Program Verification: Lecture 29

José Meseguer

University of Illinois at Urbana-Champaign

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied.

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied. But the whole point of the **DM-Check** design is to combine the proving power of symbolic model checking with that of inductive theorem proving.

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied. But the whole point of the **DM-Check** design is to combine the proving power of symbolic model checking with that of inductive theorem proving.

The NuITP is used as a backend with a default strategy for **DM-Check** to try to prove implications between constraints when trying to fold into (subsume) one constrained pattern into another.

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied. But the whole point of the **DM-Check** design is to combine the proving power of symbolic model checking with that of inductive theorem proving.

The NuITP is used as a backend with a default strategy for **DM-Check** to try to prove implications between constraints when trying to fold into (subsume) one constrained pattern into another. But there are many other uses of the NuITP that can play a crucial role in proving invariants.

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied. But the whole point of the **DM-Check** design is to combine the proving power of symbolic model checking with that of inductive theorem proving.

The NuITP is used as a backend with a default strategy for **DM-Check** to try to prove implications between constraints when trying to fold into (subsume) one constrained pattern into another. But there are many other uses of the NuITP that can play a crucial role in proving invariants.

In this lecture I will show some examples of NuITP uses that complement the automatic nature of **DM-Check**.

Synergy between **DM-Check** and Maude's **NuITP**

In Lecture 28 I explained that **DM-Check** applies sufficient conditions that need not be always satisfied. But the whole point of the **DM-Check** design is to combine the proving power of symbolic model checking with that of inductive theorem proving.

The NuITP is used as a backend with a default strategy for **DM-Check** to try to prove implications between constraints when trying to fold into (subsume) one constrained pattern into another. But there are many other uses of the NuITP that can play a crucial role in proving invariants.

In this lecture I will show some examples of NuITP uses that complement the automatic nature of **DM-Check**. Let us begin with the dadlock freedom invariant that we could not prove for `R&W-FAIR` in Lecture 28:

A Pending Proof: Deadlock Freedom of R&W-FAIR

```
DM-Check> check  in R&W-FAIR : (([N':NzNat]< 0,0 >[ 0 | N':NzNat]) | true) \/
(([N':NzNat]< 0,1 >[ 0 | N':NzNat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< M:Nat,0 >[N':NzNat | K:Nat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< N':NzNat,0 >[M:Nat | K:Nat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< M:Nat,0 >[K:Nat | N':NzNat]) | true) subsumed-by
((([N:Nat]< 0,0 >[ 0 | N:Nat]) | true) \/ (([N:Nat]< 0,1 >[ 0 | N:Nat]) | true)
\/ (([K:Nat + N:Nat + M:Nat + 1]< N:Nat,0 >[M:Nat + 1 | K:Nat]) | true) \/
(([K:Nat + N:Nat + M:Nat + 1]< (N:Nat + 1), 0 >[M:Nat | K:Nat]) | true)) .

  Constrained terms on the left that could not be subsumed:

Term 7: [N':NzNat + K:Nat + M:Nat] < M:Nat, 0 >[N':NzNat | K:Nat]
Constraint 7: true

Term 8: [N':NzNat + K:Nat + M:Nat] < N':NzNat, 0 >[M:Nat | K:Nat]
Constraint 8: true

Term 9: [N':NzNat + K:Nat + M:Nat] < M:Nat, 0 >[K:Nat | N':NzNat]
Constraint 9: true
```

A Pending Proof: Deadlock Freedom of R&W-FAIR

```
DM-Check> check  in R&W-FAIR : (([N':NzNat]< 0,0 >[ 0 | N':NzNat]) | true) \/
(([N':NzNat]< 0,1 >[ 0 | N':NzNat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< M:Nat,0 >[N':NzNat | K:Nat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< N':NzNat,0 >[M:Nat | K:Nat]) | true) \/
(([N':NzNat + K:Nat + M:Nat]< M:Nat,0 >[K:Nat | N':NzNat]) | true) subsumed-by
((([N:Nat]< 0,0 >[ 0 | N:Nat]) | true) \/ (([N:Nat]< 0,1 >[ 0 | N:Nat]) | true)
\/ (([K:Nat + N:Nat + M:Nat + 1]< N:Nat,0 >[M:Nat + 1 | K:Nat]) | true) \/
(([K:Nat + N:Nat + M:Nat + 1]< (N:Nat + 1), 0 >[M:Nat | K:Nat]) | true)) .

  Constrained terms on the left that could not be subsumed:

Term 7: [N':NzNat + K:Nat + M:Nat] < M:Nat, 0 >[N':NzNat | K:Nat]
Constraint 7: true

Term 8: [N':NzNat + K:Nat + M:Nat] < N':NzNat, 0 >[M:Nat | K:Nat]
Constraint 8: true

Term 9: [N':NzNat + K:Nat + M:Nat] < M:Nat, 0 >[K:Nat | N':NzNat]
Constraint 9: true
```

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed;

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained).

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained). If a constrained pattern $u \mid \varphi$ could not be shown contained in $\bigvee_{j \in J} v_j \mid \psi_j$ by the `subsumed-by` command for a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B)$, we can:

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained). If a constrained pattern $u \mid \varphi$ could not be shown contained in $\bigvee_{j \in J} v_j \mid \psi_j$ by the `subsumed-by` command for a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B)$, we can:

1. Extend the equational theory $(\Sigma, E \cup B)$ by adding a predicate $p : State \to [Bool]$ defined by the conditional equations:

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained). If a constrained pattern $u \mid \varphi$ could not be shown contained in $\bigvee_{j \in J} v_j \mid \psi_j$ by the `subsumed-by` command for a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B)$, we can:

1. Extend the equational theory $(\Sigma, E \cup B)$ by adding a predicate $p : State \rightarrow [Bool]$ defined by the conditional equations: $\{p(v_j) = true \textit{ if } \psi_j\}_{j \in J}$.

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained). If a constrained pattern $u \mid \varphi$ could not be shown contained in $\bigvee_{j \in J} v_j \mid \psi_j$ by the `subsumed-by` command for a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B)$, we can:

1. Extend the equational theory $(\Sigma, E \cup B)$ by adding a predicate $p : State \to [Bool]$ defined by the conditional equations: $\{p(v_j) = true \ if \ \psi_j\}_{j \in J}$.

2. Prove in the exteded equational theory the inductive theorem:

Proving Containment of Constrained Terms with NuITP

The `subsumed-by` command is quite useful, even when it does not succeed; because it identifies the constrained patterns that could not be subsumed automatically.

The following general method uses the NuITP to prove containment for constrained patterns (if actually contained). If a constrained pattern $u \mid \varphi$ could not be shown contained in $\bigvee_{j \in J} v_j \mid \psi_j$ by the `subsumed-by` command for a topmost rewrite theory $\mathcal{R} = (\Sigma, E \cup B)$, we can:

1. Extend the equational theory $(\Sigma, E \cup B)$ by adding a predicate $p : \mathit{State} \to [\mathit{Bool}]$ defined by the conditional equations: $\{p(v_j) = \mathit{true} \textit{ if } \psi_j\}_{j \in J}$.

2. Prove in the exteded equational theory the inductive theorem:

$$\varphi \to p(u) = \mathit{true}$$

Proving Deadlock Freedom of `R&W-FAIR`

We can extend the equational theory of `R&W-FAIR` thus:

Proving Deadlock Freedom of R&W-FAIR

We can extend the equational theory of R&W-FAIR thus:

```
set include BOOL off .

fmod R&W-FAIR is
  sorts NzNat Nat Conf .
  subsorts NzNat < Nat .
  op 0 : -> Nat [ctor metadata "2"] .
  op 1 : -> NzNat [ctor metadata "3"] .
  op _+_ : Nat Nat -> Nat [metadata "4" assoc comm id: 0] .
  op _+_ : NzNat Nat -> NzNat [ctor metadata "4" assoc comm id: 0] .
  op [_]<_,_>[_|_] : Nat Nat Nat Nat Nat -> Conf [ctor metadata "5"] .
  op init : NzNat -> Conf .
endfm

fmod ENABLED is protecting  R&W-FAIR .
  sort  MyBool .
  op true : -> MyBool [ctor metadata "0"] .
  op false : -> MyBool [ctor metadata "1"] .
  op enabled : Conf -> [MyBool] [metadata "6"] .

  vars N M K I J L : Nat .  var N' M' : NzNat .
```

Proving Deadlock Freedom of R&W-FAIR

```
  eq enabled([N]< 0,0 >[ 0 | N]) = true .
  eq enabled([N]< 0,1 >[ 0 | N]) = true .
  eq enabled([K + N + M + 1]< N,0 >[M + 1 | K]) = true .
  eq enabled([K + N + M + 1]< N + 1,0 >[M | K]) = true .
endfm
```

Proving Deadlock Freedom of R&W-FAIR

```
  eq enabled([N]< 0,0 >[ 0 | N]) = true .
  eq enabled([N]< 0,1 >[ 0 | N]) = true .
  eq enabled([K + N + M + 1]< N,0 >[M + 1 | K]) = true .
  eq enabled([K + N + M + 1]< N + 1,0 >[M | K]) = true .
endfm
```

Then, the proof that R&W-FAIR is deadlock free is obtained by the following sequence of **NuITP** Case (CAS) commands:

Proving Deadlock Freedom of R&W-FAIR

```
  eq enabled([N]< 0,0 >[ 0 | N]) = true .
  eq enabled([N]< 0,1 >[ 0 | N]) = true .
  eq enabled([K + N + M + 1]< N,0 >[M + 1 | K]) = true .
  eq enabled([K + N + M + 1]< N + 1,0 >[M | K]) = true .
endfm
```

Then, the proof that R&W-FAIR is deadlock free is obtained by the following sequence of **NuITP** Case (CAS) commands:

```
genset NZNATG for NzNat is 1 ;; (1 + X:NzNat) .

genset NATG for Nat is 0 ;; 1 ;; (1 + X:NzNat) .

set goal (enabled([N':NzNat + K:Nat + M:Nat]< M:Nat, 0 >[N':NzNat | K:Nat]) =
true)/\(enabled([N':NzNat + K:Nat + M:Nat]< N':NzNat, 0 >[M:Nat | K:Nat])= true

apply cas! to 0 on $3:NzNat .

set goal enabled([N':NzNat + K:Nat + M:Nat]< M:Nat, 0 >[K:Nat | N':NzNat])= tru

apply cas! to 0 on $2:Nat .

apply cas! to 0.1 on $1:Nat .
```

Proving an Inductive Invariant for BAKERY

Recall the `BAKERY` example in Lecture 27.

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes.

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

Proving an Inductive Invariant for BAKERY

Recall the `BAKERY` example in Lecture 27. Its initial state is parametric on a set `IS` of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

```
(lfmc [100] N ; N ; IS |= [] mutex  .)

result: no counterexample found within bound 100
```

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

```
(lfmc [100] N ; N ; IS |= [] mutex  .)

result: no counterexample found within bound 100
```

However, combining the proving powers of **DM-Check** and the **NuITP** we will be able to:

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

```
(lfmc [100] N ; N ; IS |= [] mutex  .)

result: no counterexample found within bound 100
```

However, combining the proving powers of **DM-Check** and the **NuITP** we will be able to:

1. Prove an inductive invariant for BAKERY from `N ; N ; IS`.

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

```
(lfmc [100] N ; N ; IS |= [] mutex  .)

result: no counterexample found within bound 100
```

However, combining the proving powers of **DM-Check** and the **NuITP** we will be able to:

1. Prove an inductive invariant for BAKERY from `N ; N ; IS`.
2. Prove the mutual exclusion invariant just by unification.

Proving an Inductive Invariant for BAKERY

Recall the BAKERY example in Lecture 27. Its initial state is parametric on a set IS of idle processes. Maude's LTL Logical Model Checker was able to prove the mutual exclusion invariant, but only for a fixed number of idle processes, e.g.,

```
(lfmc N:Name ; N:Name ; [idle] [idle] |= [] mutex .)

result: true (complete with depth 5)
```

For the parametric intial state the LTL Logical Model Checker could not fold the symbolic states into a finte graph: only bounded model checking was possible:

```
(lfmc [100] N ; N ; IS |= [] mutex  .)

result: no counterexample found within bound 100
```

However, combining the proving powers of **DM-Check** and the **NuITP** we will be able to:

1. Prove an inductive invariant for BAKERY from `N ; N ; IS`.
2. Prove the mutual exclusion invariant just by unification.

Proving an Inductive Invariant for `BAKERY` (II)

Recall again the `BAKERY` specification, now extended with a few auxiliary functions used in the constraints of patterns:

Proving an Inductive Invariant for BAKERY (II)

Recall again the BAKERY specification, now extended with a few auxiliary functions used in the constraints of patterns:

```
fmod NAME is
  sorts Name MyBool .
  op True : -> MyBool [ctor] .
  op False : -> MyBool [ctor] .
  op _or'_ : MyBool MyBool -> MyBool [comm] .
  op _and'_ : MyBool MyBool -> MyBool [comm] .
  op 0 : -> Name [ctor metadata "2"] .
  op s : -> Name [ctor metadata "4"] .
  op __ : Name Name -> Name [ctor comm assoc id: 0] .

  vars N M : Name .  var B : MyBool .

  eq True or' B = True .            eq False or' B = B .
  eq True and' B = B .              eq False and' B = False .

  op _<=_ : Name Name -> MyBool [metadata "9"] .

  eq N <= N M = True [variant] .
  eq s N M <= M = False [variant] .
endfm
```

Proving an Inductive Invariant for BAKERY (III)

```
fmod MSET is  protecting NAME .
  sort MSet  .  subsort Name < MSet .
  op null : -> MSet [ctor] .
  op _,_ : MSet MSet -> MSet [ctor assoc comm id: null] .

  vars N M : Name .  var MS : MSet .

  op _in_ : Name MSet -> MyBool .
  eq M in (M, MS) = True .
  eq M in null = False .
  eq M in ((s N M), MS) = M in MS .
  eq s N M in (M, MS) = s N M in MS .
endfm

mod BAKERY is protecting NAME .
  sorts ModeIdle ModeWait ModeCrit Mode .
  subsorts ModeIdle ModeWait ModeCrit < Mode .
  sorts ProcIdle ProcWait Proc ProcIdleSet ProcWaitSet ProcSet .
  subsorts ProcIdle < ProcIdleSet .
  subsorts ProcWait < ProcWaitSet .
  subsorts ProcIdle ProcWait < Proc < ProcSet .
  subsorts ProcIdleSet < ProcWaitSet < ProcSet .
```

Proving an Inductive Invariant for BAKERY (IV)

```
  op idle : -> ModeIdle [ctor] .
  op wait : Name -> ModeWait [ctor] .
  op crit : Name -> ModeCrit [ctor] .
  op [_] : ModeIdle -> ProcIdle [ctor] .
  op [_] : ModeWait -> ProcWait [ctor] .
  op [_] : Mode -> Proc [ctor] .
  op none : -> ProcIdleSet  [ctor] .
  op __ : ProcIdleSet ProcIdleSet -> ProcIdleSet [ctor assoc comm id: none] .
  op __ : ProcWaitSet ProcWaitSet -> ProcWaitSet [ctor assoc comm id: none] .
  op __ : ProcSet ProcSet -> ProcSet [ctor assoc comm id: none] .

  sort Conf .
  op _;_;_ : Name Name ProcSet -> Conf .

  var PS : ProcSet .  vars N M I J K M1 M2 : Name . var IS : ProcIdleSet .
  var WS : ProcWaitSet .

  rl [wake] : N ; M ; [idle] PS    => s N ; M ; [wait(N)] PS [narrowing] .
  rl [crit] : N ; M ; [wait(M)] PS => N ; M ; [crit(M)] PS [narrowing] .
  rl [exit] : N ; M ; [crit(M)] PS => N ; s M ; [idle] PS [narrowing] .
endm
```

Proving an Inductive Invariant for BAKERY (IV)

```
mod BAKERY-AUX is
  protecting BAKERY .  protecting MSET .

  op [_,_] : Name Name -> MSet .
  op tickets : ProcSet -> MSet .

  vars N M I J K M1 M2 : Name .
  var IS : ProcIdleSet .
  var WS : ProcWaitSet .
  var PS : ProcSet .

  eq [N,N] = N .
  eq [(s N M),N] = null .
  eq [N,(s N M)] = [N,(N M)], (s N M) .

  *** interval of numbers as a set
  eq tickets(none) = null .
  eq tickets([idle] IS PS) = tickets(PS) .
  eq tickets([wait(N)] PS) = N , tickets(PS) .
  eq tickets([crit(N)] PS) = N , tickets(PS) .
endm
```

Proving an Inductive Invariant for BAKERY (IV)

Using the above auxiliary functions we can conjecture the following inductive invariant from the parametric initial state `K ; K ; IS`:

Proving an Inductive Invariant for BAKERY (IV)

Using the above auxiliary functions we can conjecture the following inductive invariant from the parametric initial state `K ; K ; IS`:

```
K ; K ; IS | true
\/
s M ; N ; WS | tickets(WS) = [N,M] /\ N <= M
\/
s M ; N ; [crit(N)] WS | tickets(WS) = [s N,M] /\ N <= M
```

Proving an Inductive Invariant for BAKERY (IV)

Using the above auxiliary functions we can conjecture the following inductive invariant from the parametric initial state `K ; K ; IS`:

```
K ; K ; IS | true
\/
s M ; N ; WS | tickets(WS) = [N,M] /\ N <= M
\/
s M ; N ; [crit(N)] WS | tickets(WS) = [s N,M] /\ N <= M
```

The parametric initial state is subsumed by the invariant:

Proving an Inductive Invariant for BAKERY (IV)

Using the above auxiliary functions we can conjecture the following inductive invariant from the parametric initial state `K ; K ; IS`:

```
K ; K ; IS | true
\/
s M ; N ; WS | tickets(WS) = [N,M] /\ N <= M
\/
s M ; N ; [crit(N)] WS | tickets(WS) = [s N,M] /\ N <= M
```

The parametric initial state is subsumed by the invariant:

```
DM-Check> check in BAKERY-AUX :
((K:Name ; K:Name ; IS:ProcIdleSet) | true) subsumed-by
(((K:Name ; K:Name ; IS:ProcIdleSet) | true)  \/
((s M:Name ; N:Name ; WS:ProcWaitSet) | ((tickets(WS:ProcWaitSet) =
([N:Name, M:Name])) /\ ((N:Name <= M:Name) = True)))  \/
((s M:Name ; N:Name ; [crit(N:Name)] WS:ProcWaitSet) |
((tickets(WS:ProcWaitSet) = [s N:Name, M:Name]) /\
((N:Name <= M:Name) = True)))) .

  Subsumption satisfied.
```

Proving an Inductive Invariant for BAKERY (V)

The invariant check cannot fold a generated constrained pattern:

```
DM-Check> check-inv in BAKERY-AUX : (K:Name ; K:Name ; IS:ProcIdleSet) | true
\/  (s M:Name ; N:Name ; WS:ProcWaitSet) | (tickets(WS:ProcWaitSet) =
([N:Name, M:Name])) /\ ((N:Name <= M:Name) = True)  \/
(s M:Name ; N:Name ; [crit(N:Name)] WS:ProcWaitSet) | (tickets(WS:ProcWaitSet)
= [s N:Name, M:Name]) /\ ((N:Name <= M:Name) = True) .

  Invariant could not be proved (no match).

Parent: 1
Term: K:Name ; K:Name ; IS:ProcIdleSet
Constraint: true

Parent: 2
Term: (s M:Name) ; N:Name ; WS:ProcWaitSet
Constraint: (tickets(WS:ProcWaitSet) =[N:Name, M:Name]) /\ N:Name <= M:Name =
    True
```

Proving an Inductive Invariant for BAKERY (V)

```
Parent: 3
Term: (s M:Name) ; N:Name ;[crit(N:Name)] WS:ProcWaitSet
Constraint: (tickets(WS:ProcWaitSet) =[s N:Name, M:Name]) /\ N:Name <= M:Name =
    True

Child: 9
Parent: 3
Term: (s %1:Name) ; %2:Name ; %3:ProcWaitSet[crit(%2:Name)][crit(%2:Name)]
Substitution: M:Name --> %1:Name
N:Name --> %2:Name
WS:ProcWaitSet --> %3:ProcWaitSet[wait(%2:Name)]
Constraint: ((%2:Name, tickets(%3:ProcWaitSet)) =[s %2:Name, %1:Name]) /\
    %2:Name <= %1:Name = True
```

Proving an Inductive Invariant for BAKERY (V)

```
Parent: 3
Term: (s M:Name) ; N:Name ;[crit(N:Name)] WS:ProcWaitSet
Constraint: (tickets(WS:ProcWaitSet) =[s N:Name, M:Name]) /\ N:Name <= M:Name =
    True

Child: 9
Parent: 3
Term: (s %1:Name) ; %2:Name ; %3:ProcWaitSet[crit(%2:Name)][crit(%2:Name)]
Substitution: M:Name --> %1:Name
N:Name --> %2:Name
WS:ProcWaitSet --> %3:ProcWaitSet[wait(%2:Name)]
Constraint: ((%2:Name, tickets(%3:ProcWaitSet)) =[s %2:Name, %1:Name]) /\
    %2:Name <= %1:Name = True
```

The output indicates that narrowing derived from `Parent 3` `Child 9`, which cannot be folded. It is the constrained pattern:

Proving an Inductive Invariant for BAKERY (V)

```
Parent: 3
Term: (s M:Name) ; N:Name ;[crit(N:Name)] WS:ProcWaitSet
Constraint: (tickets(WS:ProcWaitSet) =[s N:Name, M:Name]) /\ N:Name <= M:Name =
    True

Child: 9
Parent: 3
Term: (s %1:Name) ; %2:Name ; %3:ProcWaitSet[crit(%2:Name)][crit(%2:Name)]
Substitution: M:Name --> %1:Name
N:Name --> %2:Name
WS:ProcWaitSet --> %3:ProcWaitSet[wait(%2:Name)]
Constraint: ((%2:Name, tickets(%3:ProcWaitSet)) =[s %2:Name, %1:Name]) /\
    %2:Name <= %1:Name = True
```

The output indicates that narrowing derived from `Parent 3` `Child 9`, which cannot be folded. It is the constrained pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

Proving an Inductive Invariant for BAKERY (VI)

We will have proved the inductive invariant if we show that the constraint

```
N, tickets(WS) = [s N,M] /\ N <= M = True
```

Proving an Inductive Invariant for BAKERY (VI)

We will have proved the inductive invariant if we show that the constraint

```
N, tickets(WS) = [s N,M] /\ N <= M = True
```

is unsatisfiable, i.e., that its negation is an inductive theorem. We can do so for the more general constraint:

```
N, S:MSet = [s N,M] /\ N <= M = True
```

Proving an Inductive Invariant for BAKERY (VI)

We will have proved the inductive invariant if we show that the constraint

```
N, tickets(WS) = [s N,M] /\ N <= M = True
```

is unsatisfiable, i.e., that its negation is an inductive theorem. We can do so for the more general constraint:

```
N, S:MSet = [s N,M] /\ N <= M = True
```

by loading into the **NuITP** the following functional module extracted from BAKERY-AUX and endowed with an RPO order:

Proving an Inductive Invariant for BAKERY (VI)

We will have proved the inductive invariant if we show that the constraint

```
N, tickets(WS) = [s N,M] /\ N <= M = True
```

is unsatisfiable, i.e., that its negation is an inductive theorem. We can do so for the more general constraint:

```
N, S:MSet = [s N,M] /\ N <= M = True
```

by loading into the **NuITP** the following functional module extracted from BAKERY-AUX and endowed with an RPO order:

```
set include BOOL off .

fmod NAME is
  sorts Name MyBool .
  op True : -> MyBool [ctor metadata "0"] .
  op False : -> MyBool [ctor metadata "1"] .
  op _or'_ : MyBool MyBool -> MyBool [metadata "7" comm] .
  op _and'_ : MyBool MyBool -> MyBool [metadata "8" comm] .
  op 0 : -> Name [ctor metadata "2"] .
  op s : -> Name [ctor metadata "4"] .
  op __ : Name Name -> Name [ctor comm assoc id: 0 metadata "5"] .
  op _<=_ : Name Name -> MyBool [metadata "9"] .
```

Proving an Inductive Invariant for BAKERY (VI)

```
  vars N M : Name .  var B : MyBool .

  eq True or' B = True .
  eq False or' B = B .
  eq True and' B = B .
  eq False and' B = False .
  eq N <= N M = True [variant] .
  eq s N M <= M = False [variant] .
endfm

fmod MSET is  protecting NAME .
  sort MSet  .  subsort Name < MSet .
  op null : -> MSet [ctor metadata "3"] .
  op _,_ : MSet MSet -> MSet [ctor assoc comm id: null metadata "6"] .
  vars N M : Name . var MS : MSet .

  op _in_ : Name MSet -> MyBool [metadata "10"] .
  eq M in (M, MS) = True .
  eq M in null = False .
  eq M in ((s N M), MS) = M in MS .
  eq s N M in (M, MS) = s N M in MS .
endfm
```

Proving an Inductive Invariant for BAKERY (VI)

```
  vars N M : Name .  var B : MyBool .

  eq True or' B = True .
  eq False or' B = B .
  eq True and' B = B .
  eq False and' B = False .
  eq N <= N M = True [variant] .
  eq s N M <= M = False [variant] .
endfm

fmod MSET is  protecting NAME .
  sort MSet  .  subsort Name < MSet .
  op null : -> MSet [ctor metadata "3"] .
  op _,_ : MSet MSet -> MSet [ctor assoc comm id: null metadata "6"] .
  vars N M : Name . var MS : MSet .

  op _in_ : Name MSet -> MyBool [metadata "10"] .
  eq M in (M, MS) = True .
  eq M in null = False .
  eq M in ((s N M), MS) = M in MS .
  eq s N M in (M, MS) = s N M in MS .
endfm
```

Proving an Inductive Invariant for BAKERY (VI)

```
  vars N M : Name .  var B : MyBool .

  eq True or' B = True .
  eq False or' B = B .
  eq True and' B = B .
  eq False and' B = False .
  eq N <= N M = True [variant] .
  eq s N M <= M = False [variant] .
endfm

fmod MSET is  protecting NAME .
  sort MSet  .  subsort Name < MSet .
  op null : -> MSet [ctor metadata "3"] .
  op _,_ : MSet MSet -> MSet [ctor assoc comm id: null metadata "6"] .
  vars N M : Name . var MS : MSet .

  op _in_ : Name MSet -> MyBool [metadata "10"] .
  eq M in (M, MS) = True .
  eq M in null = False .
  eq M in ((s N M), MS) = M in MS .
  eq s N M in (M, MS) = s N M in MS .
endfm
```

Proving an Inductive Invariant for BAKERY (VII)

```
fmod INTERVALS is protecting MSET .
  op [_,_] : Name Name -> MSet [metadata "11"].

  vars N M K : Name .

  eq [N,N] = N .
  eq [(s N M),N] = null .
  eq [N,(s N M)] = [N,(N M)], (s N M) .
endfm
```

Proving an Inductive Invariant for BAKERY (VII)

```
fmod INTERVALS is protecting MSET .
  op [_,_] : Name Name -> MSet [metadata "11"].

  vars N M K : Name .

  eq [N,N] = N .
  eq [(s N M),N] = null .
  eq [N,(s N M)] = [N,(N M)], (s N M) .
endfm
```

In the **NuITP** the negation of the (generalized) constraint is the clause:

Proving an Inductive Invariant for BAKERY (VII)

```
fmod INTERVALS is protecting MSET .
  op [_,_] : Name Name -> MSet [metadata "11"].

  vars N M K : Name .

  eq [N,N] = N .
  eq [(s N M),N] = null .
  eq [N,(s N M)] = [N,(N M)], (s N M) .
endfm
```

In the **NuITP** the negation of the (generalized) constraint is the clause:

```
N, S:MSet = [s N,M] /\ N <= M = True -> false
```

Proving an Inductive Invariant for BAKERY (VII)

```
fmod INTERVALS is protecting MSET .
  op [_,_] : Name Name -> MSet [metadata "11"].

  vars N M K : Name .

  eq [N,N] = N .
  eq [(s N M),N] = null .
  eq [N,(s N M)] = [N,(N M)], (s N M) .
endfm
```

In the **NuITP** the negation of the (generalized) constraint is the clause:

```
N, S:MSet = [s N,M] /\ N <= M = True -> false
```

To prove it as an inductive theorem we can use the following **Cut** inference rule (not yet implemented, but easily usable):

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_, \_$ and Λ a conjunction of disjunctions of equalities.

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \to \Gamma'' \qquad \Gamma'', \Gamma' \to \Lambda}{\Gamma, \Gamma' \to \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_,\_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \to \Gamma'' \qquad \Gamma'', \Gamma' \to \Lambda}{\Gamma, \Gamma' \to \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_, \_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \;\wedge\; (A'' \wedge A') \Rightarrow B) \;\Rightarrow (A \wedge A') \Rightarrow B).$$

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \to \Gamma'' \qquad \Gamma'', \Gamma' \to \Lambda}{\Gamma, \Gamma' \to \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_, \_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \;\wedge\; (A'' \wedge A') \Rightarrow B) \;\Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \to \Lambda$ will be:

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \to \Gamma'' \qquad \Gamma'', \Gamma' \to \Lambda}{\Gamma, \Gamma' \to \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_,\_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \;\wedge\; (A'' \wedge A') \Rightarrow B) \;\Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \to \Lambda$ will be:

```
N, S:MSet = [s N,M] /\ N <= M = True -> false,
```

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as _, _ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \ \wedge \ (A'' \wedge A') \Rightarrow B) \ \Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \rightarrow \Lambda$ will be:

`N, S:MSet = [s N,M] /\ N <= M = True -> false,`

$\Gamma \rightarrow \Gamma''$ will be:

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_, \_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \ \wedge \ (A'' \wedge A') \Rightarrow B) \ \Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \rightarrow \Lambda$ will be:

```
N, S:MSet = [s N,M] /\ N <= M = True -> false,
```

$\Gamma \rightarrow \Gamma''$ will be:

```
[s N,M] =  N, S:MSet -> N in [s N,M] = N in (N, S:MSet),
```

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as _,_ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \;\wedge\; (A'' \wedge A') \Rightarrow B) \;\Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \rightarrow \Lambda$ will be:

`N, S:MSet = [s N,M] /\ N <= M = True -> false,`

$\Gamma \rightarrow \Gamma''$ will be:

`[s N,M] =  N, S:MSet -> N in [s N,M] = N in (N, S:MSet),`

and $\Gamma'', \Gamma' \rightarrow \Lambda$ will be:

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \rightarrow \Gamma'' \qquad \Gamma'', \Gamma' \rightarrow \Lambda}{\Gamma, \Gamma' \rightarrow \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as _, _ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \ \wedge \ (A'' \wedge A') \Rightarrow B) \ \Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \rightarrow \Lambda$ will be:

`N, S:MSet = [s N,M] /\ N <= M = True -> false,`

$\Gamma \rightarrow \Gamma''$ will be:

`[s N,M] =  N, S:MSet -> N in [s N,M] = N in (N, S:MSet),`

and $\Gamma'', \Gamma' \rightarrow \Lambda$ will be:

`N in [s N,M] = N in (N, S:MSet)/\ N <= M = True -> false.`

Proving an Inductive Invariant for BAKERY (VIII)

$$\frac{\Gamma \to \Gamma'' \qquad \Gamma'', \Gamma' \to \Lambda}{\Gamma, \Gamma' \to \Lambda}$$

with Γ, Γ' and Γ'' conjunctions of equalities, conjunction represented as $\_,\_$ and Λ a conjunction of disjunctions of equalities.

The soundness of the **Cut** rule follows from the fact that the following implication is a tautology in Propositional Logic:

$$(A \Rightarrow A'' \;\wedge\; (A'' \wedge A') \Rightarrow B) \;\Rightarrow (A \wedge A') \Rightarrow B).$$

In our application of **Cut**, $\Gamma, \Gamma' \to \Lambda$ will be:

`N, S:MSet = [s N,M] /\ N <= M = True -> false,`

$\Gamma \to \Gamma''$ will be:

`[s N,M] =  N, S:MSet -> N in [s N,M] = N in (N, S:MSet),`

and $\Gamma'', \Gamma' \to \Lambda$ will be:

`N in [s N,M] = N in (N, S:MSet)/\ N <= M = True -> false.`

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \to \Gamma''$:

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .`

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \to \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \to \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

is simplified by the command `apply cvul! to 0 .` to goal 0.1.1:

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

is simplified by the command `apply cvul! to 0 .` to goal 0.1.1:

```
True = $4:Name in[s $4:Name, $5:Name $4:Name] -> false
```

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

is simplified by the command `apply cvul! to 0 .` to goal 0.1.1:

```
True = $4:Name in[s $4:Name, $5:Name $4:Name] -> false
```

we use a lemma with the command:

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

is simplified by the command `apply cvul! to 0 .` to goal 0.1.1:

```
True = $4:Name in[s $4:Name, $5:Name $4:Name] -> false
```

we use a lemma with the command:

```
apply le! to 0.1.1 with ((N:Name in [s N:Name,(N:Name K:Name)]) = False) .
```

Proving an Inductive Invariant for BAKERY (IX)

The **NuITP** proof is as follows. The goal $\Gamma \rightarrow \Gamma''$:

```
(([s N:Name,M:Name]) =  (N:Name , S:MSet)) ->
((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet)))
```

is proved by the single command: `apply icc! to 0 .` The goal $\Gamma'', \Gamma' \rightarrow \Lambda$:

```
(((N:Name in [s N:Name,M:Name]) = (N:Name in (N:Name , S:MSet))) /\
((N:Name <= M:Name) = True)) -> false
```

is simplified by the command `apply cvul! to 0 .` to goal 0.1.1:

```
True = $4:Name in[s $4:Name, $5:Name $4:Name] -> false
```

we use a lemma with the command:

```
apply le! to 0.1.1 with ((N:Name in [s N:Name,(N:Name K:Name)]) = False) .
```

This proves the goal, leaving only the proof of the lemma, discharged with the commands:

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`.

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`. We can now use the inductive invariant and disjoint unification to prove **Negatively** that `BAKERY` satisfies the mutual exclusion invariant:

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`. We can now use the inductive invariant and disjoint unification to prove **Negatively** that `BAKERY` satisfies the mutual exclusion invariant:

```
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? K ; K ; IS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; WS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; [crit(N)] WS .

No unifier.
```

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`. We can now use the inductive invariant and disjoint unification to prove **Negatively** that `BAKERY` satisfies the mutual exclusion invariant:

```
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? K ; K ; IS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; WS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; [crit(N)] WS .

No unifier.
```

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`. We can now use the inductive invariant and disjoint unification to prove **Negatively** that `BAKERY` satisfies the mutual exclusion invariant:

```
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? K ; K ; IS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; WS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; [crit(N)] WS .

No unifier.
```

Proving an Inductive Invariant for BAKERY (IX)

```
apply cas! to 0.1.1.1.1 on $6:Name .

apply gsi! to 0.1.1.1.1.2.1 on $8:Name .
```

This finishes the proof of unsatisfiability for the constraint of the pattern:

```
s M ; N ; WS [crit(N)] [crit(N)] | N tickets(WS) = [s N,M] /\ N <= M = True
```

and therefore the proof that our conjectured invariant is an inductive invariant from the symbolic initial state `K ; K ; IS`. We can now use the inductive invariant and disjoint unification to prove **Negatively** that BAKERY satisfies the mutual exclusion invariant:

```
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? K ; K ; IS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; WS .

No unifier.
Maude> unify I ; J ; [crit(M1)] [crit(M2)] PS =? s M ; N ; [crit(N)] WS .

No unifier.
```