Program Verification: Lecture 25

José Meseguer

University of Illinois at Urbana-Champaign

Folding Narrowing Verification in Maude

For $\mathcal{R} = (\Omega, B, R)$ a topmost rewrite theory with state sort St, $u_1 \vee \ldots \vee u_n$ an inititial state, and $Q \subseteq T_{\Omega/B,St}$, folding narrowing verification of an invariant $(\dagger)$ $\mathbb{C}_{\mathcal{R}}, [\![u_1 \vee \ldots \vee u_n]\!] \models_{S4} \Box Q$ is supported by Maude in the following ways:

Folding Narrowing Verification in Maude

For $\mathcal{R} = (\Omega, B, R)$ a topmost rewrite theory with state sort St, $u_1 \vee \ldots \vee u_n$ an inititial state, and $Q \subseteq T_{\Omega/B,St}$, folding narrowing verification of an invariant $(\dagger)$ $\mathbb{C}_{\mathcal{R}}, [\![u_1 \vee \ldots \vee u_n]\!] \models_{S4} \Box Q$ is supported by Maude in the following ways:

A. If $Q = [\![\neg v_1 \wedge \ldots \wedge \neg v_m]\!]$, by **Method 1** in Lecture 24, $(\dagger)$ holds if the m commands `{fold} vu-narrow` $u_1 \vee \ldots \vee u_n$ `=>*` v_j, $1 \leq j \leq m$ return: `No solution`.

Folding Narrowing Verification in Maude

For $\mathcal{R} = (\Omega, B, R)$ a topmost rewrite theory with state sort St, $u_1 \vee \ldots \vee u_n$ an inititial state, and $Q \subseteq T_{\Omega/B,St}$, folding narrowing verification of an invariant (†) $\mathbb{C}_{\mathcal{R}}, [\![u_1 \vee \ldots \vee u_n]\!] \models_{S4} \Box Q$ is supported by Maude in the following ways:

A. If $Q = [\![\neg v_1 \wedge \ldots \wedge \neg v_m]\!]$, by **Method 1** in Lecture 24, (†) holds if the m commands $\{$`fold`$\}$ `vu-narrow` $u_1 \vee \ldots \vee u_n$ `=>*` v_j, $1 \leq j \leq m$ return: `No solution`.

W.L.O.G Maude assumes and requires that $vars(u_i) \cap vars(u_{i'}) = \emptyset$, $1 \leq i < i' \leq n$, and of course that $vars(u_i) \cap vars(v_j) = \emptyset$, $1 \leq i \leq n$, $1 \leq j \leq m$.

Folding Narrowing Verification in Maude

For $\mathcal{R} = (\Omega, B, R)$ a topmost rewrite theory with state sort St, $u_1 \vee \ldots \vee u_n$ an inititial state, and $Q \subseteq T_{\Omega/B,St}$, folding narrowing verification of an invariant (†) $\mathbb{C}_{\mathcal{R}}, [\![u_1 \vee \ldots \vee u_n]\!] \models_{S4} \Box Q$ is supported by Maude in the following ways:

A. If $Q = [\![\neg v_1 \wedge \ldots \wedge \neg v_m]\!]$, by **Method 1** in Lecture 24, (†) holds if the m commands `{fold} vu-narrow` $u_1 \vee \ldots \vee u_n$ `=>*` v_j, $1 \leq j \leq m$ return: `No solution`.

W.L.O.G Maude assumes and requires that $vars(u_i) \cap vars(u_{i'}) = \emptyset$, $1 \leq i < i' \leq n$, and of course that $vars(u_i) \cap vars(v_j) = \emptyset$, $1 \leq i \leq n$, $1 \leq j \leq m$.

As explained in Lecture 23, the more general the initial state, the better, since this increases the chances that `{fold} vu-narrow` commands will succeeed. Let us see an example.

Folding Narrowing Verification in Maude (II)

Recall the `R&W` module, where to enable folding narrowing rules `must be declared` with the `[narrowing]` attribute.

Folding Narrowing Verification in Maude (II)

Recall the R&W module, where to enable folding narrowing rules must be declared with the [narrowing] attribute.

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

  vars R W N M I J : Nat .

  rl < 0, 0 > => < 0, s(0) > [narrowing] .
  rl < R, s(W) > => < R, W > [narrowing] .
  rl < R, 0 > => < s(R), 0 > [narrowing] .
  rl < s(R), W > => < R, W > [narrowing] .
endm
```

Folding Narrowing Verification in Maude (II)

Recall the `R&W` module, where to enable folding narrowing rules must be declared with the `[narrowing]` attribute.

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

  vars R W N M I J : Nat .

  rl < 0, 0 > => < 0, s(0) > [narrowing] .
  rl < R, s(W) > => < R, W > [narrowing] .
  rl < R, 0 > => < s(R), 0 > [narrowing] .
  rl < s(R), W > => < R, W > [narrowing] .
endm
```

The `{fold} vu-narrow` command from initial state `< 0, 0 >` will not terminate. We can try the more general state `< R, 0 >` to verify **mutual exclusion**.

Folding Narrowing Verification in Maude (III)

```
Maude> {fold} vu-narrow < R,0 > =>* < s(N),s(M) > .

No solution.
```

Folding Narrowing Verification in Maude (III)

```
Maude> {fold} vu-narrow < R,0 > =>* < s(N),s(M) > .

No solution.
```

This command terminated because the folding variant narrowing algorithm computed at "fixpoint" P_d some depth d s.t. $F_{d+1} = \bot$.

Folding Narrowing Verification in Maude (III)

```
Maude> {fold} vu-narrow < R,0 > =>* < s(N),s(M) > .

No solution.
```

This command terminated because the folding variant narrowing algorithm computed at "fixpoint" P_d some depth d s.t. $F_{d+1} = \bot$. We can ask Maude to display such a P_d with the command:

Folding Narrowing Verification in Maude (III)

```
Maude> {fold} vu-narrow < R,0 > =>* < s(N),s(M) > .

No solution.
```

This command terminated because the folding variant narrowing algorithm computed at "fixpoint" P_d some depth d s.t. $F_{d+1} = \bot$. We can ask Maude to display such a P_d with the command:

```
Maude> show most general states .
< #1:Nat, 0 > \/
< 0, s(0) >
Maude>
```

By **Method 2** in Lecture 24, we can now verify any other invariant $Q = [\![\neg v_1 \wedge \ldots \wedge \neg v_m]\!]$ from `< R,0 >` by checking that $P_d \wedge v_j = \bot$, $1 \leq j \leq m$, which (see Appendix 1 to Lecture 24) can be computed by unification.

Folding Narrowing Verification in Maude (IV)

For example, we can verify the **One-writer** invariant by the unification commands:

Folding Narrowing Verification in Maude (IV)

For example, we can verify the **One-writer** invariant by the unification commands:

```
Maude> unify < R,0 > =? < N,s(s(M)) > .

No unifier.
Maude> unify < 0, s(0) > =? < N,s(s(M)) > .

No unifier.
```

Folding Narrowing Verification in Maude (IV)

For example, we can verify the **One-writer** invariant by the unification commands:

```
Maude> unify < R,0 > =? < N,s(s(M)) > .

No unifier.
Maude> unify < 0, s(0) > =? < N,s(s(M)) > .

No unifier.
```

That is, once we have found the fixpoint `< R,0 >` `< 0, s(0) >` there is no need to use the `{fold} vu-narrow` command to verify any other invariant from `< R,0 >`, since unification suffices.

Folding Narrowing Verification in Maude (IV)

For example, we can verify the **One-writer** invariant by the unification commands:

```
Maude> unify < R,0 > =? < N,s(s(M)) > .

No unifier.
Maude> unify < 0, s(0) > =? < N,s(s(M)) > .

No unifier.
```

That is, once we have found the fixpoint `< R,0 >` `< 0, s(0) >` there is no need to use the `{fold} vu-narrow` command to verify any other invariant from `< R,0 >`, since unification suffices. Of course, **One-writer** could also be verified with the `{fold}` `vu-narrow` command:

Folding Narrowing Verification in Maude (IV)

For example, we can verify the **One-writer** invariant by the unification commands:

```
Maude> unify < R,0 > =? < N,s(s(M)) > .

No unifier.
Maude> unify < 0, s(0) > =? < N,s(s(M)) > .

No unifier.
```

That is, once we have found the fixpoint `< R,0 >` `< 0, s(0) >` there is no need to use the {`fold`} `vu-narrow` command to verify any other invariant from `< R,0 >`, since unification suffices. Of course, **One-writer** could also be verified with the {`fold`} `vu-narrow` command:

```
Maude> {fold} vu-narrow < R,0 > =>* < N,s(s(M)) > .

No solution.
```

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$).

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$). A decidable sufficient condition is $P_d \sqsubseteq_B v_1 \vee \ldots \vee v_m$ (resp. $p \sqsubseteq_B v_1 \vee \ldots \vee v_m$).

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$). A decidable sufficient condition is $P_d \sqsubseteq_B v_1 \vee \ldots \vee v_m$ (resp. $p \sqsubseteq_B v_1 \vee \ldots \vee v_m$).

This method can be quite useful to prove, for example, that `R&W` is deadlock-free. That is, that `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` is an invariant from `< R,0 >`.

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$). A decidable sufficient condition is $P_d \sqsubseteq_B v_1 \vee \ldots \vee v_m$ (resp. $p \sqsubseteq_B v_1 \vee \ldots \vee v_m$).

This method can be quite useful to prove, for example, that `R&W` is deadlock-free. That is, that `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` is an invariant from `< R,0 >`. All we need to show is that `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >`.

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$). A decidable sufficient condition is $P_d \sqsubseteq_B v_1 \vee \ldots \vee v_m$ (resp. $p \sqsubseteq_B v_1 \vee \ldots \vee v_m$).

This method can be quite useful to prove, for example, that `R&W` is deadlock-free. That is, that `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` is an invariant from `< R,0 >`. All we need to show is that `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >`. This holds trivially for `< R,0 >`.

Folding Narrowing Verification in Maude (V)

B. Let Q be specifiable as $Q = [\![v_1 \vee \ldots \vee v_m]\!]$. By **Method 3** in Lecture 24, If we have found a P_d (resp. positive formula p) s.t. $[\![P_d]\!] = \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (resp. $[\![p]\!] \supseteq \mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$), then invariant (†) holds for any such Q iff $P_d \subseteq_B v_1 \vee \ldots \vee v_m$ (resp. if $p \subseteq_B v_1 \vee \ldots \vee v_m$). A decidable sufficient condition is $P_d \sqsubseteq_B v_1 \vee \ldots \vee v_m$ (resp. $p \sqsubseteq_B v_1 \vee \ldots \vee v_m$).

This method can be quite useful to prove, for example, that `R&W` is deadlock-free. That is, that `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` is an invariant from `< R,0 >`. All we need to show is that `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >`. This holds trivially for `< R,0 >`. It also holds for `< 0, s(0) >` because `< 0, s(0) >` $\sqsubseteq$ `< R, s(W) >`.

A Shortcut for Proving Deadlock Freedom

The subsumption check $\texttt{< R,0 >} \vee \texttt{< 0, s(0) >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< R, 0 >} \vee \texttt{< s(R), W >}$ for R&W is trivial.

A Shortcut for Proving Deadlock Freedom

The subsumption check $\texttt{< R,0 >} \vee \texttt{< 0, s(0) >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< R, 0 >} \vee \texttt{< s(R), W >}$ for $\texttt{R\&W}$ is trivial.

In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form

$(\sharp_j)\quad w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m,\ 1 \leq j \leq k,$

A Shortcut for Proving Deadlock Freedom

The subsumption check $\texttt{< R,0 >} \vee \texttt{< 0, s(0) >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< R, 0 >} \vee \texttt{< s(R), W >}$ for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)\ \ w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$.

A Shortcut for Proving Deadlock Freedom

The subsumption check `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)$ $w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$. In the worse case this may require $k \times m$ checks of the form $w_j \sqsubseteq_B v_i$.

A Shortcut for Proving Deadlock Freedom

The subsumption check `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)$ $w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$. In the worse case this may require $k \times m$ checks of the form $w_j \sqsubseteq_B v_i$. This can be automated by $k \times m$ Maude matching commands: `match [1]` v_i `<=?` w_j.

A Shortcut for Proving Deadlock Freedom

The subsumption check `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)\;\; w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$. In the worse case this may require $k \times m$ checks of the form $w_j \sqsubseteq_B v_i$. This can be automated by $k \times m$ Maude matching commands: `match [1]` v_i `<=?` w_j.

But since the v_i are the rule's lefthand sides, each $(\sharp_j)$ holds if the search command: `search [1]` w_j `=>1` $S : St$ has a solution.

A Shortcut for Proving Deadlock Freedom

The subsumption check `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)$ $w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$. In the worse case this may require $k \times m$ checks of the form $w_j \sqsubseteq_B v_i$. This can be automated by $k \times m$ Maude matching commands: `match [1]` v_i `<=?` w_j.

But since the v_i are the rule's lefthand sides, each $(\sharp_j)$ holds if the search command: `search [1]` w_j `=>1` $S : St$ has a solution. E.g.,

A Shortcut for Proving Deadlock Freedom

The subsumption check `< R,0 >` $\vee$ `< 0, s(0) >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< R, 0 >` $\vee$ `< s(R), W >` for `R&W` is trivial. In general, for $P_d = w_1 \vee \ldots \vee w_k$ we need k checks of the form $(\sharp_j)$ $w_j \sqsubseteq_B v_1 \vee \ldots \vee v_m$, $1 \leq j \leq k$, where the v_i, $1 \leq i \leq m$, are the lefthand sides of the rules in $\mathcal{R}$. In the worse case this may require $k \times m$ checks of the form $w_j \sqsubseteq_B v_i$. This can be automated by $k \times m$ Maude matching commands: `match [1]` v_i `<=?` w_j.

But since the v_i are the rule's lefthand sides, each $(\sharp_j)$ holds if the search command: `search [1]` w_j `=>1` $S : St$ has a solution. E.g.,

```
Maude> search [1] < R, 0 > =>1 C:Config .

Solution 1 (state 1) C:Config --> < s(R), 0 >

Maude> search [1] < 0, s(0) > =>1 C:Config .

Solution 1 (state 1) C:Config --> < 0, 0 >
```

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if:

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$.

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$. (2) Q is transititition-closed;

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$. (2) Q is transititition-closed; this holds iff a `{fold} vu-narrow` $v_1 \vee \ldots \vee v_m$ `=>1 $ .` command, where `$` is a fresh, unreachable constant added to $\mathcal{R}$, generates an $F_1(v_1 \vee \ldots \vee v_m)$ s.t. $F_1(v_1 \vee \ldots \vee v_m) \subset_B v_1 \vee \ldots \vee v_m$;

Folding Narrowing Verification in Maude (VI)

By **Method 4**, ($\dagger$) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$. (2) Q is transitition-closed; this holds iff a `{fold} vu-narrow` $v_1 \vee \ldots \vee v_m$ `=>1 $ .` command, where $ is a fresh, unreachable constant added to $\mathcal{R}$, generates an $F_1(v_1 \vee \ldots \vee v_m)$ s.t. $F_1(v_1 \vee \ldots \vee v_m) \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $F_1(v_1 \vee \ldots \vee v_m) = \perp$.

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$. (2) Q is transititition-closed; this holds iff a `{fold} vu-narrow` $v_1 \vee \ldots \vee v_m$ `=>1 $ .` command, where `$` is a fresh, unreachable constant added to $\mathcal{R}$, generates an $F_1(v_1 \vee \ldots \vee v_m)$ s.t. $F_1(v_1 \vee \ldots \vee v_m) \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $F_1(v_1 \vee \ldots \vee v_m) = \bot$.

This method provides, for example, an alternative way of proving that `R&W` is deadlock-free from `< R,0 >`.

Folding Narrowing Verification in Maude (VI)

By **Method 4**, (†) holds for $Q = [\![v_1 \vee \ldots \vee v_m]\!]$ if: (1) $u_1 \vee \ldots \vee u_n \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $u_1 \vee \ldots \vee u_n \sqsubset_B v_1 \vee \ldots \vee v_m$. (2) Q is transititition-closed; this holds iff a `{fold} vu-narrow` $v_1 \vee \ldots \vee v_m$ `=>1 $ .` command, where `$` is a fresh, unreachable constant added to $\mathcal{R}$, generates an $F_1(v_1 \vee \ldots \vee v_m)$ s.t. $F_1(v_1 \vee \ldots \vee v_m) \subset_B v_1 \vee \ldots \vee v_m$; a decidable sufficient condition is $F_1(v_1 \vee \ldots \vee v_m) = \bot$.

This method provides, for example, an alternative way of proving that `R&W` is deadlock-free from `< R,0 >`. The module adding the unreachable fresh constant `$` to the kind `[Config]` is:

Folding Narrowing Verification in Maude (VII)

```
mod R&W is
  sorts Nat Config .
  op <_,_> : Nat Nat -> Config [ctor] .
  op $ : -> [Config] .      *** unreachable state
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  vars R W N M I J : Nat .

  rl < 0, 0 > => < 0, s(0) > [narrowing] .
  rl < R, s(W) > => < R, W > [narrowing] .
  rl < R, 0 > => < s(R), 0 > [narrowing] .
  rl < s(R), W > => < R, W > [narrowing] .
endm
```

Folding Narrowing Verification in Maude (VII)

```
mod R&W is
  sorts Nat Config .
  op <_,_> : Nat Nat -> Config [ctor] .
  op $ : -> [Config] .     *** unreachable state
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  vars R W N M I J : Nat .

  rl < 0, 0 > => < 0, s(0) > [narrowing] .
  rl < R, s(W) > => < R, W > [narrowing] .
  rl < R, 0 > => < s(R), 0 > [narrowing] .
  rl < s(R), W > => < R, W > [narrowing] .
endm

{fold} vu-narrow in R&W : < 0, 0 > \/ < R, s(W) > \/ < N, 0 > \/ < s(M), I >
    =>1 $ .

No solution.
Maude> show frontier states .
< @1:Nat, @2:Nat >
```

Folding Narrowing Verification in Maude (VII)

```
mod R&W is
  sorts Nat Config .
  op <_,_> : Nat Nat -> Config [ctor] .
  op $ : -> [Config] .      *** unreachable state
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  vars R W N M I J : Nat .

  rl < 0, 0 > => < 0, s(0) > [narrowing] .
  rl < R, s(W) > => < R, W > [narrowing] .
  rl < R, 0 > => < s(R), 0 > [narrowing] .
  rl < s(R), W > => < R, W > [narrowing] .
endm

{fold} vu-narrow in R&W : < 0, 0 > \/ < R, s(W) > \/ < N, 0 > \/ < s(M), I >
    =>1 $ .

No solution.
Maude> show frontier states .
< @1:Nat, @2:Nat >
```

We just need to check conditions (1)–(2).

Folding Narrowing Verification in Maude (VIII)

Condition (1) is: `< R,0 >` $\sqsubseteq$ `< 0, 0 >` $\vee$ `< R, s(W) >` $\vee$ `< N, 0 >` $\vee$ `< s(M), I >`, which holds trivially.

Folding Narrowing Verification in Maude (VIII)

Condition (1) is: $\texttt{< R,0 >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$, which holds trivially.

Condition (2) is: $\texttt{< I,J >} \subseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$.

Folding Narrowing Verification in Maude (VIII)

Condition (1) is: $\texttt{< R,0 >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$, which holds trivially.

Condition (2) is: $\texttt{< I,J >} \subseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$. This holds because by the **Pattern Decomposition Lemma** in pg. 6 of Lecture 24, using the generator set $\{0, s(K)\}$ for sort `Nat`, this follows from $\texttt{< I,0 >} \vee \texttt{< I,s(K) >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$,

Folding Narrowing Verification in Maude (VIII)

Condition (1) is: $\texttt{< R,0 >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$, which holds trivially.

Condition (2) is: $\texttt{< I,J >} \subseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$. This holds because by the **Pattern Decomposition Lemma** in pg. 6 of Lecture 24, using the generator set $\{0, s(K)\}$ for sort $\texttt{Nat}$, this follows from $\texttt{< I,0 >} \vee \texttt{< I,s(K) >} \sqsubseteq \texttt{< 0, 0 >} \vee \texttt{< R, s(W) >} \vee \texttt{< N, 0 >} \vee \texttt{< s(M), I >}$, which holds trivially.

A Fair R&W Protocol Case Study

R&W is unfair. The infinite behavior below starves all readers:

A Fair R&W Protocol Case Study

R&W is unfair. The infinite behavior below starves all readers:

```
< 0,0 > => < 0,s(0) > => < 0,0 > => < 0,s(0) > ...
```

A Fair R&W Protocol Case Study

R&W is unfair. The infinite behavior below starves all readers:

```
< 0,0 > => < 0,s(0) > => < 0,0 > => < 0,s(0) > ...
```

This problem is resoved by the following R&W fair protocol:

A Fair R&W Protocol Case Study

R&W is unfair. The infinite behavior below starves all readers:

```
< 0,0 > => < 0,s(0) > => < 0,0 > => < 0,s(0) > ...
```

This problem is resoved by the following R&W fair protocol:

```
mod R&W-FAIR is sorts NzNat Nat Conf .  subsorts NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op 1 : -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
  op _+_ : NzNat Nat -> NzNat [ctor assoc comm id: 0] .
  op <_,_>[_|_] : Nat Nat Nat Nat -> Conf .
  op $ : -> [Conf] .
  op init : NzNat -> Conf .

  vars N N1 N2 N3 N4 M M1 M2 K K1 K2 I J : Nat . vars N' N1' N2' N3' M' : NzNat

  eq init(N') = < 0,0 >[ 0 | N'] .
  rl [w-in] : < 0,0 >[ 0 | N] => < 0,1 >[0 | N] [narrowing] .
  rl [w-out] : < 0,1 >[ 0 | N] => < 0,0 >[N | 0] [narrowing] .
  rl [r-in] : < N,0 >[M + 1 | K] => < N + 1,0 >[M | K] [narrowing] .
  rl [r-out] : < N + 1,0 >[M | K] => < N,0 >[M | K + 1] [narrowing] .
endm
```

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p?

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that:

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that: (i) $u_1 \vee \ldots \vee u_n \subseteq_B p$, and

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that: (i) $u_1 \vee \ldots \vee u_n \subseteq_B p$, and (ii) p is transition-closed.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that: (i) $u_1 \vee \ldots \vee u_n \subseteq_B p$, and (ii) p is transition-closed. this can be checked by command `{fold} vu-narrow` p `=>1 $ .`

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that: (i) $u_1 \vee \ldots \vee u_n \subseteq_B p$, and (ii) p is transition-closed. this can be checked by command `{fold} vu-narrow` p `=>1 $ .` Even if $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!] \subset [\![p]\!]$, p can still be very useful for Methods 3–4.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$

A possitive pattern formula p specifying the set of all reachable states $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ can be obtained by a terminating with no solution a folding narrowing search from $u_1 \vee \ldots \vee u_n$. A second approach is to guess it, by guessing the patterns that describe (or over-approximate) $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ from some initial states $u_1 \vee \ldots \vee u_n$. As Methods 3–4 show, such p can be very useful.

How can we guess p? By reflecting on the rules in $\mathcal{R}$ to guess a p such that: (i) $u_1 \vee \ldots \vee u_n \subseteq_B p$, and (ii) p is transition-closed. this can be checked by command `{fold} vu-narrow` p `=>1 $ .` Even if $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!] \subset [\![p]\!]$, p can still be very useful for Methods 3–4.

Let us do this for `R&W-FAIR` with initial state `< 0,0 >[ 0 | N']`.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden.

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules?

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

```
< 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/  < M,0 >[N1 + 1 | K] \/
< N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
```

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in < 0,0 >[0 | N'] variable N' has sort NnNat, there is at least one reading process. To guess the pattern, we can think about the case N' $=$ 1, and of the different containers in <_,_>[_|_] as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

```
< 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/  < M,0 >[N1 + 1 | K] \/
< N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
```

This guess is an invariant by **Method 4** because:

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

```
< 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/  < M,0 >[N1 + 1 | K] \/
< N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
```

This guess is an invariant by **Method 4** because: (i) it B-subsumes `< 0,0 >[ 0 | N']` when decomposed with generator set $\{n + 1\}$ for `N'`; and

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

```
< 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/  < M,0 >[N1 + 1 | K] \/
< N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
```

This guess is an invariant by **Method 4** because: (i) it B-subsumes `< 0,0 >[ 0 | N']` when decomposed with generator set $\{n+1\}$ for `N'`; and (ii) it is transition closed:

Guessing a Pattern Formula for $\mathcal{R}^*[\![u_1 \vee \ldots \vee u_n]\!]$ (II)

Since in `< 0,0 >[ 0 | N']` variable `N'` has sort `NnNat`, there is at least one reading process. To guess the pattern, we can think about the case `N'` $= 1$, and of the different containers in `<_,_>[_|_]` as places where the "pea" 1 could be hidden. Can we guess where it can be, looking at the rules? Here is a guess inspired by the "pea" idea, yet fully general:

```
< 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/  < M,0 >[N1 + 1 | K] \/
< N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
```

This guess is an invariant by **Method 4** because: (i) it B-subsumes `< 0,0 >[ 0 | N']` when decomposed with generator set $\{n+1\}$ for `N'`; and (ii) it is transition closed:

```
Maude> {fold} vu-narrow  < 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/
< M,0 >[N1 + 1 | K] \/ < N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1] =>1 $ .

Maude> show frontier states .
*** frontier is empty ***
```

Verifying Some Properties of `R&W-FAIR`

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form

Verifying Some Properties of `R&W-FAIR`

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form $\neg v_1 = \neg$ `< 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat]` and

Verifying Some Properties of `R&W-FAIR`

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form $\neg v_1 = \neg$ `< 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat]` and $\neg v_2 = \neg$ `< m:Nat , 1 + 1 + i:Nat >[j:Nat | k:Nat]`.

Verifying Some Properties of R&W-FAIR

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form $\neg v_1 = \neg$ `< 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat]` and $\neg v_2 = \neg$ `< m:Nat , 1 + 1 + i:Nat >[j:Nat | k:Nat]`. By **Method 2** we just need to check that $p \wedge v_1 = \bot$, and $p \wedge v_2 = \bot$ by unification.

Verifying Some Properties of R&W-FAIR

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form $\neg v_1 = \neg$ `< 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat]` and $\neg v_2 = \neg$ `< m:Nat , 1 + 1 + i:Nat >[j:Nat | k:Nat]`. By **Method 2** we just need to check that $p \wedge v_1 = \bot$, and $p \wedge v_2 = \bot$ by unification. But this intersection check is automated by the commands:

Verifying Some Properties of `R&W-FAIR`

The **Mutual Exclusion** and **One-writer** invariants can be specified by negative patterns of the form $\neg v_1 = \neg$ `< 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat]` and $\neg v_2 = \neg$ `< m:Nat , 1 + 1 + i:Nat >[j:Nat | k:Nat]`. By **Method 2** we just need to check that $p \wedge v_1 = \bot$, and $p \wedge v_2 = \bot$ by unification. But this intersection check is automated by the commands:

```
Maude> {fold} vu-narrow  < 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/
< M,0 >[N1 + 1 | K] \/ < N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
=>*  < 1 + m:Nat , 1 + i:Nat >[j:Nat | k:Nat] .

No solution.

Maude> {fold} vu-narrow  < 0,0 >[ 0 | N + 1] \/ < 0, 1 >[0 | N3 + 1] \/
< M,0 >[N1 + 1 | K] \/ < N2 + 1,0 >[M1 | K1] \/ < N4,0 >[M2 | K2 + 1]
=>*  < m:Nat , 1 + 1 + i:Nat >[j:Nat | k:Nat] .

No solution.
```

Verifying Some Properties of `R&W-FAIR` (II)

We can now prove deadlock freedom of `R&W-FAIR` from `< 0,0 >[ 0 | N']` by **Method 3**.

Verifying Some Properties of R&W-FAIR (II)

We can now prove deadlock freedom of R&W-FAIR from $<$ 0,0 $>$[0 | N'] by **Method 3**. That is, by showing that $p \subseteq_B$ $<$ 0,0 $>$[0 | N] $\vee$ $<$ 0,1 $>$[0 | N] $\vee$ $<$ N,0 $>$[M + 1 | K] $\vee$ $<$ N + 1,0 $>$[M | K].

Verifying Some Properties of R&W-FAIR (II)

We can now prove deadlock freedom of R&W-FAIR from $< 0,0 >$[0 | N'] by **Method 3**. That is, by showing that $p \subseteq_B$ $< 0,0 >$[0 | N] $\vee$ $< 0,1 >$[0 | N] $\vee$ $< N,0 >$[M + 1 | K] $\vee$ $< N + 1,0 >$[M | K]. Furthermore, we can do so using the shortcut suggested in pg. 7:

Verifying Some Properties of R&W-FAIR (II)

We can now prove deadlock freedom of `R&W-FAIR` from `< 0,0 >[ 0 | N']` by **Method 3**. That is, by showing that $p \subseteq_B$ `< 0,0 >[ 0 | N]` $\vee$ `< 0,1 >[ 0 | N]` $\vee$ `< N,0 >[M + 1 | K]` $\vee$ `< N + 1,0 >[M | K]`. Furthermore, we can do so using the shortcut suggested in pg. 7:

```
search [1] < 0,0 >[ 0 | N + 1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 0, 1 >[0 | 1 + N]

search [1] < 0, 1 >[0 | N3 + 1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 0, 0 >[1 + N3 | 0]
```

Verifying Some Properties of R&W-FAIR (III)

```
search [1] < M,0 >[N1 + 1 | K] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 1 + M, 0 >[N1 | K]

search [1] < N2 + 1,0 >[M1 | K1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < N2, 0 >[M1 | 1 + K1]

search [1] < N4,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

Verifying Some Properties of R&W-FAIR (III)

```
search [1] < M,0 >[N1 + 1 | K] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 1 + M, 0 >[N1 | K]

search [1] < N2 + 1,0 >[M1 | K1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < N2, 0 >[M1 | 1 + K1]

search [1] < N4,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

The problem with pattern `< N4,0 >[M2 | K2 + 1]` is that is too general to be rewritten by the rules of `R&W-FAIR`.

Verifying Some Properties of R&W-FAIR (III)

```
search [1] < M,0 >[N1 + 1 | K] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 1 + M, 0 >[N1 | K]

search [1] < N2 + 1,0 >[M1 | K1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < N2, 0 >[M1 | 1 + K1]

search [1] < N4,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

The problem with pattern `< N4,0 >[M2 | K2 + 1]` is that is too general to be rewritten by the rules of `R&W-FAIR`. But we can use the **Pattern Decomposition Lemma** of Lecture 24 to show that it is semantically equivalent to a disjunction of patterns that can be rewritten.

Verifying Some Properties of R&W-FAIR (III)

```
search [1] < M,0 >[N1 + 1 | K] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 1 + M, 0 >[N1 | K]

search [1] < N2 + 1,0 >[M1 | K1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < N2, 0 >[M1 | 1 + K1]

search [1] < N4,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

The problem with pattern `< N4,0 >[M2 | K2 + 1]` is that is too general to be rewritten by the rules of `R&W-FAIR`. But we can use the **Pattern Decomposition Lemma** of Lecture 24 to show that it is semantically equivalent to a disjunction of patterns that can be rewritten. We instantiate `N4` with generator set `{0, n:Nat + 1}`.

Verifying Some Properties of R&W-FAIR (IV)

```
search [1] < n:Nat + 1,0 >[M2 | K2 + 1]  =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < n:Nat, 0 >[M2 | 1 + 1 + K2]

search [1] < 0,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

Verifying Some Properties of R&W-FAIR (IV)

```
search [1] < n:Nat + 1,0 >[M2 | K2 + 1]  =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < n:Nat, 0 >[M2 | 1 + 1 + K2]

search [1] < 0,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

Finally, we instantiate `M2` with generator set {`0, n:Nat + 1`}.

Verifying Some Properties of R&W-FAIR (IV)

```
search [1] < n:Nat + 1,0 >[M2 | K2 + 1]  =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < n:Nat, 0 >[M2 | 1 + 1 + K2]

search [1] < 0,0 >[M2 | K2 + 1] =>1 C:Conf .

No solution.
```

Finally, we instantiate `M2` with generator set {`0, n:Nat + 1`}.

```
search [1] < 0,0 >[0 | K2 + 1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 0, 1 >[0 | 1 + K2]

search [1] < 0,0 >[n:Nat + 1 | K2 + 1] =>1 C:Conf .

Solution 1 (state 1)
C:Conf --> < 1, 0 >[n:Nat | 1 + K2]
```