting of just the constants `sleep`, `wait`, and `crit`. States are represented by terms of sort `BState`, which are constructed by a 4-tuple operator `<_,_,_,_>`; the first two components describe the status of the first process (the mode it is currently in, and its priority as given by the number according to which it will be served) and the last two the status of the second process. E consists of just the equations imported from `NAT`, plus the above equation defining the `initial` state. R consists of eight rewrite rules, four for each process. These rules describe how each process passes from being sleeping to waiting, from waiting to its critical section, and then back to sleeping. In this case, the chosen kind k for states is of course the kind `[BState]` associated with the sort `BState`. Note that in Maude each entity in (Σ, E, R) is introduced by a corresponding keyword, such as `sorts` for sorts, `op` for an operator, `eq` (resp. `ceq`) for equations (resp. conditional equations), and `rl` (resp. `crl`) for rules (resp. conditional rules) that optionally can be labeled.

Rewriting logic then has the inference rules in Figure 2 to infer all the possible concurrent computations in a system [33, 7], in the sense that, given two states $[u], [v] \in T_{\Sigma/E,k}$, we can reach $[v]$ from $[u]$ by some possibly complex concurrent computation iff we can prove $u \longrightarrow v$ in the logic; we denote this provability by $\mathcal{R} \vdash u \longrightarrow v$. In particular we can easily define the *one-step $\mathcal{R}$-rewriting relation*, which is a binary relation $\rightarrow^1_{\mathcal{R},k}$ on $T_{\Sigma,k}$ that holds between terms $u, v \in T_{\Sigma,k}$ iff there is a one-step proof of $u \longrightarrow v$. More precisely, $u \rightarrow^1_{\mathcal{R},k} v$ if either there is a derivation of $u \longrightarrow v$ whose last rule is **(Replacement)**, or **(Equality)** applied to a pair of terms already in the relation, or if, for some $f \in \Sigma_{k_1 \ldots k_n, k}$, $u = f(t_1, \ldots, t_n)$ and $v = f(t'_1, \ldots, t'_n)$, and there exists i such that $t_i \rightarrow^1_{\mathcal{R},k_i} t'_i$ and $t_j = t'_j$ for all $j \neq i$. **(Transitivity)** is thus allowed, but only to solve the conditions that may arise in **(Replacement)**. We can get a binary relation (with the same name) $\rightarrow^1_{\mathcal{R},k}$ on $T_{\Sigma/E,k}$ by defining $[u] \rightarrow^1_{\mathcal{R},k} [v]$ iff $u' \rightarrow^1_{\mathcal{R},k} v'$ for some $u' \in [u]$, $v' \in [v]$. This then makes unnecessary the **(Equality)** rule, because $[u] \rightarrow^1_{\mathcal{R},k} [v]$ is defined at the level of E-equivalence classes.

The relationship with Kripke structures is now almost obvious: we can associate to a concurrent system axiomatized by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ with a chosen kind k of states a Kripke structure

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet, L_\Pi)\,.$$

We say "almost obvious," because nothing has yet been said about the choice of state predicates Π and the associated labeling function L_Π. The reason for this is methodological: Π, L_Π, and the LTL formulas φ describing properties of the system specified by $\mathcal{R}$ belong to the *property specification level*. Indeed, for the same system specification $\mathcal{R}$ we may come up with different predicates Π, labeling functions L_Π, and properties φ, depending on the properties of interest.

$$\frac{t \in T_\Sigma(\vec{x})}{(\forall \vec{x})\, t \longrightarrow t}\ \textbf{Reflexivity} \qquad \frac{(\forall \vec{x})\, t_1 \longrightarrow t_2 \qquad (\forall \vec{x})\, t_2 \longrightarrow t_3}{(\forall \vec{x})\, t_1 \longrightarrow t_3}\ \textbf{Transitivity}$$

$$\frac{E \vdash (\forall \vec{x})\, t = u \qquad (\forall \vec{x})\, u \longrightarrow u' \qquad E \vdash (\forall \vec{x})\, u' = t'}{(\forall \vec{x})\, t \longrightarrow t'}\ \textbf{Equality}$$

$$\frac{f \in \Sigma_{k_1 \ldots k_n, k} \qquad (\forall \vec{x})\, t_i \longrightarrow t_i' \;\; t_i, t_i' \in T_{\Sigma, k_i}(\vec{x}) \quad 1 \leq i \leq n}{(\forall \vec{x})\, f(t_1, \ldots, t_n) \longrightarrow f(t_1', \ldots, t_n')}\ \textbf{Congruence}$$

$$(\forall \vec{x})\, \lambda : t \longrightarrow t' \ \textbf{if} \ \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t_l' \in R$$

$$\theta : \vec{x} \to T_\Sigma(\vec{y}) \qquad (\forall \vec{y})\, \theta(t_l) \longrightarrow \theta(t_l') \;\; l \in L$$

$$\frac{E \vdash (\forall \vec{y})\, \theta(p_i) = \theta(q_i) \;\; i \in I \qquad E \vdash (\forall \vec{y})\, \theta(w_j) : s_j \;\; j \in J}{(\forall \vec{y})\, \theta(t) \longrightarrow \theta(t')}\ \textbf{Replacement}$$

Fig. 2. Deduction rules for rewrite theories.

The question of when a rewrite theory $\mathcal{R}$ is executable is closely related with wanting $T_{\Sigma/E,k}$ to be a *computable* set, and $(\to^1_{\mathcal{R},k})^\bullet$ to be a *computable* relation in the above Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$, an obvious precondition for any model checking. We say that $\mathcal{R} = (\Sigma, E \cup A, R)$ is *executable* if:

1. there exists a *matching algorithm modulo* the equational axioms A;[4]
2. the equational theory $(\Sigma, E \cup A)$ is (ground) *Church-Rosser and terminating modulo* A [18]; and
3. the rules R are (ground) *coherent* [40] relative to the equations E modulo A.

Conditions 1 and 2 ensure that $T_{\Sigma/E \cup A,k}$ is a computable set, since each ground term t can be simplified by applying the equations E from left to right modulo A to reach a *canonical form* $can_{E/A}(t)$ which is unique modulo the axioms A. We can then reduce the equality problem $[u]_{E \cup A} = [v]_{E \cup A}$ to the decidable equality problem $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$.

Condition 3 means that for each ground term t, whenever we have $t \to^1_{\mathcal{R}} u$ we can always find $can_{E/A}(t) \to^1_{\mathcal{R}} v$ such that $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$. This implies that $(\to^1_{\mathcal{R},k})^\bullet$ is a computable binary relation on $T_{\Sigma/E \cup A,k}$, since we can decide $[t]_{E \cup A} \to^1_{\mathcal{R}} [u]_{E \cup A}$ by enumerating the finite set of all one-step $\mathcal{R}$-rewrites modulo A of $can_{E/A}(t)$, and for any such rewrite, say v, we can decide $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$.

Coherence can be checked by critical-pair-like techniques similar to those used for checking equational confluence and performing Knuth-Bendix completion; the general theory is developed in [40]. Intuitively, the idea is to first establish that E is Church-Rosser and terminating modulo A, and then check the coherence of "relative critical pairs" (that is, overlaps on nonvariable subterms obtained by unification) between the equations E and the rules R modulo the axioms A; see Section 7 for examples.

4.3 LTL Properties of Rewrite Theories and Model Checking

One appealing feature of rewriting logic is that it provides a seamless integration of the system specification level and the property specification level, because we can specify the

[4] In the rewriting logic language Maude, the axioms A for which the rewrite engine supports matching modulo are any combination of *associativity*, *commutativity*, and *identity* axioms for different binary operators.

relevant state predicates Π *equationally*, and this then determines the labeling function L_Π and the semantics of the LTL formulas φ in a unique way. Indeed, to associate LTL properties to a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ with a chosen kind k of states we only need to make explicit the relevant state predicates Π, which need not be part of the system specification $\mathcal{R}$. The state predicates Π can be defined by means of equations D in an equational theory $(\Sigma', E \cup A \cup D)$ that *protects* $(\Sigma, E \cup A)$; specifically, the unique Σ-homomorphism $T_{\Sigma/E \cup A} \to T_{\Sigma'/E \cup A \cup D}$ induced by the theory inclusion $(\Sigma, E \cup A) \subseteq (\Sigma', E \cup A \cup D)$ should be bijective at each sort s in Σ.

The syntax defining the state predicates consists of a subsignature $\Pi \subseteq \Sigma'$ of operators p of the general form $p : s_1 \dots s_n \longrightarrow \mathit{Prop}$ (with *Prop* the sort of propositions), reflecting the fact that state predicates can be *parametric*. The semantics of the state predicates Π is defined by D with the help of an operator $\_ \models \_ : k \ \ [\mathit{Prop}] \longrightarrow [\mathit{Bool}]$ in Σ'. By definition, given ground terms $u_1, \dots, u_n$, we say that the state predicate $p(u_1, \dots, u_n)$ *holds* in the state $[t]$ iff

$$E \cup A \cup D \vdash_{ind} t \models p(u_1, \dots, u_n) = \mathit{true}\,.$$

We can now associate to $\mathcal{R}$ a Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$, whose atomic predicates are specified by the set $AP_\Pi = \{\theta(p) \mid p \in \Pi, \theta \text{ ground substitution}\}$.[5]

Definition 5. *The Kripke structure associated to a rewrite theory $\mathcal{R}$ is given by $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\to^1_{\mathcal{R},k})^\bullet, L_\Pi)$, where*

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

In practice we want the equality $t \models p(u_1, \dots, u_n) = \mathit{true}$ to be *decidable*. This can be achieved by giving equations in $E \cup D$ that are Church-Rosser and terminating modulo A. Then, if we begin with an *executable* rewrite theory $\mathcal{R}$ and define decidable state predicates Π by the method just described, we obtain a *computable* Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$ which, if it has finite sets of reachable states, can be used for model checking.

Since its 2.0 version, the Maude system has an on-the-fly, explicit-state LTL model checker [22] which supports the methodology just mentioned. Given an executable rewrite theory specified in Maude by a module `M`, and an initial state, say `initial` of sort $\texttt{State}_\texttt{M}$, we can model check different LTL properties beginning at this state. For that, a new module `M-PREDS` must be defined importing both `M` and the predefined module `SATISFACTION`, and a subsort declaration $\texttt{State}_\texttt{M}$ `<` `State` must be added (this declaration can be omitted if `State` = $\texttt{State}_\texttt{M}$). Then, the syntax of the state predicates must be declared by means of operators of sort `Prop` and their semantics must be given by equations involving the satisfaction operator `op _|=_ : [State] [Prop] -> [Bool]`. Once the semantics of the state predicates has been defined, and assuming that the set of states reachable from `initial` is finite, we define a new module `M-CHECK` that imports both `M-PREDS` and the predefined module `MODEL-CHECKER`; then we can model check any LTL `formula` in LTL(AP_Π) by giving to Maude the command:

```
reduce modelCheck(initial, formula) .
```

Continuing with our bakery protocol example, two basic properties that we may wish to verify are:

1. *mutual exclusion*: the two processes are never simultaneously in their critical section; and
2. *liveness*: any process in waiting mode will eventually enter its critical section.

In order to specify these properties it is enough to specify in Maude the following set Π of state predicates:

[5] By convention, if p has n parameters, $\theta(p)$ denotes the term $\theta(p(x_1, \dots, x_n))$.

```
mod BAKERY-PREDS is
  protecting BAKERY .
  including SATISFACTION .
  subsort BState < State .
  ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .
  vars P Q : Mode .
  vars X Y : Nat .

  eq < wait, X, Q, Y > |= 1wait = true .
  eq < sleep, X, Q, Y > |= 1wait = false .
  eq < crit, X, Q, Y > |= 1wait = false .
  eq < P , X, wait, Y > |= 2wait = true .
  eq < P , X, sleep, Y > |= 2wait = false .
  eq < P , X, crit, Y > |= 2wait = false .
  eq < crit , X, Q, Y > |= 1crit = true .
  eq < sleep, X, Q, Y > |= 1crit = false .
  eq < wait, X, Q, Y > |= 1crit = false .
  eq < P , X, crit, Y > |= 2crit = true .
  eq < P , X, sleep, Y > |= 2crit = false .
  eq < P , X, wait, Y > |= 2crit = false .
endm
```

Mutual exclusion is then expressed by the formula `[]~(1crit /\ 2crit)`, and liveness by `(1wait |-> 1crit) /\ (2wait |-> 2crit)`, where `[]`, `~`, and `|->` are respectively the symbols used by the model checker to represent $\Box$, $\neg$, and $\leadsto$.

Since the set of states reachable from `initial` (defined in the `BAKERY` module) is *infinite*, we should not model check the above specification as given. Instead, we should first define an *abstraction* of it where `initial` has only finitely many reachable states and then model check the abstraction.

5 Equational Abstractions

Let $\mathcal{R} = (\Sigma, E \cup A, R)$ be a rewrite theory. A quite general method for defining abstractions of the Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E\cup A,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet, L_\Pi)$ is by specifying an equational theory extension of the form

$$(\Sigma, E \cup A) \subseteq (\Sigma, E \cup A \cup E').$$

Since this defines an equivalence relation $\equiv_{E'}$ on $T_{\Sigma/E\cup A,k}$, namely,

$$[t]_{E\cup A} \equiv_{E'} [t']_{E\cup A} \iff E \cup A \cup E' \vdash t = t' \iff [t]_{E\cup A\cup E'} = [t']_{E\cup A\cup E'},$$

we can obviously define our quotient abstraction as $\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'}$. We call this the *equational quotient abstraction* of $\mathcal{K}(\mathcal{R}, k)_\Pi$ defined by E'.

But can $\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'}$, which we have just defined in terms of the underlying Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$, be understood as the Kripke structure associated to *another* rewrite theory? Let us take a closer look at

$$\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'} = (T_{\Sigma/E\cup A,k}/\equiv_{E'}, (\rightarrow^1_{\mathcal{R},k})^\bullet/\equiv_{E'}, L_{\Pi/\equiv_{E'}}).$$

The first observation is that, by definition, we have $T_{\Sigma/E\cup A,k}/\equiv_{E'} \;\cong\; T_{\Sigma/E\cup A\cup E',k}$. A second observation is that if $\mathcal{R}$ is *k-deadlock free*, that is, if we have $(\rightarrow^1_{\mathcal{R},k})^\bullet = \rightarrow^1_{\mathcal{R},k}$, then the rewrite theory $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$ is also k-deadlock free and we have, under some mild requirements (see Lemma 2 later):

$$(\rightarrow^1_{\mathcal{R}/E',k})^\bullet = \rightarrow^1_{\mathcal{R}/E',k} = (\rightarrow^1_{\mathcal{R},k})^\bullet/\equiv_{E'}.$$

Therefore, for $\mathcal{R}$ k-deadlock free, our obvious candidate for a rewrite theory having $\mathcal{K}(\mathcal{R},k)_{\Pi}/\equiv_{E'}$ as its underlying Kripke structure is the rewrite theory $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$. That is, we just add to $\mathcal{R}$ the equations E' and do not change at all the rules R.

How restrictive is the requirement that $\mathcal{R}$ is k-deadlock free? There is no essential loss of generality: in Section 6 we show how we can always associate to an executable rewrite theory $\mathcal{R}$ with no rewrites appearing in the conditions of its rules a semantically equivalent (from the LTL point of view) theory $\mathcal{R}_{df}$ which is both deadlock free and executable. All theories we have come across for our case studies satisfy that requirement.

In this way, at a purely mathematical level, $\mathcal{R}/E'$ seems to be what we want. Assuming that we have an A-matching algorithm, two problems may arise from the following two *executability questions* about $\mathcal{R}/E'$, which are essential for $\mathcal{K}(\mathcal{R},k)_{\Pi}/\equiv_{E'}$ to be computable and therefore for model checking:

- Are the equations $E \cup E'$ ground Church-Rosser and terminating modulo A?
- Are the rules R ground coherent relative to $E \cup E'$ modulo A?

The answer to each of these questions may be affirmative or negative. In practice, sufficient care on the part of the user when specifying E' should result in an affirmative answer to the first question. In any case, we can always try to check such a property with a tool such as Maude's Church-Rosser checker [19]; if the check fails, we can try to complete the equations with a Knuth-Bendix completion tool, for example [15], to get a theory $(\Sigma, E'' \cup A)$ equivalent to $(\Sigma, E \cup A \cup E')$ for which the first question has an affirmative answer. Likewise, we can try to check whether the rules R are ground coherent relative to $E \cup E'$ (or to E'') modulo A using the tool described in [20]. If the check fails, we can again try to complete the rules R to a semantically equivalent set of rules R', using also that tool [20]. By this process we can hopefully arrive at an *executable* rewrite theory $\mathcal{R}' = (\Sigma, E'' \cup A, R')$ which is semantically equivalent to $\mathcal{R}/E'$. We can then use $\mathcal{R}'$ to try to model check properties about $\mathcal{R}$.

But we are not finished yet. What about the state predicates Π? Recall (see Section 4.3) that these (possibly parameterized) state predicates will have been defined by means of equations D in a Maude module importing the specification of $\mathcal{R}$ and also the module `SATISFACTION`. The question is whether the state predicates Π are *preserved* under the equations E'. This indeed may be a problem. We need to unpack a little the definition of the labeling function $L_{\Pi/\equiv_{E'}}$, which is defined by the intersection formula

$$L_{\Pi/\equiv_{E'}}([t]_{E\cup A\cup E'}) = \bigcap_{[x]_{E\cup A}\subseteq[t]_{E\cup A\cup E'}} L_{\Pi}([x]_{E\cup A}).$$

In general, computing such an intersection and coming up with new equational definitions D' capturing the new labeling function $L_{\Pi/\equiv_{E'}}$ may not be easy. It becomes much easier if the state predicates Π are *preserved* under the equations E'. By definition, we say that the state predicates Π are preserved under the equations E' if for any $[t]_{E\cup A}, [t']_{E\cup A} \in T_{\Sigma/E\cup A,k}$ we have the implication

$$[t]_{E\cup A\cup E'} = [t']_{E\cup A\cup E'} \quad \Longrightarrow \quad L_{\Pi}([t]_{E\cup A}) = L_{\Pi}([t']_{E\cup A}).$$

Note that in this case, assuming that the equations $E \cup E' \cup D$ (or $E'' \cup D$) are ground Church-Rosser and terminating modulo A, we *do not need to change the equations D* to define the state predicates Π on $\mathcal{R}/E'$ (or its semantically equivalent $\mathcal{R}'$). Therefore, we have an isomorphism (given by a pair of invertible bisimulation maps)

$$\mathcal{K}(\mathcal{R},k)_{\Pi}/\equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E',k)_{\Pi},$$

or, in case we need the semantically equivalent $\mathcal{R}'$, an isomorphism

$$\mathcal{K}(\mathcal{R}, k)_{\Pi}/\equiv_{E'} \cong \mathcal{K}(\mathcal{R}', k)_{\Pi}.$$

The crucial point in both isomorphisms is that the labeling function of the righthand side Kripke structure is now equationally defined by the same equations D as before. Since by construction either $\mathcal{R}/E'$ or $\mathcal{R}'$ are executable theories, for an initial state $[t]_{E \cup A \cup E'}$ having a *finite* set of reachable states we can use the Maude model checker to model check any LTL formula in this equational quotient abstraction. Furthermore, since the quotient AP_{Π}-simulation map

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} \longrightarrow \mathcal{K}(\mathcal{R}/E', k)_{\Pi}$$

is then by construction *strict*, by Theorem 1 it reflects satisfaction of *arbitrary* LTL formulas. (Indeed, also of arbitrary ACTL* formulas.)

A practical problem remains: how can we actually try to *prove* the implication

$$[t]_{E \cup A \cup E'} = [t']_{E \cup A \cup E'} \quad \Longrightarrow \quad L_{\Pi}([t]_{E \cup A}) = L_{\Pi}([t']_{E \cup A})$$

to show the desired preservation of state predicates? A first result in solving that problem is the following, where `BOOL` is the predefined theory of Boolean values.

Theorem 2. *Let $\mathcal{R} = (\Sigma, E \cup A, R)$ be a k-deadlock free rewrite theory and let D be equations defining (possibly parametric) state predicates Π fully defined for all states of kind k as either* `true` *or* `false`*, and assume that $(\Sigma', E \cup A \cup D)$ protects* `BOOL`*. Let then E' be a set of Σ-equations such that $(\Sigma', E \cup A \cup E' \cup D)$ also protects* `BOOL`*. Then, the state predicates Π are preserved under E'.*

Proof. We have to check that $\equiv_{E'}$ is label-preserving, which is equivalent to proving the following equivalences for each $p \in \Pi$ and ground substitution θ:

$$E \cup A \cup D \vdash_{ind} t \models \theta(p) = \texttt{true} \iff E \cup A \cup E' \cup D \vdash_{ind} t \models \theta(p) = \texttt{true}$$

$$E \cup A \cup D \vdash_{ind} t \models \theta(p) = \texttt{false} \iff E \cup A \cup E' \cup D \vdash_{ind} t \models \theta(p) = \texttt{false}$$

The implications from left to right follow by monotonicity of equational reasoning. The converse implications follow from the protecting `BOOL` assumption, since we can reason by contradiction. Suppose, for example, that $E \cup A \cup E' \cup D \vdash_{ind} t \models \theta(p) = \texttt{true}$ but $E \cup A \cup D \nvdash_{ind} t \models \theta(p) = \texttt{true}$; by the protecting `BOOL` assumption this forces $E \cup A \cup D \vdash_{ind} t \models \theta(p) = \texttt{false}$, which implies $E \cup A \cup E' \cup D \vdash_{ind} t \models \theta(p) = \texttt{false}$, contradicting the protection of `BOOL`. □

The fact that `BOOL` is protected can be automatically checked with the sufficient completeness checker (SCC) for Maude [25]. This tool accepts a module as input and checks whether it is sufficiently complete, in the intuitive sense that enough equations are specified so that every term can be reduced to a canonical form in which only constructor operators are used; for `BOOL`, these constructors are `true` and `false`. The SCC tool assumes that the specification is terminating and confluent, which can also be proved automatically with tools like the Church-Rosser Checker (CRC) [19] and Maude Termination Tool (MTT) [21] if all equations are unconditional; otherwise, conditional critical pairs appear that complicate the proof. So Theorem 2 is especially useful in the unconditional case. We show an example of its application in Section 7.2; [12, Chapter 13] contains an abstraction for the bakery protocol different from the one discussed in Section 7.1, which can be proved correct with this theorem.

We now present a more general and powerful condition to prove preservation of predicates. A signature Σ is *k-encapsulated* if the kind k only appears as the codomain of a single operator $f : k_1 \dots k_n \longrightarrow k$, and does not appear as an argument in any operator in Σ. Then,

a particularly easy case for proving the preservation of predicates is that of *k-encapsulated* rewrite theories, for k the kind of states. This condition is very mild, since any rewrite theory $\mathcal{R}$ can be transformed into a semantically equivalent k'-encapsulated one by enclosing the original states in the kind k into new states in a kind k' through an operator $\{\_\} : k \longrightarrow k'$, 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 operator $\{\_\} : k \longrightarrow k'$. $\mathcal{R}'$ so defined is k'-encapsulated.*

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[6] *$(t \models p) = b$ **if** C in D into $(\{t\} \models p) = b$ **if** C. Then, the function $h : T_{\Sigma'/E,k'} \longrightarrow T_{\Sigma/E,k}$ given by $h([\{t\}]_E) = [t]_E$ defines a bijective bisimulation $\mathcal{K}(\mathcal{R}, k)_\Pi \cong \mathcal{K}(\mathcal{R}', k')_\Pi$.*

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 maps the term $\{t\}$ to t, we have that the transition relation is preserved in both directions. As for the state predicates, by the transformation 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. □

Besides being useful for the study of preservation of properties, encapsulation offers a way to tackle the deadlock freedom of theories.

Lemma 2. *Suppose that $\mathcal{R} = (\Sigma, E \cup A, R)$ is a k-encapsulated rewrite theory and that E' is a set of equations of the form $t = t'$ **if** C, with $t, t' \in T_{\Sigma,k}(\vec{x})$. Then, if $\mathcal{R}$ is k-deadlock free and no terms of kind k appear in the conditions of the rewrite rules in R, the rewrite theory $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$ is also k-deadlock free and we have*

$$\rightarrow^1_{\mathcal{R}/E',k'} = \rightarrow^1_{\mathcal{R},k'} /\equiv_{E'} .$$

for all kinds k' in Σ.

Proof. It is clear that $\mathcal{R}/E'$ is k-deadlock free because every rewrite in $\mathcal{R}$ is also a rewrite in $\mathcal{R}/E'$. For the same reason, the second relation is included in the first one. Now, assume that $[t] \rightarrow^1_{\mathcal{R}/E',k'} [t']$. By induction on the definition of $\rightarrow^1_{\mathcal{R}/E',k'}$:

- If there is a rule $l \longrightarrow r$ **if** C in R and a ground substitution θ such that $[t] = [\theta(l)]$, $[t'] = [\theta(r)]$, and $E \cup A \cup E' \vdash \theta(C)$ then, because of the restrictions on E' and R, we have $E \cup A \vdash \theta(C)$ (see Lemma 3 for the details of a similar proof) and therefore $[t] = [\theta(l)] \rightarrow^1_{\mathcal{R},k'} /\equiv_{E'} [\theta(r)] = [t']$.
- If $[t] = [f(u_1, \ldots, u_n)]$, $[t'] = [f(u'_1, \ldots, u'_n)]$, and $[u_i] \rightarrow^1_{\mathcal{R}/E',k_i} [u'_i]$ for some i, the result follows by induction hypothesis. □

Now, a useful fact about k-encapsulated theories, easy to prove from the rules of equational deduction and needed in the proof of our main result, is:

Lemma 3. *Let (Σ, E) be k-encapsulated and let E' be a set of (possibly conditional) equations whose left and righthand sides are terms of kind k. Then, if no terms of kind k appear in any conditions in E, we have $T_{\Sigma/E,k'} = T_{\Sigma/E \cup E',k'}$ for each kind k' different from k.*

[6] The explicit quantification of variables in equations is necessary to avoid inconsistencies when there are kinds with no ground terms. In all our proofs we take that into account but, in order to ease the presentation, we will not write them whenever it is more convenient.

Proof. We will prove that

$$E \cup E' \vdash (\forall \vec{x})\, u = v \quad \text{implies} \quad E \vdash (\forall \vec{x})\, u = v$$

by structural induction on the derivation:

- **(Reflexivity)**, **(Symmetry)**, **(Transitivity)**, and **(Membership)**. Trivial.
- **(Congruence)**. If

$$\frac{u_1 = v_1 \ \ldots \ u_n = v_n}{f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)}$$

  is the last step of a derivation in $E \cup E'$ then, since the theory is k-encapsulated, none of the u_i or v_i is of kind k and we can apply the induction hypothesis to get $E \vdash u_i = v_i$, whence the result follows.
- **(Replacement)**. If

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

  is the last step of a derivation in $E \cup E'$ for some equation $t = t'$ **if** C in E (note that by hypothesis it cannot belong to E'), we can apply the induction hypothesis to $\theta(C)$ since it cannot contain equations between terms of kind k, and the result follows. □

We can now give a sufficient condition under which preservation of atomic predicates is guaranteed. Actually, the following result proves much more since it shows that `BOOL` is protected and that the resulting theory is sort-decreasing and terminating.

Theorem 3. *Let* $(\Sigma', E \cup D)$ *be the extension of* (Σ, E) *with the operator* $\_ \models \_$ *and equations for the state predicates. Assume that* $(\Sigma', E \cup D)$ *and* $(\Sigma, E \cup E')$ *are both ground confluent, sort-decreasing, and terminating, and both protect* `BOOL`*. Assume also that for any* $f : k_1 \ldots k_n \longrightarrow k'$ *in* Σ'*, if* `[Bool]` *appears among the argument kinds* $k_1, \ldots, k_n$*, then* k' *is not* `[Bool]`*.*

Furthermore, assume that (Σ, E) *is* k*-encapsulated, the left and righthand side terms of the equations in* E' *are of kind* k*, and no terms of kind* k *appear in any conditions in* E *or* E'*. Then, if for each equation* $(\forall \vec{x})\ t = t'$ **if** C *in* E' *and each* $p \in \Pi$ *we have*

$$(\dagger) \qquad E \cup D \vdash_{ind} \ (\forall \vec{x}, \vec{y})\ C \rightarrow (t(\vec{x}) \models p(\vec{y}) = t'(\vec{x}) \models p(\vec{y}))$$

then $(\Sigma', E \cup E' \cup D)$ *is ground confluent, sort-decreasing, and terminating, and protects* `BOOL`*.*

Proof. Sort-decreasingness is obvious, since all the equations in $E \cup E' \cup D$ are sort-decreasing by hypothesis.

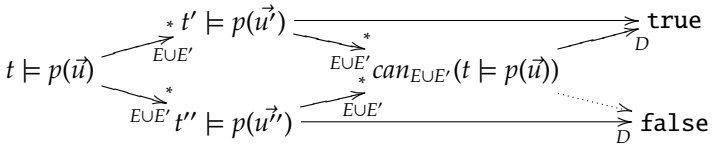
We show confluence and termination for each kind. Note that, by the above assumptions, for any kind k other than `[Bool]` or `[Prop]` we have $T_{\Sigma',k} = T_{\Sigma,k}$. Therefore, the only equations applying to ground terms of those kinds are those in $E \cup E'$, which are ground confluent and terminating by hypothesis. Similarly, any ground term $p(t_1, \ldots, t_n)$ has subterms $t_1, \ldots, t_n$ with kinds different from `[Bool]` or `[Prop]`, and the ground confluence and termination for each of those kinds, plus the absence of equations for p, easily yields ground confluence and termination. So we are left with terms in $T_{\Sigma',[\texttt{Bool}]}$ which, by the assumptions, are either:

1. terms in $T_{\Sigma,[\texttt{Bool}]}$, or
2. ground terms of the form $t \models p(\vec{u})$, or
3. Boolean combinations of `true`, `false`, and terms of the forms (1)–(2) above.

Since for terms of type (1) only equations in $E \cup E'$ apply, their ground confluence and termination follows by hypothesis. It all then boils down to showing ground confluence and termination of terms of type (2), because then the type (3) case follows easily by case analysis and a non-overlap confluence argument from types (1)–(2).

Note that termination for terms of type (2) follows from the observation that all sequences rewriting a term of the form $t \models p(\vec{u})$ must be either of the form $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t' \models p(\vec{u'})$, or of the form $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t' \models p(\vec{u'}) \longrightarrow_D b$, with b either `true` or `false`. The second kind of sequences are already terminating, and since $E \cup E'$ is by hypothesis terminating, we cannot have infinite sequences of the first kind: they must all eventually reach a unique normal form.

Confluence now follows easily from the fact that, given any two $E \cup E' \cup D$-rewrite sequences starting at a ground term $t \models p(\vec{u})$, we can always extend them to terminating sequences of the form $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t' \models p(\vec{u'}) \longrightarrow_D b$, $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t'' \models p(\vec{u''}) \longrightarrow_D b'$. If b equals b', we are done. Otherwise, we have, say, $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t' \models p(\vec{u'}) \longrightarrow_D \texttt{true}$, $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} t'' \models p(\vec{u''}) \longrightarrow_D \texttt{false}$. But, since $E \cup E'$ is ground confluent and terminating, we also have a sequence of the form $t \models p(\vec{u}) \longrightarrow^*_{E \cup E'} can_{E \cup E'}(t \models p(\vec{u})) \longrightarrow_D b''$, with b'' either `true` or `false`, say $b'' = \texttt{true}$ (the other case is analogous). Graphically:



Since we also have a sequence $t'' \models p(\vec{u''}) \longrightarrow^*_{E \cup E'} can_{E \cup E'}(t \models p(\vec{u}))$, we will reach a contradiction (against the protecting `BOOL` hypothesis for $E \cup D$) if we show that we must have $E \cup D \vdash_{ind} can_{E \cup E'}(t \models p(\vec{u})) = \texttt{false}$. This we can easily prove by induction on the number of steps in the sequence $t'' \models p(\vec{u''}) \longrightarrow^*_{E \cup E'} can_{E \cup E'}(t \models p(\vec{u}))$. For a single step resulting from an equation φ of the form $l = r$ **if** C and substitution θ, it must be $E \cup E' \vdash \theta(C)$; the conditions in Lemma 3 are satisfied and hence $E \vdash \theta(C)$. If $\varphi \in E$, it follows that $E \cup D \vdash t'' \models p(\vec{u''}) = can_{E \cup E'}(t \models p(\vec{u}))$ and we are done. Otherwise, because of the main hypothesis we have

$$E \cup D \vdash_{ind} (\forall \vec{x}, \vec{y})\ C \rightarrow (l \models p(\vec{y}) = r \models p(\vec{y}))\,;$$

then, since $E \cup D \vdash \theta(C)$, also $E \cup D \vdash_{ind} \theta(u \models p(\vec{y})) = \theta(v \models p(\vec{y}))$, perhaps extending θ to the variables in $\vec{y}$. The result now follows by induction. □

Summing up, to prove the preservation of state predicates when the abstraction equations E' are unconditional, often Theorem 2 will be enough. In the conditional case, however, we need to resort to the more powerful Theorem 3. As a consequence of this theorem, to prove that the state predicates Π are preserved in an equational abstraction we can use a tool like Maude's ITP [13] to mechanically discharge proof obligations of the form (†), under the above assumptions on $\mathcal{R}$, E', and D. In particular, the theory has to be k-encapsulated but, as Lemma 1 has shown, this implies no loss of generality. We illustrate the use of this more general theorem with the abstraction for the bakery protocol presented in Section 7.1.

Notice that the fact that the state kind is encapsulated does not preclude the use of recursive data structures in state components, for example a history variable. For instance, the case study in Section 7.2 shows indeed encapsulated states involving such recursive structures. In fact, the encapsulation requirement poses no real restriction in practice since Lemma 1 allows us to transform any rewrite theory $\mathcal{R}$ with state kind k into an equivalent k'-encapsulated one.

6 The Deadlock Difficulty

The reason why we have focused on deadlock-free rewrite theories is because deadlocks can pose a problem, due to a technical point in the Kripke structure semantics of LTL. As emphasized in its definition, the transition relation of a Kripke structure is *total*, and this requirement is also imposed on the Kripke structures arising from rewrite theories. Consider then the following specification of a rewrite theory, together with the declaration of two state predicates:

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

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

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

The transition relation of the Kripke structure corresponding to this specification has three elements: $\mathtt{a} \to \mathtt{b}$, $\mathtt{b} \to \mathtt{c}$, and $\mathtt{c} \to \mathtt{c}$, the last one consistently added as a deadlock transition according to the definition of $(\to^1_{\mathcal{R},[\mathrm{State}]})^\bullet$.

Suppose now that we wanted to abstract this system and that we decided to identify states `a` and `c` by means of a simulation map h. For that, according to the method presented in the previous sections, it would be enough to add the equation `eq c = a` to the above specification. The resulting system is coherent, and `a` and `c` satisfy the same state predicates. Note that the corresponding 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, we have $(\to_{\mathcal{R}/E',[\mathrm{State}]})^\bullet = \to_{\mathcal{R}/E',[\mathrm{State}]}$ for E' the equation `eq c = a` so that no additional deadlock transitions are added. In particular, there is no transition from the equivalence class of `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 does exist. The lack of this idle transition is a serious problem, because now we can prove properties about the supposedly simulating system that are actually false in the original one, for example, $\Box \Diamond \mathtt{p2}$.

One simple way to deal with this difficulty is to just add idle transitions for each of the states in the resulting specification by means of a rule of the form `x => x`. The resulting system, in addition to all the rules that the minimal system should contain, may in fact have some extra "junk" transitions that are not part of it. Therefore, we would 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 would be coarser than the minimal system.

A better way of addressing the problem is to characterize the set of deadlock states. For this, given a rewrite theory $\mathcal{R}$ with no rewrites appearing in the conditions of its rules, we introduce a new predicate *enabled* : $k \longrightarrow$ [*Bool*] for each kind k in $\mathcal{R}$ that will be *true* for a term iff there is a rule that can be applied to it.

Proposition 3. *Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ such that for every $l \longrightarrow r$ **if** C in R there are no rewrites in C, we define an extension (Σ', E') of its equational part by adding:*

1. *for each kind k in Σ, a new operator* $enabled : k \longrightarrow [Bool]$ *in* Σ';
2. *for each rule* $l \longrightarrow r$ **if** C *in* R, *an equation* $enabled(l) = true$ **if** C *in* E', *and*
3. *for each operator* $f : k_1 \dots k_n \longrightarrow k$ *in* Σ *and for each* i *with* $1 \leq i \leq n$, *the equation* $enabled(f(x_1, \dots, x_n)) = true$ **if** $enabled(x_i) = true$.

Then, for each term $t \in T_\Sigma$,

$$E' \vdash_{ind} enabled(t) = true \iff \text{there exists } t' \in T_\Sigma \text{ such that } t \rightarrow^1_{\mathcal{R},k} t' .$$

Proof. Notice first that since the terms are ground, the equation holds in the initial model iff it holds in every model. We prove the implication from left to right by induction on the derivation. The only nontrivial cases are when the last rule of inference used is either **(Replacement)** or **(Transitivity)**. In the case of **(Replacement)**, since *enabled* is a new operator, the equation used must have been one of those added to E. Assume then that, for $enabled(l) = true$ **if** C in E' and a substitution θ,

$$\frac{\theta(C)}{\theta(enabled(l)) = true}$$

is the last step of a derivation in E' where $l \longrightarrow r$ **if** C is a rule in R. Then, by Lemma 4 below, $E \vdash \theta(C)$ and therefore $\theta(l) \rightarrow^1_{\mathcal{R},k} \theta(r)$. When the equation used is $enabled(f(x_1, \dots, x_n)) = true$ **if** $enabled(x_i) = true$, the result follows by induction hypothesis and the **(Congruence)** rule of the rewriting logic calculus. Finally, in the case of **(Transitivity)**,

$$\frac{enabled(t) = t' \qquad t' = true}{enabled(t) = true} .$$

By Lemma 5 below we can distinguish two cases. If t' is *true* or if there is a smaller derivation of $enabled(t) = true$, we can apply the induction hypothesis. If t' is $enabled(t'')$ for some t'' such that $E' \vdash t = t''$, the result follows by the induction hypothesis applied to $E' \vdash t' = true$, and the fact that $E \vdash t = t''$ by Lemma 4.

The implication in the other direction is proved by induction on the definition of $\rightarrow^1_{\mathcal{R},k}$. If $t = \theta(l)$ and $t' = \theta(r)$ for some substitution θ and rule $l \rightarrow r$ **if** C in R, the result follows by instantiating the appropriate equation among those added to E'. If $E \vdash t = u$, $E \vdash t' = v$, and $u \rightarrow^1_{\mathcal{R},k} v$, by induction hypothesis $E' \vdash enabled(u) = true$ and therefore $E' \vdash enabled(t) = true$. Finally, if $t = f(t_1, \dots, t_n)$, $t' = f(t'_1, \dots, t'_n)$, and $t_i \rightarrow^1_{\mathcal{R},k} t'_i$ for some i, by induction hypothesis we have $E' \vdash enabled(t_i) = true$ and, again, the result follows by instantiating the appropriate equation in E'. □

Lemma 4. *Under the conditions in Proposition 3, for all terms* $t, t' \in T_\Sigma(\vec{x})$,

$$E' \vdash (\forall \vec{x})\, t = t' \quad implies \quad E \vdash (\forall \vec{x})\, t = t' .$$

Proof. It is straightforward to prove by induction that if there is a derivation of $t = t'$ in E' then there is also a derivation, with no occurrences of *enabled*, of $u = u'$, where u and u' are obtained from t and t' by replacing all subterms of the form $enabled(w)$ by *true*. Hence, when $t, t' \in T_\Sigma(\vec{x})$ what we get is a derivation in E. □

Lemma 5. *Under the conditions in Proposition 3, for all ground terms* t *and* t', *if there is a derivation of* $enabled(t) = t'$ *or of* $t' = enabled(t)$ *in* E', *then either:*

(a) t' *is true,*
(b) there is a derivation of $enabled(t) = true$ *in* E' *whose depth is less or equal, or*

(c) t' is enabled(t'') for some t'' such that $E' \vdash t = t''$.

Proof. By induction on the derivation. Only **(Transitivity)** is not immediate. Given

$$\frac{enabled(t) = t'' \qquad t'' = t'}{enabled(t) = t'},$$

we apply the induction hypothesis to $enabled(t) = t''$. If it is the case that either (*a*) or (*b*) holds, then (*b*) also holds for the original equation. Otherwise, t'' is $enabled(t''')$ and we can apply the induction hypothesis to $t'' = t'$. The cases (*a*) and (*c*) are immediate. Now, if (*b*) holds, there is a derivation of $enabled(t''') = true$ whose depth is less than or equal to the one for $enabled(t''') = t'$, and we can use it together with $enabled(t) = enabled(t''')$ to build a derivation of $enabled(t) = true$ not deeper than the original derivation. □

The *enabled* predicate and its properties are the key ingredients in the proof of the following proposition, which allows us to transform an executable rewrite theory into a semantically equivalent one that is both deadlock-free and executable.

Proposition 4. *Let $\mathcal{R} = (\Sigma, E \cup A, R)$ be an executable rewrite theory. Given a chosen kind of states k, we can construct an executable theory extension $\mathcal{R} \subseteq \mathcal{R}^k_{df} = (\Sigma', E' \cup A, R')$ such that:*

- *$\mathcal{R}^k_{df}$ is k'-deadlock free and k'-encapsulated for a certain kind k';*
- *there is a function $h : T_{\Sigma',k'} \longrightarrow T_{\Sigma,k}$ inducing a bijection $h : T_{\Sigma'/E' \cup A,k'} \longrightarrow T_{\Sigma/E \cup A,k}$ such that for each $t, t' \in T_{\Sigma',k'}$ we have*

$$h(t)(\rightarrow^1_{\mathcal{R},k})^\bullet h(t') \iff t \rightarrow^1_{\mathcal{R}^k_{df},k'} t'.$$

Furthermore, if Π are state predicates for $\mathcal{R}$ and k defined by equations D, then we can define state predicates Π for $\mathcal{R}^k_{df}$ and k' by equations D' such that the above map h becomes a bijective AP_Π-bisimulation

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

Proof. Define $\mathcal{R}^k_{df}$ by extending the equational theory (Σ, E) in $\mathcal{R}$ with an *enabled* predicate as explained in Proposition 3, and by adding a new kind k', a new operator $\{\_\} : k \longrightarrow k'$, and the rule

$$\{x\} \rightarrow \{x\} \textbf{ if } enabled(x) \neq true$$

to R.

By construction, it is clear that $\mathcal{R}^k_{df}$ is k'-encapsulated. Given a ground term $\{t\}$ with t of kind k, if there is t' in $\mathcal{R}$ such that $t \rightarrow^1_{\mathcal{R},k} t'$ then $\{t\} \rightarrow^1_{\mathcal{R}^k_{df},k'} \{t'\}$; otherwise, by Proposition 3, $E' \vdash enabled(t) \neq true$ and, by the rule we have just added, $\{t\} \rightarrow^1_{\mathcal{R}^k_{df}} \{t\}$. Hence $\mathcal{R}^k_{df}$ is k'-deadlock free. The function h can be defined as $h(\{t\}) = t$ and, since no equations between terms of kind k' have been introduced, it induces a bijection and clearly satisfies the equivalence in the second item.

Finally, regarding the state predicates, we transform each equation $(t \models p) = b$ **if** C into $(\{t\} \models p) = b$ **if** C, as in Lemma 1. This implies that $L_\Pi(\{t\}) = L_\Pi(t)$ and, together with the previous results, that h is a strict bisimulation. □

This transformation can be carried out automatically within Maude; see [12, Chapter 15] for details.

Note that we have used an *inequality* in the condition of the new rule. This is allowed in the implementation of rewriting logic in Maude under appropriate Church-Rosser and termination assumptions, but not in rewriting logic itself. However, by a metatheorem of Bergstra and Tucker [3], under the conditions of the proposition it is always possible to define such inequality in an equational way. The reason not to do it here is because it is more convenient and concise to express the rule this way, which in addition is supported by Maude in a built-in way as the inequality predicate _=/=_.

7 Case Studies

We show in detail the application of the techniques introduced in this paper with two examples: the bakery protocol presented in Section 4 and a communication protocol.

In addition to the cases presented here we have also dealt successfully with a number of examples that have been used in the literature to illustrate other abstraction methods, including a readers/writers system [29] (see also [12, Chapter 12]), the alternating bit protocol [36, 14, 30], a mutual exclusion protocol discussed in [16], and the bounded retransmission protocol [1, 2, 14], which is included in Appendix A. The abstractions were obtained simply by adding some equations to the specifications. Only in the last two cases was it necessary to add some extra rewrite rules (allowing idle/stuttering transitions of the form $x \longrightarrow x$) to guarantee coherence; the case studies not included in this paper can be found in [35].

7.1 The Bakery Protocol Example Revisited

We can use the bakery protocol example to illustrate how equational quotient abstractions can be used to verify infinite-state systems. We can define such an abstraction by adding to the equations of `BAKERY` (see page 6) a set E' of additional equations defining a quotient of the set of states. We can do so in the following module extending `BAKERY` by equations and leaving the transition rewrite rules unchanged:

```
mod ABSTRACT-BAKERY is
  including BAKERY .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
  ceq < P, s X, Q, s s Y > = < P, s s 0, Q, s 0 > if (s Y < X) .
  ceq < P, s s s X, Q, s Y > = < P, s s 0, Q, s 0 > if (Y < s s X) .
  ceq < P, s X, Q, s s Y > = < P, s 0, Q, s 0 > if not (s Y < X) .
  ceq < P, s s X, Q, s Y > = < P, s 0, Q, s 0 > if not (Y < s X) .
endm
```

Note that $\langle P, N, Q, M\rangle \equiv \langle P', N', Q', M'\rangle$ according to the above equations iff:

1. $P = P'$ and $Q = Q'$,
2. $N = 0$ iff $N' = 0$,
3. $M = 0$ iff $M' = 0$,
4. $M < N$ iff $M' < N'$.

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. (The equations in the module are more complex than necessary at first sight to rule out nontermination by means of looping rewrites.)

Three key questions are:

– Is the set of states now finite?
– Does this abstraction correspond to a rewrite theory whose equations are ground Church-Rosser and terminating?
– Are the rules still ground coherent?

The check of termination follows from that for the bigger module ABSTRACT-BAKERY-PREDS, which is discussed later.

To check local confluence we give to the Maude Church-Rosser Checker (CRC) tool a version without built-ins of this module, in which true and false are replaced by tt and ff, respectively:

```
Maude> (check Church-Rosser ABSTRACT-BAKERY .)
Church-Rosser checking of ABSTRACT-BAKERY
Checking solution :
  ccp
    < P@:Mode, s 0, Q@:Mode, s 0 >
    = < P@:Mode, s s 0, Q@:Mode, s 0 >
    if not(s Y@:Nat < s X*@:Nat)= tt /\ s Y@:Nat < s X*@:Nat = tt .
  ccp
    < P@:Mode, s s 0, Q@:Mode, s 0 >
    = < P@:Mode, s 0, Q@:Mode, s 0 >
    if s Y@:Nat < X@:Nat = tt /\ not(s Y@:Nat < X@:Nat) = tt .
```

We can conclude local ground confluence if we show that the conditions in these conditional critical pairs are *unsatisfiable*. This follows trivially if we can show that ABSTRACT-BAKERY protects both NAT and BOOL. This, in turn, follows from the following two facts:

– BAKERY itself has no equations and therefore trivially protects NAT and BOOL;
– ABSTRACT-BAKERY is [BState]-encapsulated and all its equations are of kind [BState]; therefore, by Lemma 3 all other kinds have identical data in the initial models of BAKERY and of ABSTRACT-BAKERY.

This leaves us with the ground coherence question. Checking a version without built-ins with Maude's Coherence Checker gives us:

```
Maude> (check coherence ABSTRACT-BAKERY .)
Coherence checking of ABSTRACT-BAKERY
Checking solution :
  cp
    < sleep, 0, Q@:Mode, s 0>
    = < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
  cp
    < P@:Mode, s 0, sleep, 0 >
    = < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
  ccp
    < wait, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
    = < crit, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
    if s Y*@:Nat < X*@:Nat = tt /\ s s Y*@:Nat < s X*@:Nat = ff .
  ccp
    < wait, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
    = < crit, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
    if not(s Y*@:Nat < X*@:Nat) = tt /\ s s Y*@:Nat < s X*@:Nat = ff .
  ccp
    < wait, s s X*@:Nat, Q@:Mode, s Y*@:Nat >
```

```
  = < crit, s s X*@:Nat, Q@:Mode, s Y*@:Nat >
  if not(Y*@:Nat < s X*@:Nat) = tt /\ s Y*@:Nat < s s X*@:Nat = ff .
ccp
 < wait, s s s X*@:Nat, Q@:Mode, s Y*@:Nat >
  = < crit, s s s X*@:Nat, Q@:Mode, s Y*@:Nat >
  if Y*@:Nat < s s X*@:Nat = tt /\ s Y*@:Nat < s s s X*@:Nat = ff .
ccp
  < P@:Mode, s X*@:Nat, wait, s s Y*@:Nat >
  = < P@:Mode, s X*@:Nat, crit, s s Y*@:Nat >
  if s Y*@:Nat < X*@:Nat = tt /\ s s Y*@:Nat < s X*@:Nat = tt .
ccp
  < P@:Mode, s X*@:Nat, wait, s s Y*@:Nat >
  = < P@:Mode, s X*@:Nat, crit, s s Y*@:Nat >
  if not(s Y*@:Nat < X*@:Nat) = tt /\ s s Y*@:Nat < s X*@:Nat = tt .
ccp
  < P@:Mode, s s X*@:Nat, wait, s Y*@:Nat >
  = < P@:Mode, s s X*@:Nat, crit, s Y*@:Nat >
  if not(Y*@:Nat < s X*@:Nat) = tt /\ s Y*@:Nat < s s X*@:Nat = tt .
ccp
  < P@:Mode, s s s X*@:Nat, wait, s Y*@:Nat >
  = < P@:Mode, s s s X*@:Nat, crit, s Y*@:Nat >
  if Y*@:Nat < s s X*@:Nat = tt /\ s Y*@:Nat < s s s X*@:Nat  = tt .
```

Since NAT and BOOL are protected, the only pairs with satisfiable conditions are:

```
cp
  < sleep, 0, Q@:Mode, s 0 >
  = < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp
  < P@:Mode, s 0, sleep, 0 >
  = < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
ccp
  < wait, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  = < crit, s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  if not(s Y*@:Nat < X*@:Nat) = tt /\ s s Y*@:Nat < s X*@:Nat = ff .
ccp
  < wait, s s X*@:Nat, Q@:Mode, s Y*@:Nat >
  = < crit, s s X*@:Nat, Q@:Mode, s Y*@:Nat >
  if not(Y*@:Nat < s X*@:Nat) = tt /\ s Y*@:Nat < s s X*@:Nat = ff .
ccp
  < P@:Mode, s X*@:Nat, wait, s s Y*@:Nat >
  = < P@:Mode, s X*@:Nat, crit, s s Y*@:Nat >
  if s Y*@:Nat < X*@:Nat = tt /\ s s Y*@:Nat < s X*@:Nat = tt .
ccp
  < P@:Mode, s s s X*@:Nat, wait, s Y*@:Nat >
  = < P@:Mode, s s s X*@:Nat, crit, s Y*@:Nat >
  if Y*@:Nat < s s X*@:Nat = tt /\ s Y*@:Nat < s s s X*@:Nat  = tt .
```

all of which can be inductively rewritten. We can illustrate the method of inductive proof with the first unconditional and the first conditional pair.

The first unconditional pair is:

```
cp < sleep, 0, Q@:Mode, s 0 > =
   < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
```

We can first inductively prove the equation

```
eq < wait, s s s Y:Nat, Q:Mode, s s Y:Nat > = < wait, 2, Q, 1 > .
```

in the module ABSTRACT-BAKERY, by induction on Y:Nat, which gives us the following two goals:

```
eq < wait, s s s 0, Q:Mode, s s 0 > = < wait, 2, Q, 1 > .

ceq < wait, s s s s Y:Nat, Q:Mode, s s s Y:Nat > = < wait, 2, Q, 1 >
  if < wait, s s s Y:Nat, Q:Mode, s s Y:Nat >
     = < wait, 2, Q, 1 > .
```

These two goals can be easily proved either using the ITP [13], or directly in Maude by simplifying the first goal to a syntactic identity, and by applying the *Theorem of Constants* to the second goal and adding the premise (instantiated with a constant) as an extra lemma to simplify the conclusion (also instantiated with a constant) to a syntactic identity.

We can then check that the above critical pair fills in by using the search command with the modifier =>1, which returns all one-step rewrites.

```
Maude> search in ABSTRACT-BAKERY : < sleep, 0, Q, 1 > =>1 X:BState .

Solution 1 (state 1)
X:BState --> < wait, 2, Q, 1 >

No more solutions.
```

Similarly, consider the first conditional critical pair which, eliminating the second redundant condition, we can simplify to:

```
ccp < wait, s X:Nat, Q:Mode, s s Y:Nat > =
    < crit, s X:Nat, Q:Mode, s s Y:Nat >
  if s Y:Nat < X:Nat = false .
```

Inducting on X, using the following equations as inductive lemmas,

```
eq s X < s Y = X < Y .
eq 0 < s X = true .
eq s X < 0 = false .
eq X < s X = true .
eq s X < X = false .
ceq X < s Y = true if X < Y .
ceq s X < Y = false if X < Y = false .
```

and simplifying conditions, we obtain the following two instances:

```
cp < wait, s 0, Q:Mode, s s Y:Nat > = < crit, s 0, Q:Mode, s s Y:Nat >
ccp < wait,s s X:Nat, Q:Mode, s s Y:Nat > =
    < crit, s s X:Nat, Q:Mode, s s Y:Nat >
  if Y:Nat < X:Nat = false .
```

The first pair's righthand side has canonical form < crit, 1, Q, 1 >; we can fill in the pair with the search command:

```
Maude> search [,1]
  < wait, s 0, Q:Mode, s s Y:Nat > =>+ X:BState .  --- 1 or more rewrites

Solution 1 (state 1)
X:BState --> < crit, 1, Q, 1 >

No more solutions.
```

Using the Theorem of Constants, we can convert the variables `X` and `Y` in the second pair into constants `a` and `b` and assume `not(b < a)`. Then the state `< crit, s s a, Q:Mode, s s b >` has canonical form `< crit, 1, Q, 1 >`, and we can fill in this second pair by giving the `search` command:

```
Maude> search [,1] < wait, s s a, Q:Mode, s s b > =>+ X:BState .

Solution 1 (state 1)
X:BState --> < crit, 1, Q, 1 >

No more solutions.
```

Another pending question is the deadlock freedom of `ABSTRACT-BAKERY`. To prove that it indeed holds we can specify an *enabled* predicate, as explained in Section 6, that returns *true* when applied to a term iff that term represents a non-deadlocked state. We need 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 .
```

Then, the equation we have to prove to ensure deadlock freedom is

```
  eq enabled(S) = true .
```

where `S` is a variable of sort `State`. The proof proceeds by induction on the first and third components of the state and can be done straighforwardly with the ITP. Alternatively, we could also prove the result in a more automated way by using the SCC tool. In our case the tool returns that the module is sufficiently 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.

What about state predicates? Are they preserved by the abstraction? State predicates are imported, together with `ABSTRACT-BAKERY`, in the module

```
mod ABSTRACT-BAKERY-PREDS is
  pr ABSTRACT-BAKERY .
  inc BAKERY-PREDS .
endm
```

What remaining tasks do we have left to show that we have an executable quotient equational abstraction? First of all, we need to show that the equations in `BAKERY-PREDS` are ground confluent, sort-decreasing, and terminating, and that `BAKERY-PREDS` protects `BOOL`. The check of termination follows from that of `ABSTRACT-BAKERY-PREDS`, which is discussed later. The local confluence test gives us:

```
Maude> (check Church-Rosser BAKERY-PREDS .)
Checking solution:
  All critical pairs have been joined. The specification is
    locally-confluent.
  The specification is sort-decreasing.
```

and the sufficient completeness test gives us:

```
Maude> (scc BAKERY-PREDS .)
Success: BAKERY-PREDS is sufficiently complete under the assumption
 that it is weakly-normalizing, confluent, and sort-decreasing.
```

and since `true` and `false` are in canonical form this shows that `BAKERY-PREDS` protects `BOOL`. Next we have to show that `ABSTRACT-BAKERY-PREDS` protects `BOOL` (which will ensure that the state predicates are preserved by the abstraction), and is ground confluent, sort-decreasing, and terminating. Since the equations in `ABSTRACT-BAKERY` are all of the kind `[BState]`, we can apply Theorem 3. All the equalities in Theorem 3's hypothesis can be easily proved by induction, either manually or with the ITP, using case analysis on the constants of sort `Mode`, since: (i) the equations in `ABSTRACT-BAKERY` leave modes unchanged; and (ii) the value of each state predicate only depends on the mode of one of the two processes.

All we have left is checking termination of the equations in the modules `BAKERY-PREDS` and `ABSTRACT-BAKERY`. But since their union are the equations in `ABSTRACT-BAKERY-PREDS`, it is enough to check `ABSTRACT-BAKERY-PREDS` is terminating. This check succeeds with the MTT tool [21], after replacing the predefined modules `NAT` and `BOOL` by equivalent specifications (predefined modules are not handled by the MTT tool at present).

In other words, we have just shown that, for Π the state predicates declared in the module `BAKERY-PREDS` (page 10), we have a strict quotient simulation map,

$$\mathcal{K}(\texttt{BAKERY-PREDS}, \texttt{State})_{\Pi} \longrightarrow \mathcal{K}(\texttt{ABSTRACT-BAKERY-PREDS}, \texttt{State})_{\Pi}.$$

Therefore, we can establish the mutual exclusion property of `BAKERY-PREDS` by model checking in `ABSTRACT-BAKERY-CHECK` the following:

```
Maude> reduce modelCheck(initial, []~ (1crit /\ 2crit)) .
result Bool: true
```

Likewise, we can establish the liveness property of `BAKERY-PREDS` by model checking in `ABSTRACT-BAKERY-CHECK`:

```
Maude> reduce modelCheck(initial, (1wait |-> 1crit) /\ (2wait |-> 2crit)) .
result Bool: true
```

7.2 A Communication Protocol

Our second example is a protocol for in-order communication of messages between a sender and a receiver in an asynchronous communication medium. To guarantee that the messages are received in the correct order, messages include a sequence number and 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 the next value (in the set `{a, b, c}` in this presentation) which is then paired with the sender's counter to compose a message that is then released to the medium; the value itself is also appended to a list of sent values owned by the sender. The receiver has a corresponding list of received values: 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 with a sequence number equal to its current counter, it removes it from the medium and adds its value to its list of received values.

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, MS, R >`, where `S` represents the status of the sender, `R` that of the receiver, and `MS` the asynchronous medium (a soup of messages).

```
mod PROTOCOL is
  protecting NAT .
  sorts Value ValueList LocalState Message MessageSoup State .
  subsort Value < ValueList .
  subsort Message < MessageSoup .

  ops a b c : -> Value [ctor] .
  op nil : -> ValueList [ctor] .
  op _:_ : ValueList ValueList -> ValueList [ctor assoc id: nil] .
  op ls : Nat ValueList -> LocalState [ctor] .

  op msg : Nat Value -> Message [ctor] .
  op null : -> MessageSoup [ctor] .
  op _;_ : MessageSoup MessageSoup -> MessageSoup [ctor assoc comm id: null] .

  op <_,_,_> : LocalState MessageSoup LocalState -> State [ctor] .
  op initial : -> State .

  vars N M : Nat .
  var X : Value  .
  vars L L1 L2 : ValueList .
  var MS : MessageSoup .
  vars R S : LocalState .

  eq initial = < ls(0, nil), null, ls(0, nil) > .

  rl < ls(N, L), MS, R > => < ls(s(N), L : a), MS ; msg(N, a), R > .
  rl < ls(N, L), MS, R > => < ls(s(N), L : b), MS ; msg(N, b), R > .
  rl < ls(N, L), MS, R > => < ls(s(N), L : c), MS ; msg(N, c), R > .
  rl < S, msg(N, X) ; MS, ls(N, L) > => < S, MS, ls(s(N), L : X) > .
endm
```

In this specification, terms of sort `LocalState`, constructed with the operator `ls`, are used to represent the local states of the sender and the receiver. The first argument of `ls` corresponds to the counter while the second one is the list of messages already sent or received. Note the important use of matching and rewriting *modulo* axioms of associativity (`assoc`) and identity (`id`) for the append operator `_:_` on lists, and modulo associativity (`assoc`), commutativity (`comm`), and identity (`id`) for the multiset union operator `_;_` that builds soups of messages. These axioms correspond to the axioms A in our theoretical description of a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ and are used by Maude to apply equations and rules modulo such declared axioms A.

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 $\Box$ `prefix`, where `prefix` 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 can be expressed as follows:

```
mod PROTOCOL-PREDS is
  inc SATISFACTION .
  inc PROTOCOL .
  op prefix : -> Prop [ctor] .
  vars M N : Nat .
  var V : Value .
  vars L1 L2 : ValueList .
```

```
  var MS : MessageSoup .

  eq (< ls(N, L1 : L2), MS, ls(M, L1) > |= prefix) = true .
  eq (< ls(N, nil), MS, ls(M, V : L1) > |= prefix) = false .
  eq (< ls(N, b : L2), MS, ls(M, a : L1) > |= prefix) = false .
  eq (< ls(N, c : L2), MS, ls(M, a : L1) > |= prefix) = false .
  eq (< ls(N, a : L2), MS, ls(M, b : L1) > |= prefix) = false .
  eq (< ls(N, c : L2), MS, ls(M, b : L1) > |= prefix) = false .
  eq (< ls(N, a : L2), MS, ls(M, c : L1) > |= prefix) = false .
  eq (< ls(N, b : L2), MS, ls(M, c : L1) > |= prefix) = false .
endm
```

As was the case with the bakery protocol, model checking cannot be directly applied because the set of states reachable from `initial` is infinite. There are, indeed, 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 and hence arbitrarily long lists of sent and received messages. The second one is the communication medium, which is unbounded and can contain an arbitrary number of messages. To deal with this infiniteness and to be able to apply model checking, we need to define an abstraction; the corresponding proof obligations are discharged in a way similar to that for the bakery example and hence we do not go into as much detail.

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

```
  eq < ls(N, X : L1), MS, ls(M, X : L2) > = < ls(N, L1), MS, ls(M, L2) > .
```

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.

```
  eq < ls(s(N), L1), null, ls(s(N), L2) > = < ls(0, L1), null, ls(0, L2) > .
```

(The pattern `s(N)` in this equation is used to ensure termination.)

Finally, if in the medium of the current state there is a message `msg(N, X)` and the receiver's counter is `N`, we can identify this state with one in which the message has been read by the receiver.

```
  eq < ls(M, L1 : X : L2), msg(N, X) ; MS, ls(N, L1) > =
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .
```

The equation is unconditional, but note that in order to enforce that either both states satisfy `prefix` or none does, the term corresponding to the sender is required to match a certain pattern on the lefthand side of the equation.

Before applying model checking to this new system we must again ask ourselves whether the equations are still Church-Rosser and terminating, the rules are ground coherent, and the predicates are preserved. Termination is clear because the number of messages keeps decreasing and deadlock freedom too because it is always possible to add a new element to the list of sent messages.

The Church-Rosser property is not so straightforward due to the overlapping of the first and the third equations: if the next message to be delivered appears also as the head of the lists of messages associated to the sender and the receiver, we can either append it to the end of the receiver's list using the third equation, or remove it from both lists using the first one, and in this last case it does not seem possible to further reduce (equationally) the state.

Nonetheless, the Church-Rosser property indeed holds; informally, what happens is that in order for the third equation to apply the sender and the receiver have to be such that we are going to be able to remove all messages from the receiver's list; after that, the message can be appended to the end of the receiver's list as wanted.

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

```
< ls(5, a : b : c), msg(3, b), ls(3, a) >
```

for example, can be reduced to

```
< ls(5, a : b : c), null, ls(4, a : b) >
```

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

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

On the other hand, the second equation can also raise a coherence problem. For example:

$$\begin{array}{ccc} \texttt{< ls(5,L1), null, ls(5,L2) >} & \rightarrow & \texttt{< ls(6,L1 : a), msg(5,a), ls(5,L2) >} \\ \| & & \| \, ? \\ \texttt{< ls(0,L1), null, ls(0,L2) >} & \rightarrow & \texttt{< ls(1,L1 : a), msg(0,a), ls(0,L2) >} \end{array}$$

Suppose now that `L1` and `L2` are equal: then, both terms on the righthand side can be reduced by the equations to the term

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

and hence we have coherence. This, however, is not true in general but can be enforced by requiring `L1` and `L2` to be `nil`, and thus equal, for the second equation to be applied:

```
  eq < ls(s(N), nil), null, ls(s(N), nil) > = < ls(0, nil), null, ls(0, nil) > .
```

The resulting abstraction is then given as follows:

```
mod ABSTRACT-PROTOCOL-PREDS is
  inc PROTOCOL-PREDS .

  vars M N : Nat .
  vars L1 L2 : ValueList .
  var MS : MessageSoup .
  var X : Value .

  eq < ls(N, X : L1), MS, ls(M, X : L2) > = < ls(N, L1), MS, ls(M, L2) > .
  eq < ls(s(N), nil), null, ls(s(N), nil) > = < ls(0, nil), null, ls(0, nil) > .
  eq < ls(M, L1 : X : L2), msg(N, X) ; MS, ls(N, L1) > =
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .

--- coherence
  rl < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > =>
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .
endm
```

Using, for example, the SCC tool shows that both the modules `PROTOCOL-PREDS` and `ABSTRACT-PROTOCOL-PREDS` are sufficiently complete. In particular, they both preserve `BOOL` and then, by Theorem 2, the state predicate `prefix` is preserved.

Our desired property can now be finally checked:

```
Maude> reduce modelCheck(initial, [] prefix) .
result Bool: true
```

It is worth noting the following remark about the previous lines. The reason why we achieve coherence is because the abstraction collapses almost everything! In particular, every *reachable* state is simplified by the abstraction equations to the term

```
< ls(0, nil), null, ls(0, nil) > .
```

8 Related Work and Conclusions

In [9] the simulation of a system $\mathcal{M}$ by another $\mathcal{M}'$ through a surjective function h was defined and the optimal simulation $\mathcal{M}^h_{\min}$ was identified. The idea of simulating by a quotient has been further explored in [10, 8, 2, 27, 30, 16] among others, although the construction in [16] requires a Galois connection instead of just a function. Theorem proving is proposed in [2] to construct the transition relation of the abstract system, and in [30] to prove that a function is a representative function that can be used as input to an algorithm to extract $\mathcal{M}^h_{\min}$ out of $\mathcal{M}$. While those uses of theorem proving focus on the correctness of the abstract transition relation, our method focuses on making the minimal transition relation (which is correct by construction) *computable*, and on proving the preservation of the labeling function. In [9, 10], on the other hand, the minimal model $\mathcal{M}^h_{\min}$ is discarded in favor of less precise but easier to compute approximations; this would correspond, in our approach, to the addition of rewrite rules to the specification to simplify the proofs of the proof obligations (which can indeed be a reasonable alternative way of applying some of the techniques presented here within a "lighter" methodology). In all the papers mentioned, two states can become identified only if they satisfy the same atomic propositions; our definition of simulation is more general, but we have not yet exploited this.

The equational abstraction method that we have presented seems to apply in practice to a good number of examples discussed in the literature. But we need to further test its applicability on a wider and more challenging range of examples. Also, the method itself can be generalized along several directions. For example, the equational theory extension $(\Sigma, E \cup A) \subseteq (\Sigma, E \cup A \cup E')$ is generalized in [32] to an arbitrary *theory interpretation* $H : (\Sigma, E \cup A) \longrightarrow (\Sigma', E'')$, allowing arbitrary transformations on the data representation of states. A particular instance of this is *predicate abstraction* [38, 14]. Under this approach, the abstract domain is a Boolean algebra over a set of assertions and the abstraction function, typically as part of a Galois connection, is symbolically constructed as the conjunction of all expressions satisfying a certain condition, which is proved using theorem proving. This corresponds in our framework to a theory interpretation $H : (\Sigma, E) \longrightarrow (\Sigma \cup \Sigma', E \cup E')$, with Σ' introducing operators of the form $p : \mathit{State} \longrightarrow \mathit{Bool}$, and with H mapping states S to Boolean tuples $\langle p_1(S), \ldots, p_n(S) \rangle$. Similarly, simulation maps between *different* sets AP and AP' of state predicates can be considered, yielding another increase in generality when relating systems. Yet another direction along which our methods can be generalized is considering *stuttering* notions of simulation and bisimulation [6, 37, 30] allowing changes in the atomicity levels of transitions when relating systems. All these extensions, together with the more general representations of simulations in rewriting logic by means of equationally defined functions or rewrite relations, are studied in [31].

Acknowledgements. 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, Manuel Clavel and Francisco Durán for their help in the preparation of this paper, and Roberto Bruni and Joe Hendrix for many comments on previous drafts.

References

1. Parosh Abdulla, Aurore Annichini, and Ahmed 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*, pages 208–222. Springer-Verlag, 1999.
2. 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.
3. Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *7th International Conference on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.
4. Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.
5. Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
6. Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
7. Roberto Bruni and José Meseguer. Generalized rewrite theories. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2003.
8. 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.
9. 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.
10. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
11. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
12. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
13. Manuel Clavel, Miguel Palomino, and Adrián Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.
14. 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.

15. Evelyne Contejean and Claude Marché. CiME: Completion modulo *E*. In Harald Ganzinger, editor, *Rewriting Techniques and Applications. 7th International Conference, RTA-96, New Brunswick, NJ, USA July 27 - 30, 1996. Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 416–419. Springer-Verlag, 1996.
16. Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291, 1997.
17. Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and Jan 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.
18. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.
19. Francisco Durán. The Church-Rosser checker (CRC). `http://www.lcc.uma.es/~duran/CRC/`, 2006.
20. Francisco Durán. The coherence checker (ChC). `http://www.lcc.uma.es/~duran/ChC/`, 2006.
21. Francisco Durán. The Maude termination tool (MTT). `http://www.lcc.uma.es/~duran/MTT/`, 2006.
22. Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA'02, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.
23. Kokichi Futatsugi and Răzvan Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
24. 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.
25. Joe Hendrix, José Meseguer, and Hitoshi Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning - Third International Joint Conference, IJCAR 2006, Seattle, Washington, August 17 - 20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer-Verlag, 2006.
26. 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.
27. Yonit Kesten and Amir Pnueli. Verification by augmentary finitary abstraction. *Information and Computation*, 163:203–243, 2000.
28. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
29. Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–36, 1995.
30. Panagiotis Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001.
31. Narciso Martí-Oliet, José Meseguer, and Miguel Palomino. Algebraic simulations. `http://maude.sip.ucm.es/~miguelpt/bibliography.html`, January 2005.
32. Narciso Martí-Oliet, José Meseguer, and Miguel Palomino. Theoroidal maps as algebraic simulations. In José Luiz Fiadeiro, Peter Mosses, and Fernando Orejas, editors, *Recent Trends in Algebraic Development Techniques. 17th International Workshop, WADT 2004. Barcelona, Spain, March 27-30, 2004. Revised Selected Papers*, volume 3423 of *Lecture Notes in Computer Science*, pages 126–143. Springer-Verlag, 2005.
33. José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
34. 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.

35. José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Notes on model checking and abstraction in rewriting logic. `http://maude.sip.ucm.es/~miguelpt/bibliography`, 2002.
36. 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.
37. Kedar S. Namjoshi. A simple characterization of stuttering bisimulation. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science. 17th Conference, Kharagpur, India, December 18 - 20, 1997. Proceedings*, volume 1346 of *Lecture Notes in Computer Science*, pages 284–296. Springer-Verlag, 1997.
38. 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.
39. Tomás E. Uribe Restrepo. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Department of Computer Science, Stanford University, December 1998.
40. Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

A The Bounded Retransmission Protocol

In this appendix we discuss in some detail a more complex example, the bounded retransmission protocol (BRP) [24, 17]. The BRP is an extension of the alternating bit protocol 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 different status of sender and receiver, messages, sequences of messages, and the labels of the transitions can be represented as follows:

```
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
```

Properties that the service should satisfy include 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. States of the system are represented by terms of sort `State` constructed with a 7-tuple operator `<_,...,_>`. The first and the fifth components describe the current status of the sender and the receiver, respectively. The second and the sixth are Boolean values used by the sender and the receiver for synchronization purposes. The third and fourth components of the tuple correspond to the two lossy channels through which the sender and the receiver communicate. The last component keeps track of the name of the last transition used to reach the current state (hence the name of the constants of sort `Label`: `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.

For a more detailed description of the protocol, we refer to [1]. In Maude, the protocol can be specified as follows:

```
mod BRP is
  protecting DATA .

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

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

  eq initial = < 0s, false, nil, nil, 0r, false, none > .

  rl [REQ] : < 0s, A, nil, nil, R, false, LA > =>
             < 1s, false, nil, nil, R, false, req > .
  rl [K!fst] : < 1s, A, K, L, R, RT, LA > =>
               < 2s, A, K ; fst, L, R, RT, none > .
  rl [K!fst] : < 2s, A, K, L, R, RT, LA > =>
               < 2s, A, K ; fst, L, R, RT, none > .
  rl [L?fst] : < 2s, A, K, fst ; L, R, RT, LA > =>
               < 3s, A, K, L, R, RT, none > .
  crl [L?-fst] : < 2s, A, K, M ; L, R, RT, LA > =>
                 < 2s, A, K, L, R, RT, none > if M =/= fst .
  rl [K!0] : < 3s, A, K, L, R, RT, LA > =>
             < 4s, A, K ; 0, L, R, RT, none > .
  rl [K!last] : < 3s, A, K, L, R, RT, LA > =>
                < 7s, A, K ; last, L, R, RT, none > .
  rl [K!0] : < 4s, A, K, L, R, RT, LA > =>
             < 4s, A, K ; 0, L, R, RT, none > .
```

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

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

rl [K?lastL!last] : < S, A, last ; K, L, 4r, RT, LA > =>
                    < S, A, K, L ; last, 4r, RT, none > .
rl [empty] : < S, A, last ; K, L, 4r, RT, LA > =>
```

```
               < S, A, nil, L, 0r, false, none > .
endm
```

The properties that the system should satisfy impose requirements typically of the form that certain transitions should happen before certain other transitions do. To formulate requirements of this general form, we declare a parametric atomic proposition, `tr(L)`, that is true in those states resulting from the application of a transition labeled by L.

```
mod BRP-PREDS is
  inc SATISFACTION .
  inc BRP .

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

The required four properties can then be expressed in Maude as follows:

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 is strict, i.e., that it preserves the atomic propositions.

The system is infinite but one easily realizes, by running some small examples, that the contents of the channels are always of the form $m_1^* m_2^*$, where m_1, m_2 range over {`first`, `last`, `0`, `1`}. Therefore we can use the idea of merging adjacent equal messages, which can be specified by means of the following two equations, to collapse the set of states to a finite number.

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

Note that we need not prove that the contents of the channels are actually of the form $m_1^* m_2^*$. This is only used as an intuition to guide us in the choice of the abstraction equations, which can still be used regardless of the validity of that claim (though they may not be very useful if the claim is not really true).

It is immediate to check, since the abstraction equations do 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. And since the equations apply to disjoint components of the state and there is only a finite number of messages that can be removed, we also have the Church-Rosser and termination properties.

What about the deadlock difficulty? By inspection of the lefthand sides of the rules in `BRP`, it is easy to see that the equation

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

does not hold (consider the case in which S is equal to 0s) for the enabled operator as defined in Section 6, so that the rule

```
rl < S, A, KL ; M ; K, L, R, RT, LA > => < S, A, KL ; M ; K, L, R, RT, 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 equal to 0s, all terms of those forms are enabled. Hence, instead of the previous one, we only add the rules

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

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

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

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

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

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

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

which can no longer be rewritten to t, or to any other 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, A, K, fst ; L, R, RT, LA > =>
              < 3s, A, K, fst ; L, R, RT, 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, resulting in the addition of the following rules:

```
crl [L?-fst'] : < 2s, A, K, M ; L, R, RT, LA > =>
                < 2s, A, K, M ; L, R, RT, none > if M =/= fst .
crl [L?-0'] : < 4s, A, K, M ; L, R, RT, LA > =>
              < 4s, A, K, M ; L, R, RT, none > if M =/= 0 .
rl [L?0'] : < 4s, A, K, 0 ; L, R, RT, LA > => < 5s, A, K, 0 ; L, R, RT, none > .
crl [L?-1] : < 6s, A, K, M ; L, R, RT, LA > => < 6s, A, K, M ; L, R, RT, none >
             if M =/= 1 .
crl [L?-last] : < 7s, A, K, M ; L, R, RT, LA > =>
                < 7s, A, K, M ; L, R, RT, none > if M =/= last .
rl [SOK] : < 7s, A, K, last ; L, R, RT, LA > => < 0s, A, K, last ; L, R, RT, sok > .
rl [RFST'] : < S, false, fst ; K, L, 0r, RT, LA > =>
             < S, false, fst ; K, L ; fst, 1r, true, rfst > .
rl [K?fstL!fst'] : < S, A, fst ; K, L, 1r, RT, LA > =>
                   < S, A, fst ; K, L ; fst, 1r, RT, none > .
rl [RINC'] : < S, false, 0 ; K, L, 1r, RT, LA > =>
```

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

We then get our desired executable abstraction module ABSTRACT-BRP-CHECK by importing BRP-CHECK and including the abstraction equations, the two [deadlock] rules, and the above rules.

We can then model check the abstract system specified in ABSTRACT-BRP-CHECK and verify 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.

```
Maude> red modelCheck(initial, [](tr(req) ->
                                 O (~ tr(req) W (tr(sok) \/ tr(snok) \/ tr(sdnk))))) .
result Bool: true

Maude> red modelCheck(initial, [](tr(rfst) -> (~ tr(req) W (tr(rok) \/ tr(rnok))))) .
result Bool: true

Maude> red modelCheck(initial, [](tr(req) -> (~ tr(sok) W tr(rok)))) .
result Bool: true

Maude> red modelCheck(initial, tr(req) -> (~ tr(rnok) W (tr(snok) \/ tr(sdnk)))) .
result Bool: true
```