Program Verification: Lecture 26

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign
Simulations

Given two Kripke structures \( \mathcal{A} = (A, \rightarrow_A, L_A) \), and \( \mathcal{B} = (B, \rightarrow_B, L_B) \), both having the same set \( AP \) of atomic propositions, an \( AP \)-simulation \( H : \mathcal{A} \rightarrow \mathcal{B} \) of \( \mathcal{A} \) by \( \mathcal{B} \) is a binary relation \( H \subseteq A \times B \) such that, denoting pairs \((a, b) \in H\) by \( aHb \), we have:

- if \( a \rightarrow_A a' \) and \( aHb \), then there is a \( b' \in B \) such that \( b \rightarrow_B b' \) and \( a'Hb' \), and

- \( (\forall a \in A)(\forall b \in B) \ aHb \ \Rightarrow \ L_B(b) = L_A(a) \).

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 \( H \) a bisimulation.
Simulations (II)

Note that (exercise) $AP$-simulations (resp. $AP$-simulation maps, resp. $AP$-bisimulations) compose. That is, if we have $AP$-simulations (resp. $AP$-simulation maps, resp. $AP$-bisimulations)

$$H : A \longrightarrow B \quad G : B \longrightarrow C$$

then $H ; G : A \longrightarrow C$ is also an $AP$-simulation (resp. $AP$-simulation map, resp. $AP$-bisimulation).

Note also that the identity function $1_A$ is trivially an $AP$-simulation $1_A : A \longrightarrow A$, and also an $AP$-simulation map and an $AP$-bisimulation.
We say that an $AP$-simulation $H : A \rightarrow B$ reflects the satisfaction of an $LTL$ formula $\varphi$ iff $B, b \models \varphi$ and $aHb$ imply $A, a \models \varphi$.

A fundamental result, allowing us to prove the satisfaction of an $LTL$ formula $\varphi$ in an infinite-state system $A$ by proving the same satisfaction in a finite-state system $B$ that simulates it is the following,

**Theorem:** $AP$-simulations always reflect satisfaction of $LTL(AP)$ formulae.
Simulations Reflect Satisfaction of $LTL$ Formulae (II)

**Proof:** First of all, note that we can extend $H$ to a binary relation \( \hat{H} : \text{Path}(A) \rightarrow \text{Path}(B) \), where,

\[
\pi \hat{H} \rho \iff \forall n \in \mathbb{N} \; \pi(n)H\rho(n).
\]

**Lemma 1:** If \( aHb \), then for each \( \pi \in \text{Path}(A)_a \) there is a \( \rho \in \text{Path}(B)_b \) such that \( \pi \hat{H} \rho \).

**Proof of Lemma 1:** Suppose that there is no \( \rho \in \text{Path}(B)_b \) such that \( \pi \hat{H} \rho \). This implies that there is a finite sequence \( \rho \) of length \( n \) defined for \( 0 \leq i \leq n \) with \( \rho(0) = b \), and such that
Simulations Reflect Satisfaction of $LTL$ Formulae (III)

- $\pi(i)H\rho(i), \ 0 \leq i \leq n$, and
- $\rho(i) \rightarrow_{B} \rho(i + 1), \ 0 \leq i < n$.

which cannot be extended to a similar sequence of length $n + 1$. But this is a contradiction: since $H$ is a simulation, and since $\pi(n) \rightarrow_{A} \pi(n + 1)$, we can find a $b' \in B$ such that $\rho(n) \rightarrow_{B} b'$, and $\pi(n + 1)Hb'$. Therefore, we can define $\rho(n + 1) = b'$ and extend $\rho$ to $n + 1$ steps. q.e.d.

We will be essentially done if we prove the following,

**Lemma 2**: For $H$ an $AP$-simulation, if $aHb, \pi \in Path(A)_a,$ $\rho \in Path(B)_b$, and $\pi \hat{H} \rho$, then for each $\varphi \in LTL(AP)$ we have,

$$B, \rho \models \varphi \iff A, \pi \models \varphi.$$
Proof of Lemma 2: Note that, since $H$ is a simulation, for each $a' \in A, b' \in B$, we have $a' H b' \Rightarrow L_B(b') = L_A(a')$. But since $\pi \hat{H} \rho$, this means that we have the identity of traces: $\pi; L_A = \rho; L_B$, which immediately gives us the desired equivalence $\mathcal{B}, \rho \models \varphi \Leftrightarrow \mathcal{A}, \pi \models \varphi$. q.e.d.
Simulations Reflect Satisfaction of $LTL$ Formulae (V)

We are now essentially done. Suppose $B, b \models \varphi$ and $aHb$. Then we have, $B, \rho \models \varphi$ for any $\rho \in Path(B)_b$. To prove $A, a \models \varphi$, we have to show that for each $\pi \in Path(A)_a$ we have, $A, \pi \models \varphi$. But, by Lemma1, for each such $\pi$ we have a $\rho \in Path(B)_b$ such that $\pi \stackrel{H}{\Rightarrow} \rho$. Then, since $B, \rho \models \varphi$, by Lemma2, we have, $A, \pi \models \varphi$. q.e.d.

We say that an $AP$-simulation $H : A \rightarrow B$ preserves the satisfaction of an $LTL$ formula $\varphi$ iff $A, a \models \varphi$ and $aHb$ imply $B, b \models \varphi$.

**Corollary:** $AP$-bisimulations always reflect and preserve satisfaction of $LTL(AP)$ formulae.
To prove that a, possibly infinite-state, Kripke structure $\mathcal{A}$ and initial state $a$ satisfy an LTL formula $\varphi$, with, say, $AP$ the set of atomic propositions actually appearing in $\varphi$, it is enough to find an $AP$-simulation $H : \mathcal{A} \rightarrow \mathcal{B}$ and an initial state $b \in \mathcal{B}$ such that $aHb$ and:

- the set of states reachable from $b$ in $\mathcal{B}$ is finite, and
- $\mathcal{B}, b \models \varphi$.

Then, we can model check the property $\mathcal{B}, b \models \varphi$ to prove $\mathcal{A}, a \models \varphi$. Methods to find such an $H$ are called abstraction methods. What follows describes abstraction methods developed in joint work with Narciso Martí-Oliet and Miguel Palomino for systems specified by rewrite theories.
Let $\mathcal{A} = (A, \rightarrow, L_A)$ be a Kripke structure on $AP$. We call an equivalence relation $\equiv$ on $A$ label-preserving if $a \equiv a' \Rightarrow L_A(a) = L_A(a')$. We can use a label-preserving equivalence relation $\equiv$ to define a new Kripke structure, $\left(\mathcal{A}/\equiv\right) = (\mathcal{A}/\equiv, \rightarrow/\equiv, L_A/\equiv)$, where:

- $[a_1] \rightarrow/\equiv [a_2]$ iff $\exists a'_1 \in [a_1] \exists a'_2 \in [a_2]$ s.t. $a'_1 \rightarrow A a'_2$.

- $L_A/\equiv([a]) = L_A(a)$

It is then trivial to check that the projection map to equivalence classes $q_\equiv : a \mapsto [a]$ is an $AP$-simulation map $q_\equiv : \mathcal{A} \rightarrow \mathcal{A}/\equiv$, which we call the quotient abstraction defined by $\equiv$. 

Quotient Abstractions
We are of course particularly interested in abstraction methods for systems specified by rewrite theories. Recall that, given a rewrite theory \( \mathcal{R} = (\Sigma, E, \phi, R) \) plus equations \( D \) defining state predicates \( \Pi \) in a kind \( k \) of states, we have associated to it the Kripke structure,

\[
\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow_{\mathcal{R}}^1)^\bullet, L_{\Pi})
\]

This Kripke structure may be infinite-state, so that we cannot use an LTL model checker to verify its properties; or it can have a finite state space too big to make model checking feasible. We are therefore interested in defining quotient abstractions of Kripke structures of this kind.
Equational Quotient Abstractions (II)

It is enough to focus on those state predicates actually occurring in a particular formula \( \varphi \), which we may assume have been defined by the general method described when explaining the Maude \( LTL \) model checker.

Given \( \mathcal{R} = (\Sigma, E, \phi, R) \), we assume that \( (\Sigma', E \cup D) \) protects both \( (\Sigma, E) \) and \( \text{BOOL} \). For the atomic predicates \( AP_\Pi = \{ \theta(p) \mid p \in \Pi, \theta \text{ canonical ground substitution} \} \) we then have a labeling function,

\[
L_\Pi([t]) = \{ \theta(p) \in AP_\Pi \mid E \cup D \vdash t \models \theta(p) = \text{true} \}.
\]

We are then interested in the Kripke structure, \( \mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow_\mathcal{R})^\bullet, L_\Pi) \).
Then, a quite general method for defining quotient abstractions of the Kripke structure
\( \mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\rightarrow_{\mathcal{R}}^1)^\bullet, L_\Pi) \) is adapting to Kripke structures the method followed in Lecture 23 to define equational abstractions for invariants. That is, we can add to \( E \) a set \( G \) of equations such that \( (\Sigma', E \cup G \cup D) \) also protects \( \text{BOOL} \), and consider the rewrite theory
\( \mathcal{R}/G = (\Sigma, E \cup G, \phi, R) \), which has an associated Kripke structure \( \mathcal{K}(\mathcal{R}/G, k)_\Pi = (T_{\Sigma/E\cup G,k}, (\rightarrow_{\mathcal{R}/G}^1)^\bullet, L_{\Pi/G}) \), with
\[
L_{\Pi/G}([t]_{E\cup G}) = \{ \theta(p) \in AP_\Pi \mid E \cup G \cup D \vdash t \models = \theta(p) = \text{true} \}.
\]
Note that we have an equivalence relation \( \equiv_G \) on \( T_{\Sigma/E,k} \), namely,
\[
[t]_E \equiv_G [t']_E \iff E \cup G \vdash (\forall\emptyset) t = t' \iff [t]_{E \cup G} = [t']_{E \cup G}.
\]
And of course we have a bijection $T_{\Sigma/E,k}/\equiv G \cong T_{\Sigma/E\cup G,k}$. The key question now is: under what conditions is the map $q_{\equiv G} : [t]_E \mapsto [t]_{E\cup G}$ a quotient abstraction of Kripke structures

$$q_{\equiv G} : \mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/G, k)_\Pi$$

If it is so, we will call $\mathcal{K}(\mathcal{R}/G, k)_\Pi = (T_{\Sigma/E\cup G,k}, (\rightarrow_{\mathcal{R}/G}^1)^\bullet, L_{\Pi/G})$ the equational quotient abstraction of $\mathcal{K}(\mathcal{R}, k)_\Pi$ defined by $G$.

Of course, to use $\mathcal{R}/G$ as a system module in Maude to model check $LTL$ properties of $\mathcal{R}$, we will also need to require that $\mathcal{R}/G$ satisfies the usual executability conditions.
Let $k$ be a kind in a rewrite theory, $\mathcal{R}$ is $k$-deadlock-free if we have the following identity of binary relations on $T_{\Sigma/E,k}$:

$$\left(\rightarrow_1^1\mathcal{R}\right)^\cdot = \left(\rightarrow_1\mathcal{R}\right)$$

Intuitively this means that no states in $T_{\Sigma/E,k}$ are deadlock states, so the relation $\left(\rightarrow_1\mathcal{R}\right)$ is already total.

How restrictive is the requirement that $\mathcal{R}$ is $k$-deadlock-free? There is no real loss of generality. To a theory $\mathcal{R}$ satisfying the usual executability assumptions and having equational conditions we can always associate a semantically equivalent (from the $LTL$ point of view) theory $\mathcal{R}_d.f.$ which is $k$-deadlock-free (see Ex.26.6).

Our main theorem is then the following:
Theorem 2. Let $\mathcal{R} = (\Sigma, E, \phi, R)$ be a $k$-deadlock free rewrite theory, and let $D$ be equations defining (possibly parametric) state predicates $\Pi$ fully defined for all states of kind $k$ as either true or false, and assume that $(\Sigma', E \cup D)$ protects $\text{BOOL}$. Let then $G$ be a set of $\Sigma$-equations such that $(\Sigma', E \cup G \cup D)$ also protects $\text{BOOL}$. Then the map $q_{\equiv G} : [t]_E \mapsto [t]_{E \cup G}$ is a quotient abstraction of Kripke structures

$$q_{\equiv G} : \mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/G, k)_\Pi.$$ 

**Proof:** Since $\mathcal{R}$ is $k$-deadlock free, it is trivial to check that $\mathcal{R}/G$ is also $k$-deadlock free and that therefore we have

$$(\rightarrow^1_{\mathcal{R}/G})^* = (\rightarrow^1_{\mathcal{R}/G}) = (\rightarrow^1_{\mathcal{R}})^*/\equiv_G.$$
The only remaining thing to check is that $\equiv_G$ is label-preserving. This is equivalent to proving the following equivalences for each $p \in \Pi$ and ground substitution $\theta$:

\[
E \cup D \vdash t \models \theta(p) = \text{true} \quad \iff \quad E \cup G \cup D \vdash t \models \theta(p) = \text{true}
\]
\[
E \cup D \vdash t \models \theta(p) = \text{false} \quad \iff \quad E \cup G \cup D \vdash t \models \theta(p) = \text{false}
\]

The ($\Rightarrow$) follow by monotonicity of equational reasoning. The ($\Leftarrow$) follow from the protecting BOOL assumption, since we can reason by contradiction. Suppose, for example, that $E \cup G \cup D \vdash t \models \theta(p) = \text{true}$ but $E \cup D \vdash t \models \theta(p) \neq \text{true}$. By the protecting BOOL assumption this forces $E \cup D \vdash t \models \theta(p) = \text{false}$, which implies $E \cup G \cup D \vdash t \models \theta(p) = \text{false}$, contradicting the protection of BOOL. q.e.d.
Executability of Equational Quotient Abstractions

For Theorem 2 to be useful in practice, for example to use Maude to prove LTL such properties of $\mathcal{R}$ by model checking $\mathcal{R}/G$, we need to ensure the following executability requirements:

- $(\Sigma, E \cup G)$ and $(\Sigma', E \cup G \cup D)$ should be ground confluent, sort-decreasing and terminating.

- The rules $R$ should be ground coherent relative to $E \cup G$.

These requirements can be checked using Maude’s CRC, MTT, and ChC tools. Similarly, the protecting BOOL requirements can be checked using Maude’s CRC, MTT, and SCC tools.
The checks for executability may be positive or negative. But if, say, the equations $E \cup G$ are not ground confluent, we may be able to complete them to get a semantically equivalent set of equations, say $E'$ which is ground confluent, sort-decreasing, and terminating. Similarly, if the rules $R$ are not ground coherent, we may be able to complete them to get an equivalent set $R'$ of rules that is ground coherent. In this way we would obtain a rewrite theory $\mathcal{R}' = (\Sigma, E', \phi, R')$ semantically equivalent to $\mathcal{R}/G$.

Therefore we would have an isomorphism of a Kripke structures $\mathcal{K}(\mathcal{R}', k)_\Pi \cong \mathcal{K}(\mathcal{R}/G, k)_\Pi$, but now with the crucial property that $\mathcal{R}'$ is executable, so we can use $\mathcal{R}'$ to verify our desired $LTL$ properties about $\mathcal{R}$.
In the early stages of algebraic specification all formalisms would use a STACK data type as an example. In the relatively early stages of abstraction techniques it is likewise de rigueur to show how the technique in question handles Lamport’s bakery protocol.

This is an infinite-state protocol that achieves mutual exclusion between processes by the usual method common in bakeries and deli shops: there is a number dispenser, and customers are served in sequential order according to the number that they hold.

A simple Maude specification for the case of two processes 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_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
We may wish to verify two basic properties about this protocol, namely:

- **mutual exclusion**, that is, the two processes are never simultaneously in their critical section, and

- **liveness**, that is, whenever a process enters the waiting mode, it will eventually enter its critical section.

Since the set of states reachable from initial is **infinite**, we should model check these properties using an abstraction. We can define the abstraction by adding to the equations of BAKERY a set $G$ 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 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 > .
  eq < P, s s s X, Q, s s s Y > = < P, s s X, Q, s s Y > .
  eq < P, s s s X, Q, s s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
  eq < P, s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
endm

Three key questions are: (1) is the set of states now finite? (2) does this abstraction correspond to a rewrite theory whose equations are ground confluent, sort-decreasing and terminating? (3) are the rules still ground coherent?
A Bakery Protocol Example (IV)

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

When we give to the Maude Church-Rosser Checker (CRC) tool ABSTRACT-BAKERY module to check local confluence we get:

Maude> (check Church-Rosser ABSTRACT-BAKERY .)
Church-Rosser checking of ABSTRACT-BAKERY
Checking solution:
  All critical pairs have been joined. The specification is locally-confluent.
The specification is sort-decreasing.
It is also clear that the set of states is now finite, since in the canonical forms obtained with these equations the natural numbers possible in the state can never be greater than $s(s(0))$.

Note (exercise) that the equivalence on states defined by the above equations is: $\langle P, N, Q, M \rangle \equiv \langle P', N', Q', M' \rangle$ iff:

- $P = P'$ and $Q = Q'$
- $N = 0$ iff $N' = 0$
- $M = 0$ iff $M' = 0$
- $M < N$ iff $M' < N'$
This leaves us with the ground coherence question. Checking with Maude’s Coherence Checker (ChC) a version without predefined modules, in which true and false are replaced by tt and ff, respectively, gives us:

Maude> (check coherence ABSTRACT-BAKERY .)
Coherence checking of ABSTRACT-BAKERY
Coherence checking solution:

The following critical pairs cannot be rewritten:

- \( \text{cp} < \text{sleep}, 0, \text{Q@:Mode}, s 0 > \)
  \( \rightarrow < \text{wait}, s s s Y*@:Nat, \text{Q@:Mode}, s s Y*@:Nat > \).

- \( \text{cp} < \text{sleep}, s 0, \text{Q@:Mode}, s 0 > \)
  \( \rightarrow < \text{wait}, s s s Y*@:Nat, \text{Q@:Mode}, s s Y*@:Nat > \).

- \( \text{cp} < \text{P@:Mode}, s 0, \text{sleep}, 0 > \)
  \( \rightarrow < \text{P@:Mode}, s s X*@:Nat, \text{wait}, s s s X*@:Nat > \).

- \( \text{cp} < \text{P@:Mode}, s s 0, \text{sleep}, s 0 > \)
  \( \rightarrow < \text{P@:Mode}, s s X*@:Nat, \text{wait}, s s s X*@:Nat > \).
ccp < wait, s s 0, Q@:Mode, s 0 >
  => < crit, s s 0, Q@:Mode, s 0 >
  if s 0 < s s s X*@:Nat = ff .
ccp < wait, s s 0, Q@:Mode, s 0 >
  => < crit, s s 0, Q@:Mode, s 0 >
  if s s 0 < s s s X*@:Nat = ff .
ccp < wait, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  => < crit, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = ff .
ccp < P@:Mode, s 0, wait, s 0 >
  => < P@:Mode, s 0, crit, s 0 >
  if s s Y*@:Nat < s 0 = tt .
ccp < P@:Mode, s 0, wait, s 0 >
  => < P@:Mode, s 0, crit, s 0 >
  if s s Y*@:Nat < s s 0 = tt .
ccp < P@:Mode, s s X*@:Nat, wait, s s Y*@:Nat >
  => < P@:Mode, s s X*@:Nat, crit, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = tt .
To interpret these pairs the first key observation is that both \texttt{NAT} and \texttt{BOOL} are protected in \texttt{ABSTRACT-BAKERY}. This follows from the above check for confluence, plus the (postponed) proof of termination, plus the following sufficient completeness check,

\begin{verbatim}
Maude> (scc ABSTRACT-BAKERY .)
Success: ABSTRACT-BAKERY is sufficiently complete under the assumption that it is weakly-normalizing, confluent, and sort-decreasing.
\end{verbatim}

plus the observation that no equations involving either the successor function or zero, or \texttt{tt} or \texttt{ff} have been added in \texttt{ABSTRACT-BAKERY}. But since \texttt{NAT} and \texttt{BOOL} are protected, the only pairs with satisfiable conditions are the following:
A Bakery Protocol Example (IX)

cp < sleep, 0, Q@:Mode, s 0 >
  => < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp < sleep, s 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 > .
cp < P@:Mode, s s 0, sleep, s 0 >
  => < P@:Mode, s s X*@:Nat, wait, s s s X*@: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 s s s Y*@:Nat < s s s X*@:Nat = ff .
ccp < P@:Mode, s s X*@:Nat, wait, s s Y*@:Nat >
  => < P@:Mode, s s X*@:Nat, crit, s s Y*@:Nat >
  if s s 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:

\[ \text{cp} < \text{sleep}, 0, Q@:\text{Mode}, s \ 0 > \]
\[ => < \text{wait}, s \ s \ s \ Y@:\text{Nat}, Q@:\text{Mode}, s \ s \ Y@:\text{Nat} > . \]

We can first inductively prove the equation

\[ < \text{wait}, s \ s \ s \ Y:\text{Nat}, Q:\text{Mode}, s \ s \ Y:\text{Nat} > \]
\[ = < \text{wait}, 2, Q:\text{Mode}, 1 > . \]

by using the ITP with the following proof script;

(goal unique : ABSTRACT-BAKERY |- A{Y:Nat ; Q:Mode}
  ((< wait, s s s Y:Nat, Q:Mode, s s Y:Nat >)
   = (< wait, 2, Q:Mode, 1 >)) .)

(ind on Y:Nat .)
(auto .)
(auto .)
We can then check that the above critical pair fills in by giving the `search` command:

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

Solution 1 (state 1)
states: 2 rewrites: 1 in 0ms cpu (54ms real) (~ rews/sec)
X:BState --> < wait, 2, Q, 1 >

No more solutions.

Similarly, consider the first conditional critical pair, where, using the first equation among the inductive lemmas below

\[
\begin{align*}
\text{eq } s \ X < s \ Y &= X < Y . \\
\text{eq } 0 < s \ X &= \text{true .} \\
\text{eq } s \ X < 0 &= \text{false .} \\
\text{eq } X < s \ X &= \text{true .}
\end{align*}
\]
eq s X < X = false .
ceq X < s Y = true if X < Y .
ceq s X < Y = false if X < Y = false .

we can simplify its condition as follows:

```
cp < 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 = ff .
```

Using the **Theorem of Constants**, we can convert the variables X and Y into constants a and b and add an equation assuming the condition b < a = ff. Then, using also the above equations as extra lemmas, we can fill in this conditional critical pair by giving the `search` command:

```
Maude> search in ABSTRACT-BAKERY :
  < wait, s s a, Q, s s b > =>1 X:BState .
```
Solution 1 (state 1)
X:BState --> < crit, s_~2(a), Q, s_~2(b) >

No more solutions.

What about state predicates? Are they preserved by the abstraction? In order to specify the desired mutual exclusion and liveness properties, it is enough to specify in Maude the following state predicates:

mod BAKERY-PRED 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

These predicates are then imported without change, together with ABSTRACT-BAKERY, in the module:

mod ABSTRACT-BAKERY-PREDS is
    protecting ABSTRACT-BAKERY .
    including BAKERY-PREDS .
endm

By the second part of the proof of Theorem 2 (which does
not use the deadlock-freedom assumption), the preservation of these state predicates can be guaranteed if we show that both BAKERY-PREDS and ABSTRACT-BAKERY-PREDS protect BOOL. This follows from the absence of any equations having true or false in their lefthand sides plus the following facts, all of which are checked by Maude tools (after replacing the predefined modules NAT and BOOL by equivalent specifications when necessary):

1. both BAKERY-PREDS and ABSTRACT-BAKERY-PREDS are sufficiently complete;

2. both BAKERY-PREDS and ABSTRACT-BAKERY-PREDS are locally confluent and sort-decreasing;

3. both BAKERY-PREDS and ABSTRACT-BAKERY-PREDS are terminating.
The SCC tool checks fact (1):

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

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

The CRC tool checks fact (2):

Maude> (check Church-Rosser BAKERY-PREDs .)
All critical pairs have been joined. The specification is
locally-confluent.
The specification is sort-decreasing.

Maude> (check Church-Rosser ABSTRACT-BAKERY-PREDs .)
All critical pairs have been joined. The specification is
locally-confluent.
The specification is sort-decreasing.

All we have left is checking termination of the equations in BAKERY-PREDs and ABSTRACT-BAKERY-PREDs, that is, fact (3), plus the pending proof of terminating equations for ABSTRACT-BAKERY. But since the equations in ABSTRACT-BAKERY-PREDs are precisely the union of those in ABSTRACT-BAKERY and BAKERY-PREDs, it is enough to check that ABSTRACT-BAKERY-PREDs is terminating. This check succeeds with the Maude Termination Tool (MTT).

This finishes all the checks of correctness and executability. The only remaining issue is deadlock freedom, which is required for the correctness of the abstraction. To ensure deadlock freedom we can perform the automatic module transformation described in Section 15.3 of the Maude Book, that preserves all the desired executability properties.
to obtain a semantically equivalent, deadlock-free version, say DF-BAKERY, of BAKERY. After loading the BAKERY module, we can obtain its deadlock-free version DF-BAKERY by performing this transformation in Full Maude (with the extension described in the Maude Book 15.3) as follows:

(mod DF-BAKERY is
    protecting DEADLOCK-FREE[BAKERY, BState] .
endm)
Note that the kind of states in DEADLOCK-FREE[BAKERY, BState] has changed. It is now [EConfig] and there is an operator

\[ \text{op} \{\_\} : \text{State} \to \text{EConfig} \].

wrapping each state of kind [BState] into a state of kind [EConfig]. This means that we have to slightly redefine our state predicates, since they take now states of kind [EConfig], as follows:
(mod DF-BAKERY-PREDs is
   protecting DF-BAKERY .
   including SATISFACTION .
   subsort EConfig < 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)
Our desired module DF-ABSTRACT-BAKERY has exactly the same equations as before, but now includes DF-BAKERY instead of BAKERY.

(mod DF-ABSTRACT-BAKERY is
    including DF-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 > .
    eq < P, s s X, Q, s s s Y > = < P, s s X, Q, s s Y > .
    eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
    eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
    eq < P, s s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
    eq < P, s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
    endm)

Finally we can verify our two desired properties as follows:

(mod DF-ABSTRACT-BAKERY-PREDs is
protecting DF-ABSTRACT-BAKERY.
including DF-BAKERY-PREDS.
endm)

(mod DF-ABSTRACT-BAKERY-CHECK is
   including MODEL-CHECKER.
   including DF-ABSTRACT-BAKERY-PREDS.
endm)

Maude> (red modelCheck({initial}, []~(1crit \ 2crit)) .)
reduce in DF-ABSTRACT-BAKERY-CHECK :
modelCheck({initial}, []~(1crit \ 2crit))
result Bool :
  true

Maude> (red modelCheck({initial},
                         (1wait |-\> 1crit) \ (2wait |-\> 2crit)) .)
reduce in DF-ABSTRACT-BAKERY-CHECK :
modelCheck({initial}, (1wait |-\> 1crit) \ (2wait |-\> 2crit))
result Bool :
  true