Generating Invariants by Deductive Model Checking

Kyungmin Bae [1] Santiago Escobar [2] Raúl López-Rueda [2]
José Meseguer [3] Julia Sapiña [2]

[1] Pohang University of Science and Technology, Pohang, South Korea
[2] VRAIN, Universitat Politècnica de València, Valencia, Spain
[3] University of Illinois at Urbana-Champaign, Urbana, IL, USA

16th International Workshop on Rewriting Logic and its Applications
Torino, April 11 - 12, 2026

Outline

Outline

Previous Work

- Kyungmin Bae, Santiago Escobar, Raúl López-Rueda, José Meseguer, Julia Sapiña: *Verifying Invariants by Deductive Model Checking*. WRLA 2024: 3-21.
- José Meseguer: *Symbolic Computation and Verification Methods in Maude*. LOPSTR 2025: 1-21.
- Kyungmin Bae, Santiago Escobar, Raúl López-Rueda, José Meseguer, Julia Sapiña: *DM-Check: Verifying invariants of concurrent systems by deductive model checking*. JLAMP 2026: 149, 101107.

Motivation

- Many concurrent protocols involve an unbounded number of processes, leading to infinite-state systems.
- Traditional explicit-state model checking cannot handle an infinite state space.
- Symbolic techniques can help verifying safety properties in those systems.
- In particular, proving invariants is challenging:
  - States must be represented symbolically.
  - Some invariants may require inductive reasoning.
- Fully automated approaches often fail in practice or are unavailable.

Goal

Support semi-automated generation of inductive invariants.

Methodological Approach

Core Idea

Combine constrained narrowing with inductive theorem proving to generate inductive invariants.

- Iterative fixpoint computation:
  1. Start from initial configuration.
     - It may contain variables.
     - Extra constraints may be added.
     - A disjunction may be provided.
  2. Generate one-step successors.
     - By using constrained narrowing.
     - The NuITP (Theorem Prover) deals with the constraints.
  3. Fold and generalize patterns to reach a fixpoint.
  4. Repeat the steps if needed, including new information.
- Interactive mode: Between many commands, the user can apply transformations, generalizations, and lemmas to deal with complex patterns.

QLOCK (1/10): Our Running Example

- Multiple processes compete to enter the critical section using a FIFO queue.
- Infinite state space due to:
  - Unbounded number of processes.
  - Unbounded queue of waiting processes.
- QLOCK states: $\langle U \mid W \mid C \mid Q \rangle$
  - U: reception area (processes not yet waiting).
  - W: waiting area.
  - C: critical section.
  - Q: FIFO list of processes waiting for the lock.

```
rl [r2w] : < U i | W | C | Q > => < U | W i | C | Q ; i > .

rl [w2c] : < U | W i | C | i ; Q > => < U | W | C i | i ; Q > .

rl [c2r] : < U | W | C i | i ; Q > => < U i | W | C | Q > .
```

Outline

System Theories

Rewrite Theory

A rewrite theory is a triple

$$\mathcal{R} = (\Sigma, E \cup B, R)$$

where:

- Σ is a signature defining the system symbols.
- $E \cup B$ are (variant) equations and axioms defining the state structure.
- R is a set of rewrite rules describing system transitions.

We consider infinite-state concurrent systems specified as rewrite theories.

- States: $\mathbb{T}_{\Sigma/E \cup B}$
- Transitions: rewrite rules R.
- Verification goal: check invariants and other properties.
- Main challenge: infinite state spaces.

Symbolic Representation of States

Constrained Patterns

Infinite sets of states are represented as:

$$u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n$$

- u : constructor term representing symbolic states.
- φ : constraint over variables.
- All ground instances of u satisfying φ are included.
- Constraints may include recursive functions.

Pattern Containment for Invariant Checking

Pattern Containment

To verify invariants we must check:

$$(u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n) \subseteq (v_1 \mid \psi_1 \vee \ldots \vee v_m \mid \psi_m)$$

Meaning

- Every state represented by a pattern $u \mid \varphi$ must also satisfy a pattern $v \mid \psi$.

What is required

- Matching modulo equations E and axioms B.
- Reasoning on constraints.

Checking transition closure reduces to pattern containment.

Symbolic Reachability via Constrained Narrowing

Constrained Narrowing

Symbolically computes successors of a pattern using R modulo $E \cup B$ while propagating and solving constraints.

$$(u_1 \mid \varphi_1 \vee \ldots \vee u_n \mid \varphi_n) \rightsquigarrow_{E \cup B} (v_1 \mid \psi_1 \vee \ldots \vee v_m \mid \psi_m)$$

Interpretation

- Each step expands a symbolic reachability tree.
- Patterns represent sets of states, not individual states.

Advantages

- Finite symbolic representation of infinite state sets.
- Automatic exploration of transitions.
- Basis for iterative fixpoint computation.

Inductive Invariants

Inductive Invariant

A property P is an invariant if it is transition-closed:

$$Post(P) \subseteq P$$

- Reachability computation can be automated.
- Proving transition closure requires deductive reasoning.
- This is the central difficulty in invariant verification.

QLOCK (2/10): Initial State and Invariant

- Parametric initial state:

$$\langle\ S \mid mt \mid mt \mid nil\ \rangle \mid set(S) = tt$$

- Conjectured inductive invariant (mutual exclusion):

$$\langle\ U \mid W \mid mt \mid Q\ \rangle \mid non\text{-}mt(U\ W) = tt \wedge set(U\ W) = tt \wedge l2ms(Q) = W$$
$$\vee\ \langle\ V \mid T \mid i \mid i\ ;\ Q'\ \rangle \mid non\text{-}mt(V\ T\ i) = tt \wedge set(V\ T\ i) = tt \wedge l2ms(i\ ;\ Q') = T\ i$$

Outline

DM-Check Overview

- DM-Check [1] is a deductive model checking tool for symbolic state systems.
- Automatically:
  - Computes symbolic reachable states.
  - Checks transition closure.
- Key techniques:
  - Constrained narrowing for reachability.
  - Folding to build a finite symbolic graph.
  - Inductive theorem proving for subsumptions.
- Enables verification of invariants in infinite-state systems.

[1] More info: https://safe-tools.dsic.upv.es/dmc

Constrained Narrowing and Folding

- Repeated narrowing creates a potentially infinite forest.
- Folding produces a finite representation using subsumption:

$$u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$$

- Interpretation:
  - $u \mid \varphi$ represents a subset of states in $v \mid \psi$.
  - Exploration can be safely folded into the existing node.
- Transforms an infinite narrowing forest into a finite symbolic graph.
- Inductive theorem proving justifies subsumption:

$$\mathbb{T}_{\Sigma/E \cup B} \models \varphi \Rightarrow (\psi\alpha)$$

Integration with NuITP

- NuITP [2] is the inductive theorem prover used by DM-Check.
- Acts as an oracle for:
  - Subsumption checking in the constructor subtheory.
  - Inductive unsatisfiability of constrained patterns.
- Supports advanced symbolic techniques:
  - Narrowing, variant unification, satisfiability.
  - Strategy-based rewriting, congruence closure.
- Enables semi-automatic or interactive invariant proving.

[2] More info: https://nuitp.webs.upv.es

Previous Commands: Checking Inductive Invariants

`check ind-invariant` *INV*

Goal: Verify that the invariant is transition-closed.

- Performs one-step constrained narrowing.
- Computes symbolic successor states.
- Checks whether all successors are already covered by the invariant.
- Any uncovered patterns are returned as proof obligations.

$$Post(INV) \subseteq INV$$

QLOCK (3/10): Verifying Mutual Exclusion

```
DM-Check> check ind-invariant \
>     ((< U:MSet | W:MSet | mt | Q:List >) | \
>         (non-mt(U:MSet W:MSet) = tt) \
>          /\ (set(U:MSet W:MSet) = tt) \
>          /\ (l2ms(Q:List) = W:MSet)) \
>     \/ \
>     ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
>         (non-mt(V:MSet T:MSet i:Nat) = tt) \
>          /\ (set(V:MSet T:MSet i:Nat) = tt) \
>          /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) .

    Invariant satisfied.
```

- Mutual exclusion verified for all instances of initial state.
- Subsumption check ensures the invariant holds from initial states.

Previous Commands: Checking Pattern Containment

check <P_1> subsumed by <P_2>

$$P_1 \subseteq P_2$$

Goal: Verify containment between symbolic pattern disjunctions.

- Determines whether every pattern in P_1 is included in P_2.
- Uses inductive reasoning to prove subsumptions.
- Patterns that cannot be proved automatically become proof obligations.

QLOCK (4/10): Verifying Deadlock Freedom

```
DM-Check> check \
>     ((< U:MSet | W:MSet | mt | Q:List >) | \
>         (non-mt(U:MSet W:MSet) = tt) \
>          /\ (set(U:MSet W:MSet) = tt) \
>          /\ (l2ms(Q:List) = W:MSet)) \
>     \/ \
>     ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
>         (non-mt(V:MSet T:MSet i:Nat) = tt) \
>          /\ (set(V:MSet T:MSet i:Nat) = tt) \
>          /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) \
> subsumed by \
>     ((< U1:MSet n1:Nat | W1:MSet | C1:MSet | Q1:List >) | true) \
>     \/ ((< U2:MSet | W2:MSet n2:Nat | C2:MSet | n2:Nat ; Q2:List >) | true) \
>     \/ ((< U3:MSet | W3:MSet | C3:MSet n3:Nat | n3:Nat ; Q3:List >) | true) .

    Constrained terms on the left that could not be subsumed:

Term 4:
  < $17:MSet | $18:MSet | mt | $16:List >
Matching: no matching found
Constraint 4:
  (non-mt($17:MSet $18:MSet) = tt) /\ (set($17:MSet $18:MSet) = tt) /\ l2ms($16:List) = $18:MSet
```

- `Term 4` is not subsumed, but it is too general.
- We need commands for semantic equivalence transformations.

Previous Commands: Checking Disjointness

`intersect` $<P_1>$ `with` $<P_2>$

Goal: Verify that two symbolic pattern disjunctions cannot overlap.

- Symbolically computes the intersection of the two disjunctions.
- Uses inductive reasoning to discard unsatisfiable constraints.
- If the result is empty, the property holds.

$$P_1 \cap P_2 = \emptyset$$

Previous Commands: Proof Obligations and Lemmas

Proof Obligations

Some constrained patterns cannot be proved automatically, generating proof obligations.

- Proof obligations can be discharged by:
  - Adding lemmas manually.
  - Delegating the proof to the NuITP.
- DM-Check provides several commands:
  - `empty` — prove that a pattern has no valid instances.
  - `lemma` — generate a lemma from a proved condition.
  - `contained` — fold a pattern into the invariant and generate a lemma.
- Iteratively adding lemmas improves verification efficiency.

Previous Commands: Case Splitting

`case` *<id>* `of` *<parent-id>* `on` *<variable>* `with` *<generator-set>*

Goal: Refine a constrained pattern by performing case analysis on a variable.

- Instantiates a variable in the constrained pattern `id`.
- The pattern `id` must be a child generated by one narrowing step from the pattern `parent-id` in the inductive invariant.
- The variable is instantiated using a constructor-based generator set for its sort.
- Produces several refined constrained patterns representing the possible cases.

$$P(x) \Rightarrow P(c_1) \vee P(c_2) \vee \ldots$$

Outline

The New Semantic-Equivalence Commands (1/4)

Variant Unify

- Pattern: $u \mid \varphi \wedge t = t'$
- t, t' belong to a subtheory with the finite variant property
- Variant unifiers: $\theta_1, \ldots, \theta_k$
- Semantically equivalent to the pattern disjunction:

$$u\theta_1 \mid \varphi\theta_1 \ \vee \ \ldots \ \vee \ u\theta_k \mid \varphi\theta_k$$

QLOCK (5/10): variant unify

```
Term 4:
  < $17:MSet | $18:MSet | mt | $16:List >
Matching: no matching found
Constraint 4:
  (non-mt($17:MSet $18:MSet) = tt) /\ (set($17:MSet $18:MSet) = tt) /\ l2ms($16:List) = $18:MSet

DM-Check> vunify to 4 with non-mt($17:MSet $18:MSet) =? tt .

    Constrained terms on the left that could not be subsumed:

Term 5:
  < $19:NeMSet | mt | mt | $20:List >
Matching: no matching found
Constraint 5:
  (set($19:NeMSet) = tt) /\ l2ms($20:List) = mt

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet

Term 7:
  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
Constraint 7:
  (set($23:NeMSet $24:NeMSet) = tt) /\ l2ms($25:List) = $24:NeMSet
```

- `Term 4` becomes now equivalent to those `Term 5`, `Term 6` and `Term 7`.

QLOCK (6/10): case

```
DM-Check> case to 5 on $19:NeMSet with m:Nat ;; k:Nat S:NeMSet .

    Constrained terms on the left that could not be subsumed:

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet

Term 7:
  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
Constraint 7:
  (set($23:NeMSet $24:NeMSet) = tt) /\ l2ms($25:List) = $24:NeMSet

DM-Check> case to 7 on $23:NeMSet with m:Nat ;; k:Nat S:NeMSet .

    Constrained terms on the left that could not be subsumed:

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet
```

- Subsumption is automatically rechecked, leaving only `Term 6`.

The New Semantic-Equivalence Commands: (2/4)

Narrowing

- Pattern: $u \mid \varphi \wedge t[f(u_1, \ldots, u_n)]_p = t'$
- $u_1, \ldots, u_n$ are constructor terms
- Narrow $f(u_1, \ldots, u_n)$ using equations E_f modulo B_Ω
- Semantically equivalent to the pattern disjunction:

$$(u \mid \varphi \wedge t[w_1]_p = t')\alpha_1 \ \vee \ \ldots \ \vee \ (u \mid \varphi \wedge t[w_k]_p = t')\alpha_k$$

QLOCK (7/10): narrow

```
Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  (set($21:NeMSet) = tt) /\ l2ms($22:List) = $21:NeMSet

DM-Check> narrow to 6 on l2ms($22:List) .

    Constrained terms on the left that could not be subsumed:

Term 7:
  < mt | $23:NeMSet | mt | nil >
Matching: no matching found
Constraint 7:
  (set($23:NeMSet) = tt) /\ mt = $23:NeMSet

Term 8:
  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
Matching: no matching found
Constraint 8:
  (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet
```

- We are now only left with two patterns: `Term 7` and `Term 8`.
- These patterns are easier to discard.

QLOCK (8/10): variant unify

```
Term 7:
  < mt | $23:NeMSet | mt | nil >
Matching: no matching found
Constraint 7:
  (set($23:NeMSet) = tt) /\ mt = $23:NeMSet

Term 8:
  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
Matching: no matching found
Constraint 8:
  (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet

DM-Check> vunify to 7 with $23:NeMSet =? mt .

    Constrained terms on the left that could not be subsumed:

Term 8:
  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
Matching: no matching found
Constraint 8:
  (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet
```

- `Term 7` is folded, and only `Term 8` is left.

New Semantic-Equivalence Commands (3/4)

Variable Abstraction

- Pattern: $u \mid \varphi \wedge t[v]_p = t'$ with v a non-variable subterm
- Replace v by a fresh variable x of the same sort
- Semantically equivalent to:

$$u \mid \varphi \wedge t[x]_p = t' \wedge x = v$$

QLOCK (9/10): variable abstraction

```
Term 8:
  < mt | $24:NeMSet | mt | $25:Nat ; $26:List >
Matching: no matching found
Constraint 8:
  (set($24:NeMSet) = tt) /\ ($25:Nat l2ms($26:List)) = $24:NeMSet

DM-Check> va to 8 on l2ms($26:List) with $27:MSet .

    Constrained terms on the left that could not be subsumed:

Term 9:
  < mt | $27:NeMSet | mt | $28:Nat ; $29:List >
Matching: no matching found
Constraint 9:
  (tt = set($27:NeMSet)) /\ ($30:MSet = l2ms($29:List)) /\ $27:NeMSet = $28:Nat $30:MSet
```

- We get a semantically equivalent pattern to `Term 8`: `Term 9`.

New Semantic-Equivalence Commands (4/4)

Substitution

- Pattern: $u \mid \varphi \wedge x = v$
- Conditions: $x \notin Var(v)$ and $sort(v) \leq sort(x)$
- Apply substitution $x \mapsto v$
- Semantically equivalent to:

$$(u \mid \varphi)\{x \mapsto v\}$$

QLOCK (10/10): substitution

```
Term 9:
  < mt | $27:NeMSet | mt | $28:Nat ; $29:List >
Matching: no matching found
Constraint 9:
  (tt = set($27:NeMSet)) /\ ($30:MSet = l2ms($29:List)) /\ $27:NeMSet = $28:Nat $30:MSet

DM-Check> substitution to 9 on $27:NeMSet with $28:Nat $30:MSet .

    Subsumption satisfied.
```

- Every pattern is subsumed now, so we deadlock freedom is proved for QLOCK.

The New Constrained Pattern Generalization Command

Generalization Command

- Original pattern: $u \mid \varphi$
- Generalized pattern: $v \mid \psi$
- Semantics:

$$u \mid \varphi \sqsubseteq_{B_\Omega} v \mid \psi$$

- The generalized pattern subsumes the original one
- Allows over-approximation of states in fixpoint computations
- Supports inductive invariant generation by simplifying patterns

Outline

Generating Inductive Invariants

- The new features of DM-Check, including `generalize`, allow interactive invariant generation.
- Methodology combines:
  - Folding narrowing search for symbolic exploration.
  - Deductive reasoning for subsumption and pattern equivalence.
  - Interactive commands for generalization and lemma generation.
- Illustrated through a case study: Lamport's bakery protocol.

Fixpoint Generation: Methodology

- Start with initial pattern disjunction:

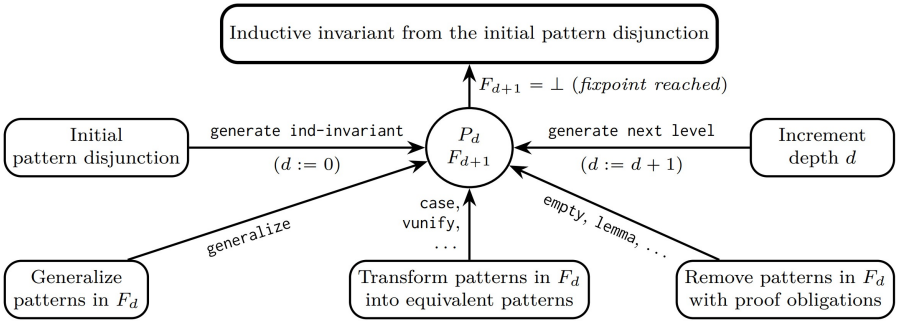
$$P_0 = F_0 = u_1 \mid \varphi_1 \vee \cdots \vee u_n \mid \varphi_n$$

- Iteratively compute:
  - F_{d+1}: successors of F_d via constrained narrowing, excluding patterns subsumed by P_d.
  - $P_{d+1} = P_d \vee F_{d+1}$.
- Fixpoint reached when $F_{d+1} = \emptyset$:

$$P_{d+1} = P_d \quad \Rightarrow \quad P_d \text{ is an inductive invariant.}$$

- Interactive mode helps overcome challenges:
  - Generalizing patterns for over-approximation.
  - Proving semantic equivalence.
  - Adding inductive lemmas for constraints.

Fixpoint Generation: Workflow



- Automated reachability analysis is combined with interactive commands.
- Several interactive commands are available.
- Remaining proof obligations can be discharged with NuITP.

Lamport's Bakery Protocol (1/11): Representation of the States

- Multiple processes compete to enter the critical section using ticket numbers.
- Infinite state space due to:
  - Unbounded number of processes.
  - Unbounded ticket numbers.
- Maude state: $n \mid m \mid [md_1]\ ;\ \ldots\ ;\ [md_k]$
  - n: current number in dispenser
  - m: number being served
  - $[md_i]$: process mode (`idle`, `wait(j)`, `crit(j)`)

Lamport's Bakery Protocol (2/11): Rewrite Rules and Initial State

- Rewrite rules:

```
rl [wake] : N | M | [idle]     ; PS  =>  1 + N | M | [wait(N)] ; PS .
rl [crit] : N | M | [wait(M)] ; PS  =>  N | M | [crit(M)] ; PS .
rl [exit] : N | M | [crit(M)] ; PS  =>  N | 1 + M | [idle] ; PS .
```

- Parametric initial state:

$$(N \mid N \mid IS) \mid true$$

- Error pattern (mutual exclusion violation):

$$(M \mid N \mid [crit(K)]\ ;\ [crit(L)]\ ;\ PS) \mid true$$

Lamport's Bakery Protocol (3/11): `generate ind-invariant`

```
DM-Check> generate ind-invariant \
>     (N:iNat | N:iNat | IS:ProcIdleSet) | true .

    Invariant could not be proved.

Parent 1
Term:
  $2:iNat | $2:iNat | $1:ProcIdleSet
Constraint:
  true

Child 2 (Parent 1)
Term:
  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Rule: wake
Unifier:
$1:ProcIdleSet <- $3:ProcIdleSet ; [idle], $2:iNat <- $4:iNat
Matching: no matching found
Constraint:
  true
```

- The inductive invariant cannot be generated in only one step.

Lamport's Bakery Protocol (4/11): `generate next level`

```
DM-Check> generate next level .

    Invariant could not be proved.

Parent 1
Term:
  $2:iNat | $2:iNat | $1:ProcIdleSet
Constraint:
  true

Parent 2
Term:
  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Constraint:
  true

Child 3 (Parent 2)
Term:
  (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ; [wait($5:iNat)] ; [wait(1 + $5:iNat)]
Rule: wake
Unifier:
$3:ProcIdleSet <- $6:ProcIdleSet ; [idle], $4:iNat <- $5:iNat
Matching: no matching found
Constraint:
  true

Child 4 (Parent 2)
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
Rule: crit
Unifier:
$3:ProcIdleSet <- $8:ProcIdleSet, $4:iNat <- $7:iNat
Matching: no matching found
Constraint:
  true
```

- One more level is not enough: we need additional reasoning.

Lamport's Bakery Protocol (5/11): generalize

```
DM-Check> generalize 3 to \
>       (M:iNat + N:iNat | N:iNat | WS:ProcWaitSet) | \
>             (tickets(WS:ProcWaitSet) = i(N:iNat, M:iNat + N:iNat)) .

    Invariant could not be proved.

Parent 1
Term:
  $2:iNat | $2:iNat | $1:ProcIdleSet
Constraint:
  true

Parent 2
Term:
  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Constraint:
  true

Child 3 (Parent 2)
Term:
  ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
Rule: none
Unifier:

Matching: no matching found
Constraint:
  tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)

Child 4 (Parent 2)
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
```

- `Child 3` generalized into a pattern with `ProcWaitSet`.

Lamport's Bakery Protocol (6/11): `generate next level`

```
DM-Check> generate next level .

    Invariant could not be proved.

Parent 1
Term:
  ($3:iNat + $1:iNat) | $1:iNat | $2:ProcWaitSet
Constraint:
  tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat)

Parent 2
Term:
  ($4:iNat + $5:iNat) | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]
Constraint:
  tickets($6:ProcWaitSet ; [wait($4:iNat)]) = i($4:iNat, $5:iNat + $4:iNat)

Child 4 (Parent 2)
Term:
  ($10:iNat + $11:iNat) | 1 + $10:iNat | $12:ProcWaitSet ; [idle]
Rule: exit
Unifier:
$4:iNat <- $10:iNat, $5:iNat <- $11:iNat, $6:ProcWaitSet <- $12:ProcWaitSet
Matching: no matching found
Constraint:
  tickets($12:ProcWaitSet ; [wait($10:iNat)]) = i($10:iNat, $11:iNat + $10:iNat)

Child 5 (Parent 2)
Term:
  ($13:iNat + $14:iNat) | $13:iNat | $15:ProcWaitSet ; [crit($13:iNat)] ; [crit($13:iNat)]
Rule: crit
Unifier:
$4:iNat <- $13:iNat, $5:iNat <- $14:iNat, $6:ProcWaitSet <- $15:ProcWaitSet ; [wait($13:iNat)]
Matching: no matching found
Constraint:
  tickets(($15:ProcWaitSet ; [wait($13:iNat)]) ; [wait($13:iNat)]) = i($13:iNat, $14:iNat + $13:iNat)
```

- Now we generate two more levels.

Lamport's Bakery Protocol (7/11): `empty`

```
DM-Check> empty 5 .

    Invariant could not be proved.

Parent 1
Term:
  ($3:iNat + $1:iNat) | $1:iNat | $2:ProcWaitSet
Constraint:
  tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat)

Parent 2
Term:
  ($4:iNat + $5:iNat) | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]
Constraint:
  tickets($6:ProcWaitSet ; [wait($4:iNat)]) = i($4:iNat, $5:iNat + $4:iNat)

Child 4 (Parent 2)
Term:
  ($10:iNat + $11:iNat) | 1 + $10:iNat | $12:ProcWaitSet ; [idle]
Rule: exit
Unifier:
$4:iNat <- $10:iNat, $5:iNat <- $11:iNat, $6:ProcWaitSet <- $12:ProcWaitSet
Matching: no matching found
Constraint:
  tickets($12:ProcWaitSet ; [wait($10:iNat)]) = i($10:iNat, $11:iNat + $10:iNat)
```

- `Term 5` has no valid instances (`empty` command).

Lamport's Bakery Protocol (8/11): case

```
Child 5 (Parent 2)
Term:
  $13:iNat | 1 + $13:iNat | $14:ProcWaitSet ; [idle]
Rule: exit
Unifier:
$4:iNat <- $13:iNat, $5:iNat <- 0, $6:ProcWaitSet <- $14:ProcWaitSet
Matching: no matching found
Constraint:
  (e($13:iNat), tickets($14:ProcWaitSet)) = null

Child 6 (Parent 2)
Term:
  (1 + $15:iNat) | 1 + $15:iNat | $16:ProcWaitSet ; [idle]
Rule: exit
Unifier:
$4:iNat <- $15:iNat, $5:iNat <- 1, $6:ProcWaitSet <- $16:ProcWaitSet
Matching parent #1: 1
Matching substitution #1:
$1:iNat <- 1 + $15:iNat, $2:ProcWaitSet <- $16:ProcWaitSet ; [idle], $3:iNat <- 0
NuITP implication #1 (open):
(e($15:iNat), tickets($16:ProcWaitSet)) = e($15:iNat) -> tickets($16:ProcWaitSet ; [idle]) = i(1 + $15:iNat, 0 + 1 +
    $15:iNat)

Child 7 (Parent 2)
Term:
  (1 + $18:iNat + $17:iNzNat) | 1 + $18:iNat | $19:ProcWaitSet ; [idle]
Rule: exit
Unifier:
$4:iNat <- $18:iNat, $5:iNat <- 1 + $17:iNzNat, $6:ProcWaitSet <- $19:ProcWaitSet
Matching parent #1: 1
Matching substitution #1:
$1:iNat <- 1 + $18:iNat, $2:ProcWaitSet <- $19:ProcWaitSet ; [idle], $3:iNat <- $17:iNzNat
NuITP implication #1 (open):
(e($18:iNat), tickets($19:ProcWaitSet)) = e($18:iNat + $17:iNzNat), i($18:iNat, $18:iNat + $17:iNzNat) -> tickets(
    $19:ProcWaitSet ; [idle]) = i(1 + $18:iNat, $17:iNzNat + 1 + $18:iNat)
```

- We can do a case analysis on `Term 4`.

Lamport's Bakery Protocol (9/11): Invariant Generated

```
  The following invariant is satisfied

(((($3:iNat + $1:iNat) | $1:iNat | $2:ProcWaitSet) | (tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat))) \/
(((($4:iNat + $5:iNat) | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) | (tickets($6:ProcWaitSet ; [wait($4:iNat)]) = i(
  $4:iNat, $5:iNat + $4:iNat)))

  with the following proof obligations:

(e($13:iNat), tickets($14:ProcWaitSet)) = null -> false
(e($15:iNat), tickets($16:ProcWaitSet)) = e($15:iNat) -> tickets($16:ProcWaitSet ; [idle]) = i(1 + $15:iNat, 0 + 1 +
  $15:iNat)
(e($18:iNat), tickets($19:ProcWaitSet)) = e($18:iNat + $17:iNzNat), i($18:iNat, $18:iNat + $17:iNzNat) -> tickets(
  $19:ProcWaitSet ; [idle]) = i(1 + $18:iNat, $17:iNzNat + 1 + $18:iNat)
tickets(($15:ProcWaitSet ; [wait($13:iNat)]) ; [wait($13:iNat)]) = i($13:iNat, $14:iNat + $13:iNat) -> false
```

- We obtain `Term 5`, `Term 6` and `Term 7`, but:
  - `Term 5` has no valid instances (`empty` command).
  - We generate lemmas from `Term 6` and `Term 7` (`lemma` command)
- The inductive invariant is generated with proof obligations.
- The proof obligations can be discharged by using the NuITP.

Lamport's Bakery Protocol (10/11): Prove Other Invariants

```
DM-Check> check ($3:iNat + $1:iNat | $1:iNat | $2:ProcWaitSet) | \
>         tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat) \/ \
>       ($4:iNat + $5:iNat | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) | \
>         tickets($6:ProcWaitSet ; [wait($4:iNat)]) = \
>         i($4:iNat, $5:iNat + $4:iNat) \
> subsumed by (M:iNat + N:iNat | N:iNat | PS:ProcSet) | \
>         tickets(PS:ProcSet) = i(N:iNat, N:iNat + M:iNat) .

    Subsumption satisfied with the following proof obligations:

  (e($13:iNat), tickets($14:ProcWaitSet)) = null -> false
  (e($15:iNat), tickets($16:ProcWaitSet)) = e($15:iNat) -> tickets($16:ProcWaitSet ; [idle]) = i(1 + $15:iNat, 0 + 1 +
    $15:iNat)
  (e($18:iNat), tickets($19:ProcWaitSet)) = e($18:iNat + $17:iNzNat), i($18:iNat, $18:iNat + $17:iNzNat) -> tickets(
    $19:ProcWaitSet ; [idle]) = i(1 + $18:iNat, $17:iNzNat + 1 + $18:iNat)
  tickets(($15:ProcWaitSet ; [wait($13:iNat)]) ; [wait($13:iNat)]) = i($13:iNat, $14:iNat + $13:iNat) -> false
```

- We can verify other invariants using the generated inductive invariant.

Lamport's Bakery Protocol (11/11): Mutual Exclusion

```
DM-Check> intersect \
>       ($3:iNat + $1:iNat | $1:iNat | $2:ProcWaitSet) | \
>         tickets($2:ProcWaitSet) = i($1:iNat, $3:iNat + $1:iNat) \/ \
>       ($4:iNat + $5:iNat | $4:iNat | $6:ProcWaitSet ; [crit($4:iNat)]) | \
>         tickets($6:ProcWaitSet ; [wait($4:iNat)]) = \
>         i($4:iNat, $5:iNat + $4:iNat) \
> with \
>       (M:iNat | N:iNat | [crit(K:iNat)] ; [crit(L:iNat)] ; PS:ProcSet) | true .

    No intersection.
```

- We can confirm that the mutual exclusion property holds for all reachable states.

Outline

Conclusions

- Previous deductive model checking with DM-Check was limited to proving inductive invariants (depth-1 bounded model checking).
- In this work we extend this methodology with:
  - New inference rules for semantic equivalence of constrained patterns.
  - A generalization command to over-approximate sets of states.
  - An unbounded deductive model checking approach to generate inductive invariants.
  - A new version of the DM-Check tool supporting these features.
- Demonstrated on the QLOCK and BAKERY protocols.

Future Work

- Further experimentation with additional case studies and improvements to DM-Check to increase scalability.
- Investigating further automation of the methodology:
  - Using more powerful NuITP strategies to automatically fold more constrained patterns.
  - Automatically applying some DM-Check inference rules via strategies.
- Example: automatically applying the `generalize` command when looping patterns are detected.

Thank you!