# A Verified SAT Solver with Watched Literals Using Imperative HOL

Peter Lammich

Mathias Fleury

Jasmin C. Blanchette

# SAT Solving

Given a formula in conjunctive normal form

$$\varphi = \bigwedge_i \bigvee_j L_{i,j}$$

is there an assignment making the formula true?

Most used algorithm: CDCL, an improvement over DPLL

# How reliable are SAT solvers?

Two ways to ensure correctness:

- certify the certificate
  - certificates are huge
- verification of the code
  - code will not be competitive
  - allows to study metatheory

I certify your proof

Isabelle

IsaFoL

# IsaFoL project

Isabelle Formalisation of Logic

# IsaFoL

- FO resolution
  by Schlichtkrull (ITP 2016)

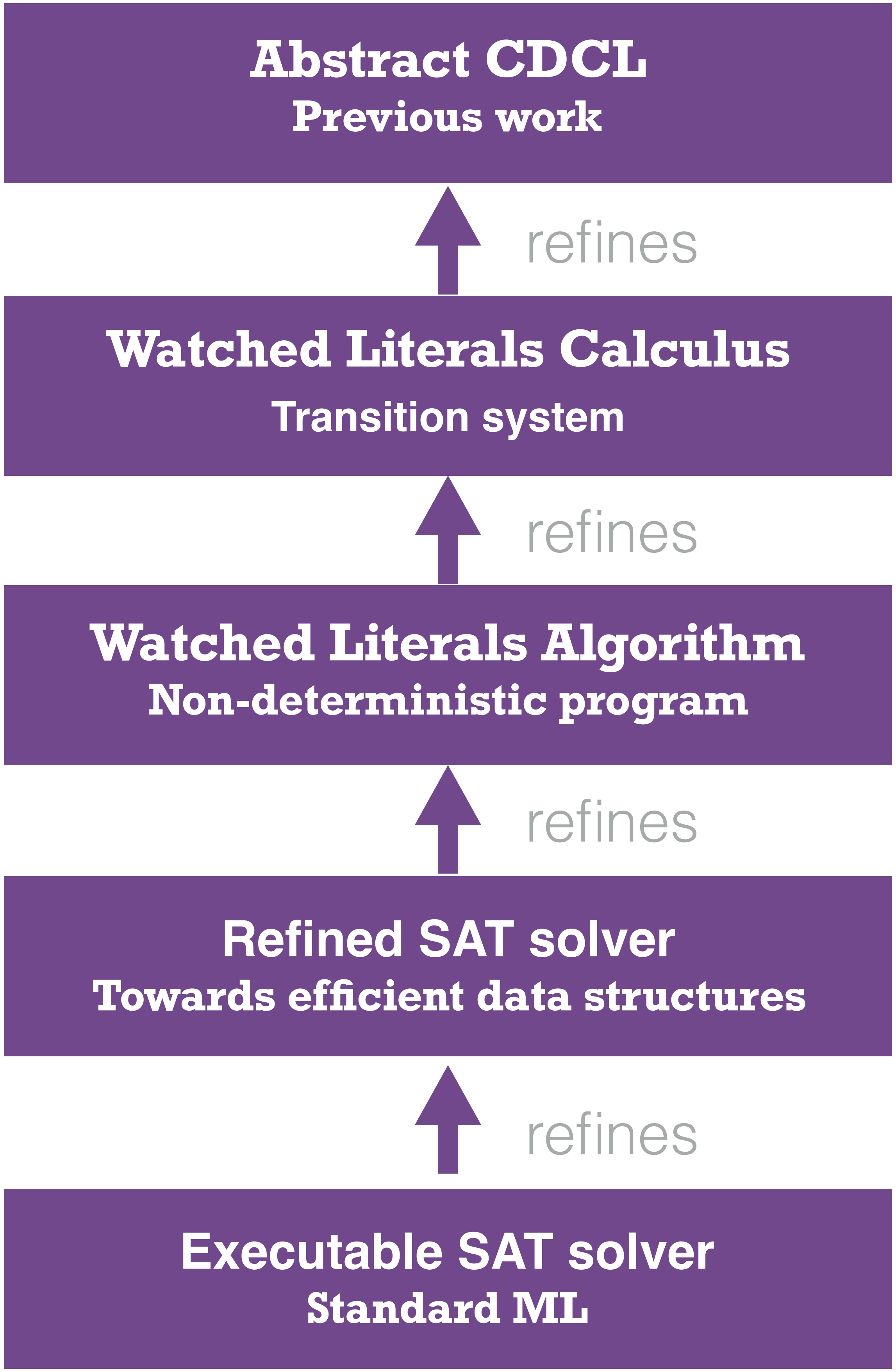
- CDCL with learn, forget, restart, and incrementality
  by Blanchette, Fleury, Weidenbach (IJCAR 2016)

- GRAT certificate checker
  by Lammich (CADE-26, 2017)

- A verified SAT solver with watched literals
  by Fleury, Blanchette, Lammich (now)

# IsaFoL

- FO resolution
  by Schlichtkrull (ITP 2016)
- CDCL with learn, forget, restart, and incrementality
  by Blanchette, Fleury, Weidenbach (IJCAR 2016)
- GRAT certificate checker
  by Lammich (CADE-26, 2017)
- A verified SAT solver with watched literals
  by Fleury, Blanchette, Lammich (now)



**Abstract CDCL**
Previous work

refines

**Watched Literals Calculus**
Transition system

refines

**Watched Literals Algorithm**
Non-deterministic program

refines

**Refined SAT solver**
Towards efficient data structures

refines

**Executable SAT solver**
Standard ML

# Abstract CDCL

## Previous work

# DPLL

**Candidate model**

**Clause**

1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

$A^?$

**Clause**

1. $\neg B \vee C \vee A$
2. $\neg C \vee \neg B \vee \neg A$
3. $\neg A \vee \neg B \vee C$
4. $\neg A \vee B$

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

$A^?$ B

**Clause**

1. $\neg B \lor C \lor A$
2. $\neg C \lor \neg B \lor \neg A$
3. $\neg A \lor \neg B \lor C$
4. $\neg A \lor B$

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

$A^{?}$ B ¬C

**Clause**

1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

¬A

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

**Clause**

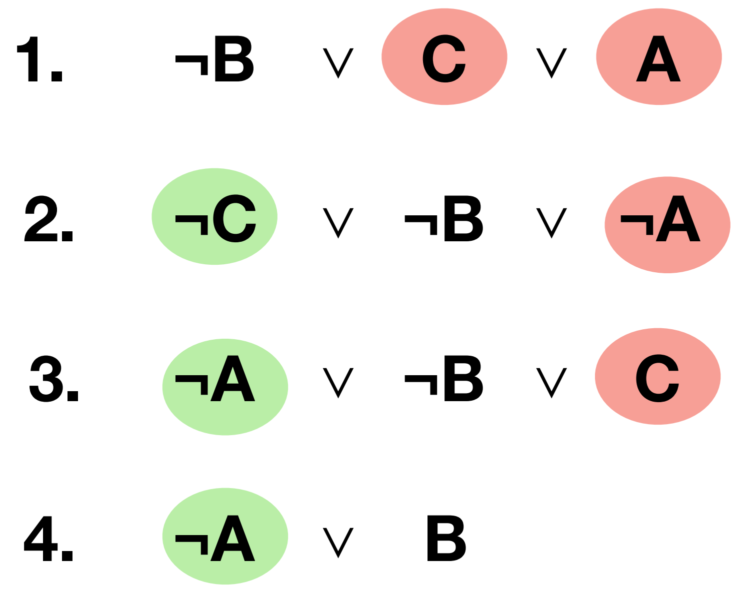
1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

# DPLL

**Candidate model**

¬A ¬C?

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

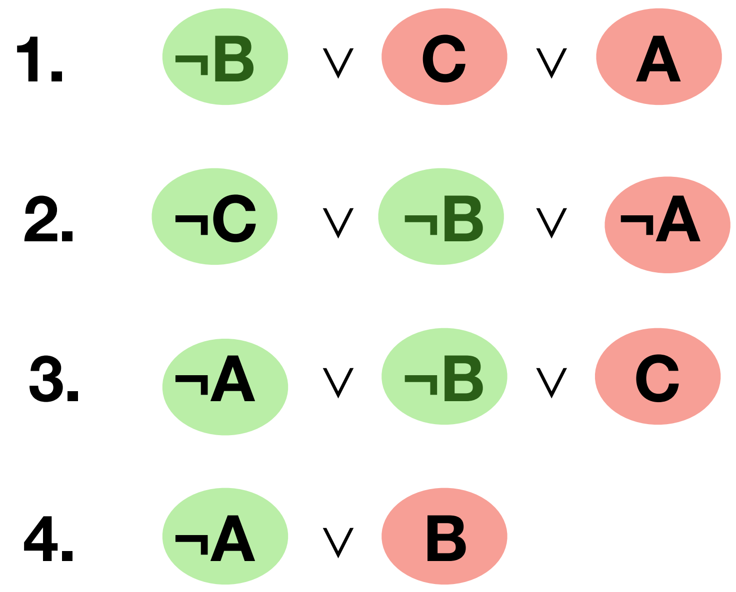
1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

$\neg A \quad \neg C^{?} \quad \neg B$

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Guess
2. or propagate information
3. or take the opposite of the last guess if there is a conflict

# DPLL

**Candidate model**

¬A

**Clause**

1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B
5. ¬A

CDCL = DPLL +
non-chronological backtracking +
learning

in Isabelle

## Propagate rule

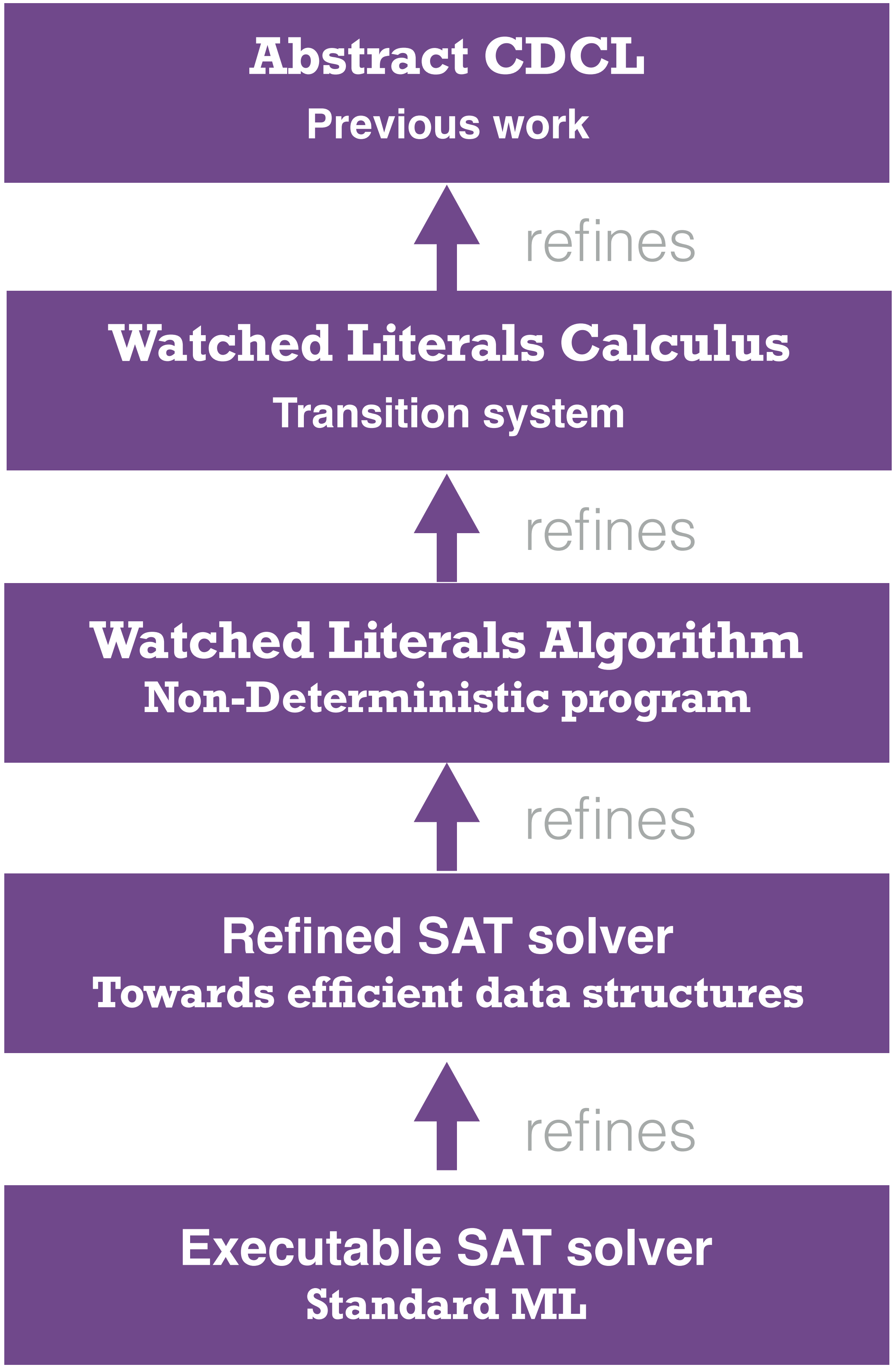
```
C ∨ L ∈ N ⟹ M ⊨as ¬C ⟹ undefined_lit M L ⟹
    (M, N) ⇒CDCL (L # M, N)
```

in Isabelle

## Propagate rule

`C ∨ L ∈ N ⟹ M ⊨as ¬C ⟹ undefined_lit M L ⟹`
`    (M, N) ⇒CDCL (L # M, N)`

**Problem:**
Iterating over the clauses is inefficient



**Abstract CDCL**

Previous work

refines

**Watched Literals Calculus**

Transition system

refines

**Watched Literals Algorithm**

**Non-Deterministic program**

refines

**Refined SAT solver**

**Towards efficient data structures**

refines

**Executable SAT solver**

**Standard ML**

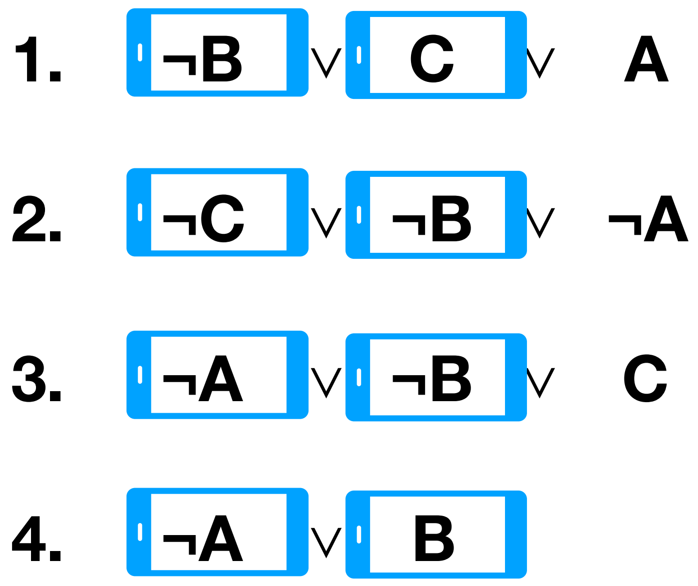
# Watched Literals Calculus

## Transition system

# DPLL with Watched Literals

**Candidate model**

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

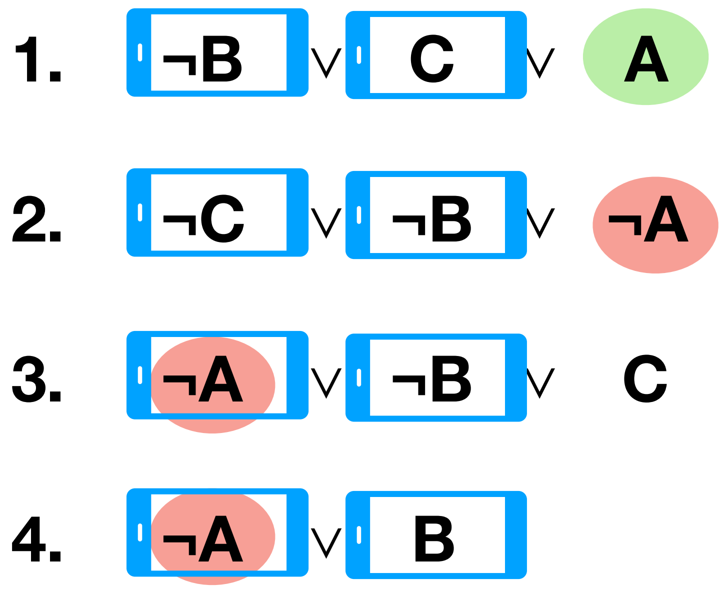
To update:

# DPLL with Watched Literals

**Candidate model**

$A^?$

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

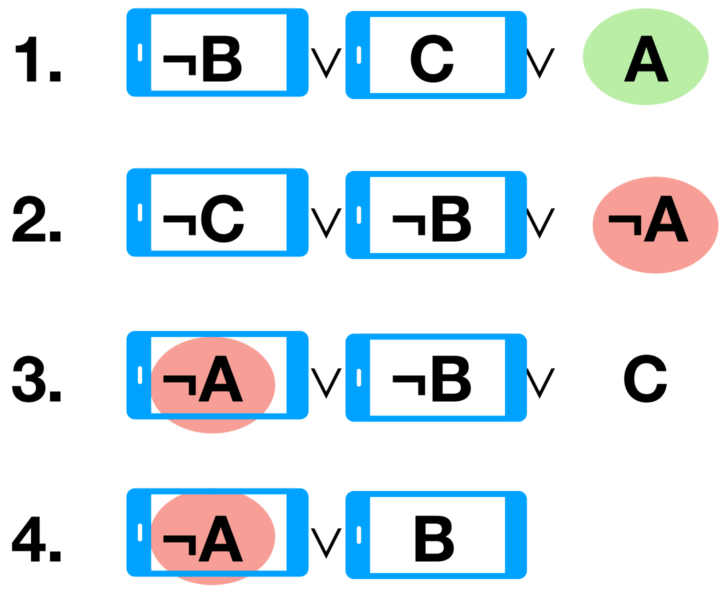
To update:

# DPLL with Watched Literals

**Candidate model**

A?

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

To update: **3. 4.**

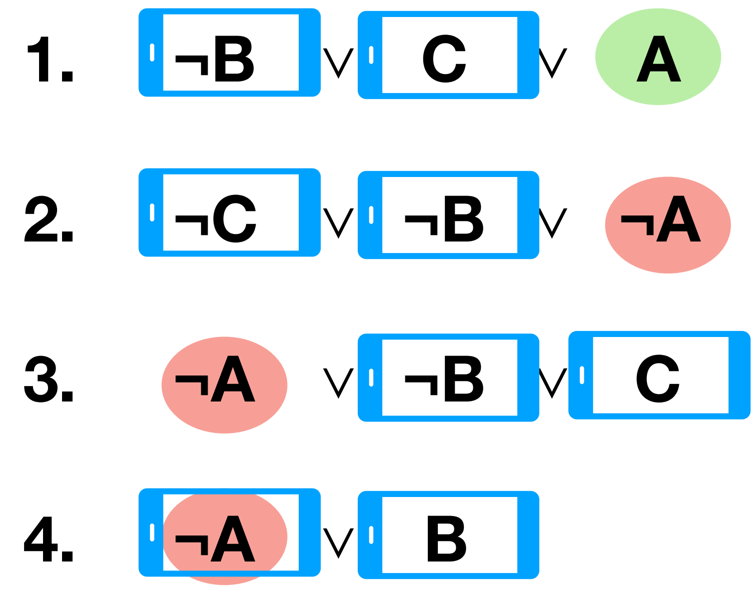
# DPLL with Watched Literals

**Candidate model**

**A?**

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

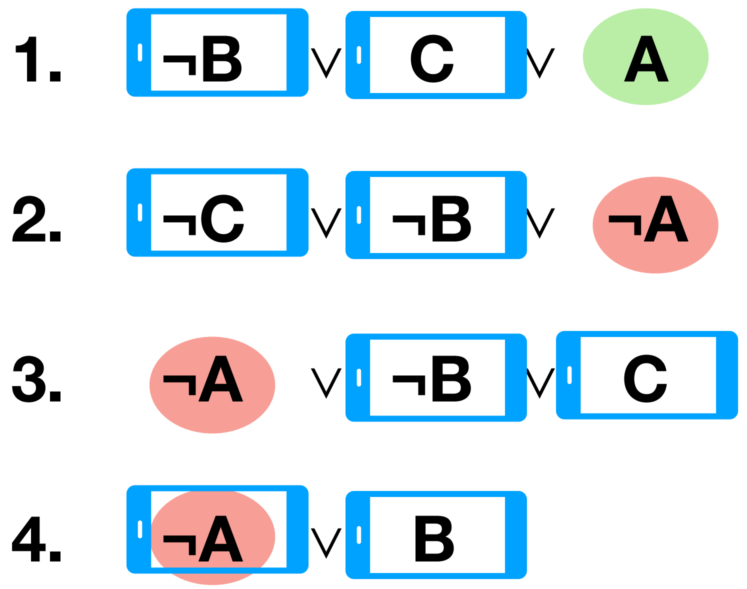
To update: **4.**

# DPLL with Watched Literals

**Candidate model**

**A?**

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

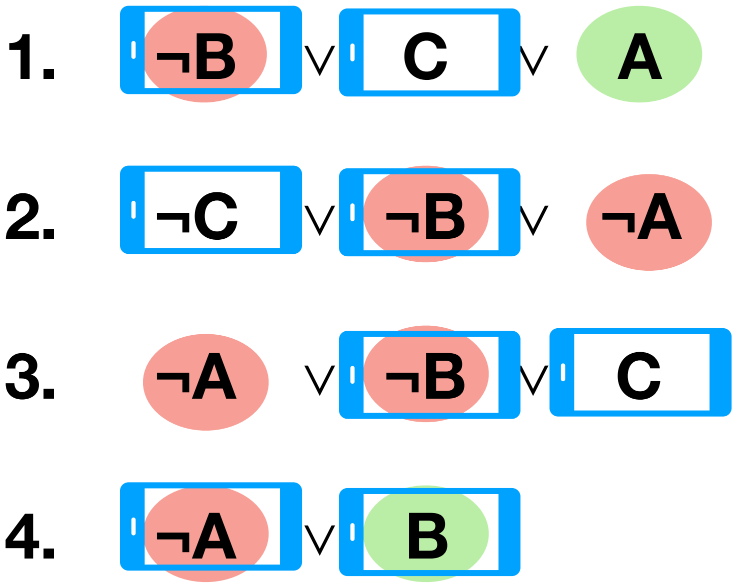
To update:

# DPLL with Watched Literals

**Candidate model**

**$A^?$ B**

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

To update:

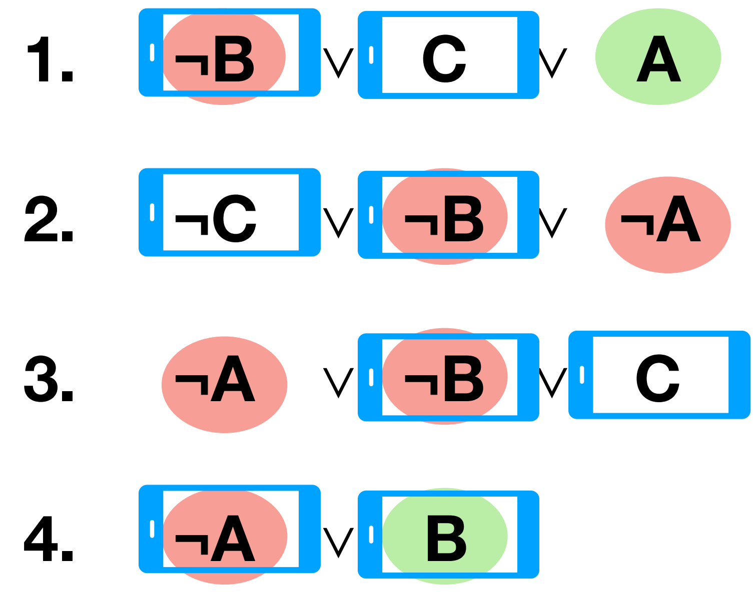
**B**

# DPLL with Watched Literals

**Candidate model**

**$A^?$ B**

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

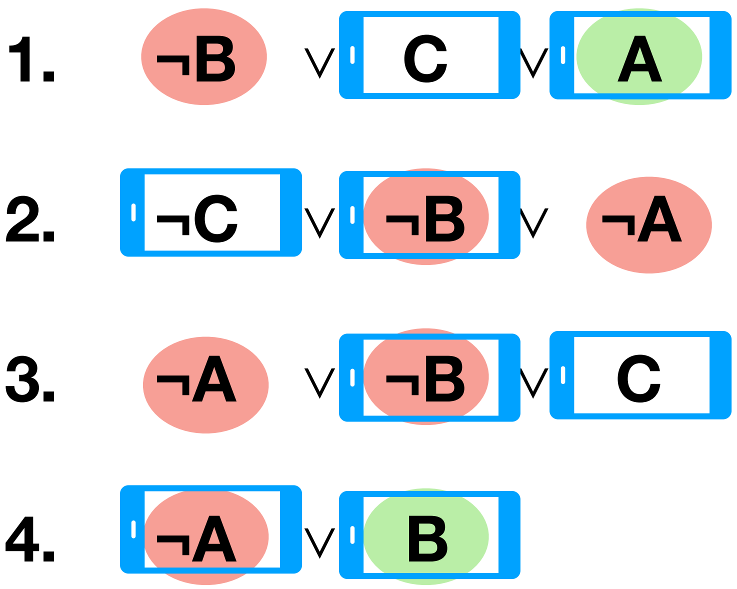
To update: **1. 2. 3.**

# DPLL with Watched Literals

**Candidate model**

**Clause**

**$A^?$ B**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

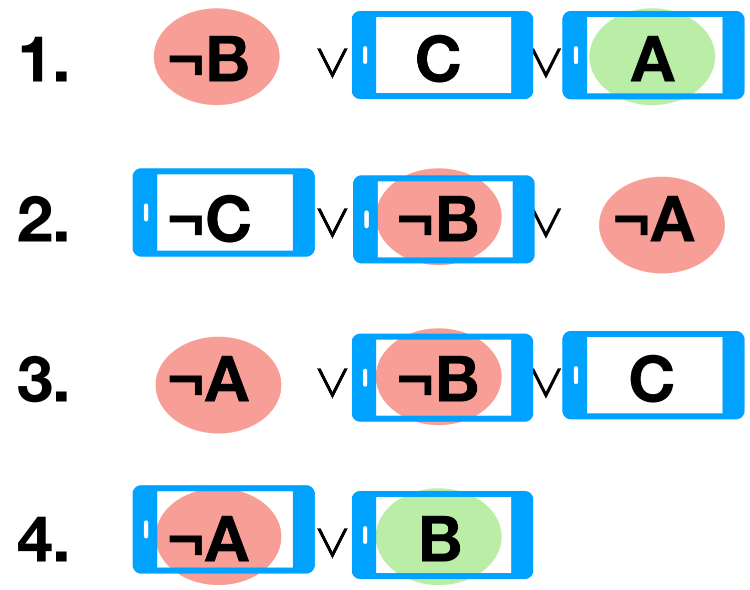
To update: **2. 3.**

# DPLL with Watched Literals

**Candidate model**

**$A^?$ B**

**Clause**



**1. ¬B ∨ C ∨ A**

**2. ¬C ∨ ¬B ∨ ¬A**

**3. ¬A ∨ ¬B ∨ C**

**4. ¬A ∨ B**

**1. Watch one true literals**
**2. or watch two unset literals**
**3. or watch a false literals**
**if all other literals are false**

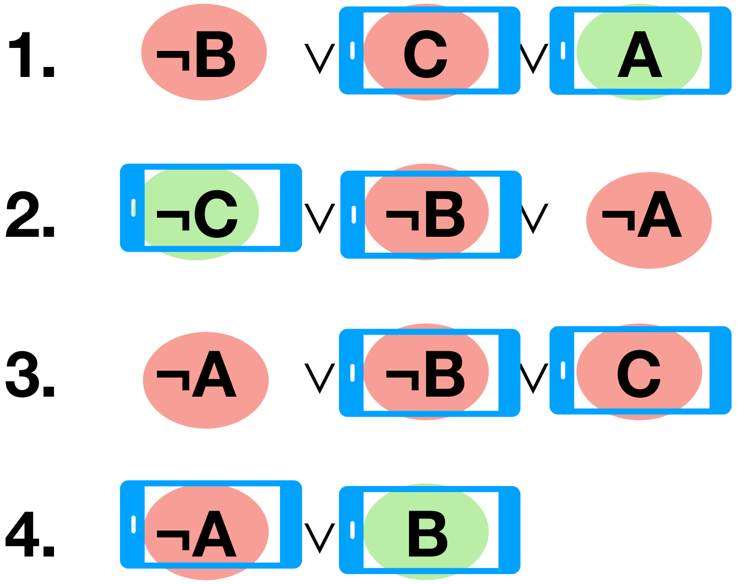
To update: **3.**

# DPLL with Watched Literals

**Candidate model**

**$A^?$ B ¬C**

**Clause**



**1.** ¬B ∨ C ∨ A

**2.** ¬C ∨ ¬B ∨ ¬A

**3.** ¬A ∨ ¬B ∨ C

**4.** ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

To update: **3.**

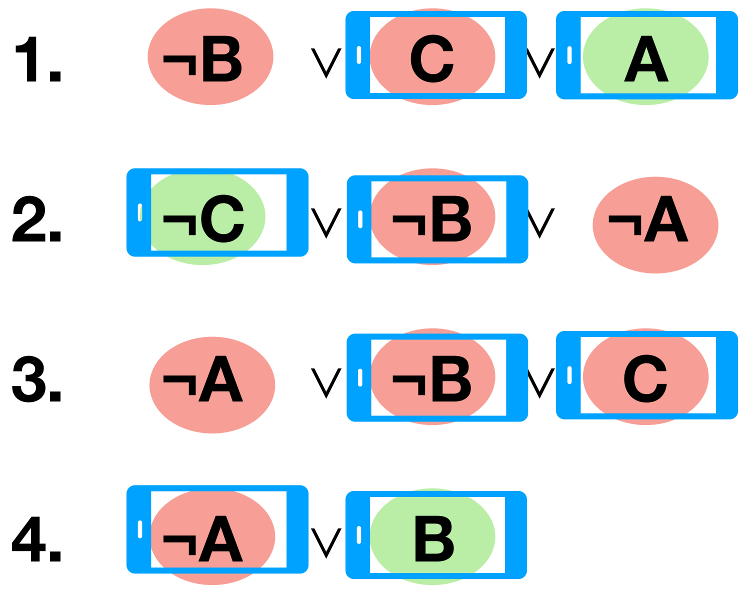
**C**

# DPLL with Watched Literals

**Candidate model**

$A^?$ B ¬C

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

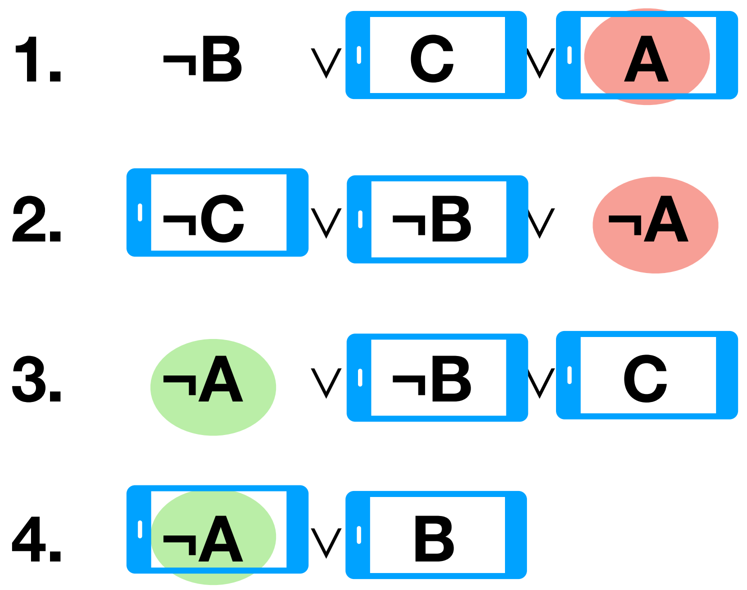
To update:

# DPLL with Watched Literals

**Candidate model**

¬A

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

To update:

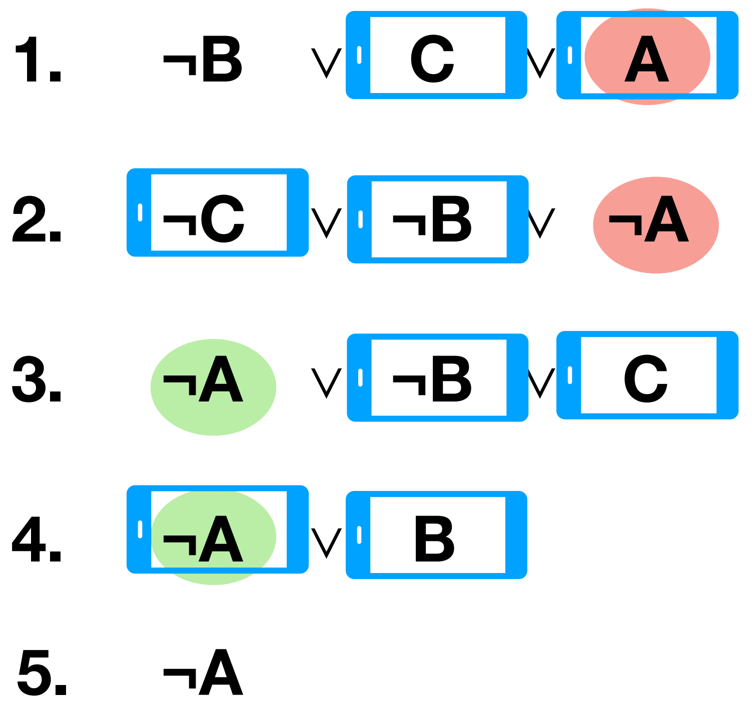
¬A

# DPLL with Watched Literals

**Candidate model**

¬A

**Clause**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B
5. ¬A

To update:

¬A

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

# Watched literals invariant

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

# Watched literals invariant

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

unless a conflict has been found

# Watched literals invariant

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

unless a conflict has been found

or an update is pending

# Watched literals invariant (less wrong)

this literal has been set earlier

1. Watch one true literals
2. or watch two unset literals
3. or watch a false literals
   if all other literals are false

unless a conflict has been found

or an update is pending

- 🔑 Finding invariants (11 new ones)
- ✗ No high-level description
- ✓ sledgehammer

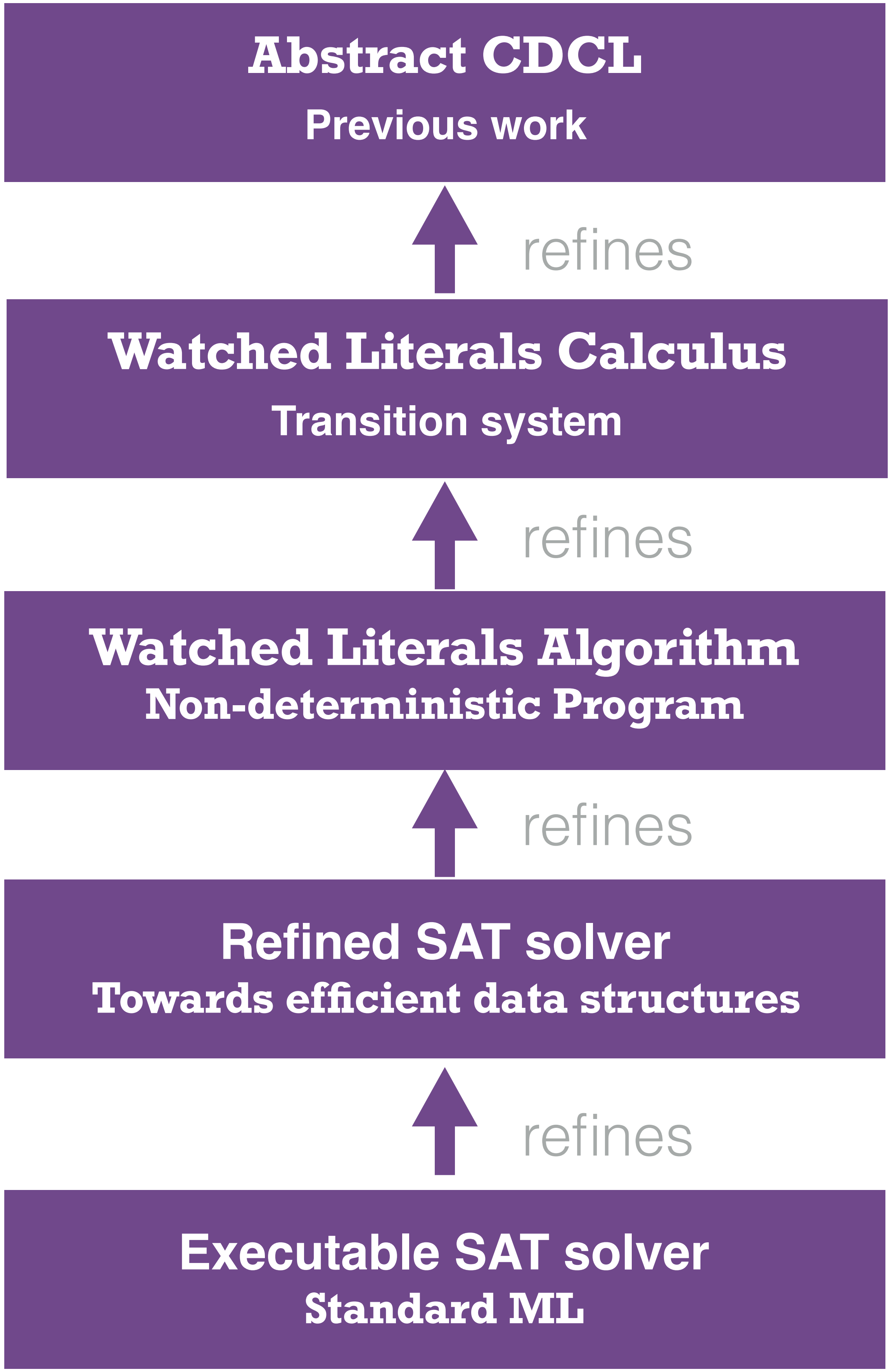
- 🔑 Finding invariants (11 new ones)
- ✗ No high-level description
- ✓ sledgehammer

**Correctness theorem**

in Isabelle

If S is well-formed and $S \Rightarrow_{\mathtt{TWL}}^{!} T$ then

$$S \Rightarrow_{\mathtt{CDCL}}^{!} T$$



**Abstract CDCL**
Previous work

refines

**Watched Literals Calculus**
Transition system

refines

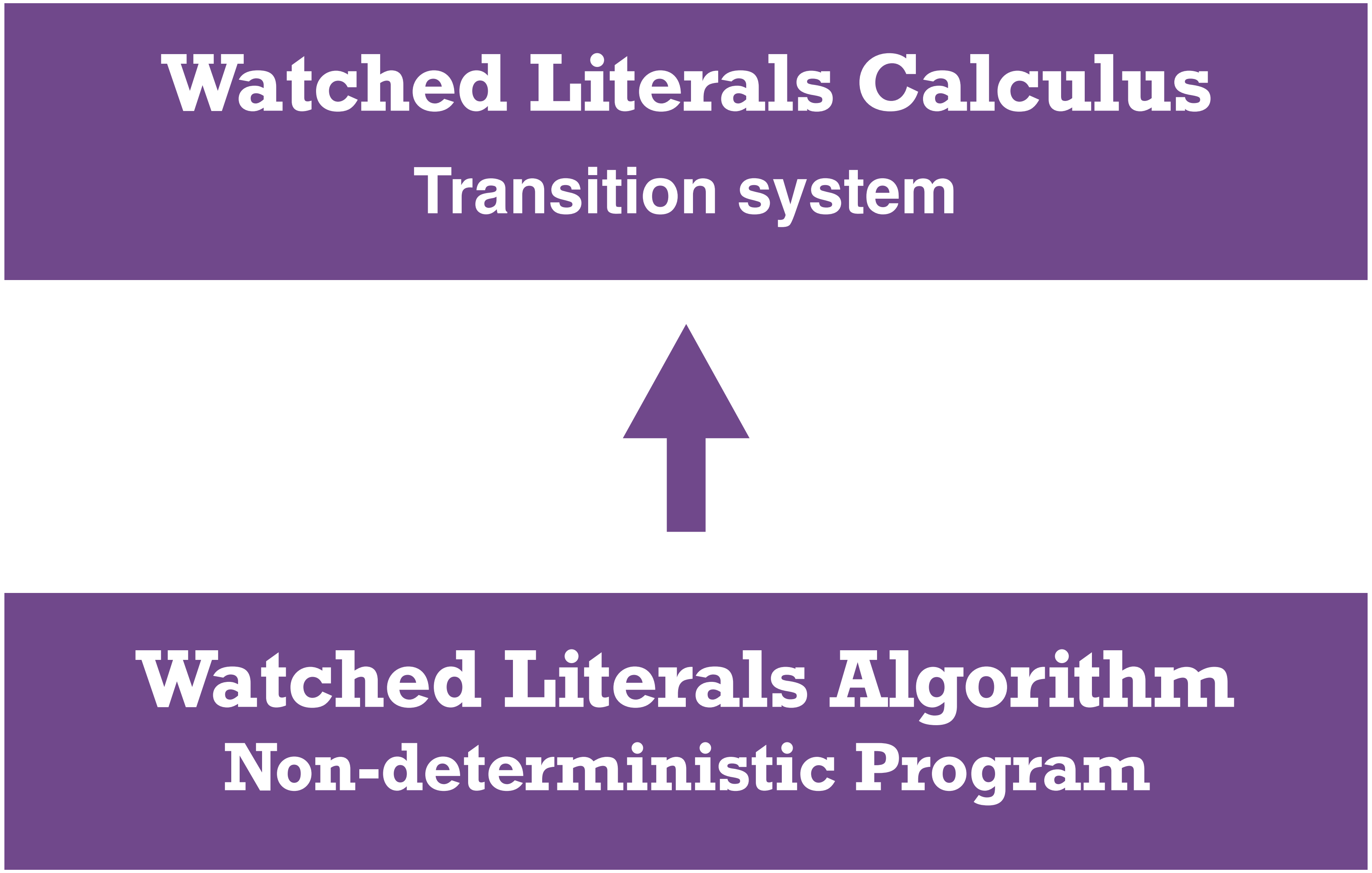
**Watched Literals Algorithm**
**Non-deterministic Program**

refines

**Refined SAT solver**
**Towards efficient data structures**

refines

**Executable SAT solver**
**Standard ML**



**Watched Literals Calculus**

Transition system

**Watched Literals Algorithm**

**Non-deterministic Program**

# Picking Next Clause

```
propagate_conflict_literal L S :=
  WHILE$_T$
     (λT. clauses_to_update T ≠ {})

     (λT. do {
        ASSERT(clauses_to_update T ≠ {})
        C ← SPEC (λC. C ∈ clauses_to_update T);
        U ← remove_from_clauses_to_update C T;
        update_clause (L, C) U
      }
     )

     S
```

# Refinement Framework:
non-deterministic exception monad

```
propagate_conflict_literal L S :=
  WHILE$_T$
    (λT. clauses_to_update T ≠ {})

    (λT. do {
       ASSERT(clauses_to_update T ≠ {})
       C ← SPEC (λC. C ∈ clauses_to_update T);
       U ← remove_from_clauses_to_update C T;
       update_clause (L, C) U
     }
    )

    S
```

## Refinement Framework:
## non-deterministic exception monad

**Assertions**

```
propagate_conflict_literal L S :=
  WHILE$_T$
    (λT. clauses_to_update T ≠ {})

    (λT. do {
      ASSERT(clauses_to_update T ≠ {})
      C ← SPEC (λC. C ∊ clauses_to_update T);
      U ← remove_from_clauses_to_update C T;
      update_clause (L, C) U
     }
    )

    S
```

# Refinement Framework:
# non-deterministic exception monad

```
propagate_conflict_literal L S :=
  WHILE_T
    (λT. clauses_to_update T ≠ {})

    (λT. do {
       ASSERT(clauses_to_update T ≠ {})
       C ← SPEC (λC. C ∈ clauses_to_update T);
       U ← remove_from_clauses_to_update C T;
       update_clause (L, C) U
     }
    )

    S
```

**Non-deterministic getting of a clause**

# Refinement Framework: non-deterministic exception monad

```
propagate_conflict_literal L S :=
  WHILE$_T$
    (λT. clauses_to_update T ≠ {})

    (λT. do {
       ASSERT(clauses_to_update T ≠ {})
       C ← SPEC (λC. C ∈ clauses_to_update T);
       U ← remove_from_clauses_to_update C T;
       update_clause (L, C) U
     }
    )

    S
```

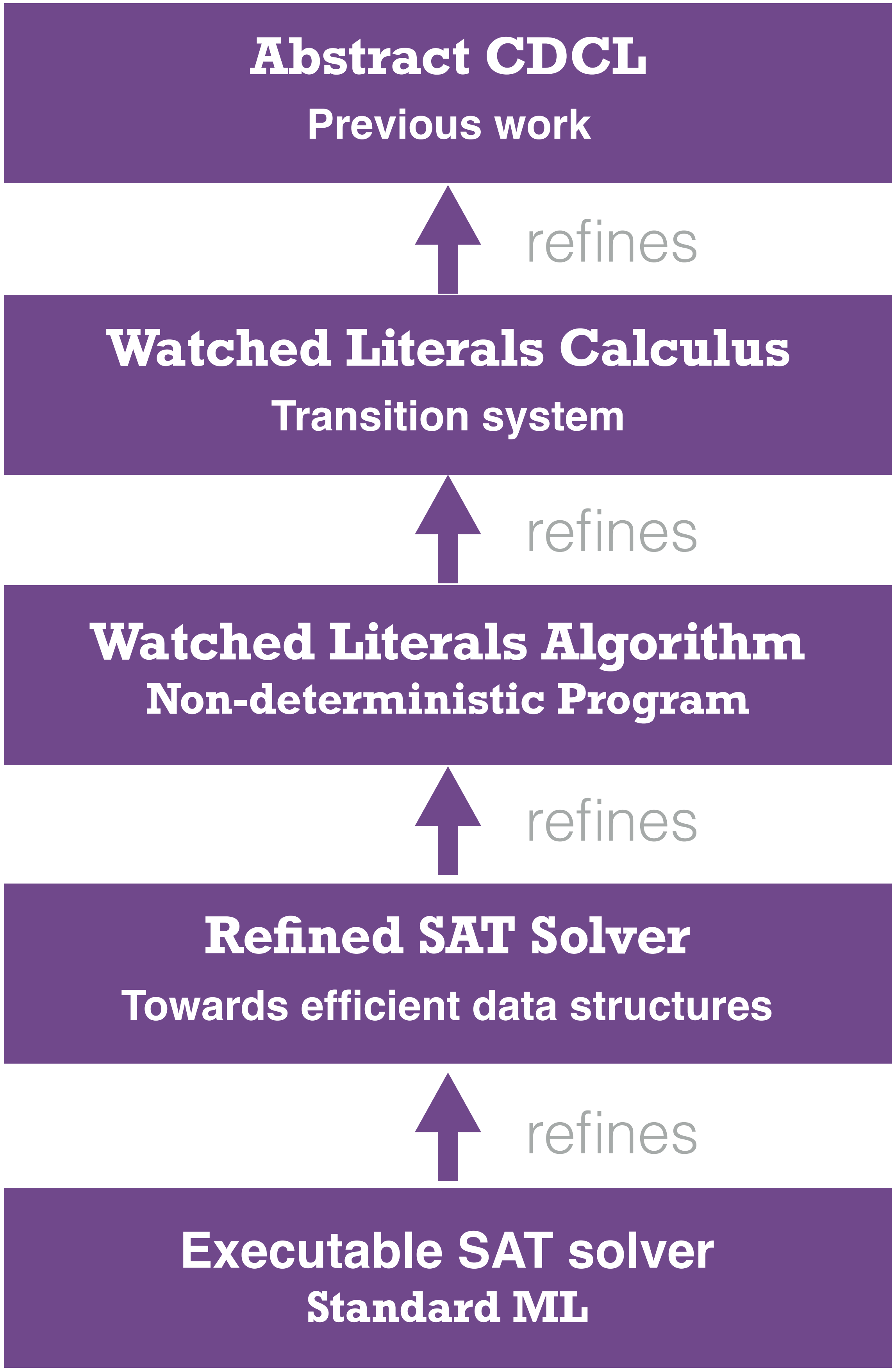
- More deterministic (order of the rules)
- But still non deterministic (decisions)
- Goals of the form

- More deterministic (order of the rules)
- But still non deterministic (decisions)
- Goals of the form

```
propagate_conflict_literal L S ≤ SPEC(λT. S ⇒TWL* T)
```

in Isabelle

- ✗ VCG’s goals hard to read
- ✗ Very tempting to write fragile proofs
- ✓ sledgehammer



**Abstract CDCL**
Previous work

refines

**Watched Literals Calculus**
Transition system

refines

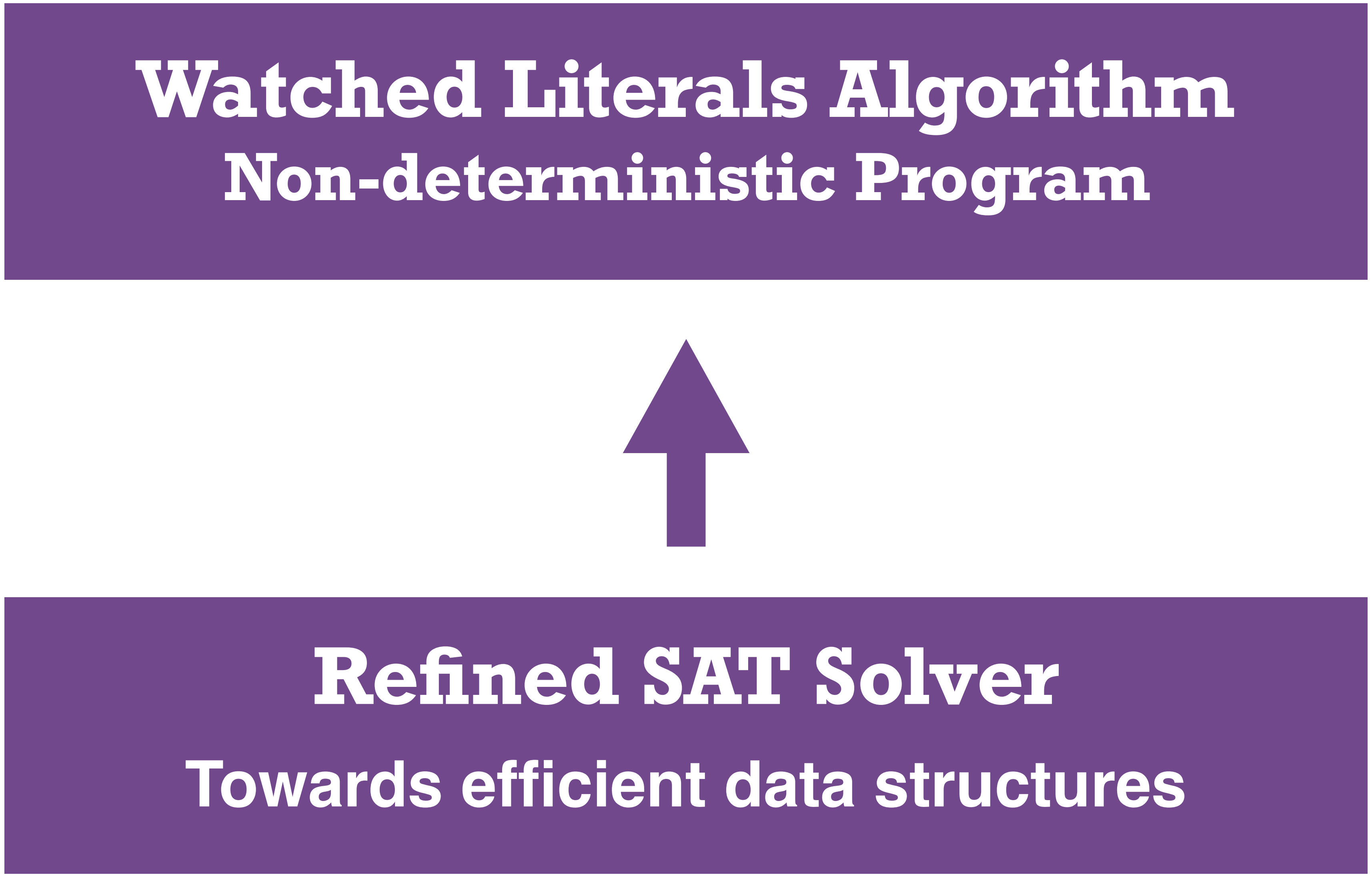
**Watched Literals Algorithm**
**Non-deterministic Program**

refines

**Refined SAT Solver**
Towards efficient data structures

refines

**Executable SAT solver**
**Standard ML**



**Watched Literals Algorithm**

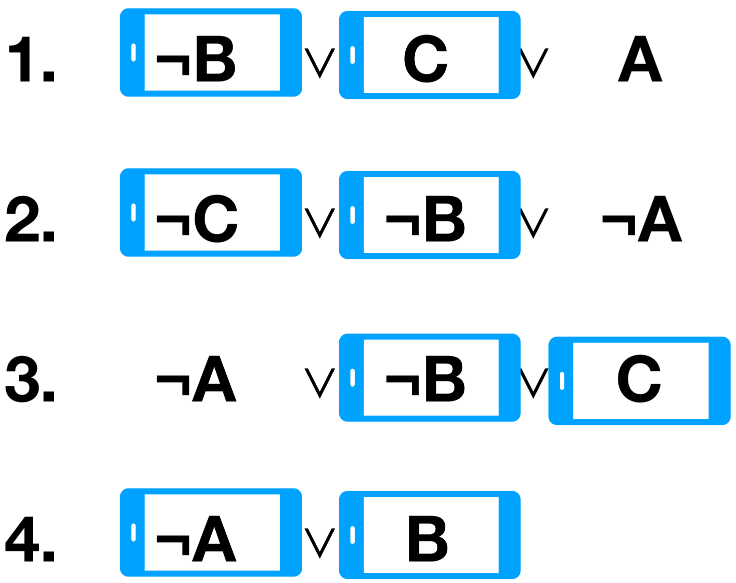
**Non-deterministic Program**

**Refined SAT Solver**

Towards efficient data structures

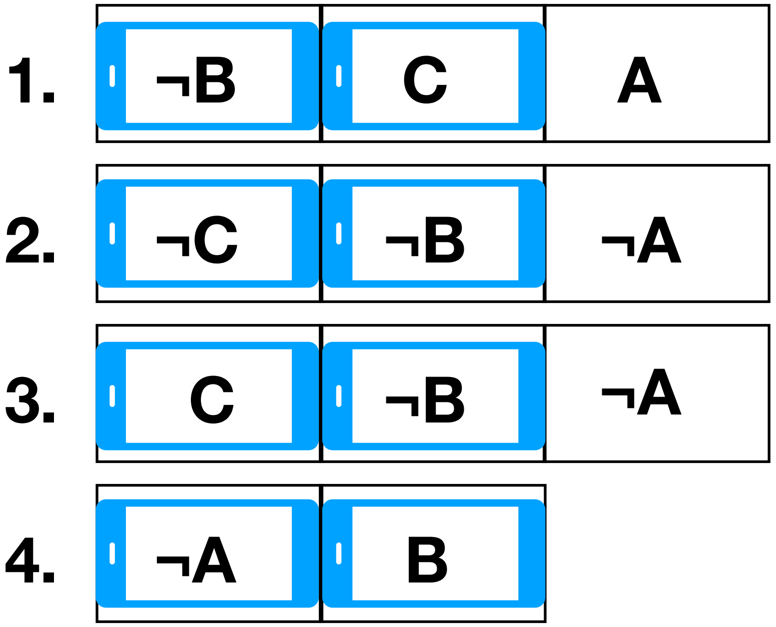
# DPLL with Watched Literals

**Clauses (multisets)**



1. ¬B ∨ C ∨ A
2. ¬C ∨ ¬B ∨ ¬A
3. ¬A ∨ ¬B ∨ C
4. ¬A ∨ B

**Clauses after refinement (lists)**



| | | | |
|---|---|---|---|
| 1. | ¬B | C | A |
| 2. | ¬C | ¬B | ¬A |
| 3. | C | ¬B | ¬A |
| 4. | ¬A | B | |

To update:

A: ¬A: 4 C: 1,3 ¬C: 2

B: 4 ¬B: 1,2,3

```
propagate_conflict_literal L S :=
  WHILE$_T$
    (λT. clauses_to_update T ≠ {})

    (λT. do {
       ASSERT(clauses_to_update T ≠ {})
       C ← SPEC (λC. C ∈ clauses_to_update T);
       U ← remove_from_clauses_to_update C T;
       update_clause L C U
     }
    )

    S
```

```
propagate_conflict_literal_list L S :=
  WHILE$_T$
    (λ(w, T). w < length (watched_by T L))

    (λ(w, T). do {
       C ← (watched_by T L) ! w;
       update_clause_list L C T
     }
    )

    (S, 0)
```

```
propagate_conflict_literal L S :=
  WHILE_T
    (λT. clauses_to_update T ≠ {})

    (λT. do {
       ASSERT(clauses_to_update T ≠ {})
       C ← SPEC (λC. C ∈ clauses_to_update T);
       U ← remove_from_clauses_to_update C T;
       update_clause L C U
     }
    )

    S
```

```
propagate_conflict_literal_list L S :=
  WHILE_T
    (λ(w, T). w < length (watched_by T L))

    (λ(w, T). do {
       C ← (watched_by T L) ! w;
       update_clause_list L C T
     }
    )

    (S, 0)
```

```
propagate_conflict_literal_list L S ≤ ⇓ conversion_between_states
   (propagate_conflict_literal L T)
```

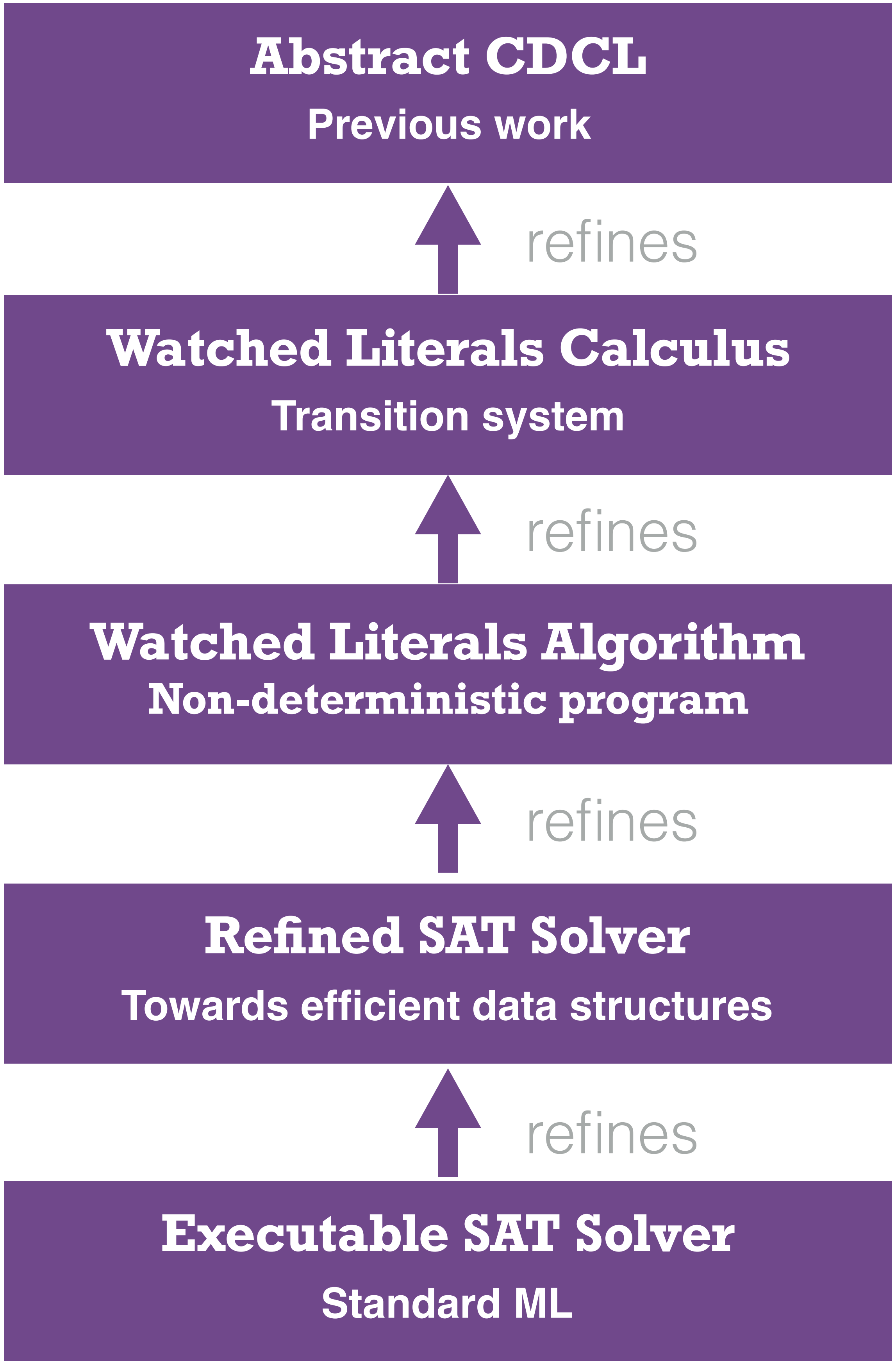
in Isabelle

- 🔑 More new invariants
- ✗ Aligning goals is hard...
- ✗ Fast code uses many invariants
- ✗ Forgotten and new invariants
- ✓ sledgehammer

- Choice on the heuristics
- Choice on the data structures
- Prepare code synthesis

# Decision heuristic

- Variable-move-to-front heuristic
- No correctness w.r.t. a standard implementation
- Behaves correctly:
  - returns an unset literal if there is one
  - no exception (out-of-bound array accesses)



**Abstract CDCL**
Previous work

↑ refines

**Watched Literals Calculus**
Transition system

↑ refines

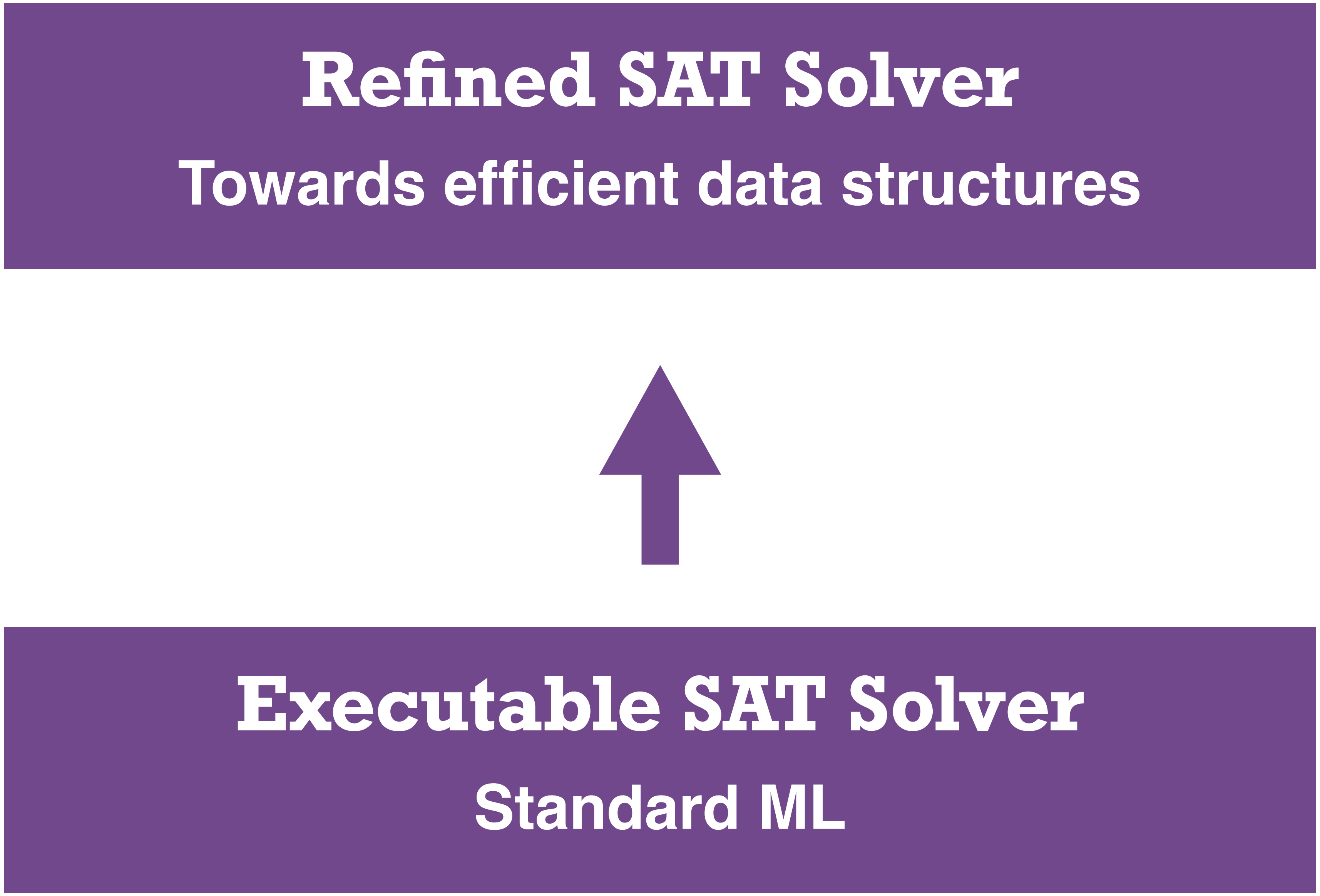
**Watched Literals Algorithm**
**Non-deterministic program**

↑ refines

**Refined SAT Solver**
Towards efficient data structures

↑ refines

**Executable SAT Solver**
Standard ML



**Refined SAT Solver**

Towards efficient data structures

**Executable SAT Solver**

Standard ML

```
sepref_definition executable_version
  is ‹propagate_conflict_literal_heuristics›
  :: ‹unat_lit_assnᵏ *ₐ state_assnᵈ →ₐ state_assn›
  by sepref
```

Synthesise imperative code and a refinement relation

```
sepref_definition executable_version
  is ‹propagate_conflict_literal_heuristics›
  :: ‹unat_lit_assn^k *_a state_assn^d →_a state_assn›
  by sepref
```

Synthesise imperative code and a refinement relation

```
main_loop S :=
   heap_WHILET
       (λ(finished, _). return (¬ finished))
       (λ(_, state).
           propagate state ≫=
           analyse_or_decide)
       (False, state) ≫=
    (λ(_, final_state). return final_state)
```

```
sepref_definition executable_version
  is ‹propagate_conflict_literal_heuristics›
  :: ‹unat_lit_assnᵏ *ₐ state_assnᵈ →ₐ state_assn›
  by sepref
```

Synthesise imperative code and a refinement relation

```
fun main_loop state =
 fn () =>
  let
   val (_, final_state) =
    heap_WHILET
      (fn (done, _) => (fn () => not done))
      (fn (_, state) =>
        (analyse_or_decide (propagate state ()) ()))
      (false, xi)
      ();
  in final_state end;
```

```
sepref_definition executable_version
  is ‹propagate_conflict_literal_heuristics›
  :: ‹unat_lit_assnᵏ *ₐ state_assnᵈ →ₐ state_assn›
  by sepref
```

Synthesise imperative code and a refinement relation

```
fun cdcl_twl_stgy_prog_wl_D_code x =
  (fn xi => fn () =>
    let
      val a =
        heap_WHILET (fn (a1, _) => (fn () => (not a1)))
          (fn (_, a2) =>
            (fn f_ => fn () => f_ ((unit_propagation_outer_loop_wl_D a2) ()) ())
              cdcl_twl_o_prog_wl_D_code)
          (false, xi) ();
    in
      let
        val (_, aa) = a;
      in
        (fn () => aa)
      end
        ()
```

```
in final_state end;
```

# 🔑 Choice on the data structures

Clauses: resizable arrays of (fixed sized) arrays

However, no aliasing

- Indices instead of pointers
- N[C] makes a copy, so only use N[C][i]

✓ Generates imperative code

✗ No error messages

✗ Transformations before generating code

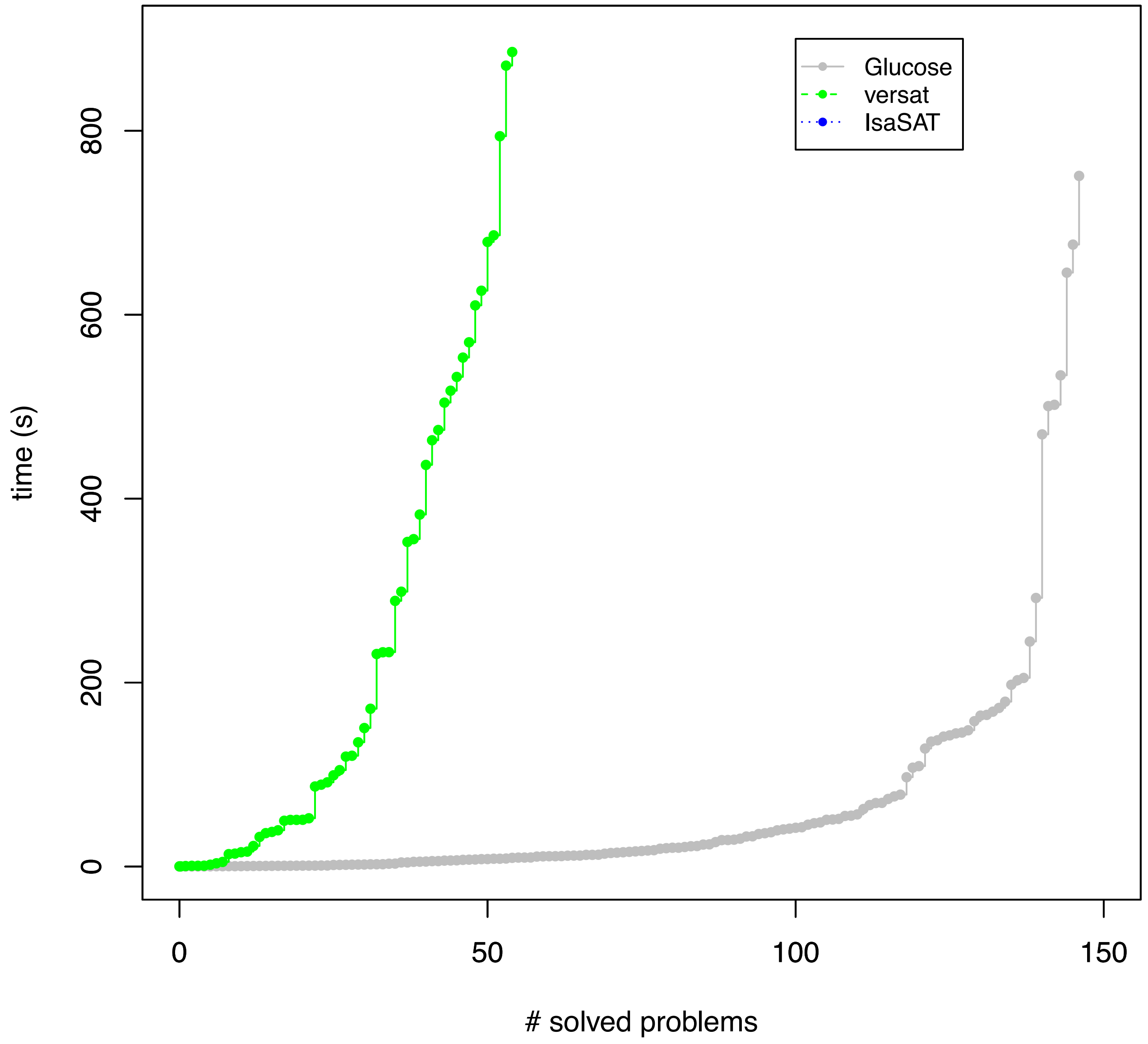
Clauses of length 0 and 1

Once combined with an initialisation:

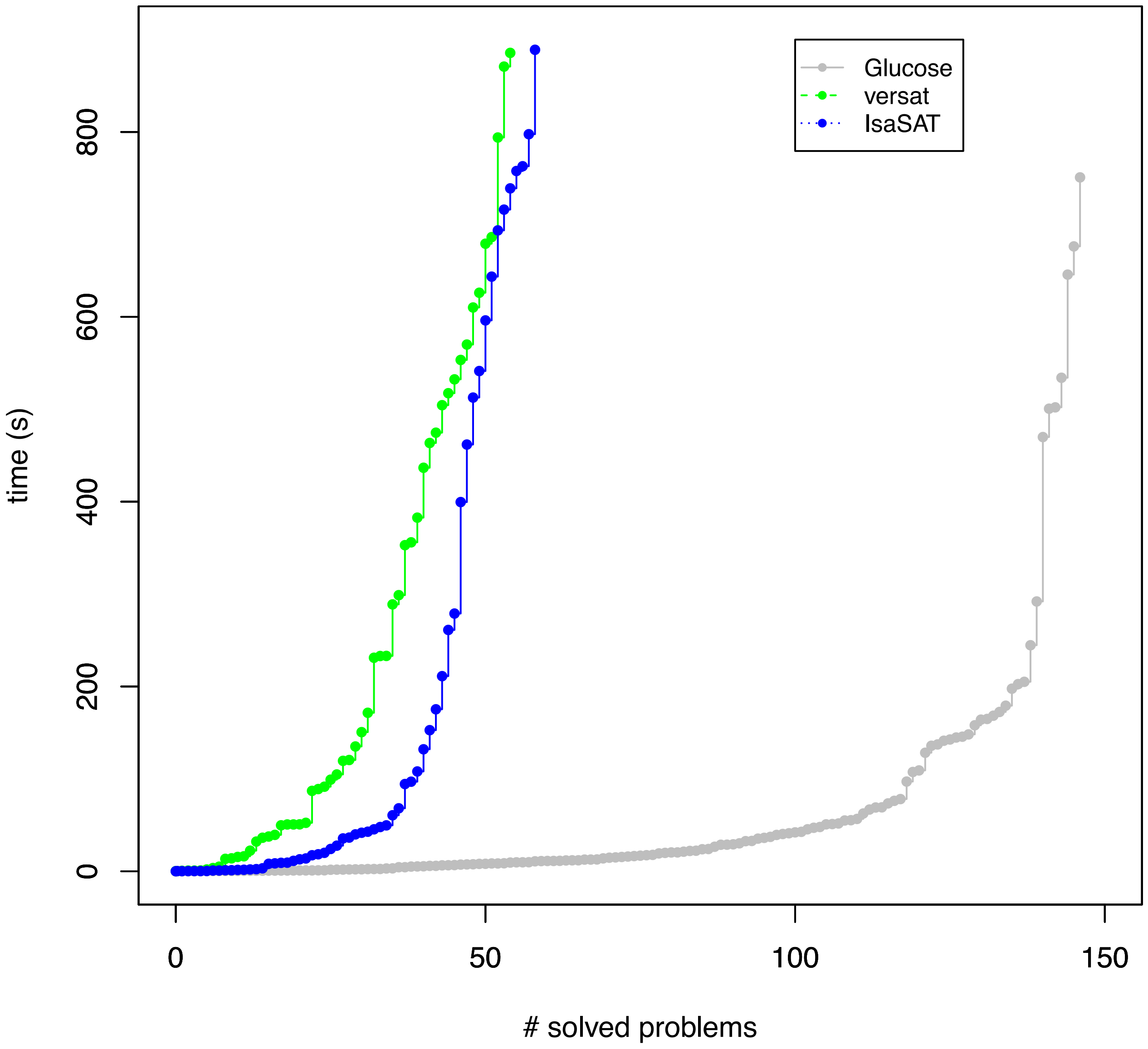
in Isabelle

```
‹(IsaSAT_code, model_if_satisfiable)
    ∈ [λN. each_clause_is_distinct N ∧
        literals_fit_in_32_bit_integer N]ₐ
      clauses_as_listsᵏ → model›
```

Exported code tested with an unchecked parser
(easy and medium problems from the SAT competition 2009)



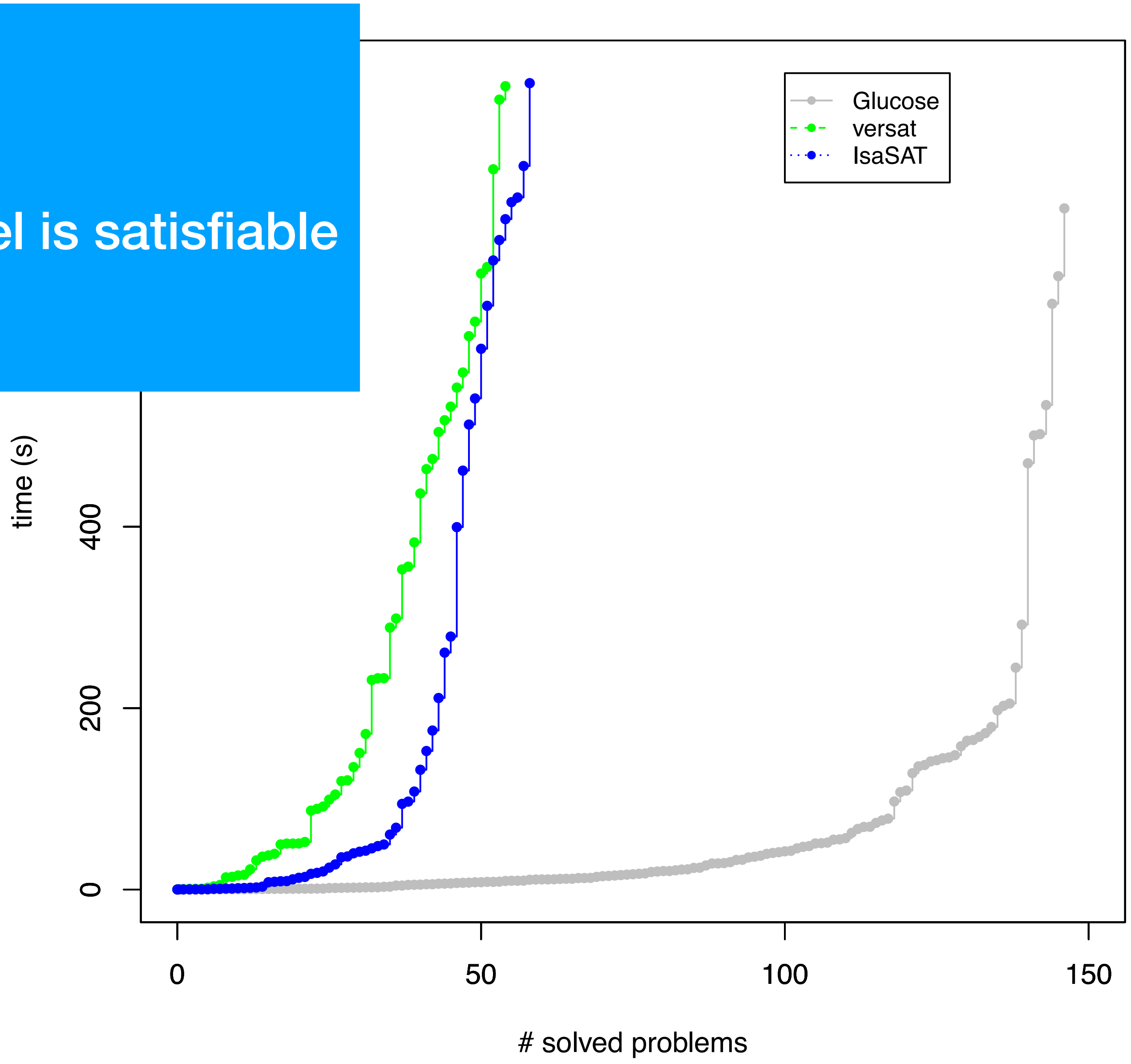
**Performance of the first executable version**



Performance of IsaSAT

Correct up to:

- run-time checks
- checking the model is satisfiable



**Performance of IsaSAT**

**Abstract CDCL**

Previous work

↑ refines

**Watched Literals Calculus**

Transition system

↑ refines

**Watched Literals Algorithm**

**Non-deterministic program**

↑ refines

**Refined SAT Solver**

Towards efficient data structures

↑ refines

**Executable SAT solver**

**Standard ML**

**Abstract CDCL**
Previous work

↑ refines

**Watched Literals Calculus**
Transition system

↑ refines

**Watched Literals Algorithm**
**Non-deterministic program**

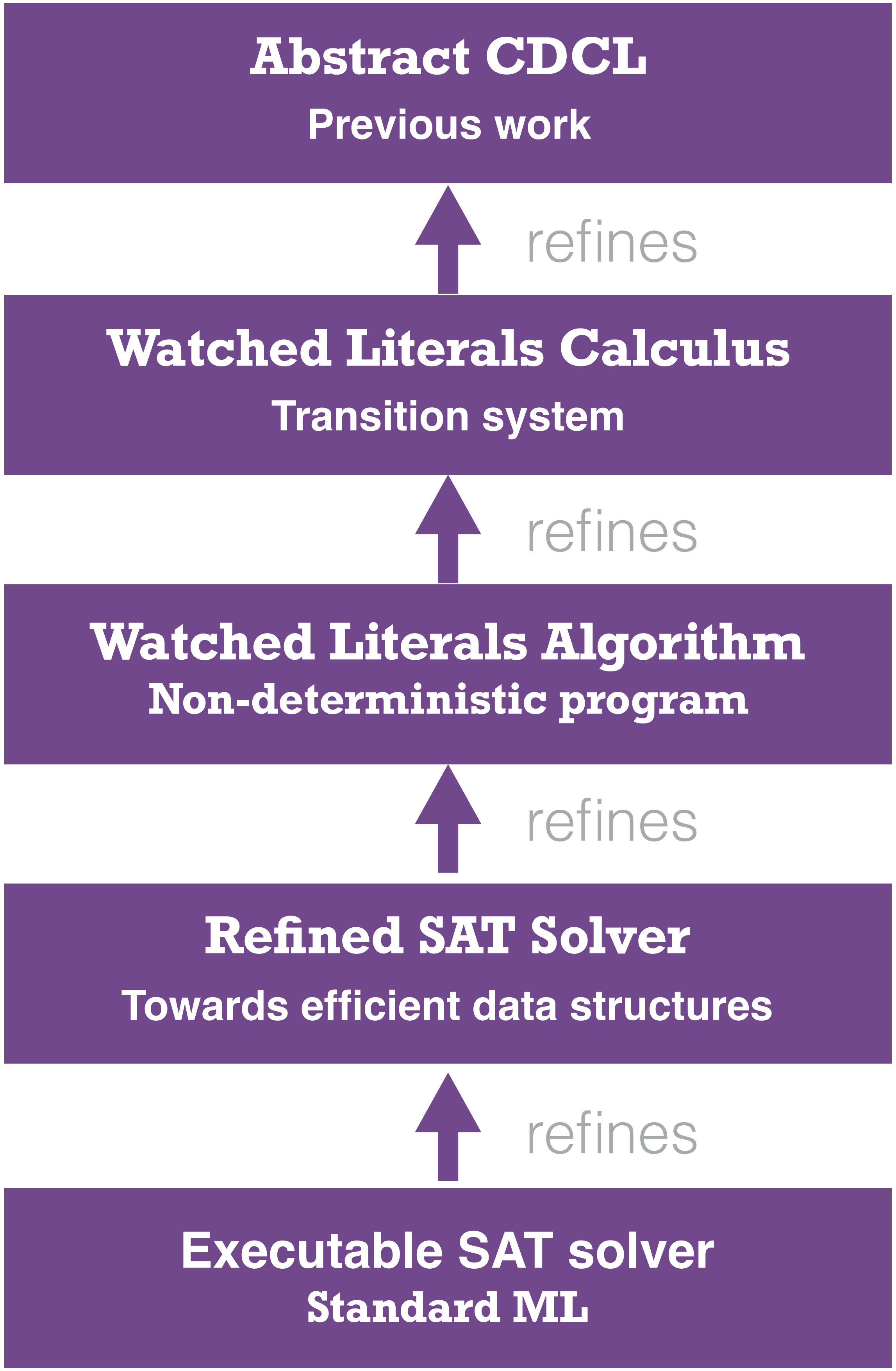
↑ refines

**Refined SAT Solver**
Towards efficient data structures

↑ refines

**Executable SAT solver**
**Standard ML**

- better implementation (trail, conflict)
- dynamic decision heuristic



**Abstract CDCL**
Previous work

refines

**Watched Literals Calculus**
Transition system

refines

**Watched Literals Algorithm**
**Non-deterministic program**

refines

**Refined SAT Solver**
Towards efficient data structures

refines

**Executable SAT solver**
**Standard ML**

- allow learned clause minimisation

- better implementation (trail, conflict)
- dynamic decision heuristic

- learned clause minimisation

**Abstract CDCL**
Previous work

- allow learned clause minimisation

refines

**Watched Literals Calculus**
Transition system

- more invariants

refines

**Watched Literals Algorithm**
**Non-deterministic program**

refines

**Refined SAT Solver**
Towards efficient data structures

- better implementation (trail, conflict)
- dynamic decision heuristic

- learned clause minimisation

refines

**Executable SAT solver**
**Standard ML**

# How hard is it?

| | Paper | Proof assistant |
|---|---|---|
| **Very abstract** | 13 pages | 50 pages |
| **Abstract CDCL** | 9 pages<br>(½ month) | 90 pages<br>(5 months) |
| **Watched Literals** | 1 page<br>(C++ code of MiniSat) | 600 pages<br>(15 months) |

VirginiaTech
Invent the Future

VU VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik

# Conclusion

## Concrete outcome

- Watched literals optimisation
- Verified executable SAT solver

## Methodology

- Refinement using the Refinement Framework

## Future work

- Restarts
- Use SAT solver in IsaFoR
- SAT Modulo Theories (e.g., CVC or z3)

# Annex

```
for (i = j = 1; i < out_learnt.size(); i++)
  if (reason(var(out_learnt[i])) == CRef_Undef ll
     !litRedundant(out_learnt[i]))
     out_learnt[j++] = out_learnt[i];
```

```
fun minimize_and_extract_highest_lookup_conflict_code x =
 (fn ai => fn bid => fn bic => fn bib => fn bia => fn bi => fn () =>
  let
   val a =
    heap_WHILET
     (fn (_, (a1a, (_, a2b))) =>
      (fn f_ => fn () => f_ ((length_arl_u_code heap_uint32 a2b) ()) ())
       (fn x_a => (fn () => (Word32.< (a1a, x_a)))))
     (fn (a1, (a1a, (a1b, a2b))) =>
      (fn f_ => fn () => f_
       (((fn () => Array.sub (fst a2b, Word32.toInt a1a))) ()) ())
       (fn x_a =>
        (fn f_ => fn () => f_
         ((literal_redundant_wl_lookup_code ai bid a1 a1b x_a bia) ())
         ())
         (fn (a1c, (_, a2d)) =>
          (if not a2d
           then (fn () =>
              (a1, (Word32.+ (a1a, (Word32.fromInt 1)),
                 (a1c, a2b))))
           else (fn f_ => fn () => f_
              ((delete_from_lookup_conflict_code x_a a1) ()) ())
              (fn x_e =>
               (fn f_ => fn () => f_ ((arl_last heap_uint32 a2b)
                ()) ())
                (fn xa =>
                 (fn f_ => fn () => f_
                  ((arl_set_u heap_uint32 a2b a1a xa) ()) ())
                  (fn xb =>
                   (fn f_ => fn () => f_
((arl_butlast heap_uint32 xb) ()) ())
(fn xc => (fn () => (x_e, (a1a, (a1c, xc))))))))))))))
     (bic, ((Word32.fromInt 1), (bib, bi))) ();
  in
   let
    val (a1, (_, (a1b, a2b))) = a;
```

# What is in IsaSAT?

## Conflict Analysis

- conflict as lookup table (Minisat)
- and as explicit array (Minisat’s “outl”, to simplify proofs)

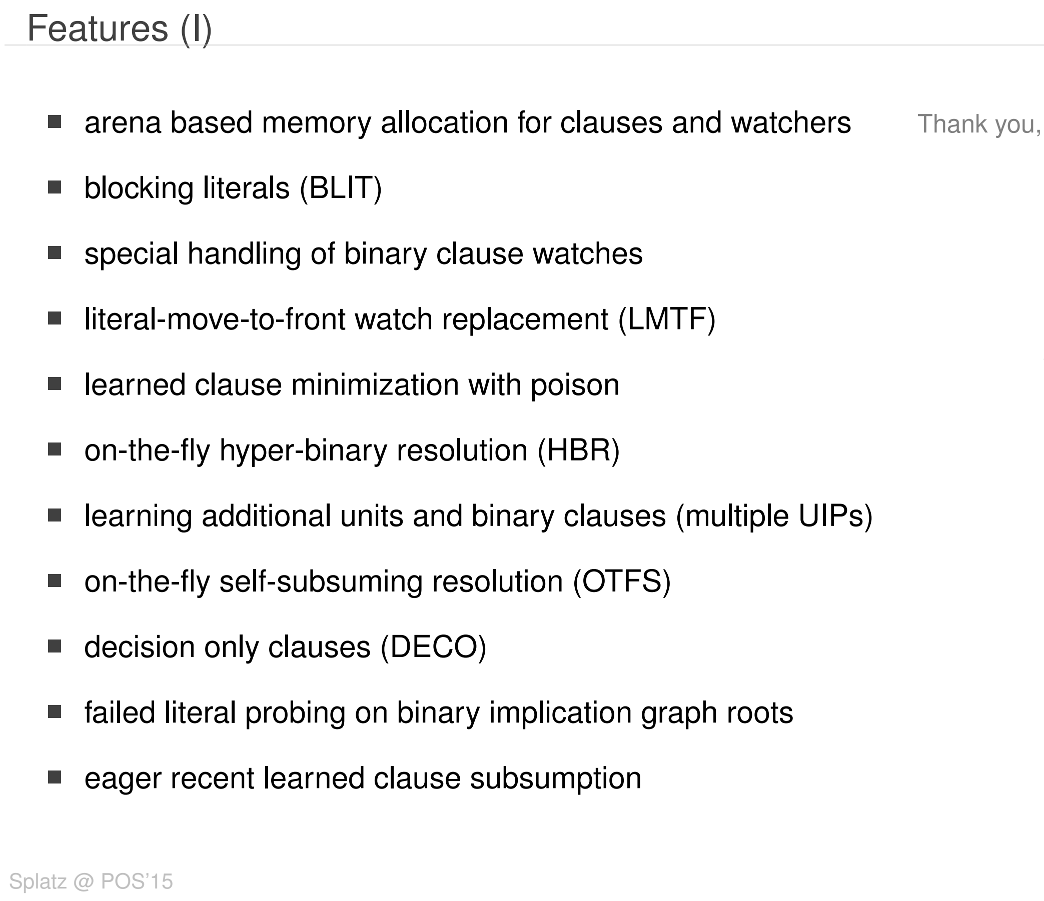
## Decisions

- Variable move to front (Splatz, cadical)

## Propagations

- Mostly following MiniSAT (without BLIT)

# How much is missing?



## Features (I)

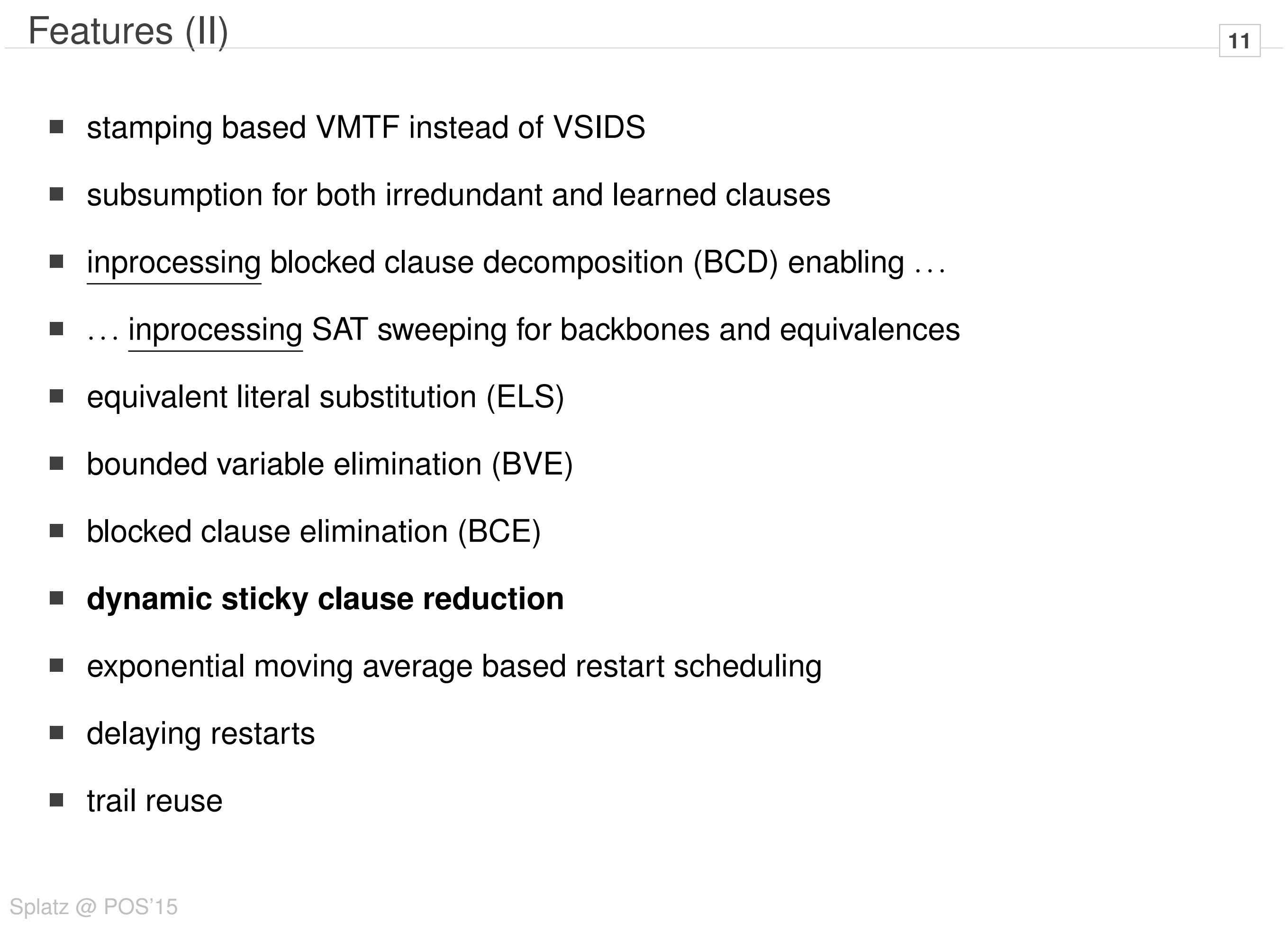
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Thank you, Norbert & Mate!

Splatz @ POS'15

**Slides by Armin Biere**



## Features (II)

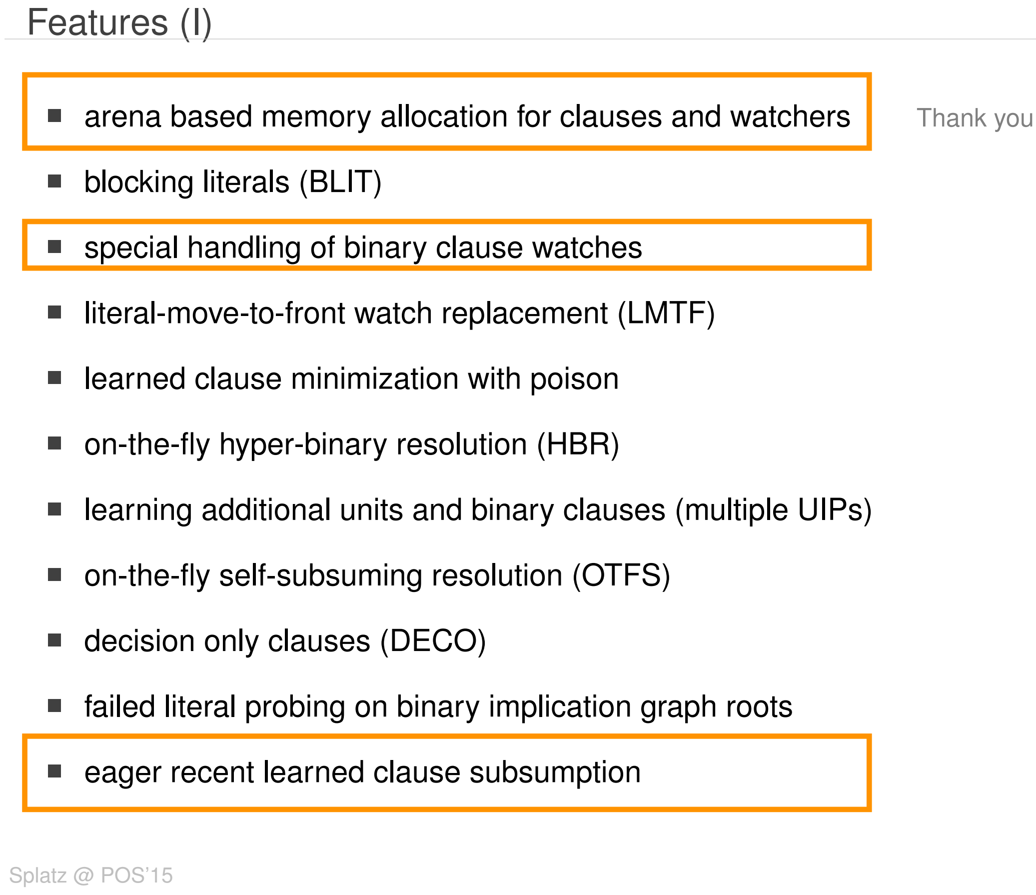
11

- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling ...
- ... inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

# How much is missing?

**Slides by Armin Biere**



Features (I)

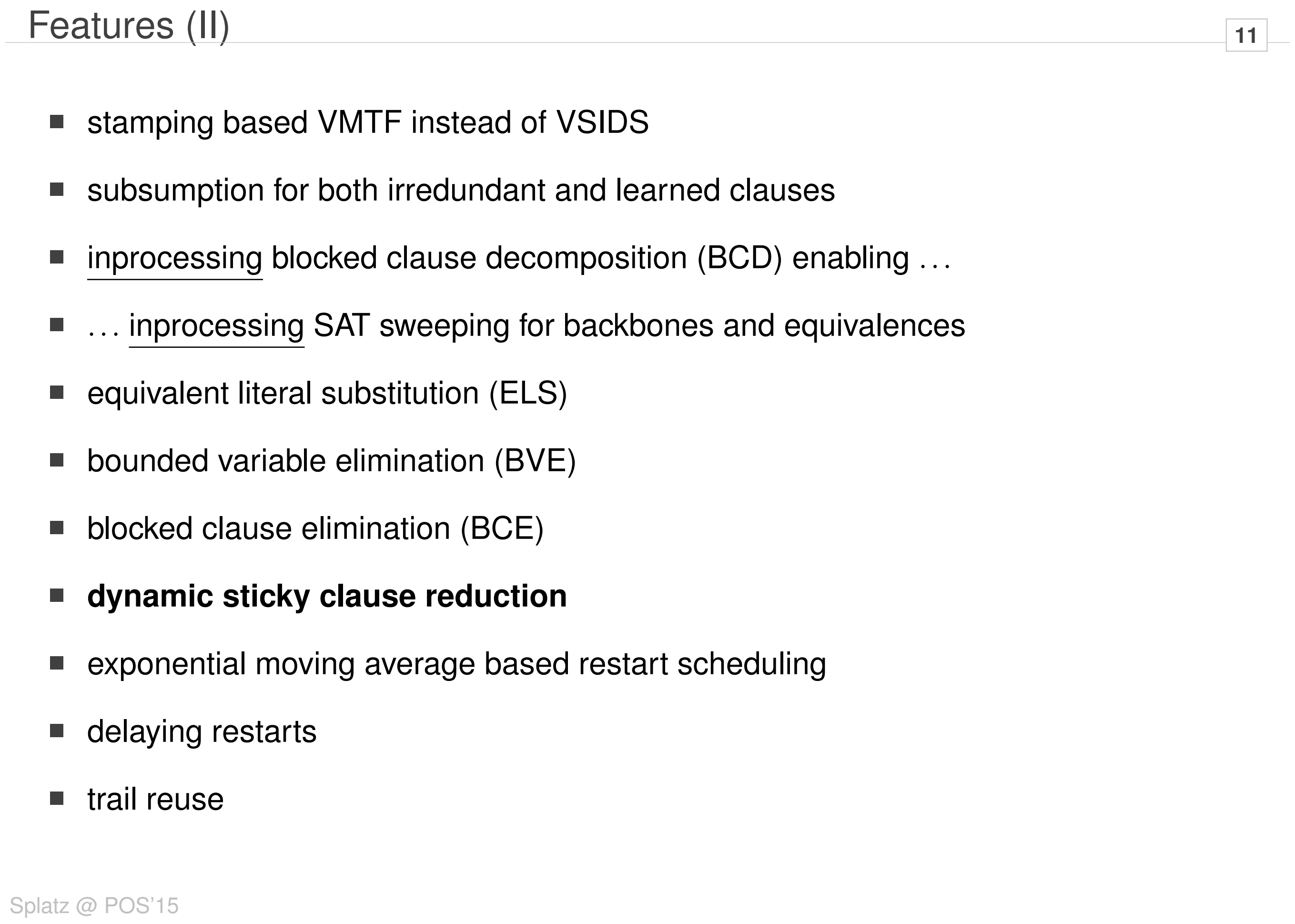
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Thank you, Norbert & Mate!

Splatz @ POS'15

*Code only*



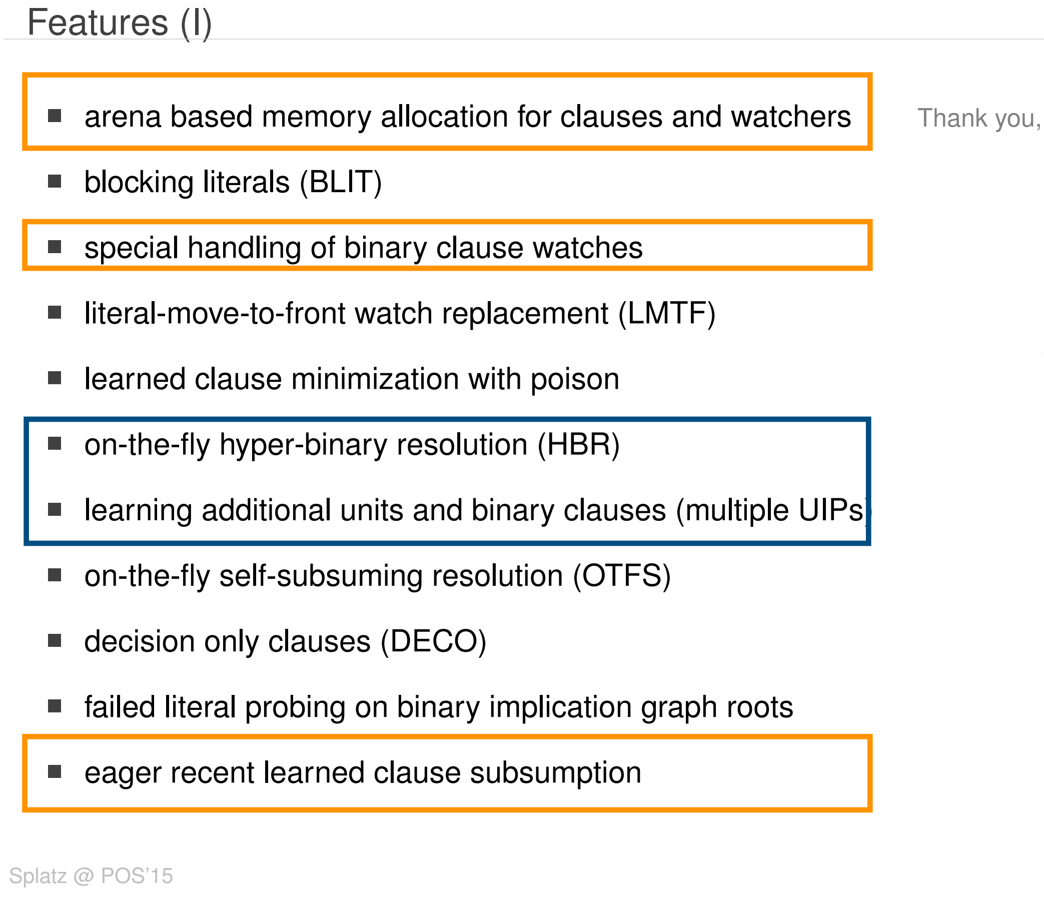
Features (II)

11

- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling . . .
- . . . inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

# How much is missing?



Features (I)

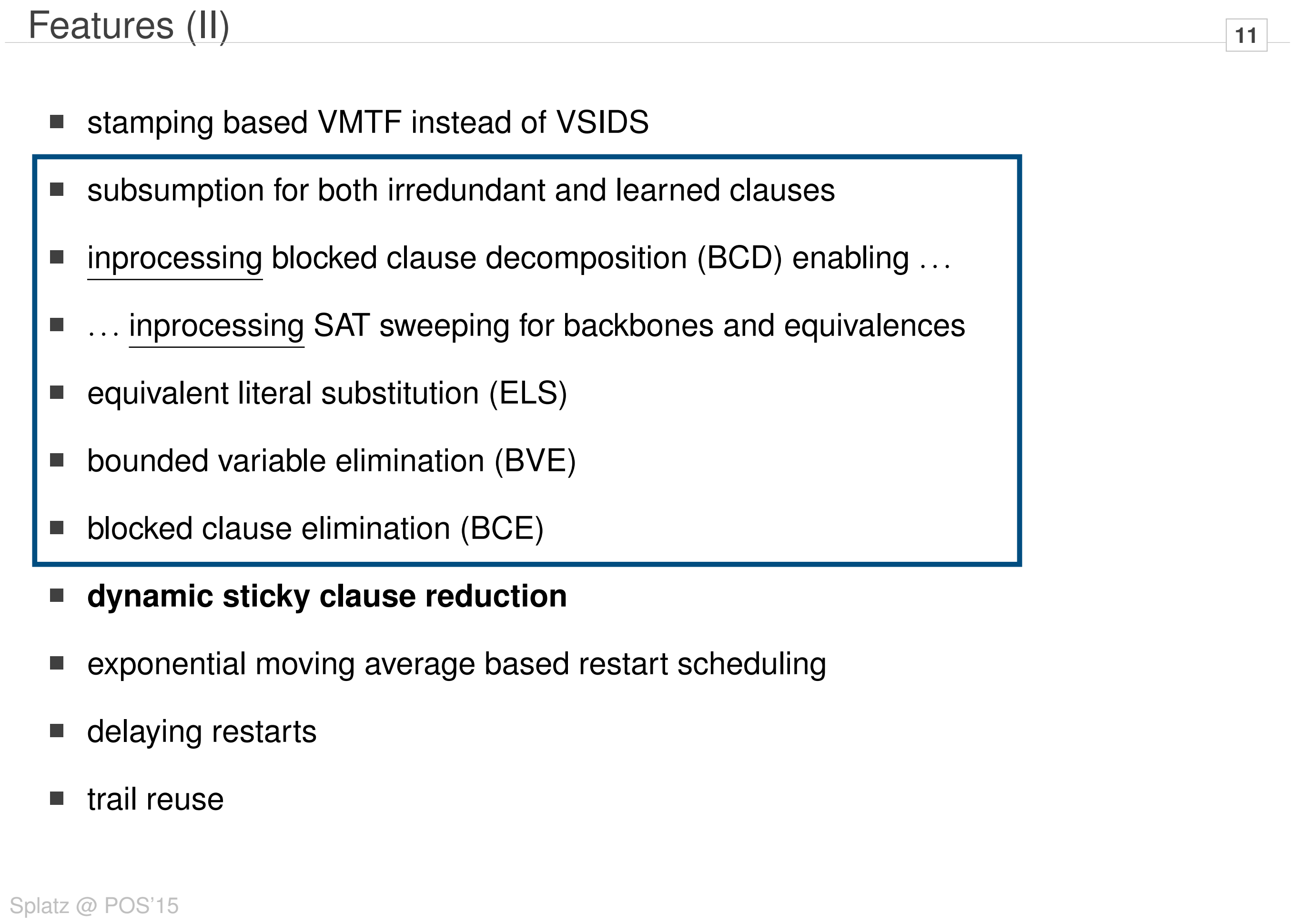
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Splatz @ POS'15

Thank you, Norbert & Mate!

**Slides by Armin Biere**



Features (II)

11

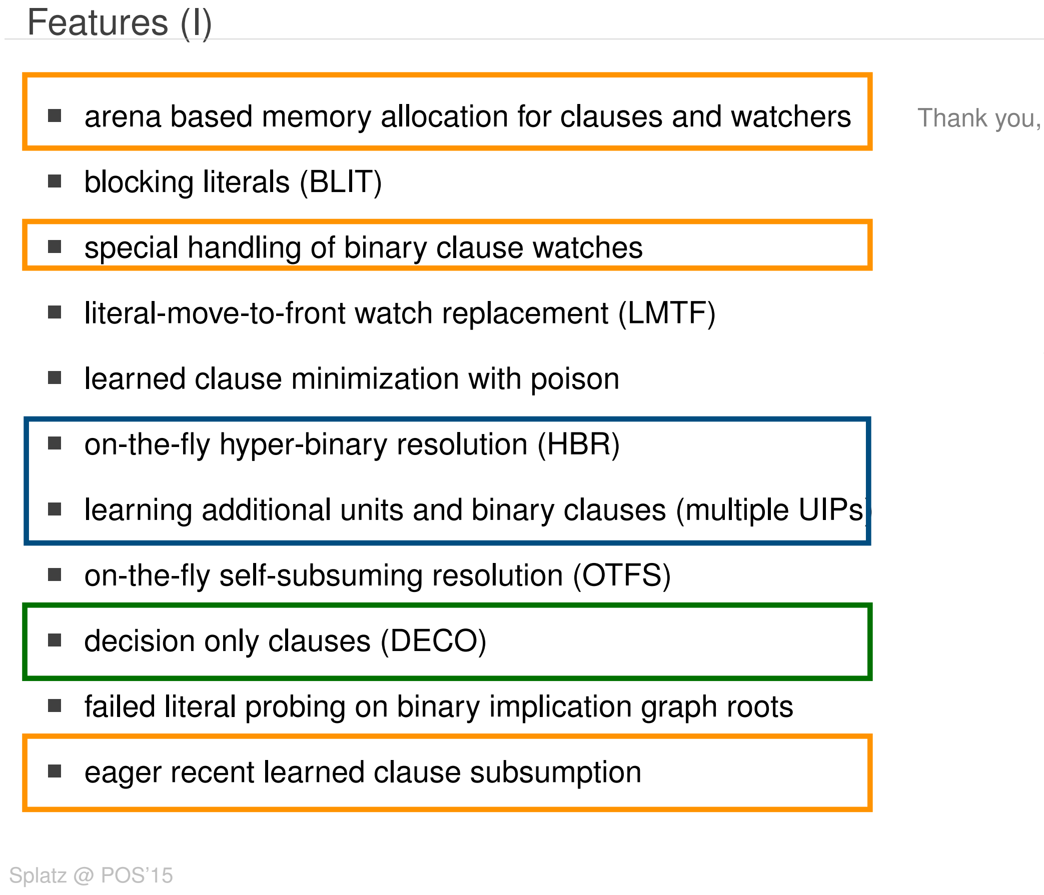
- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling . . .
- . . . inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

*Code only*

*Strengthening*

# How much is missing?



## Features (I)

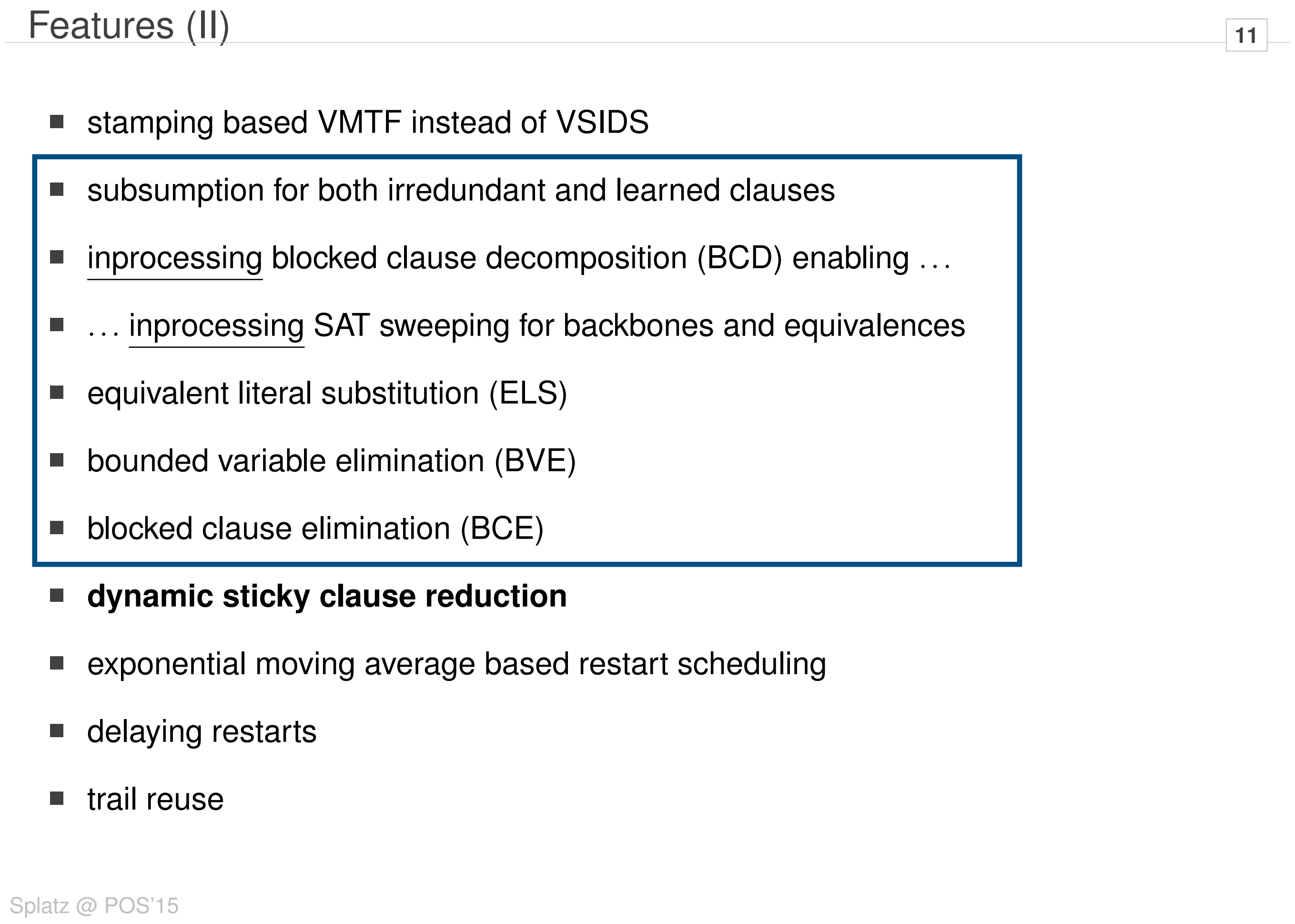
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Splatz @ POS'15

Thank you, Norbert & Mate!

**Slides by Armin Biere**



## Features (II)

11

- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling . . .
- . . . inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

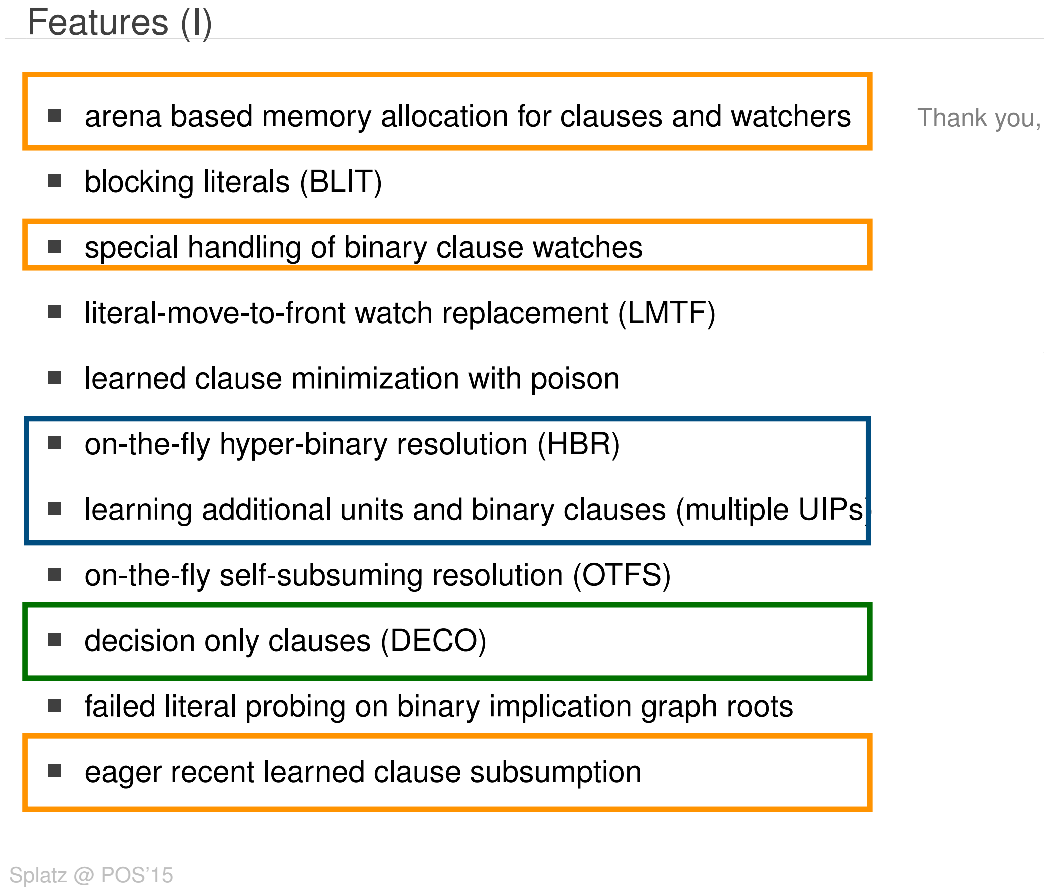
Splatz @ POS'15

*Code only*

*Strengthening*

*Change CDCL*

# How much is missing?



Features (I)

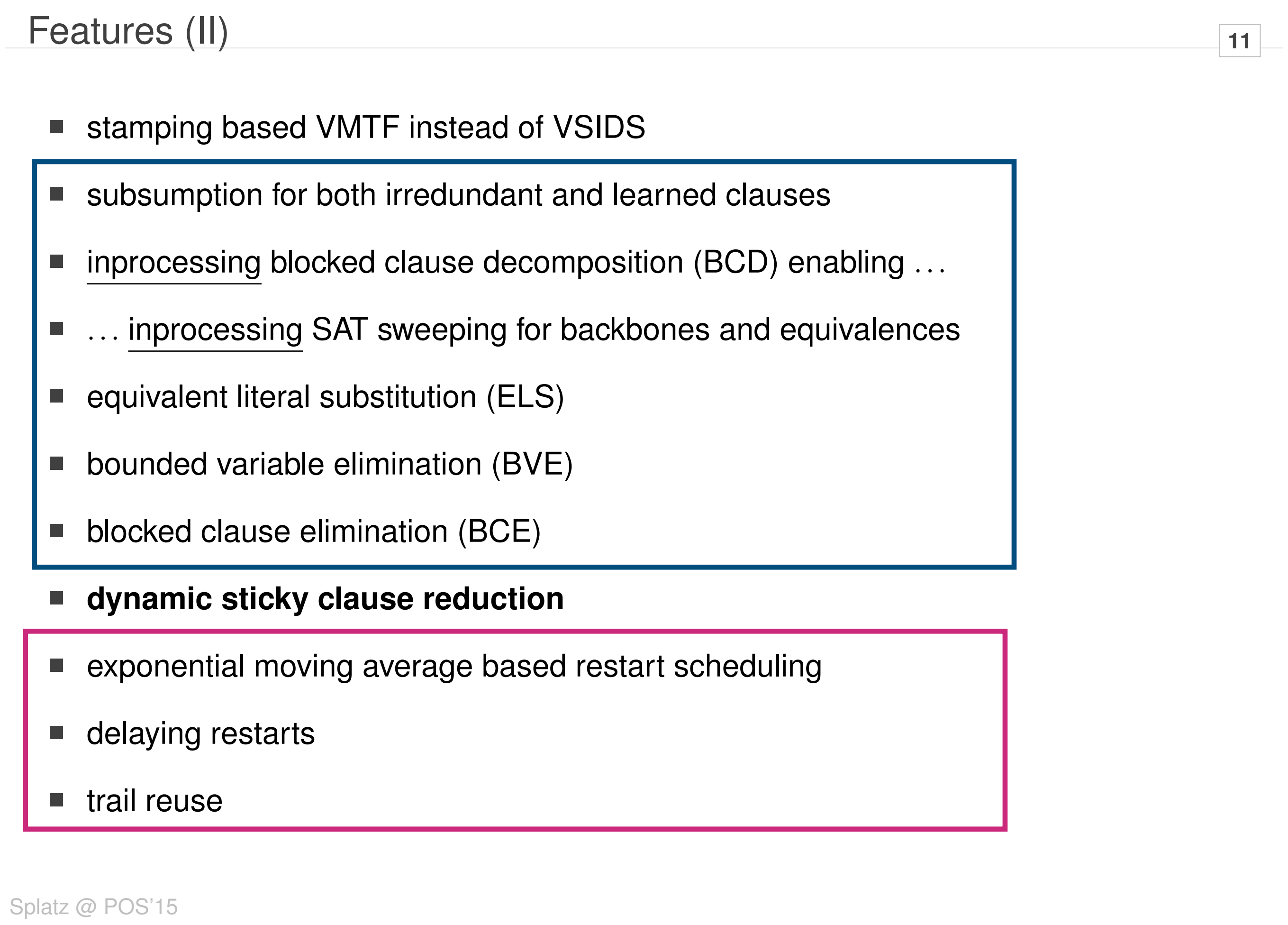
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Thank you, Norbert & Mate!

Splatz @ POS'15

**Slides by Armin Biere**



Features (II)

11

- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling . . .
- . . . inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

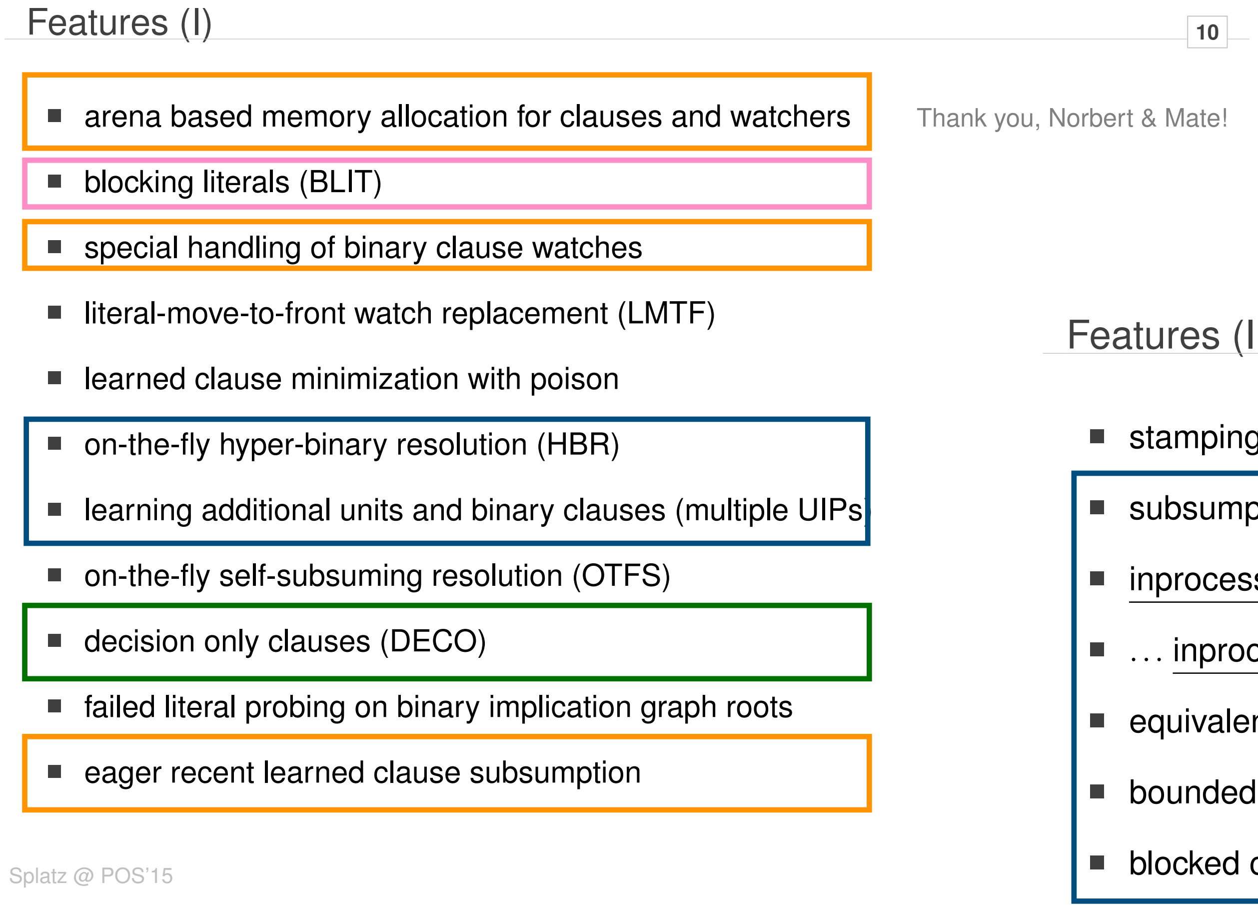
*Code only*

*Restarts (future)*

*Strengthening*

*Change CDCL*

# How much is missing?



Features (I)

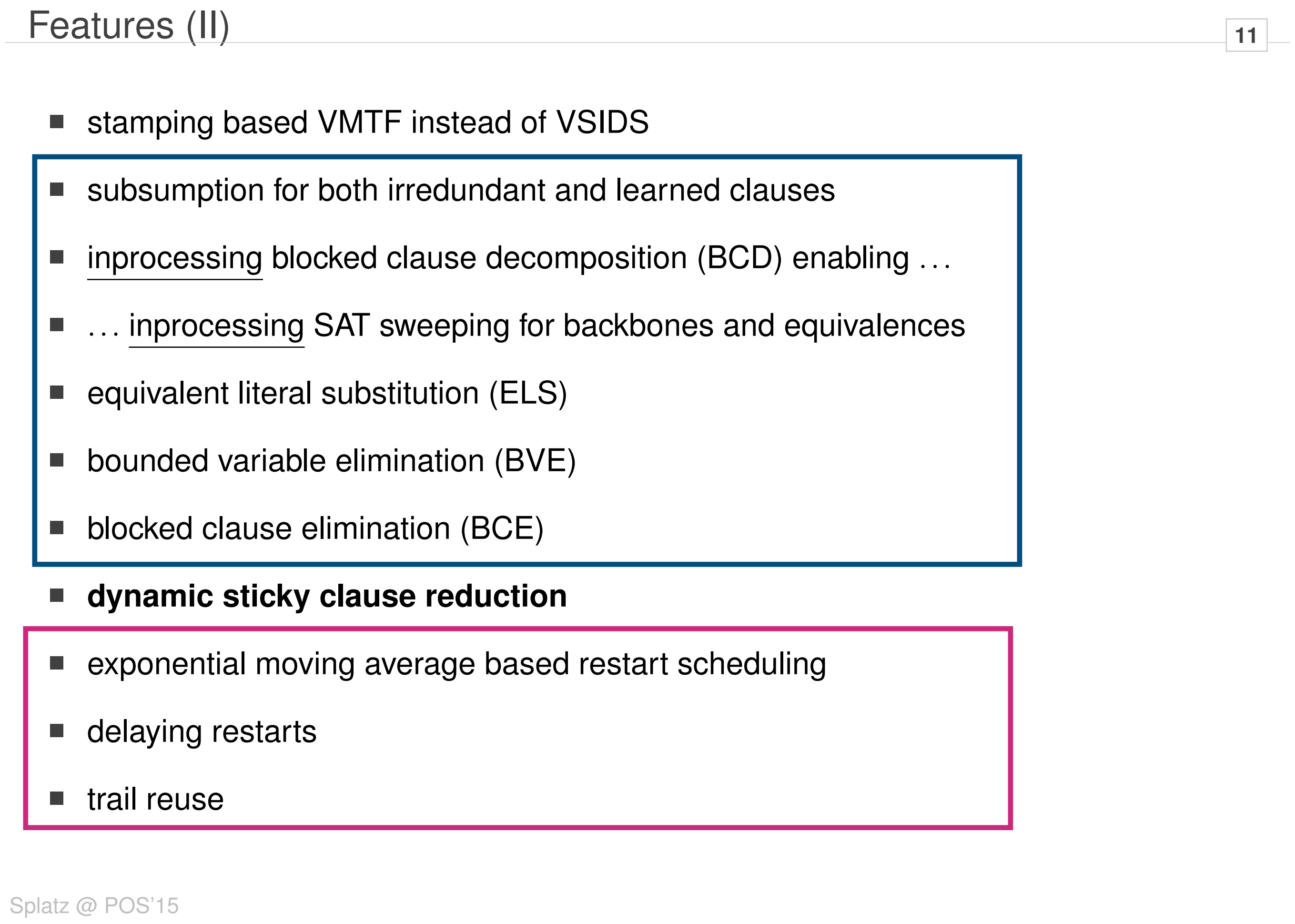
10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Thank you, Norbert & Mate!

Splatz @ POS'15

**Slides by Armin Biere**



Features (II)

11

- stamping based VMTF instead of VSIDS
- subsumption for both irredundant and learned clauses
- inprocessing blocked clause decomposition (BCD) enabling . . .
- . . . inprocessing SAT sweeping for backbones and equivalences
- equivalent literal substitution (ELS)
- bounded variable elimination (BVE)
- blocked clause elimination (BCE)
- **dynamic sticky clause reduction**
- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

*Code only*

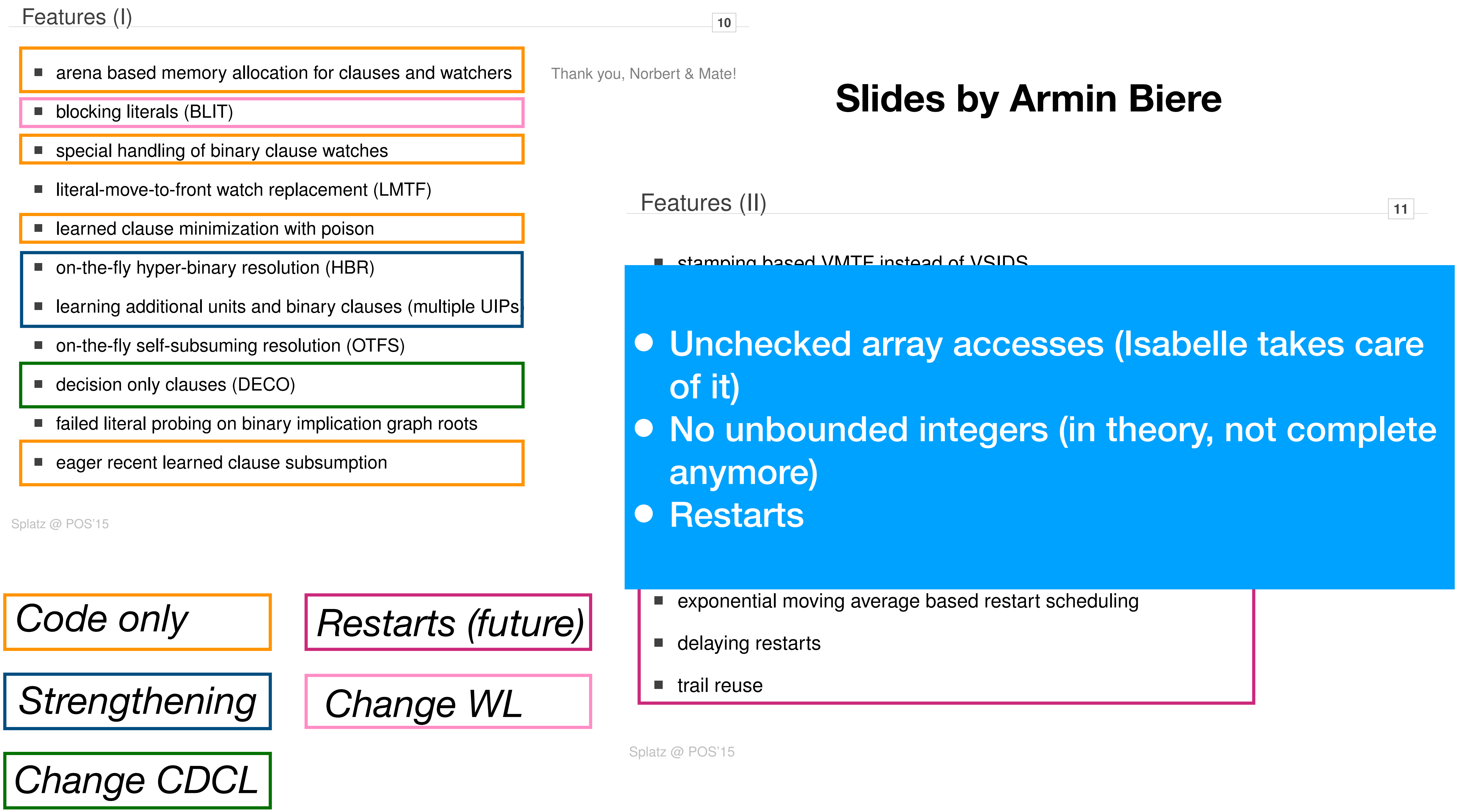
*Restarts (future)*

*Strengthening*

*Change WL*

*Change CDCL*

# How much is missing?



Features (I)

10

- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary clauses (multiple UIPs)
- on-the-fly self-subsuming resolution (OTFS)
- decision only clauses (DECO)
- failed literal probing on binary implication graph roots
- eager recent learned clause subsumption

Splatz @ POS'15

Thank you, Norbert & Mate!

**Slides by Armin Biere**

Features (II)

11

- stamping based VMTF instead of VSIDS

- Unchecked array accesses (Isabelle takes care of it)
- No unbounded integers (in theory, not complete anymore)
- Restarts

- exponential moving average based restart scheduling
- delaying restarts
- trail reuse

Splatz @ POS'15

*Code only*

*Restarts (future)*

*Strengthening*

*Change WL*

*Change CDCL*

# Update Strategy

A first idea

**M**

**Clauses N**

A

B

C

A better strategy

**CBAM**

**Clauses N**

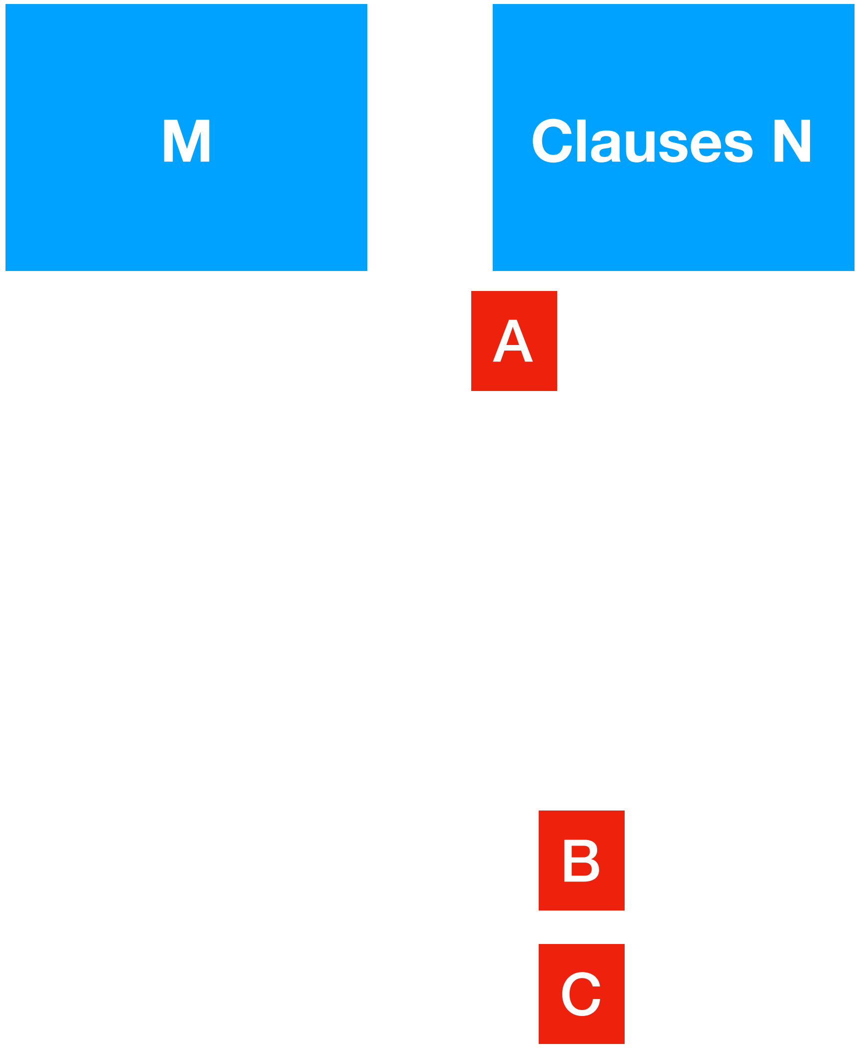
A

B

C

# Update Strategy

## A first idea



**M**

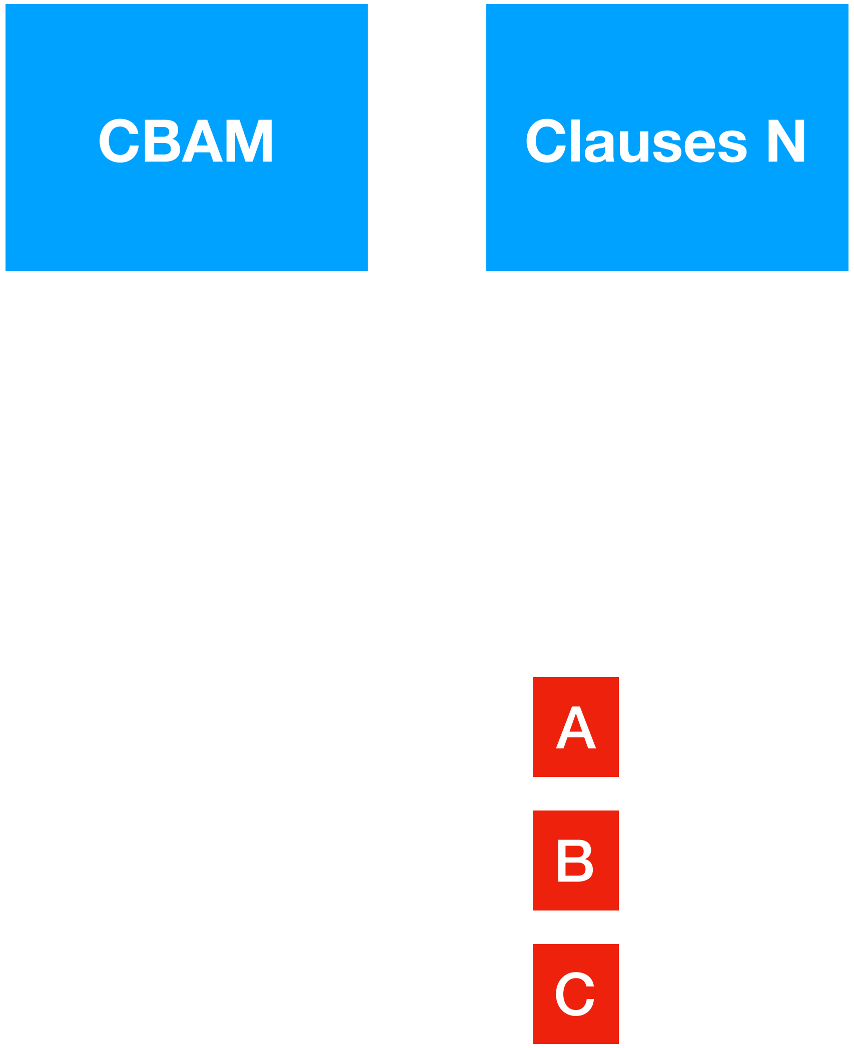
**Clauses N**

A

B

C

## A better strategy



**CBAM**

**Clauses N**

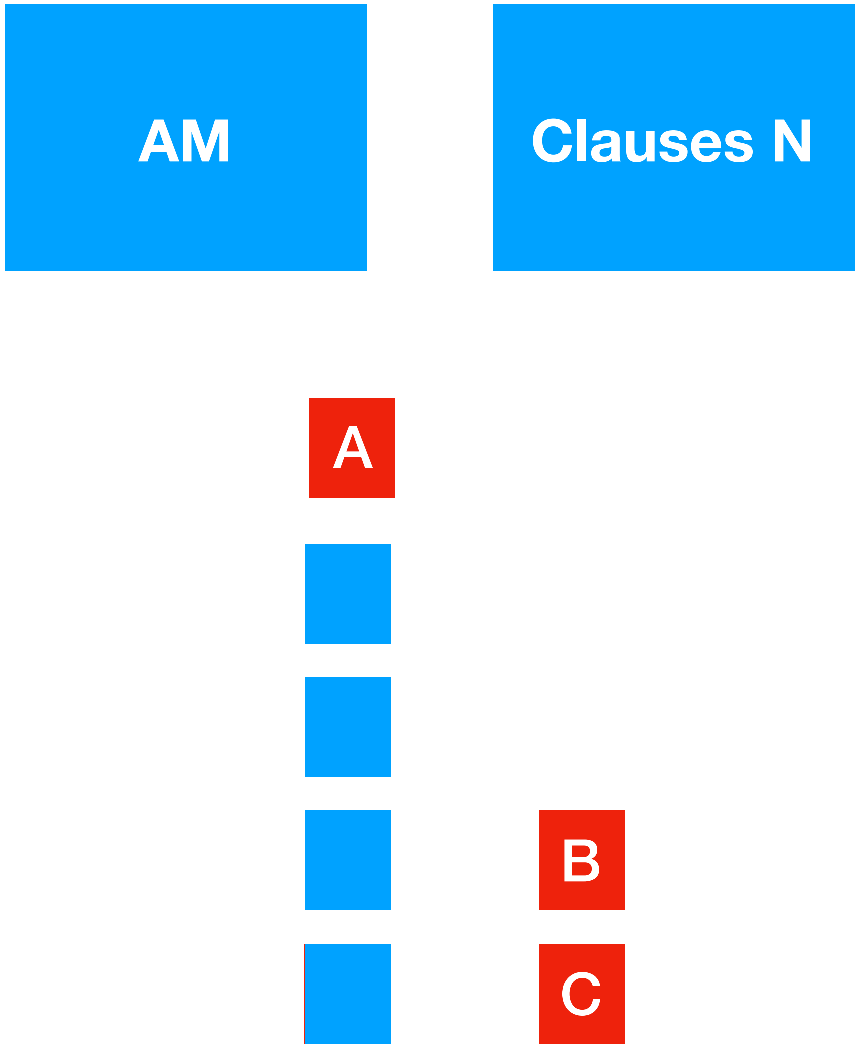
A

B

C

# Update Strategy

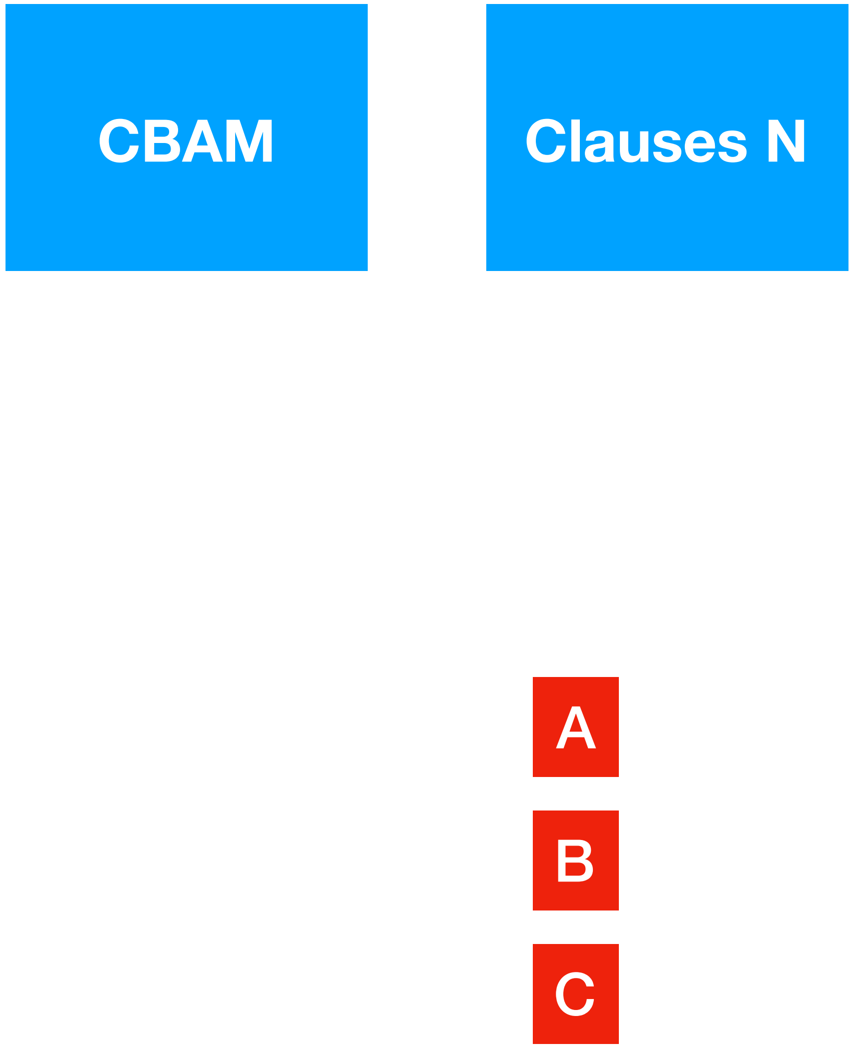
## A first idea



**AM**

**Clauses N**

## A better strategy



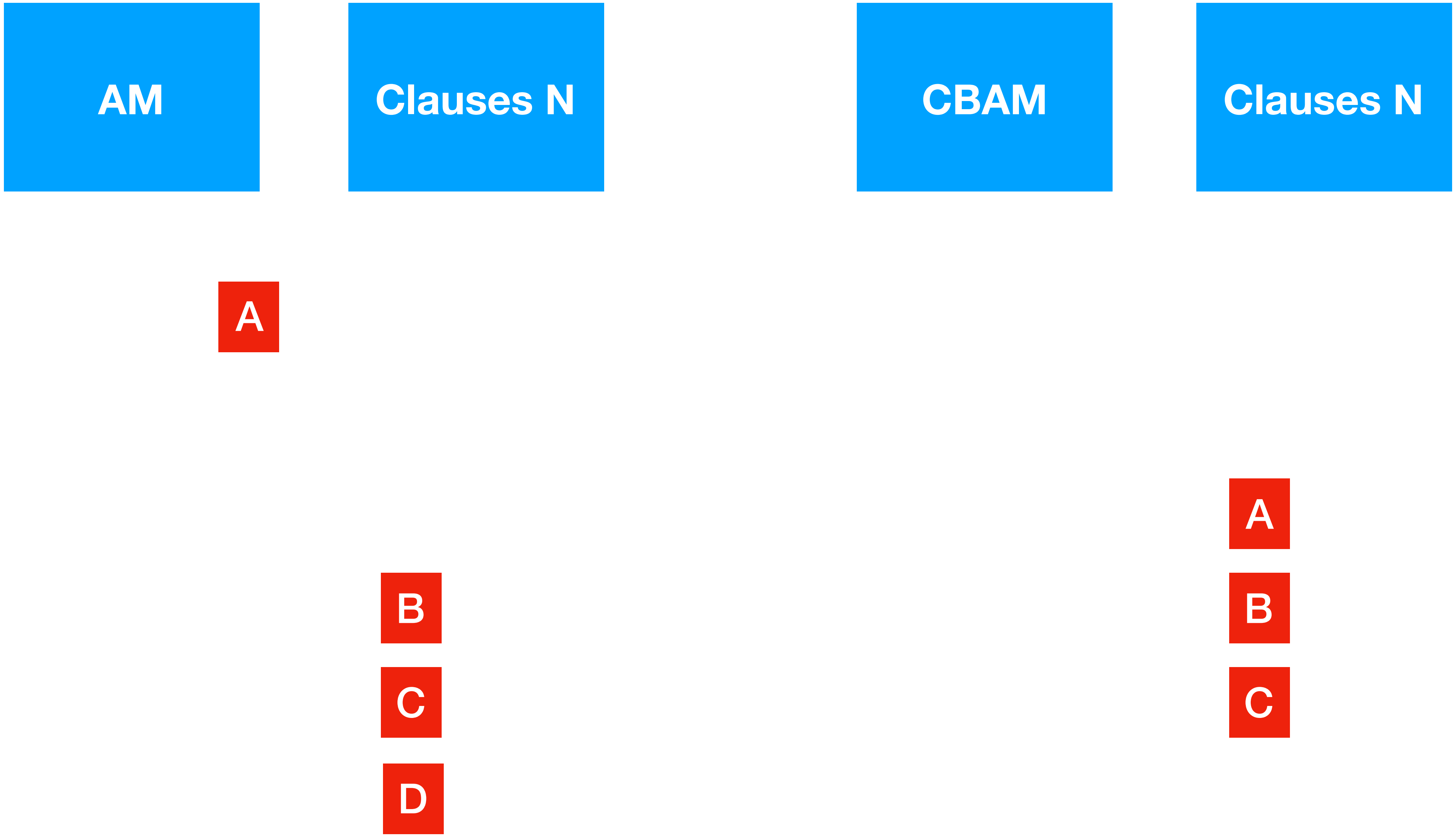
**CBAM**

**Clauses N**

# Update Strategy

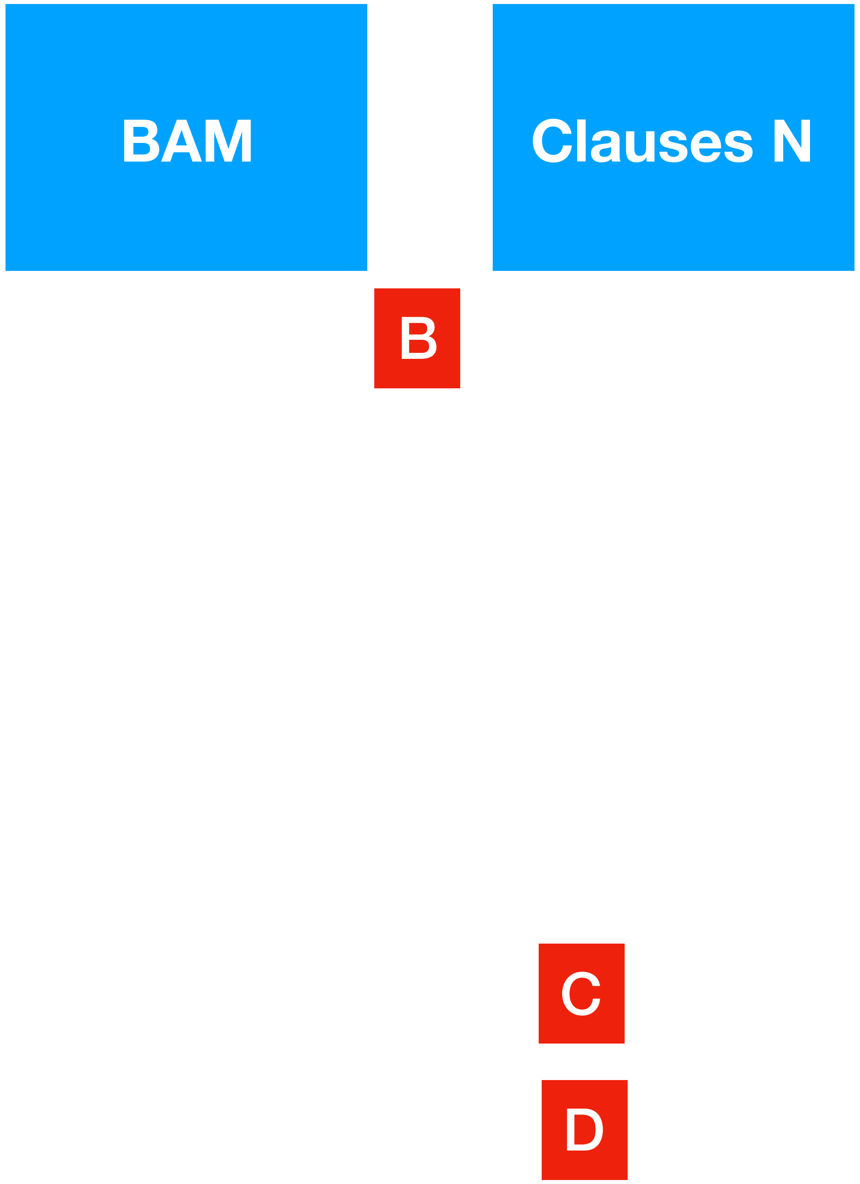
A first idea

A better strategy

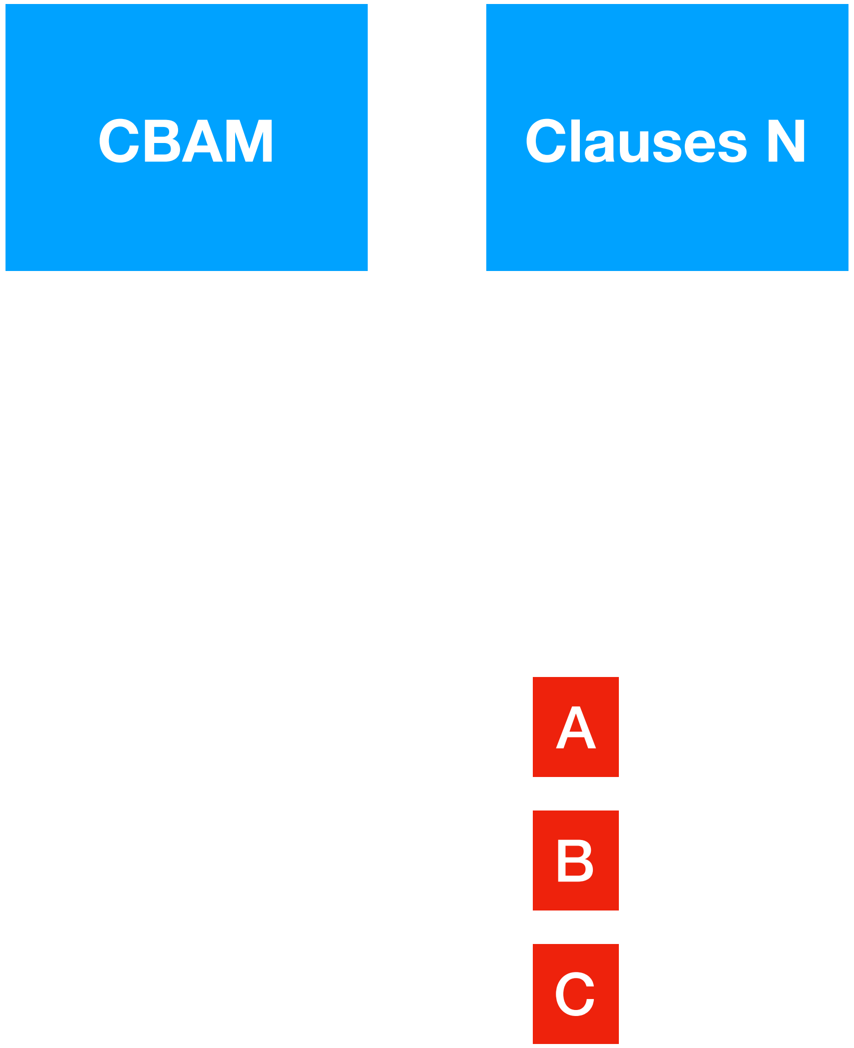


# Update Strategy

## A first idea

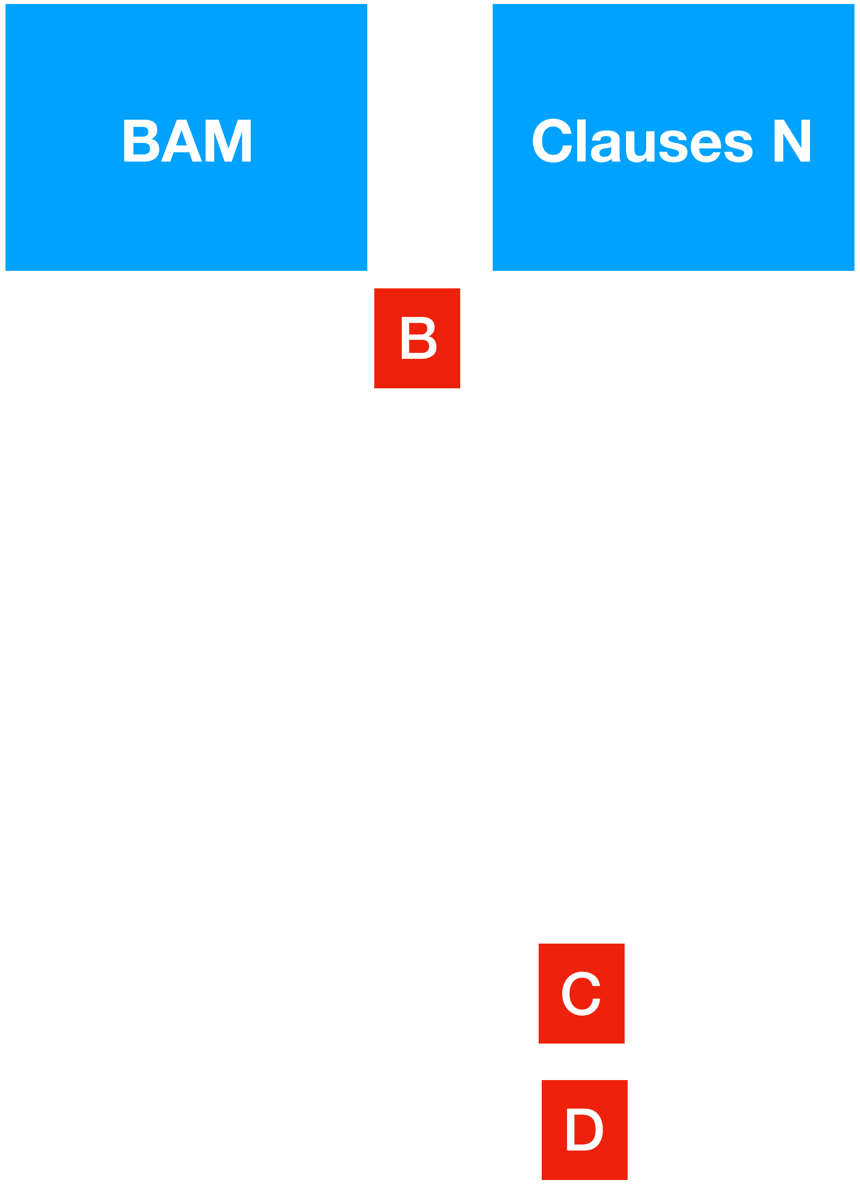


## A better strategy



# Update Strategy

## A first idea



**BAM**

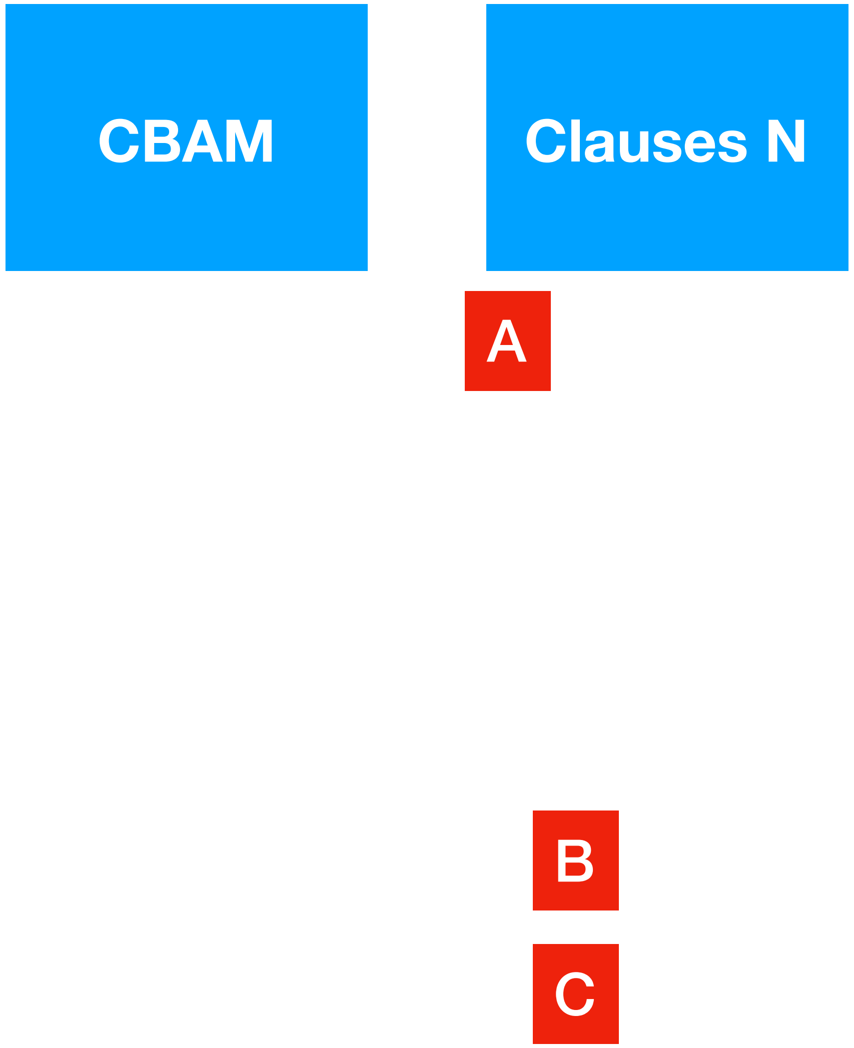
**Clauses N**

B

C

D

## A better strategy



**CBAM**

**Clauses N**

A

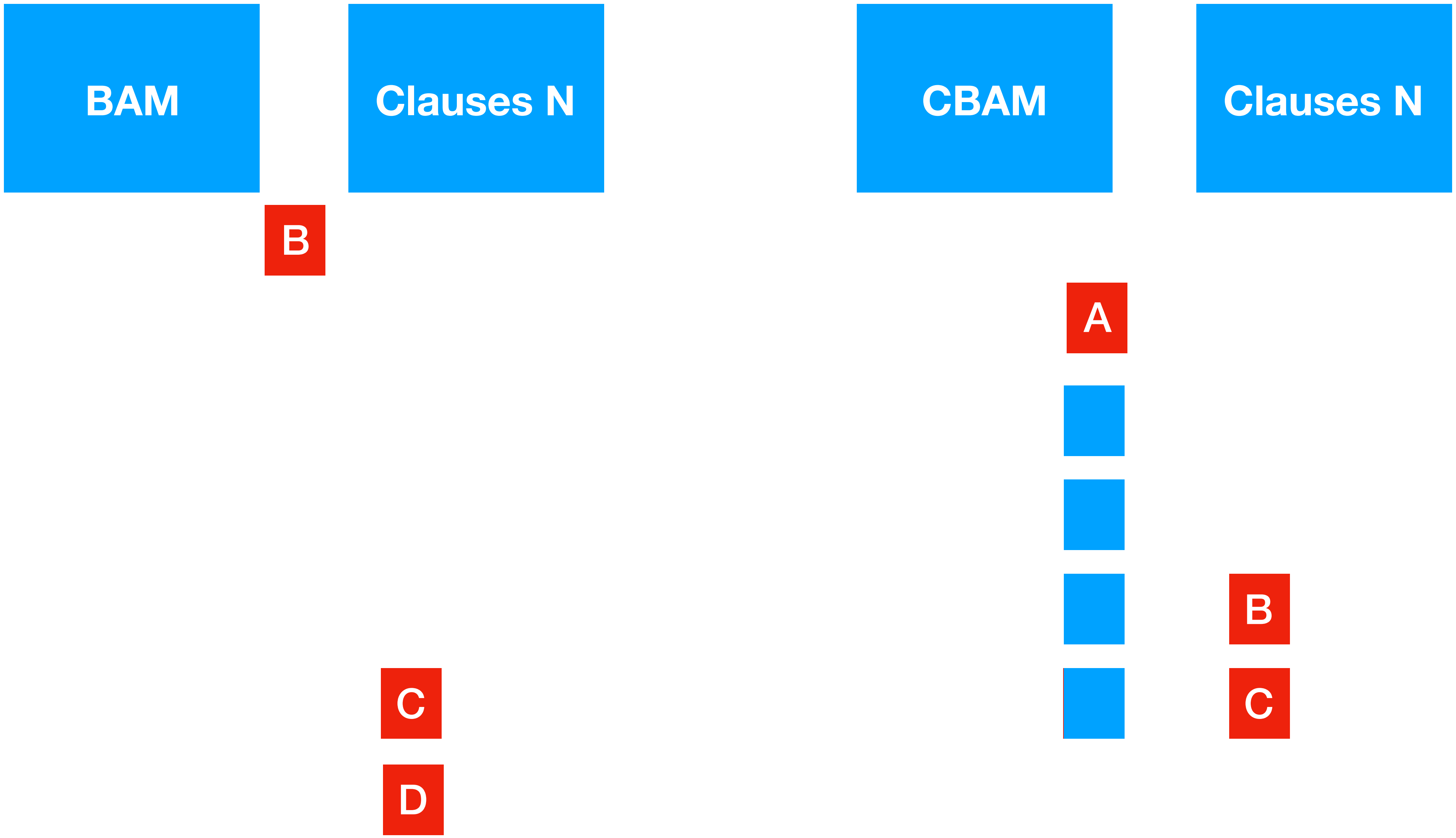
B

C

# Update Strategy

A first idea

A better strategy



BAM

Clauses N

B

C

D

CBAM

Clauses N

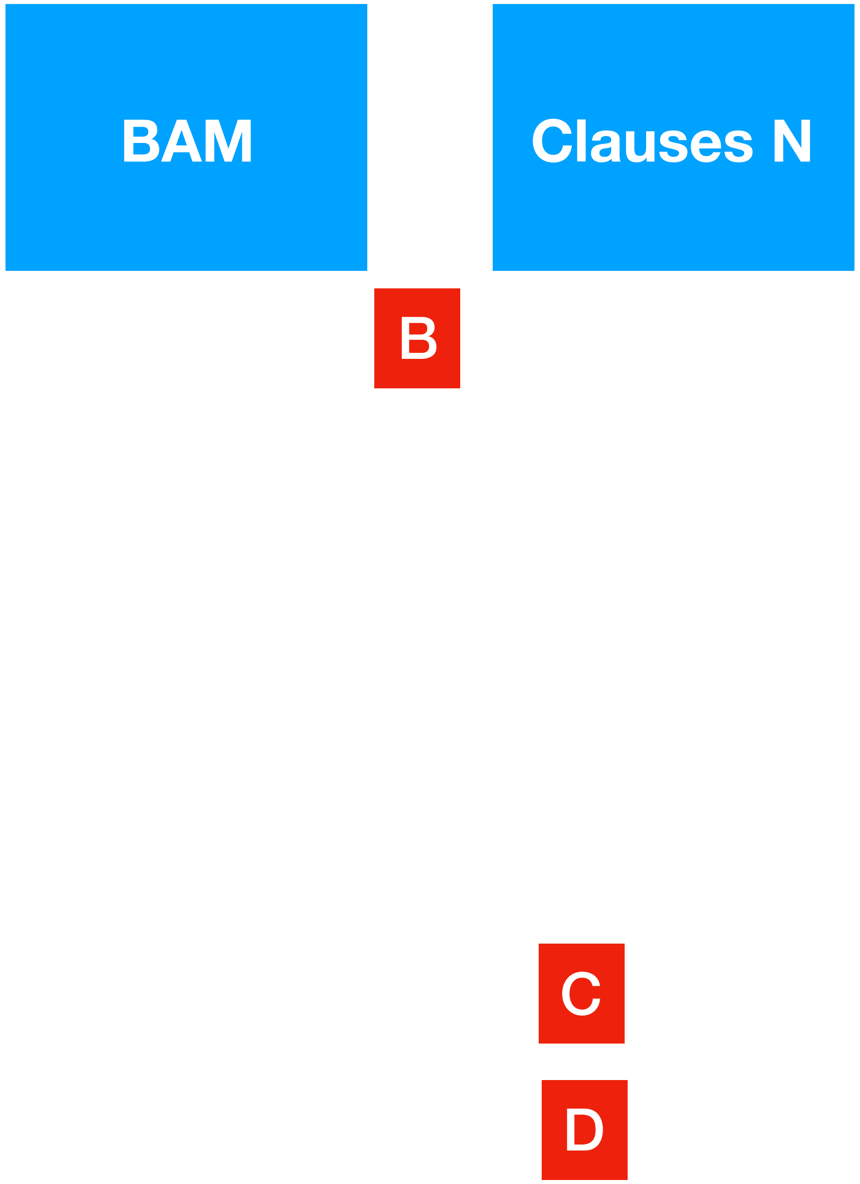
A

B

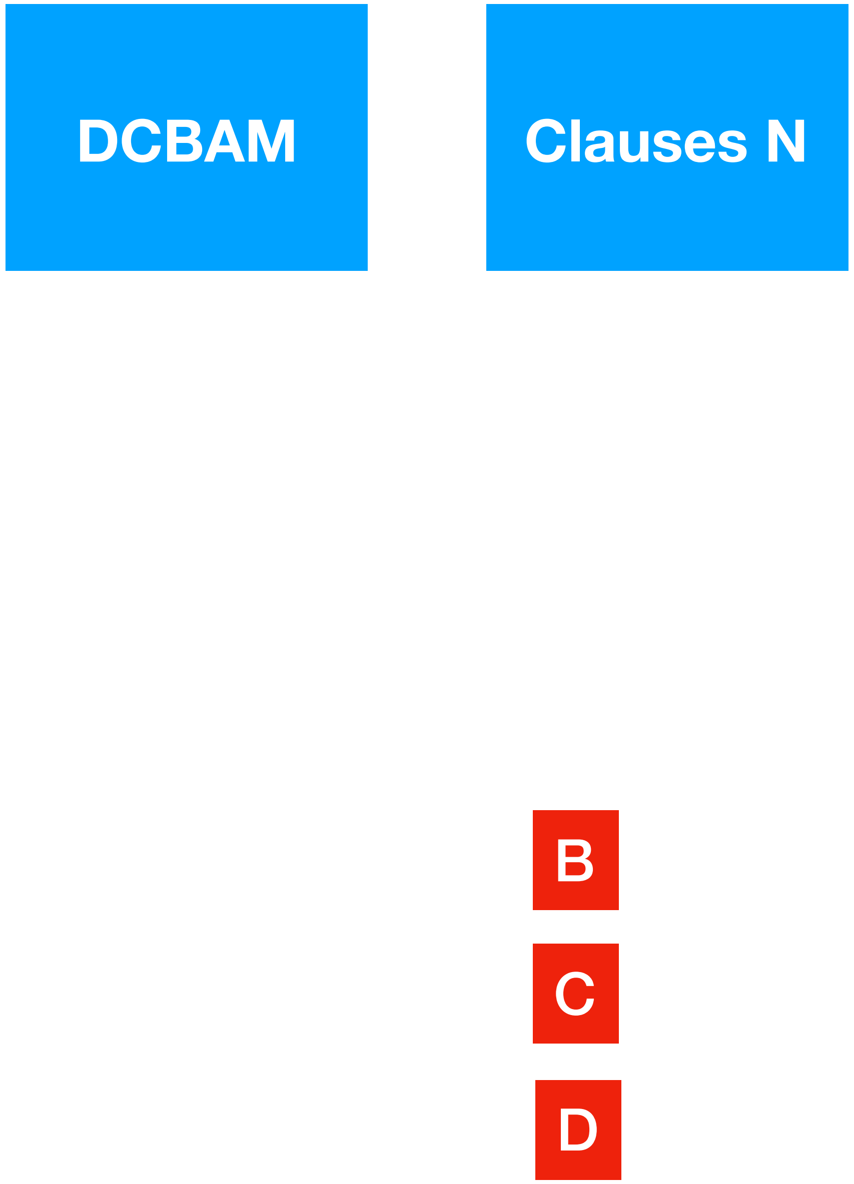
C

# Update Strategy

## A first idea

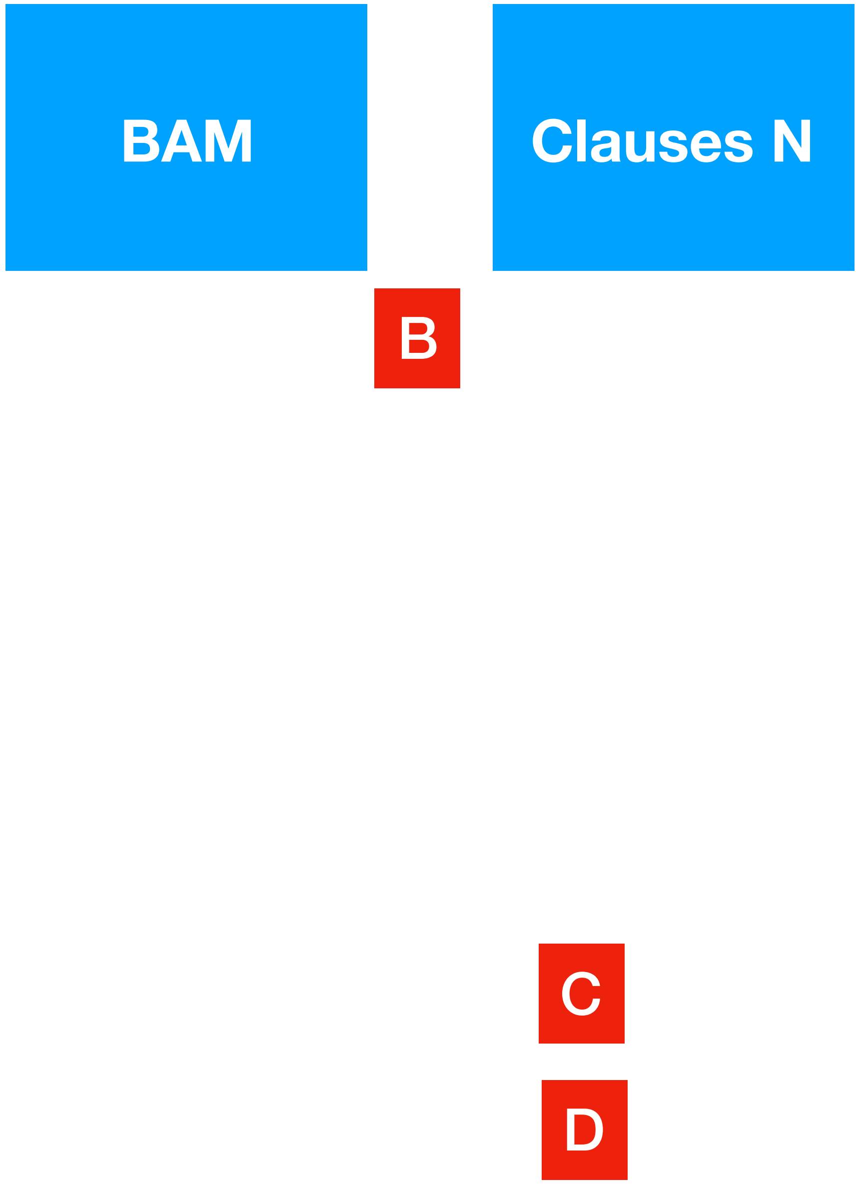


## A better strategy

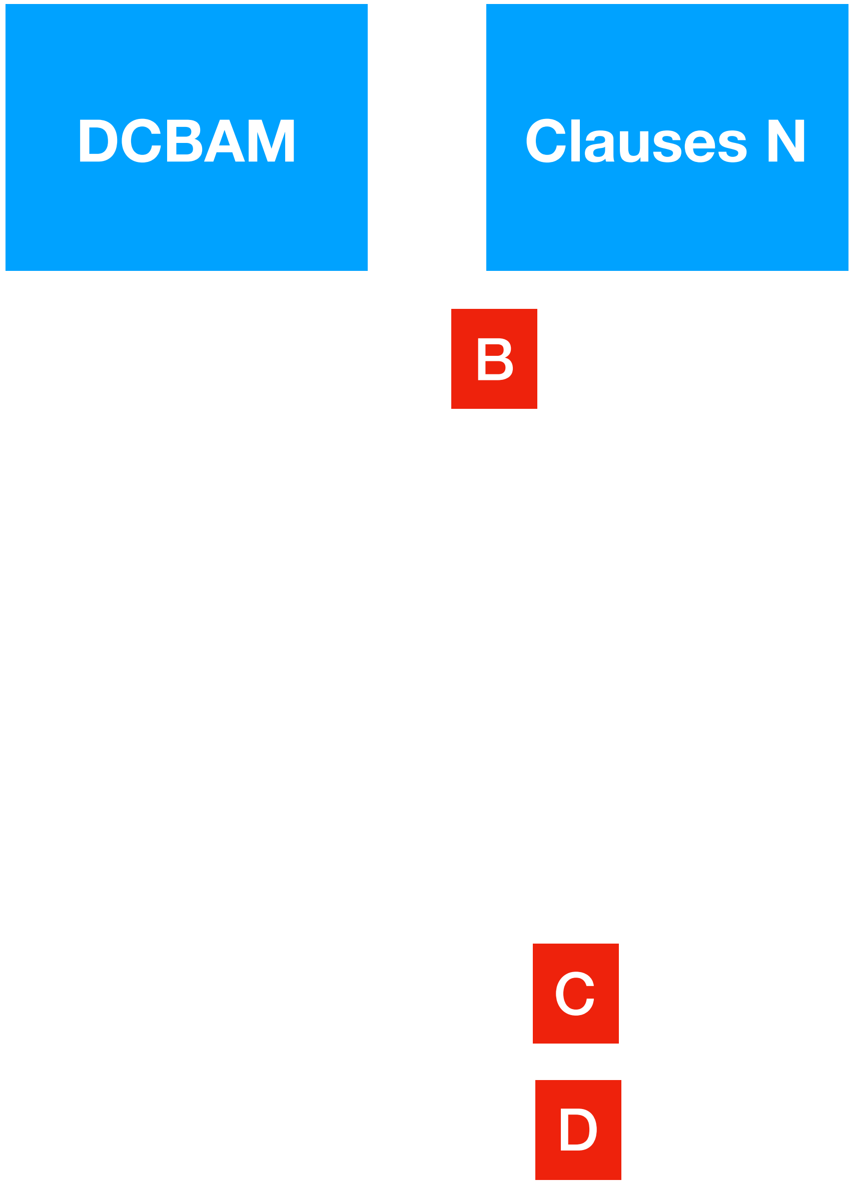


# Update Strategy

## A first idea

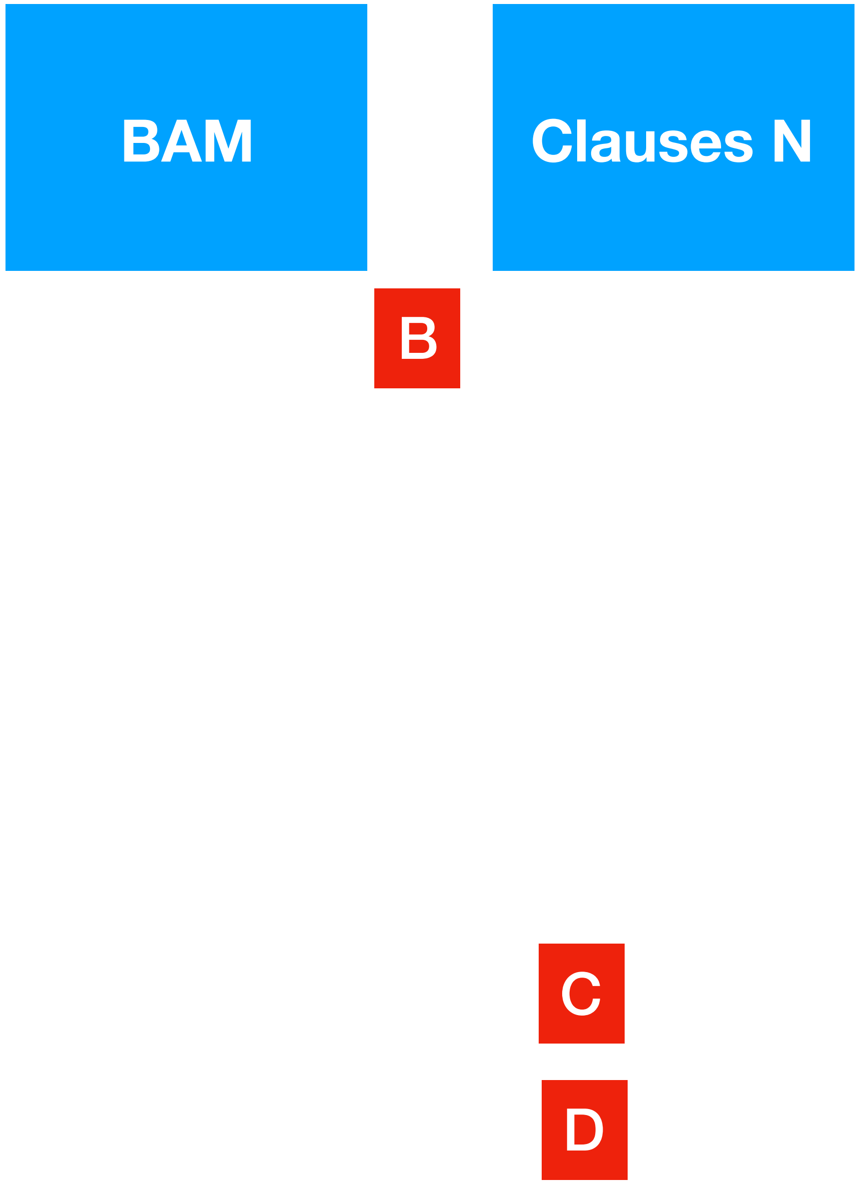


## A better strategy

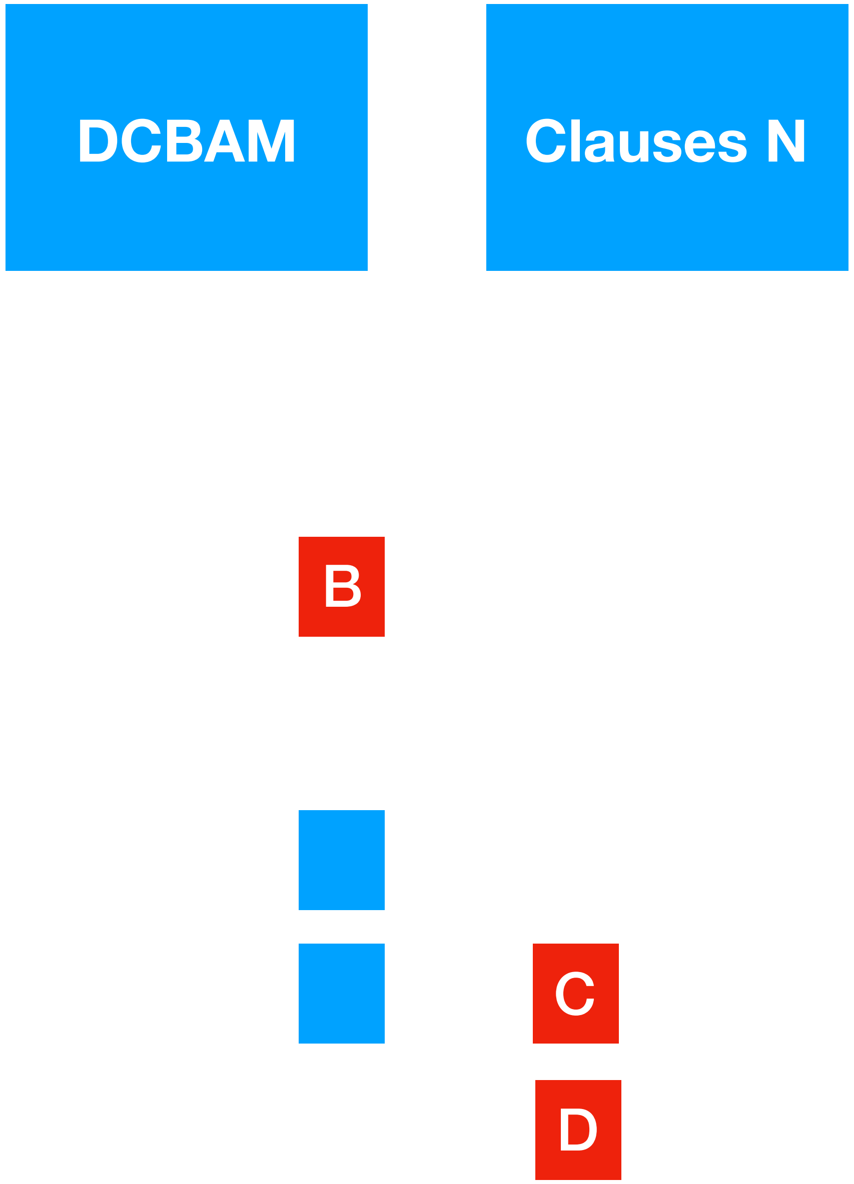


# Update Strategy

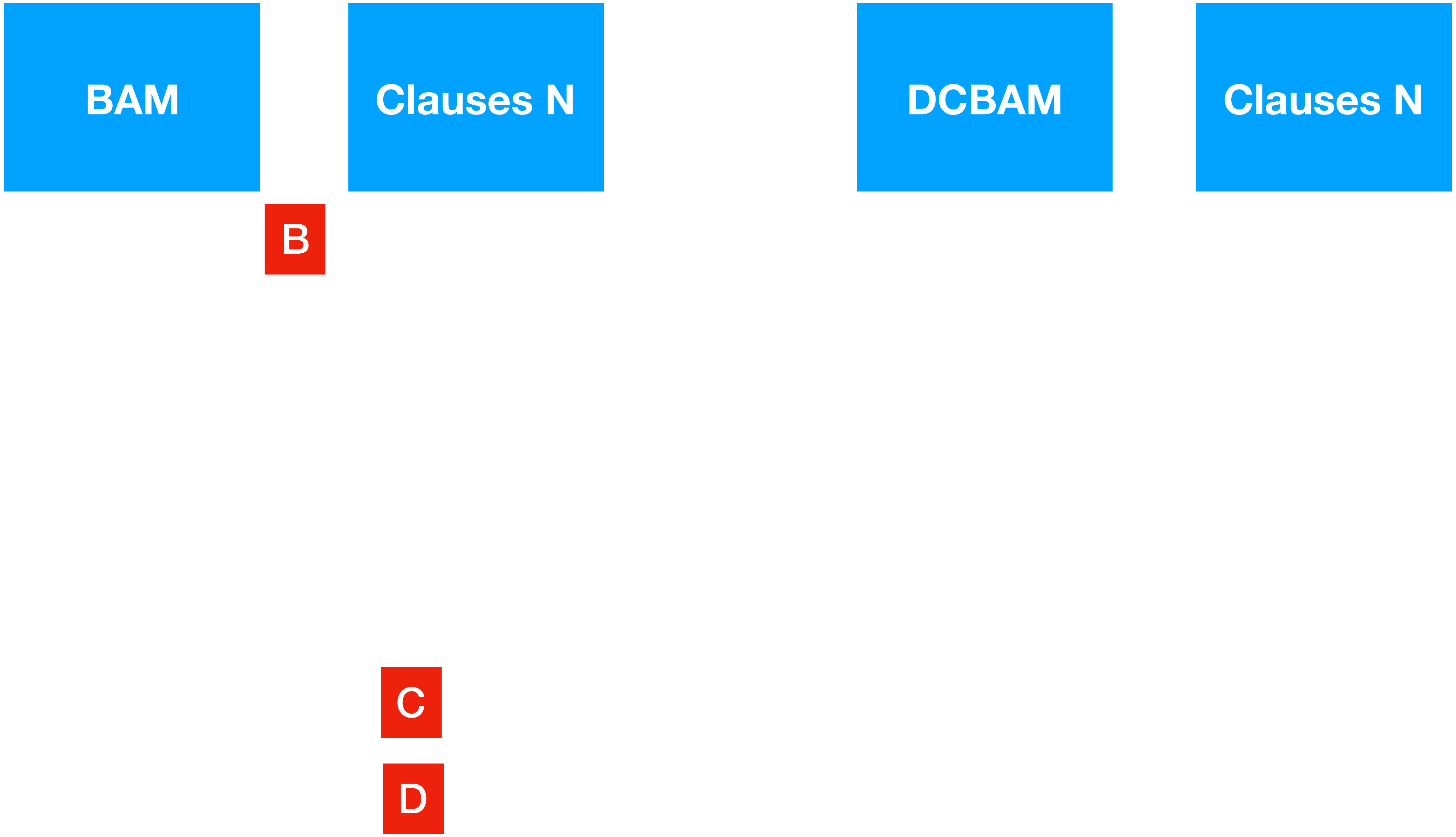
## A first idea



## A better strategy



# Update Strategy



A first idea

**BAM**

**Clauses N**

B

C

D

A better strategy

**DCBAM**

**Clauses N**