max planck institut
informatik

Saarland
Informatics Campus

# A Verified SAT Solver with Watched Literals Using Imperative HOL (Extended Abstract)

Mathias Fleury

Jasmin C. Blanchette

Peter Lammich

VirginiaTech
Invent the Future

SAARLAND
UNIVERSITY
SAARBRÜCKEN
GRADUATE SCHOOL OF
COMPUTER SCIENCE

VU
VRIJE
UNIVERSITEIT
AMSTERDAM

# 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
  - useful if non-checkable techniques are required

# How reliable are SAT solvers?

Two ways to ensure correctness:

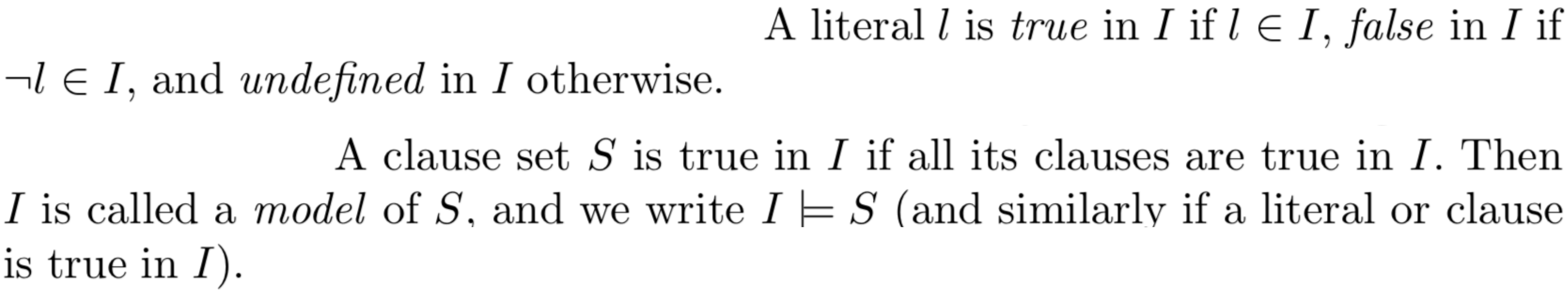
- certify the certificate
  - certificates are huge
- verification of the code
  - code will not be competitive
  - allows to study metatheory

# How reliable is the theory?

Conference version

Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates

Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell (SAT 2009)



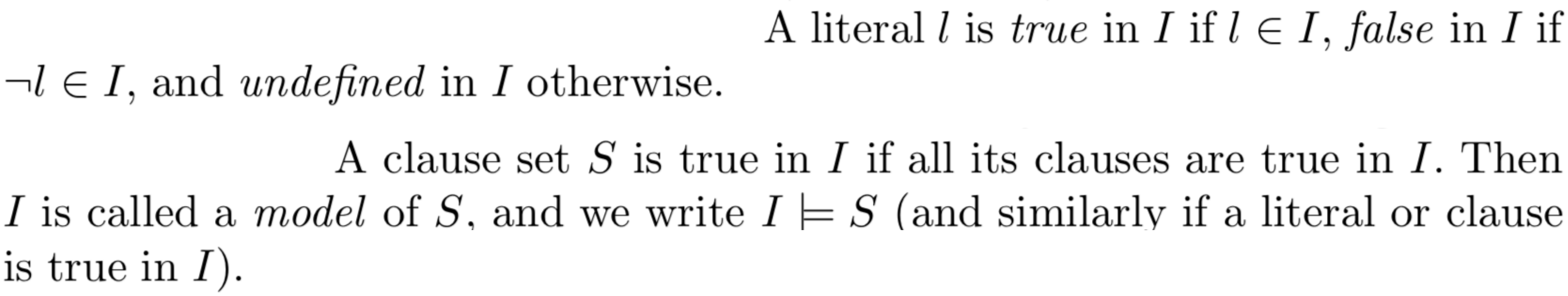
A literal $l$ is *true* in $I$ if $l \in I$, *false* in $I$ if $\neg l \in I$, and *undefined* in $I$ otherwise.

A clause set $S$ is true in $I$ if all its clauses are true in $I$. Then $I$ is called a *model* of $S$, and we write $I \models S$ (and similarly if a literal or clause is true in $I$).

# How reliable is the theory?

Conference version

Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates
Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell (SAT 2009)



A literal $l$ is *true* in $I$ if $l \in I$, *false* in $I$ if $\neg l \in I$, and *undefined* in $I$ otherwise.

A clause set $S$ is true in $I$ if all its clauses are true in $I$. Then $I$ is called a *model* of $S$, and we write $I \models S$ (and similarly if a literal or clause is true in $I$).

Journal version

A Framework for Certified Boolean Branch-and-Bound Optimization
Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell (JAR 2011)

literals of a clause $C$ are false in $I$. A clause set $S$ is true in $I$ if all its clauses are true in $I$; if $I$ is also total, then $I$ is called a *total model* of $S$, and we write $I \models S$.

**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 (CPP 2018, now)

VU VRIJE UNIVERSITEIT AMSTERDAM

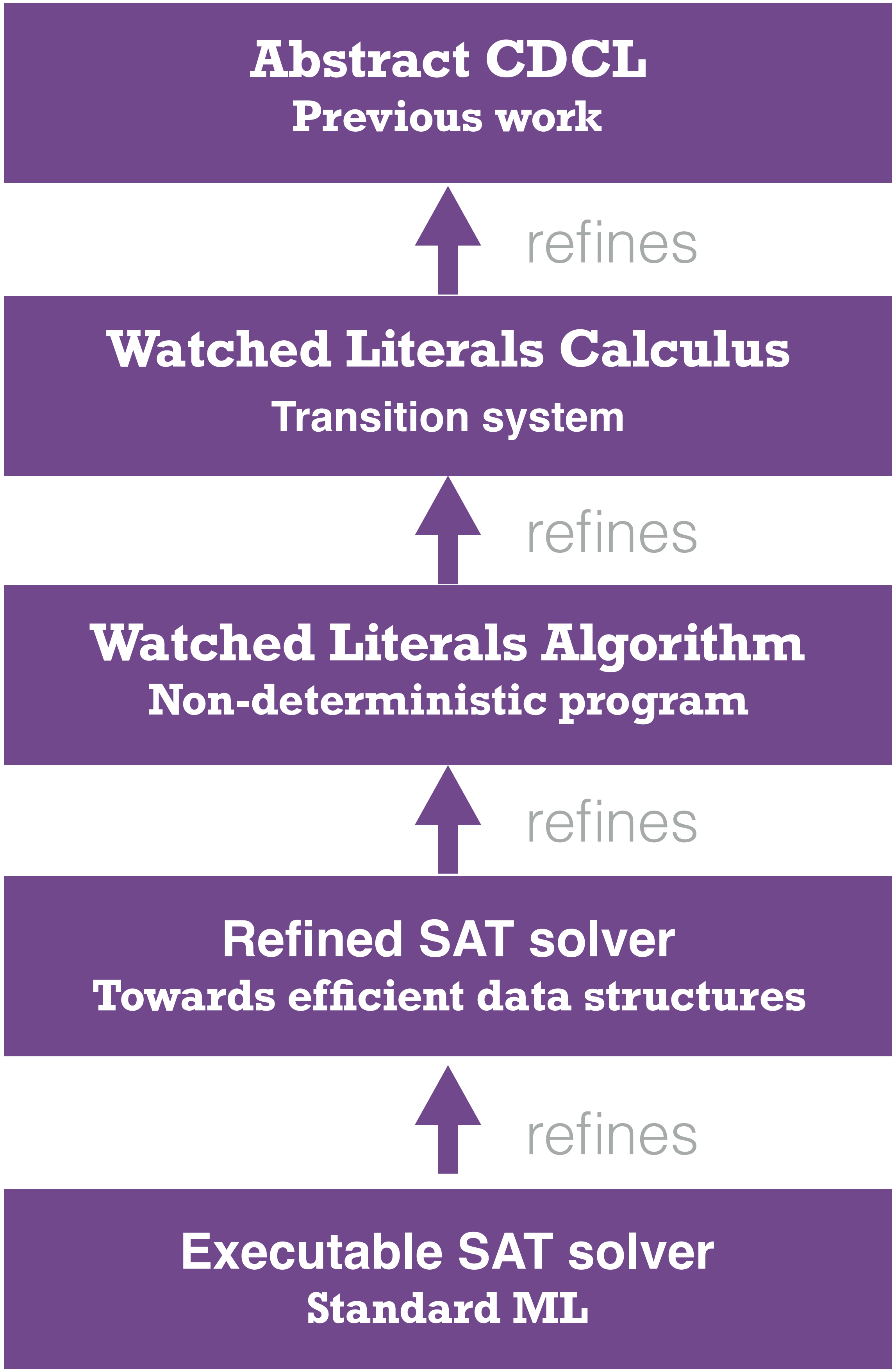
max planck institut informatik

# 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 (CPP 2018, now)

VRIJE
UNIVERSITEIT
AMSTERDAM

max planck institut
informatik



**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**

VRIJE
UNIVERSITEIT
AMSTERDAM

max planck institut
informatik

# Abstract CDCL

## Previous work

in Isabelle

## Propagate rule

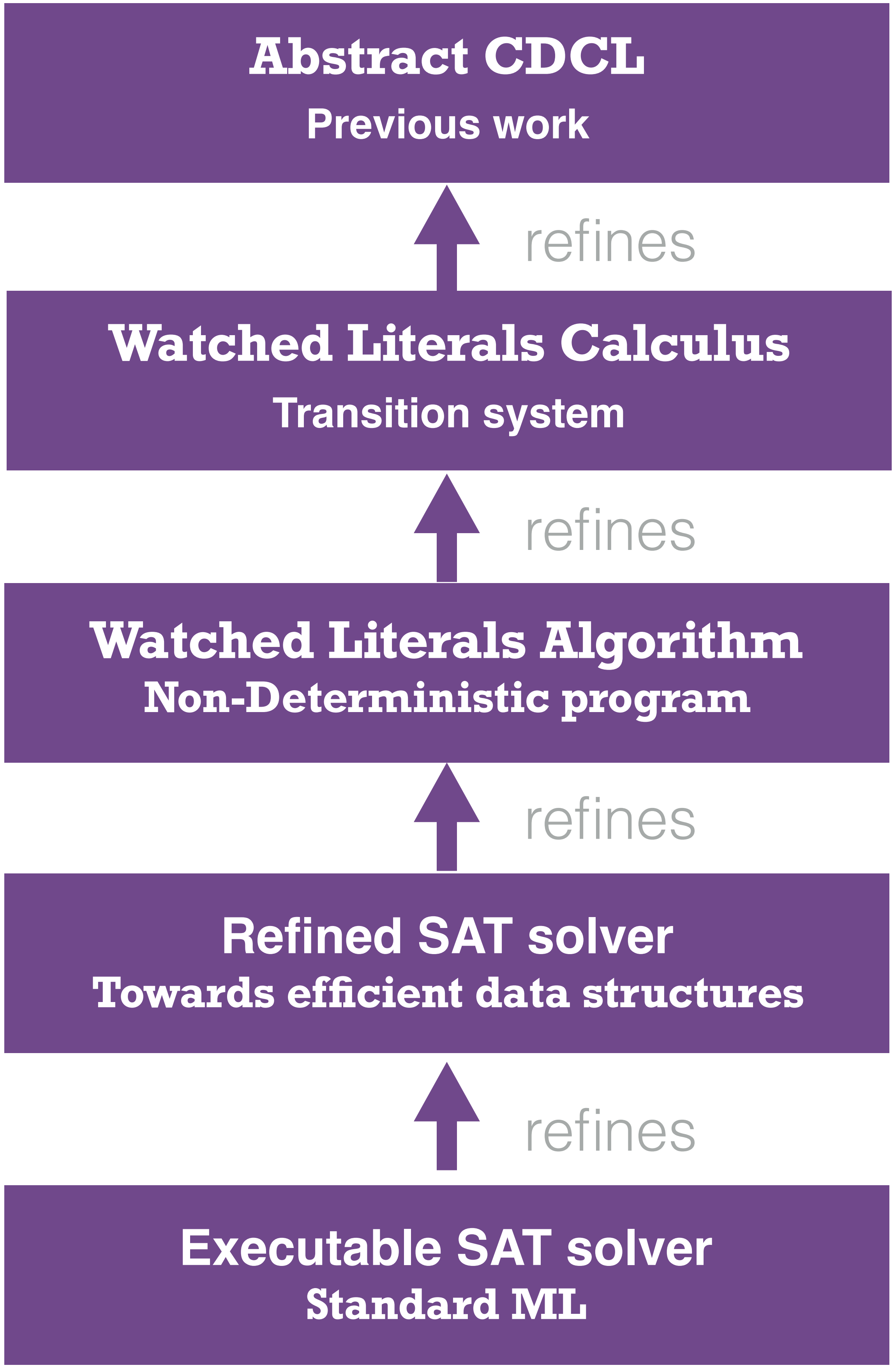
```
C ∨ L ∈ N ⟹ M ⊨as ¬C ⟹ undefined_lit M L ⟹
   (M, N) ⇒CDCL (L # M, N)
```

in Isabelle

## Propagate rule

$$C \lor L \in N \Longrightarrow M \models_{as} \neg C \Longrightarrow \mathtt{undefined\_lit}\ M\ L \Longrightarrow (M, N) \Rightarrow_{\mathrm{CDCL}} (L \mathbin{\#} M, N)$$

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

# Watched literals invariant

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

# Watched literals invariant

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

unless a conflict has been found

# Watched literals invariant

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

unless a conflict has been found

or an update is pending

# Watched literals invariant (less wrong)

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

this literal has been set earlier

unless a conflict has been found

or an update is pending

# Watched literals invariant

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

→

1. Watch any literal
   if there is a true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

## Watched literals invariant

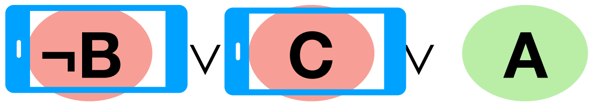
1. Watch one true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

## with blocking literals

1. Watch any literal
   if there is a true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

## Watched literals invariant

## with blocking literals

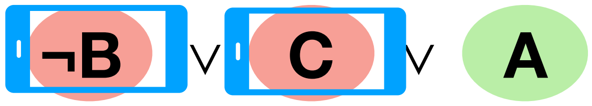


1. Watch one true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

1. Watch any literal
   if there is a true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

## Watched literals invariant

## with blocking literals



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

1. Watch any literal
   if there is a true literal
2. or watch two unset literals
3. or watch a false literal
   if all other literals are false

~~(not yet refined to code)~~

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

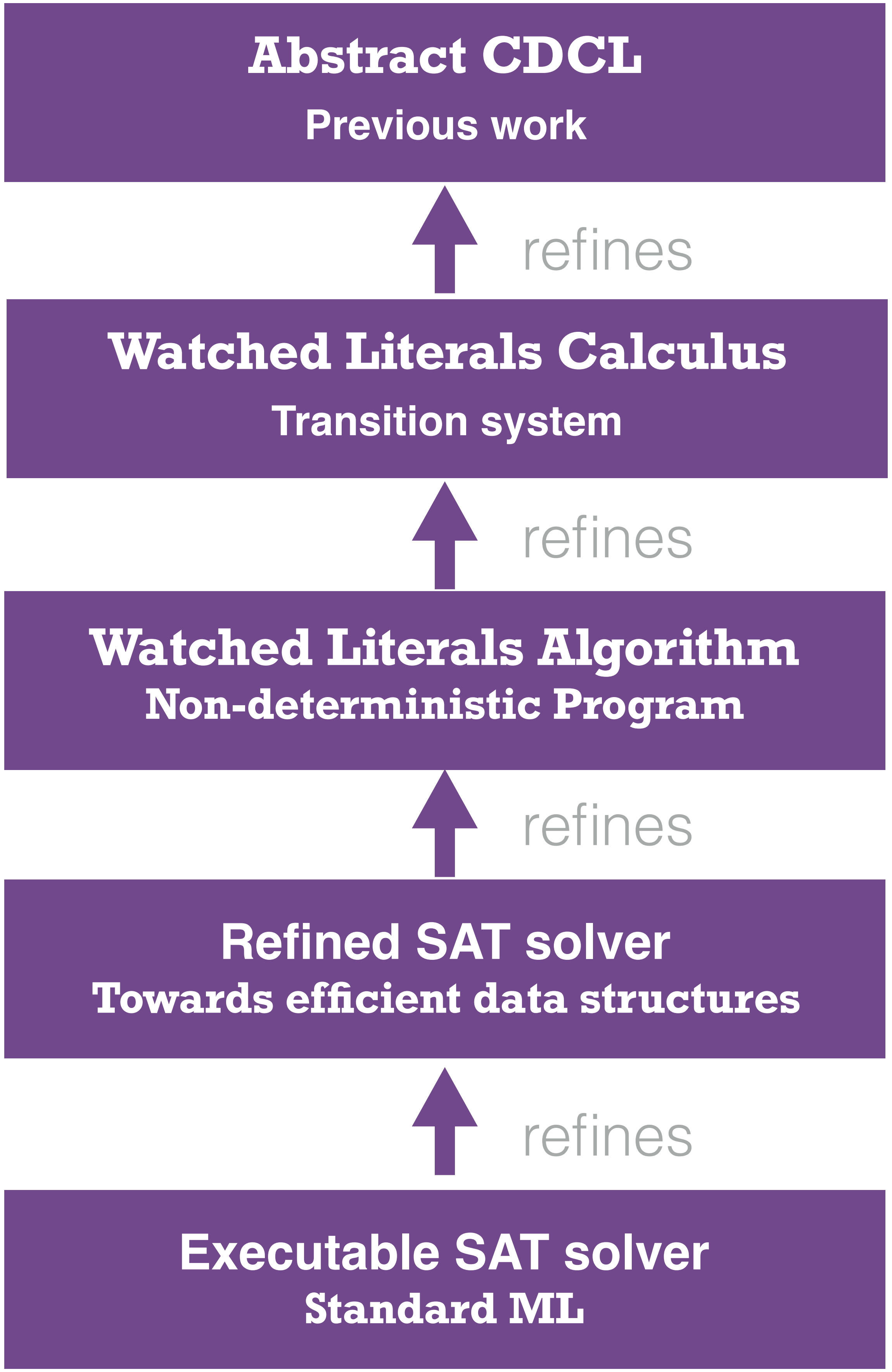
- 🔑 Finding invariants (11 new ones)
- ✗ No high-level description
- ✓ sledgehammer

## Correctness theorem

in Isabelle

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

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



**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**

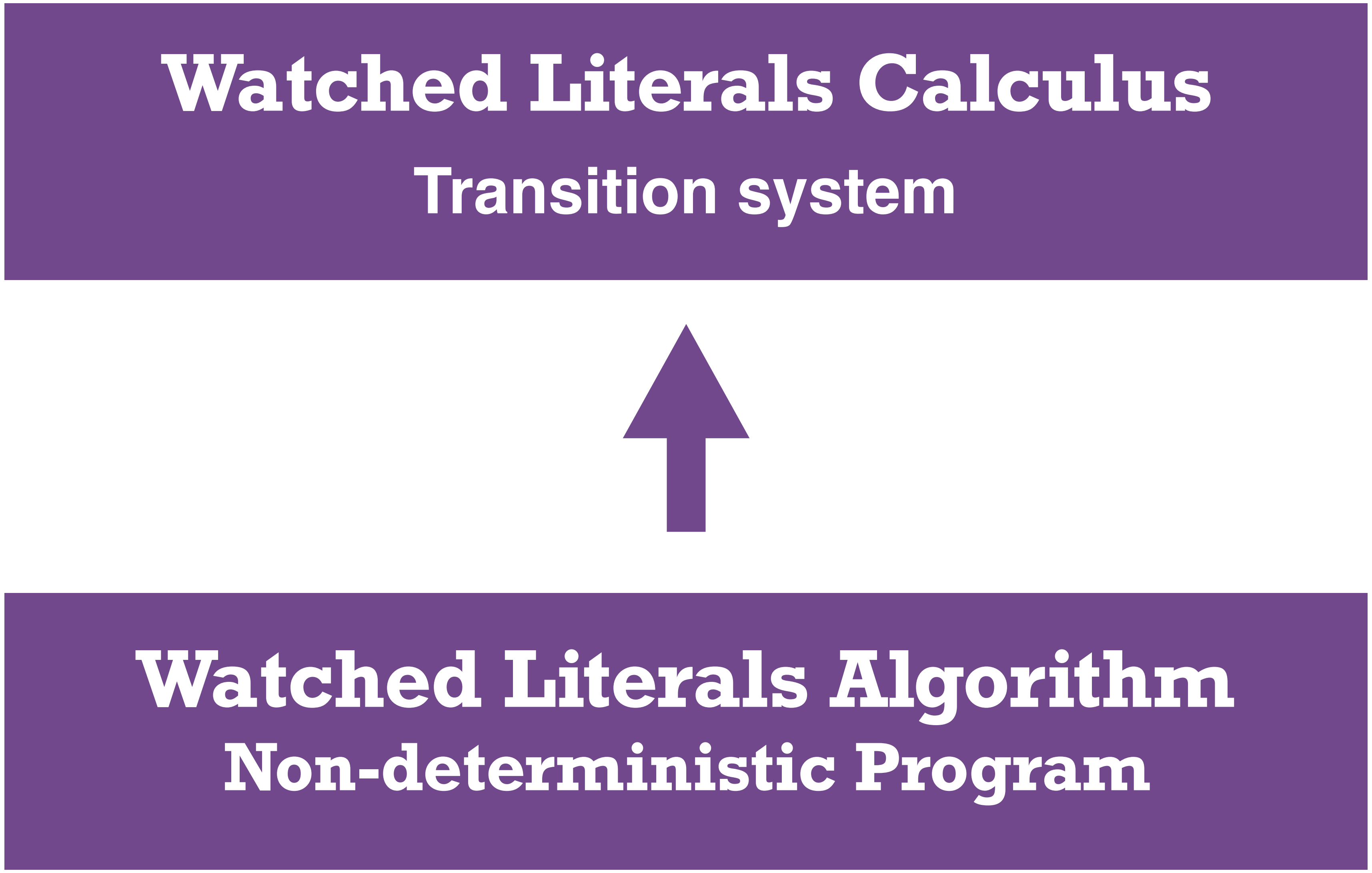
refines

**Executable SAT solver**

**Standard ML**

VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik



**Watched Literals Calculus**

Transition system

**Watched Literals Algorithm**

**Non-deterministic Program**

# DEMO I

# Picking Next Clause

```
propagate_conflict_literal L S :=
  WHILE$_T$
    ($\lambda$T. clauses_to_update T $\neq$ {})

    ($\lambda$T. do {
      ASSERT(clauses_to_update T $\neq$ {})
      C $\leftarrow$ SPEC ($\lambda$C. C $\in$ clauses_to_update T);
      U $\leftarrow$ 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$_T$
    ($\lambda$T. clauses_to_update T $\neq$ {})

    ($\lambda$T. do {
       ASSERT(clauses_to_update T $\neq$ {})
       C ← SPEC ($\lambda$C. C $\in$ 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

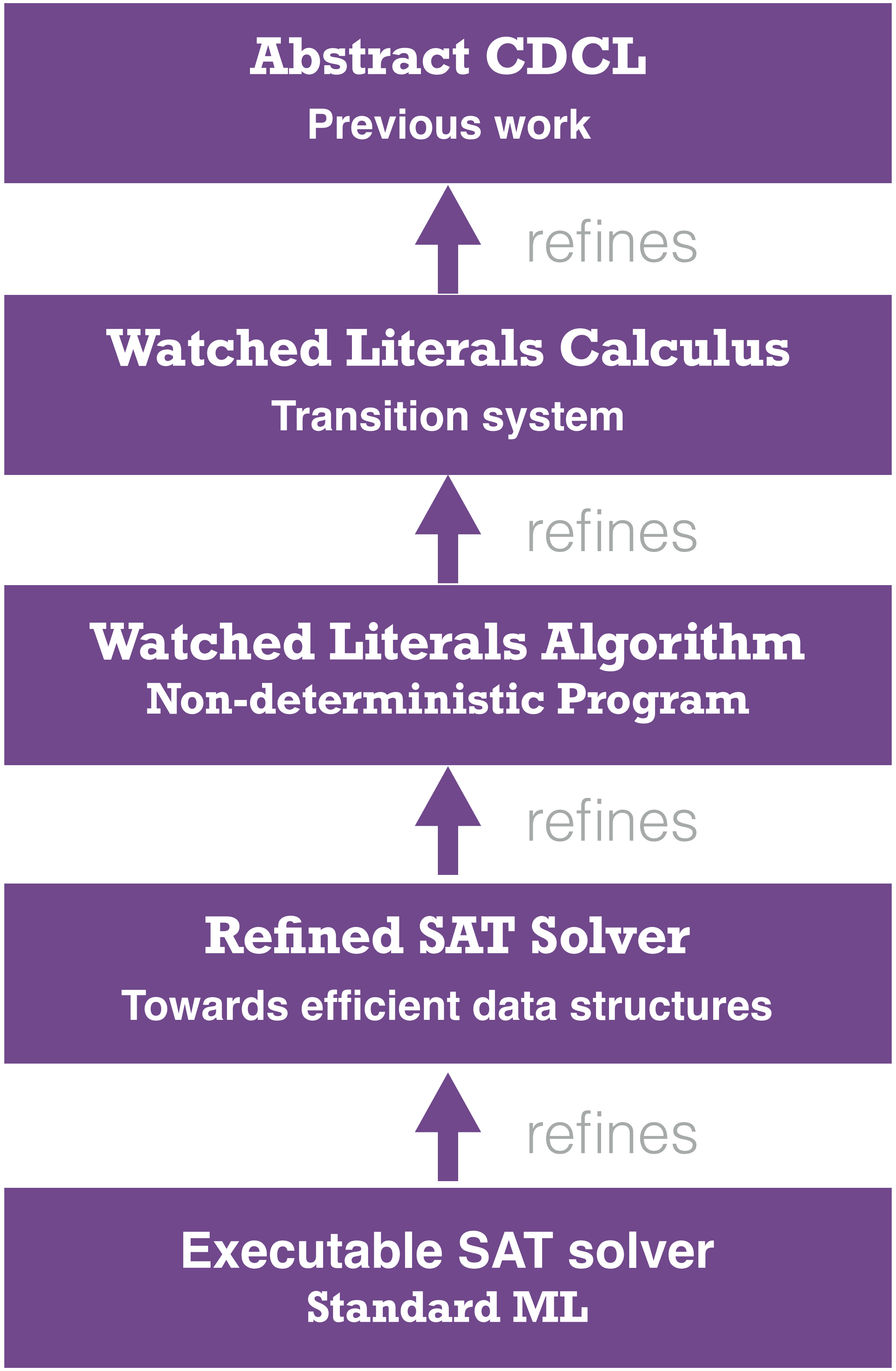
$$\texttt{propagate\_conflict\_literal}\ L\ S \le \texttt{SPEC}(\lambda T.\ S \Rightarrow_{\texttt{TWL}}^{*} T)$$

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

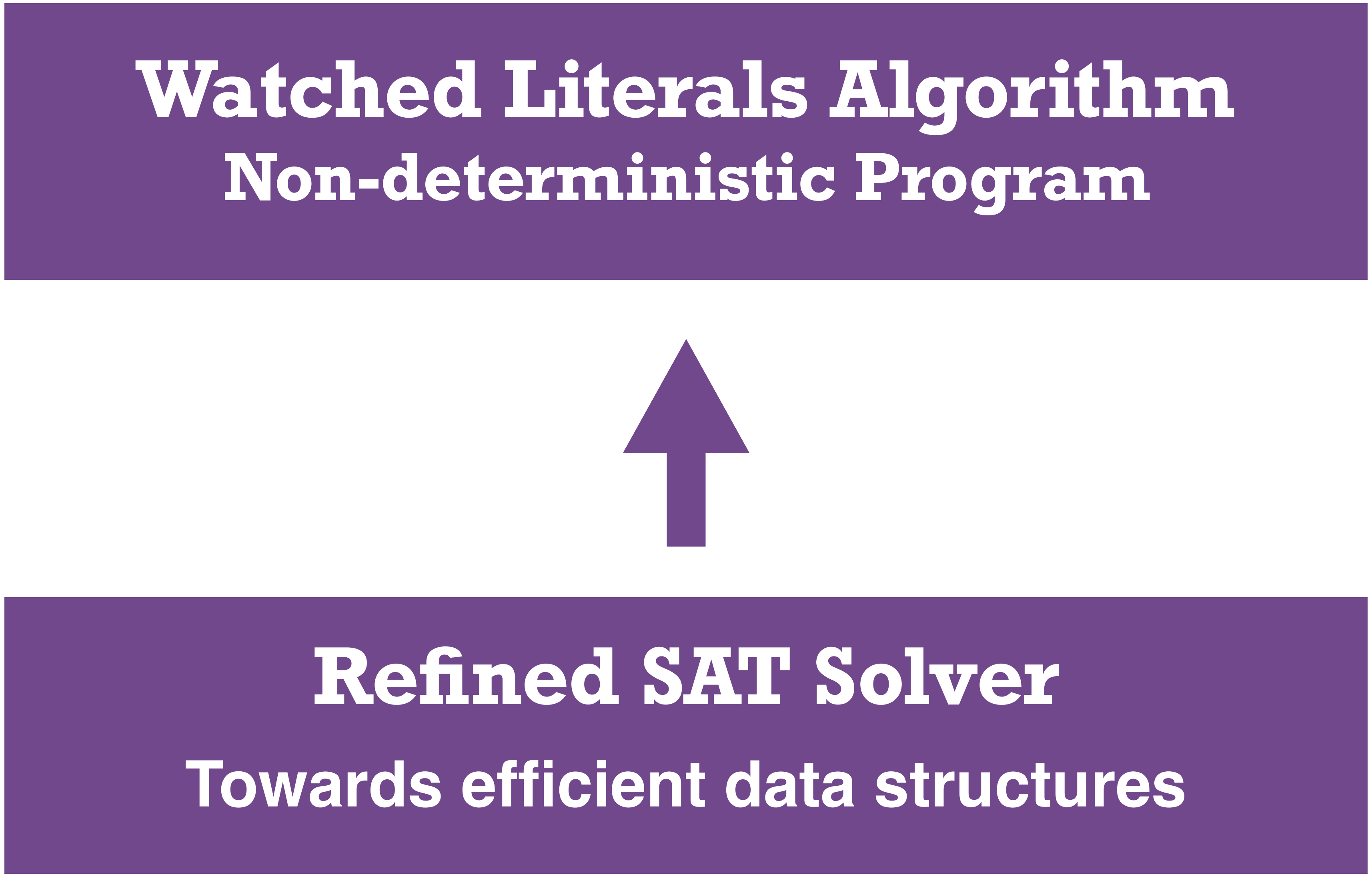
**Refined SAT Solver**

Towards efficient data structures

refines

**Executable SAT solver**

**Standard ML**



## Watched Literals Algorithm

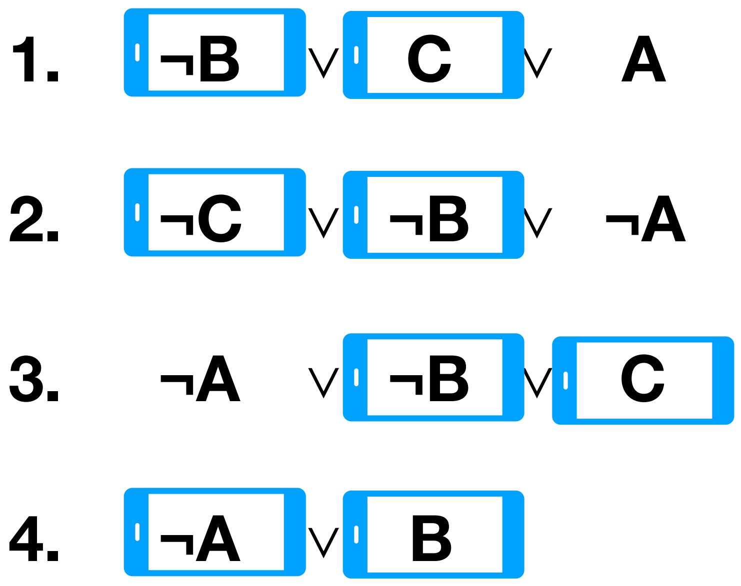
### Non-deterministic Program

## Refined SAT Solver

Towards efficient data structures

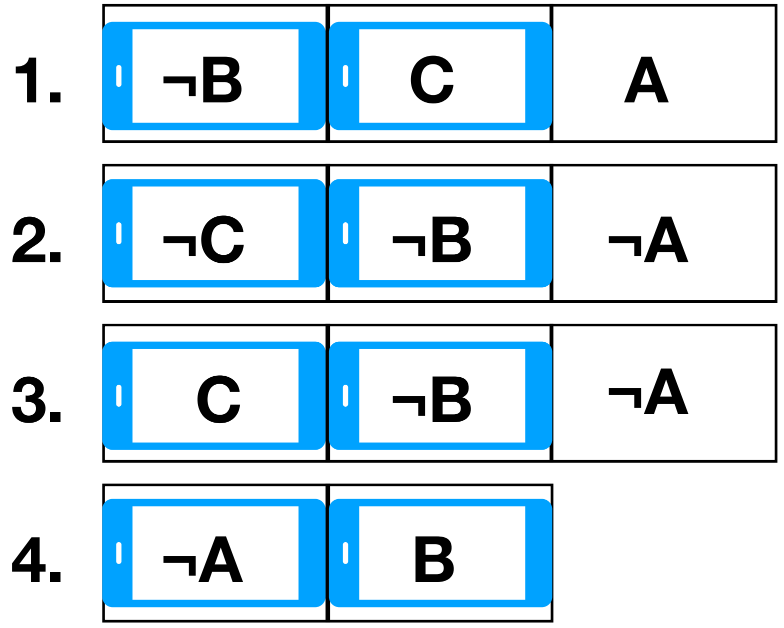
# DPLL with Watched Literals

**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

- 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)

# DEMO II

```
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)
```

VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik

```
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

In Isabelle

many simp rules along:

$$(S_{\mathrm{list}}, T_{\mathrm{mset}}) \in R_{\mathrm{list\_mset}} \Longrightarrow \mathrm{trail}_{\mathrm{mset}}\ T = \mathrm{trail}_{\mathrm{list}}\ S$$

many invariant along:

$$(\exists T_{\mathrm{mset}}.\ (S_{\mathrm{list}}, T_{\mathrm{mset}}) \in R_{\mathrm{list\_mset}} \wedge \mathrm{inv}_{\mathrm{mset}}\ T) \wedge \mathrm{inv}_{\mathrm{mset}}\ T$$

# Fly, you fool

```
lemma
  ‹P S ⟹ ∃S. P S› for S :: ‹'a × 'b›
  by auto
```

# Fly, you fool

```
lemma
  ‹P S ⟹ ∃S. P S› for S :: ‹'a × 'b›
  by auto
```

## How to deactivate this in Isabelle

```
text ‹Find a less hack-like solution›
setup ‹map_theory_claset
  (fn ctxt => ctxt delSWrapper "split_all_tac")›
```

```
lemma
  fixes S :: ‹'a × 'b × 'c›
  assumes
   H: ‹∃T. (S,T) ∈ R ∧ P (fst S)› and
   [simp]: ‹⋀S T. (S, T) ∈ R ⟹ fst T = fst S›
 shows
   ‹∃T. (S, T) ∈ R ∧ P (fst T)›
  using H
  by auto
```

**vs**

```
lemma
  fixes S :: ‹'a × 'b × 'c›
  assumes
   H: ‹∃T. (S,T) ∈ R ∧ P (fst S)› and
   [simp]: ‹⋀S T. (S, T) ∈ R ⟹ fst S = fst T›
 shows
   ‹∃T. (S, T) ∈ R ∧ P (fst T)›
  using H
  by auto
```

**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

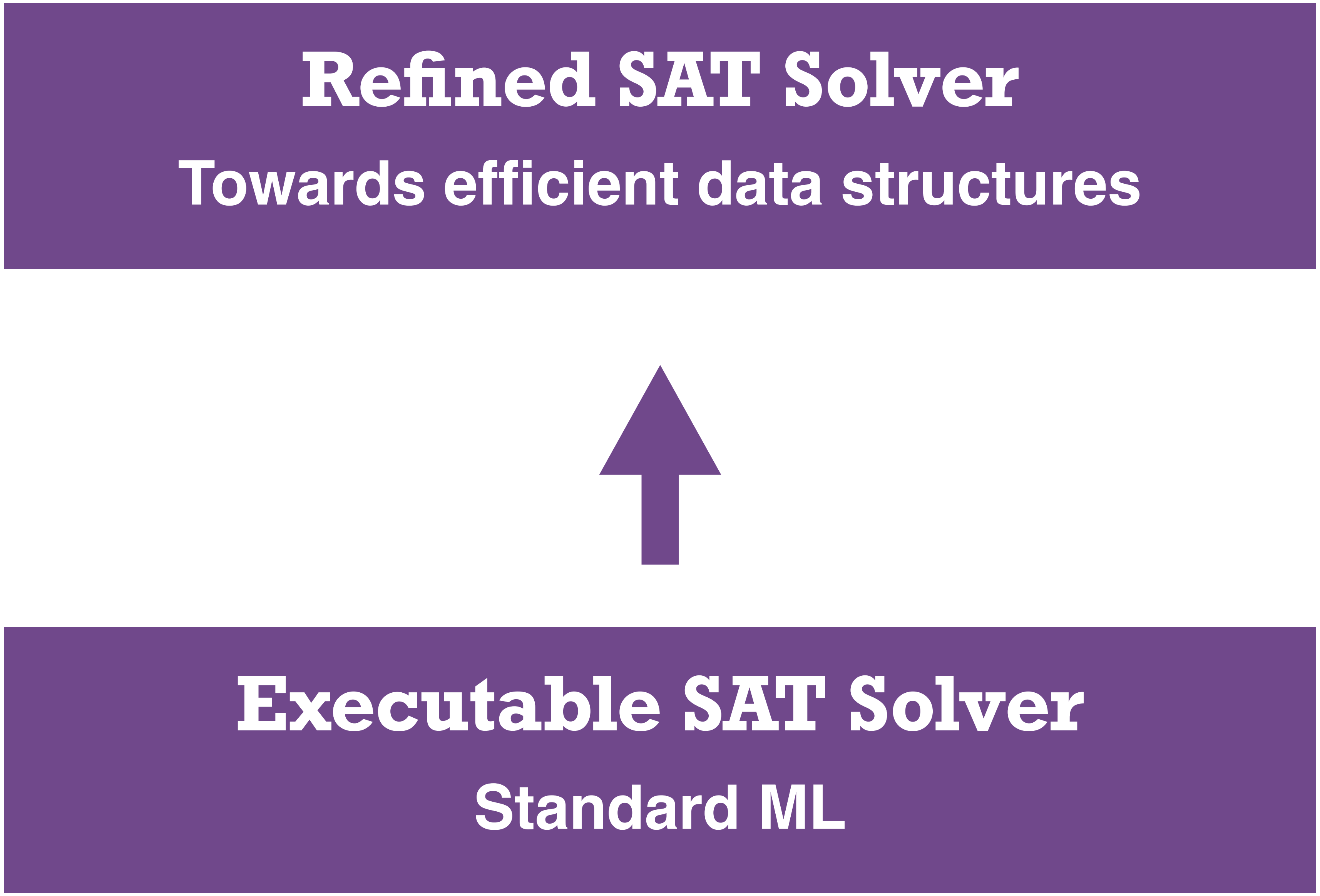
refines

**Executable SAT Solver**

Standard ML

VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik



**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_assnᵏ *ₐ state_assnᵈ →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_assn^k *_a state_assn^d →_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

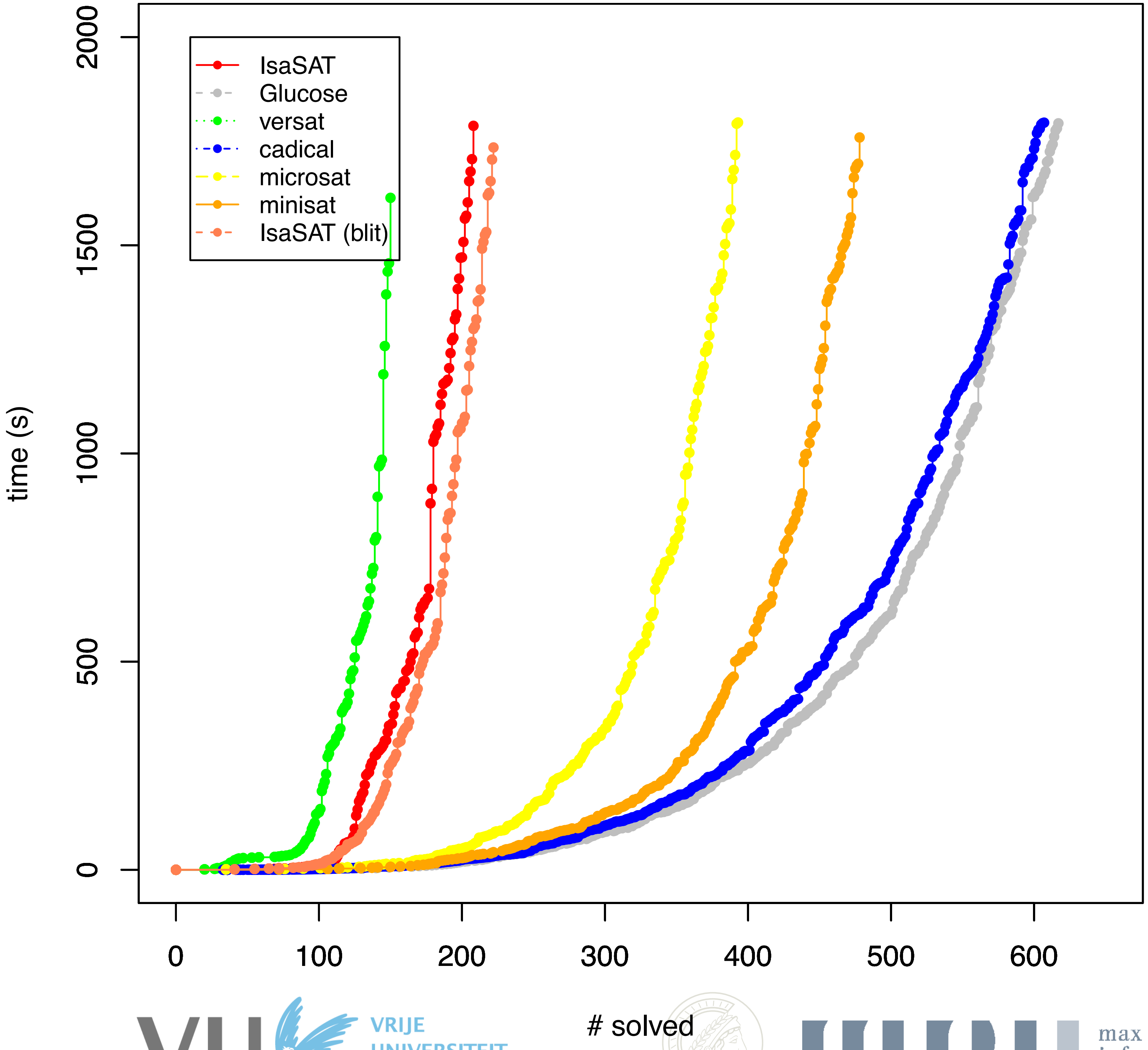
```
‹(IsaSAT_code, model_if_satisfiable)
    ∈ [λN. each_clause_is_distinct N ∧
        literals_fit_in_32_bit_integer N]ₐ
      clauses_as_listsᵏ → model›
```

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

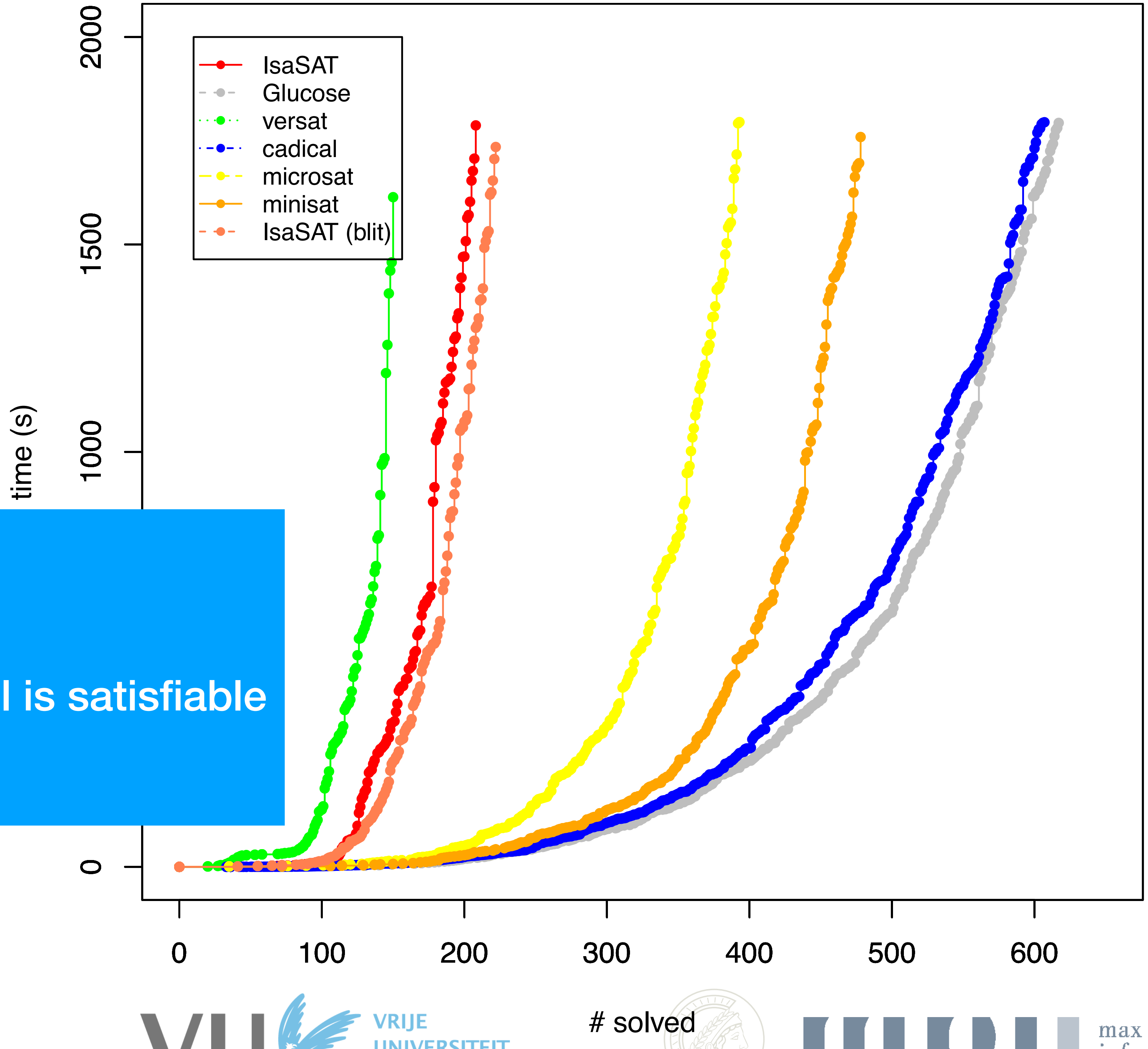
VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik

# SAT-Comp ’09, ’15 (main track), and ’14 (all submitted problems), already preprocessed

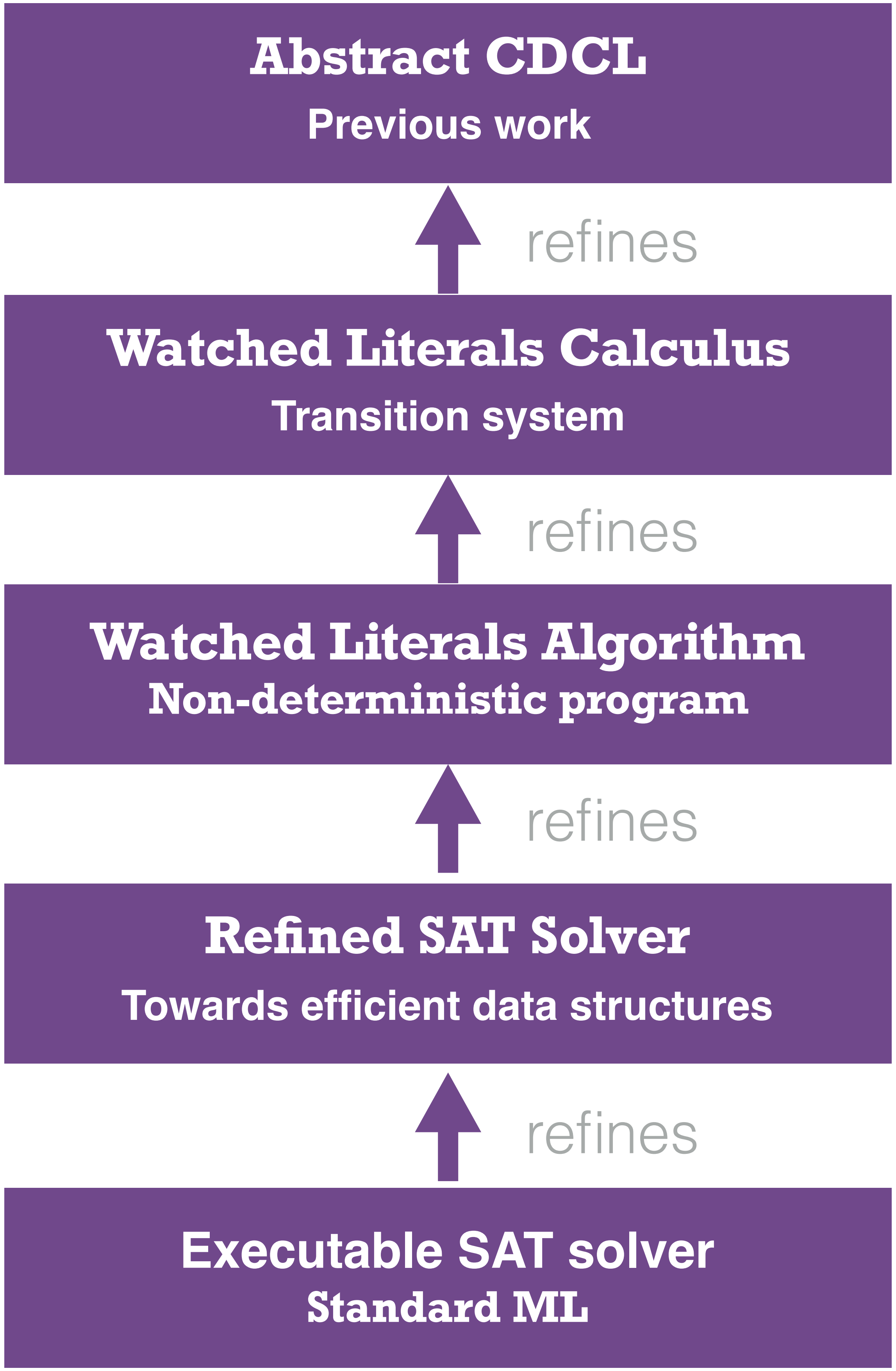


# SAT-Comp ’09, ’15 (main track), and ’14 (all submitted problems), already preprocessed



Correct up to:

- run-time checks
- checking the model is satisfiable



**Abstract CDCL**

Previous work

refines

**Watched Literals Calculus**

Transition system

refines

**Watched Literals Algorithm**

**Non-deterministic program**

refines

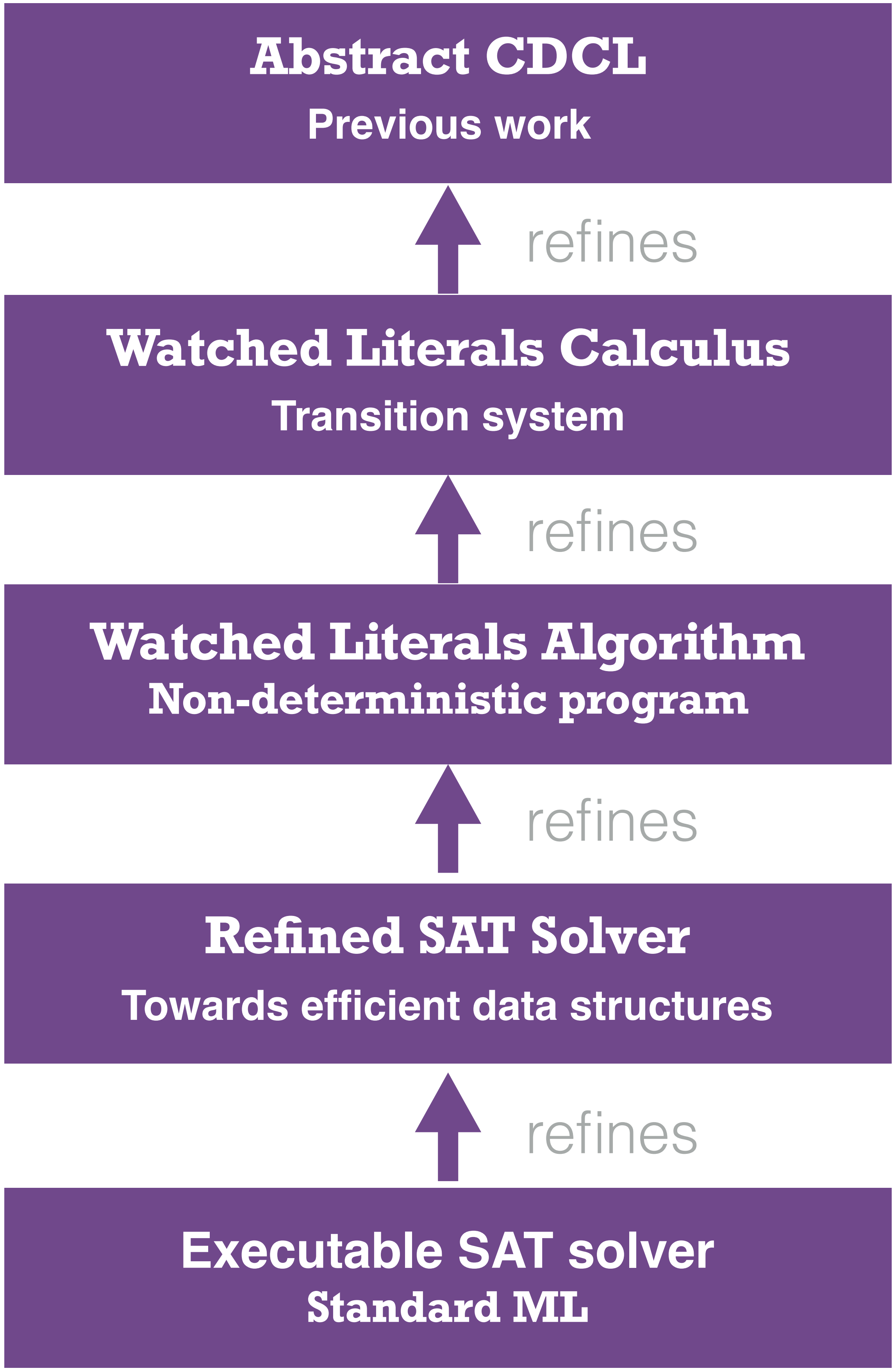
**Refined SAT Solver**

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

VRIJE UNIVERSITEIT AMSTERDAM

max planck institut informatik

**Abstract CDCL**

Previous work

- allow learned clause minimisation
- no reuse of restarts

refines

**Watched Literals Calculus**

Transition system

refines

**Watched Literals Algorithm**

**Non-deterministic program**

refines

**Refined SAT Solver**

Towards efficient data structures

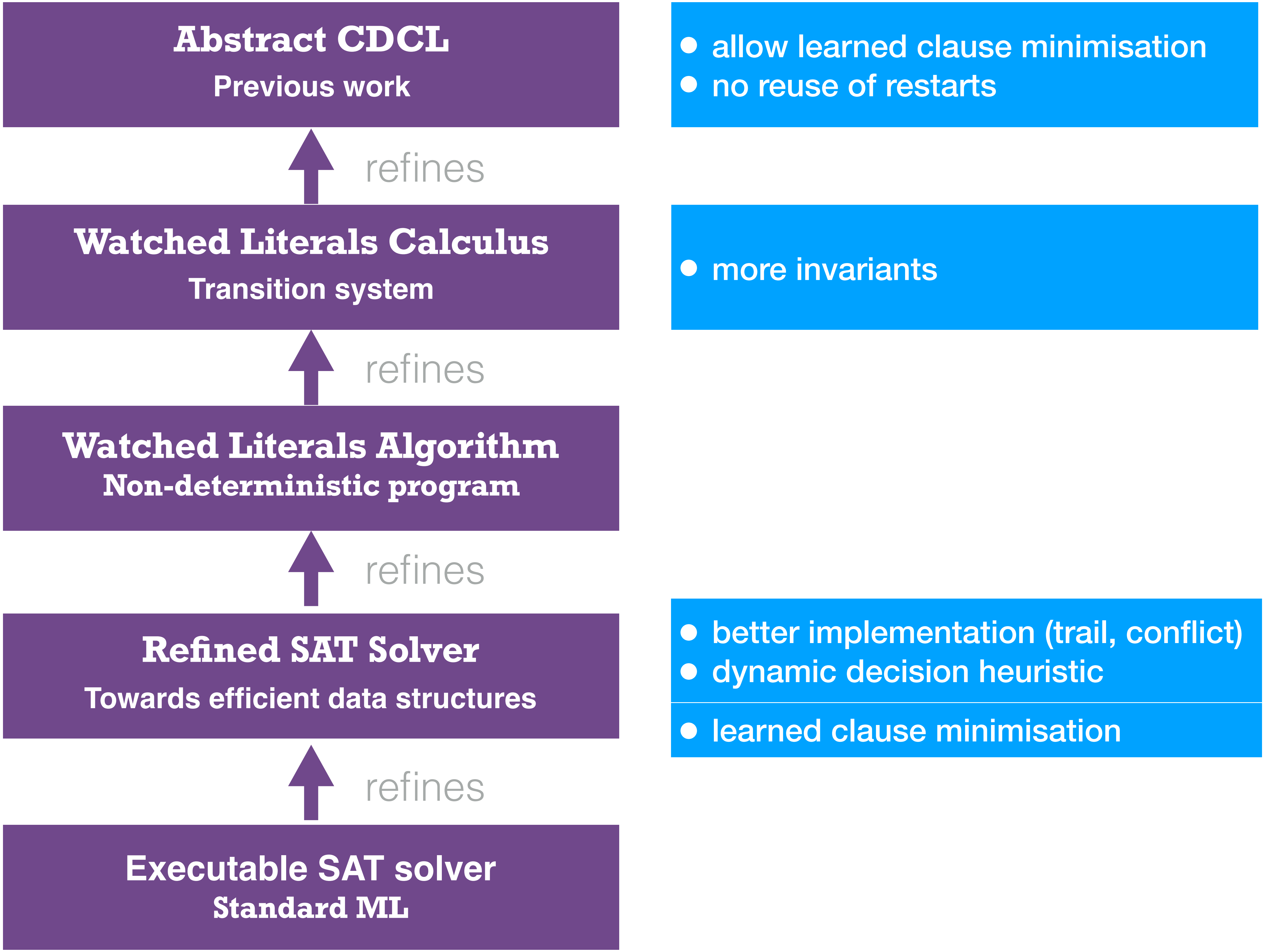
- better implementation (trail, conflict)
- dynamic decision heuristic

- learned clause minimisation

refines

**Executable SAT solver**

**Standard ML**



**Abstract CDCL**
Previous work

- allow learned clause minimisation
- no reuse of restarts

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

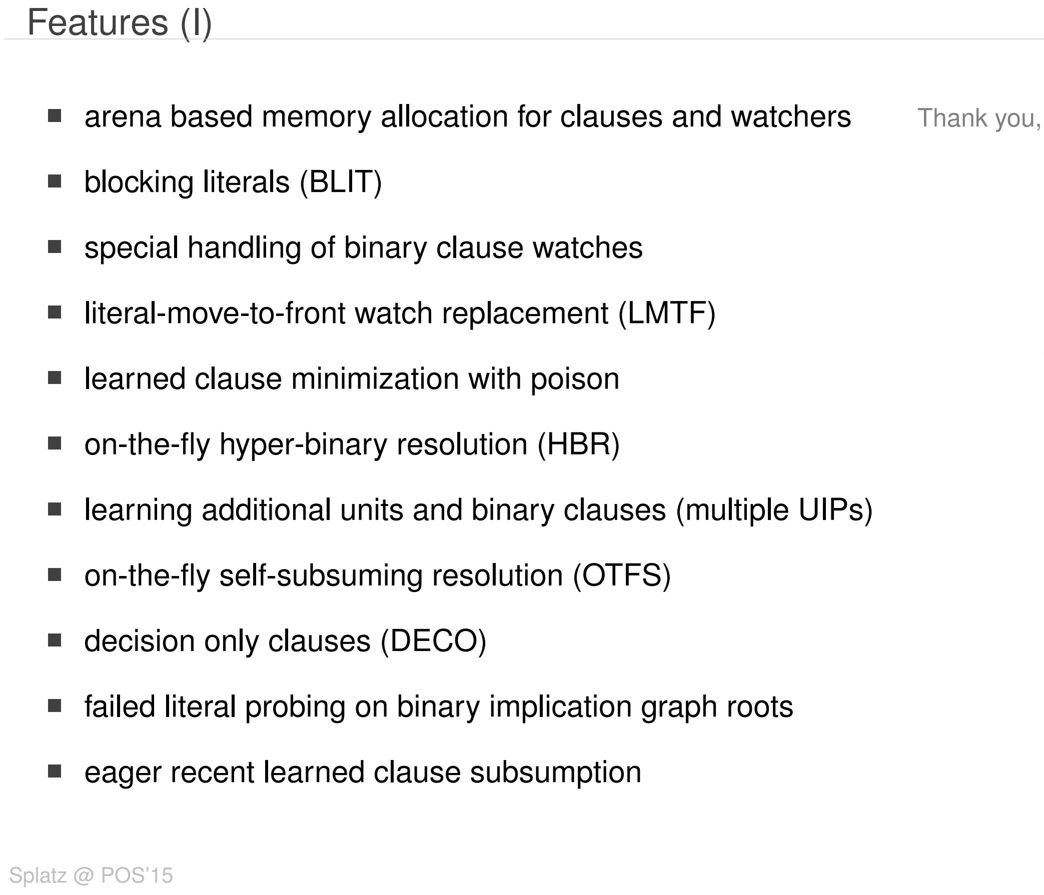
refines

**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) |
| **IsaSAT** | 1 page<br>(C++ code of MiniSat) | 900 pages<br>(2 years) |

# How much is missing?

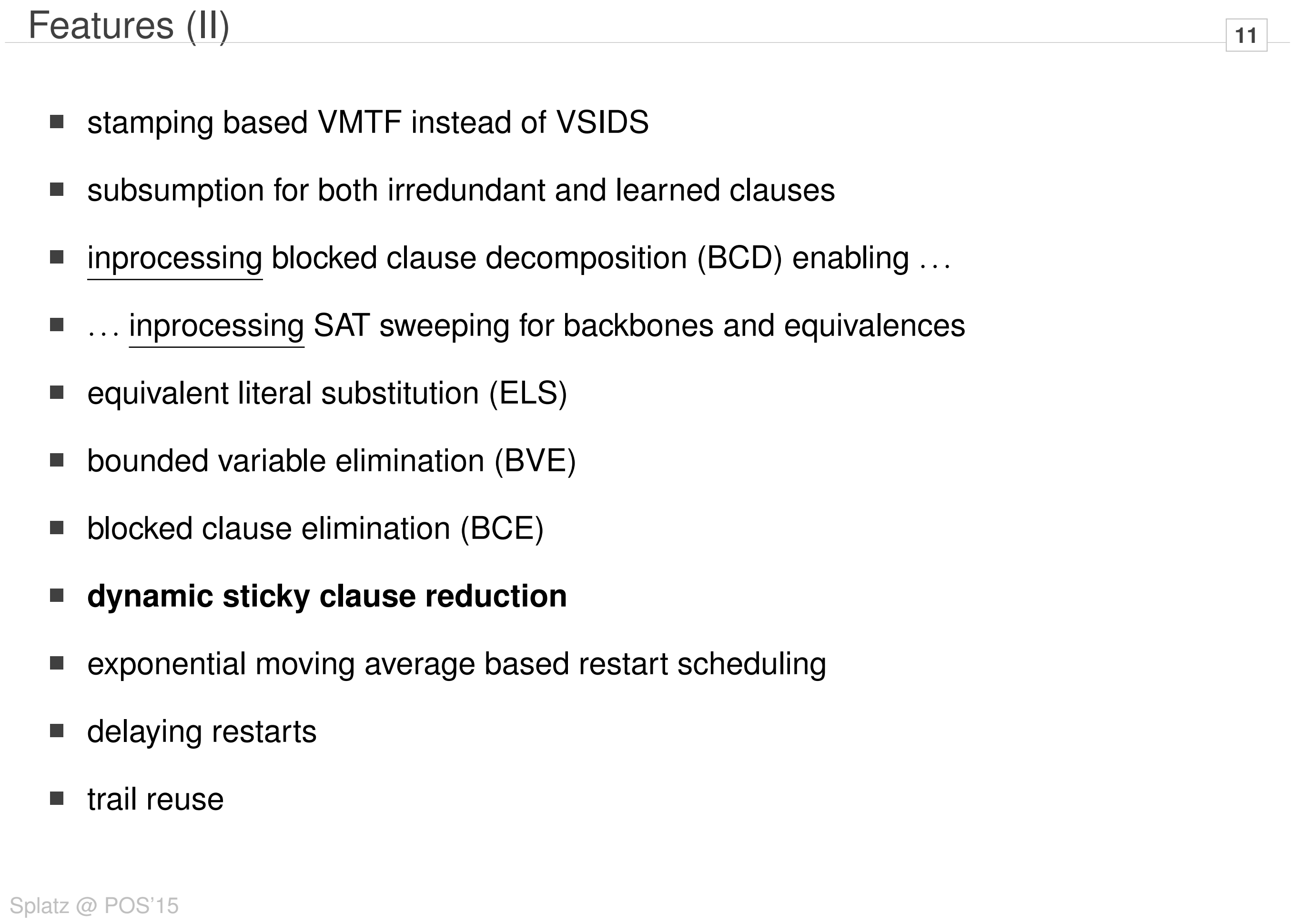


## Features (I)

Thank you, Norbert & Mate!

- 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

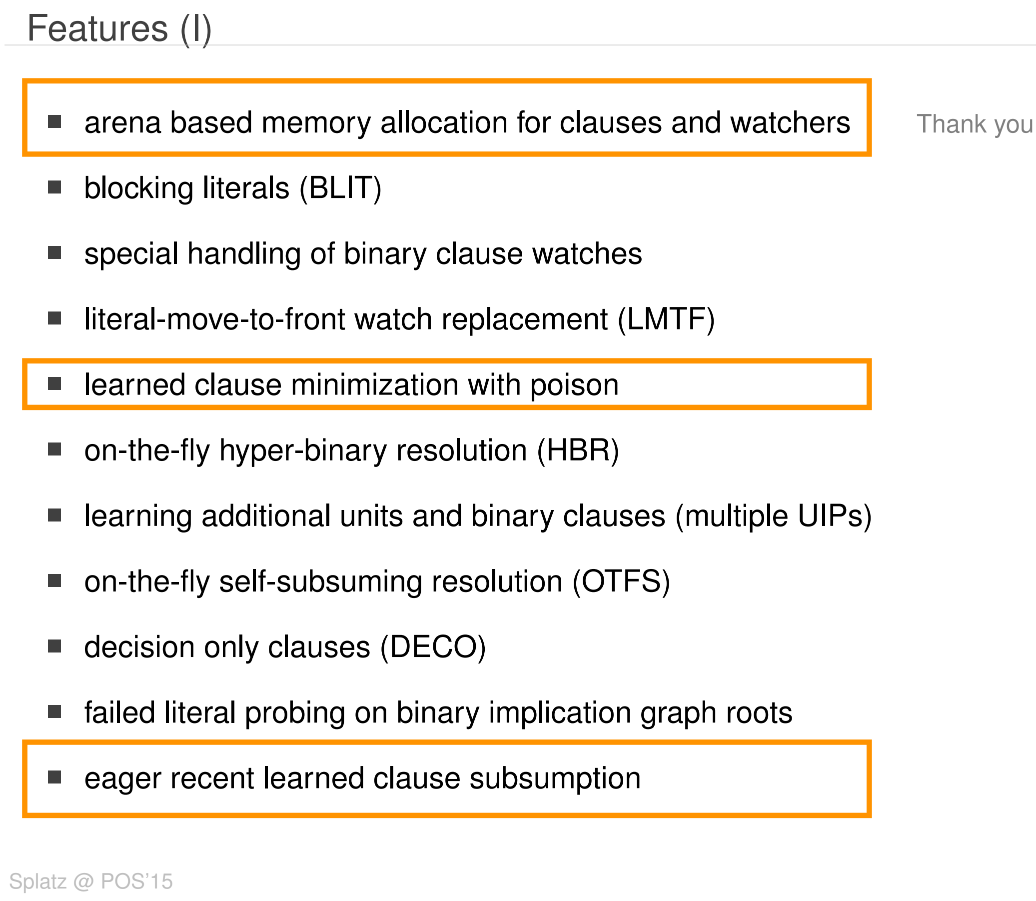
**Slides by Armin Biere**



## Features (II)

- 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

# How much is missing?



## Features (I)

10

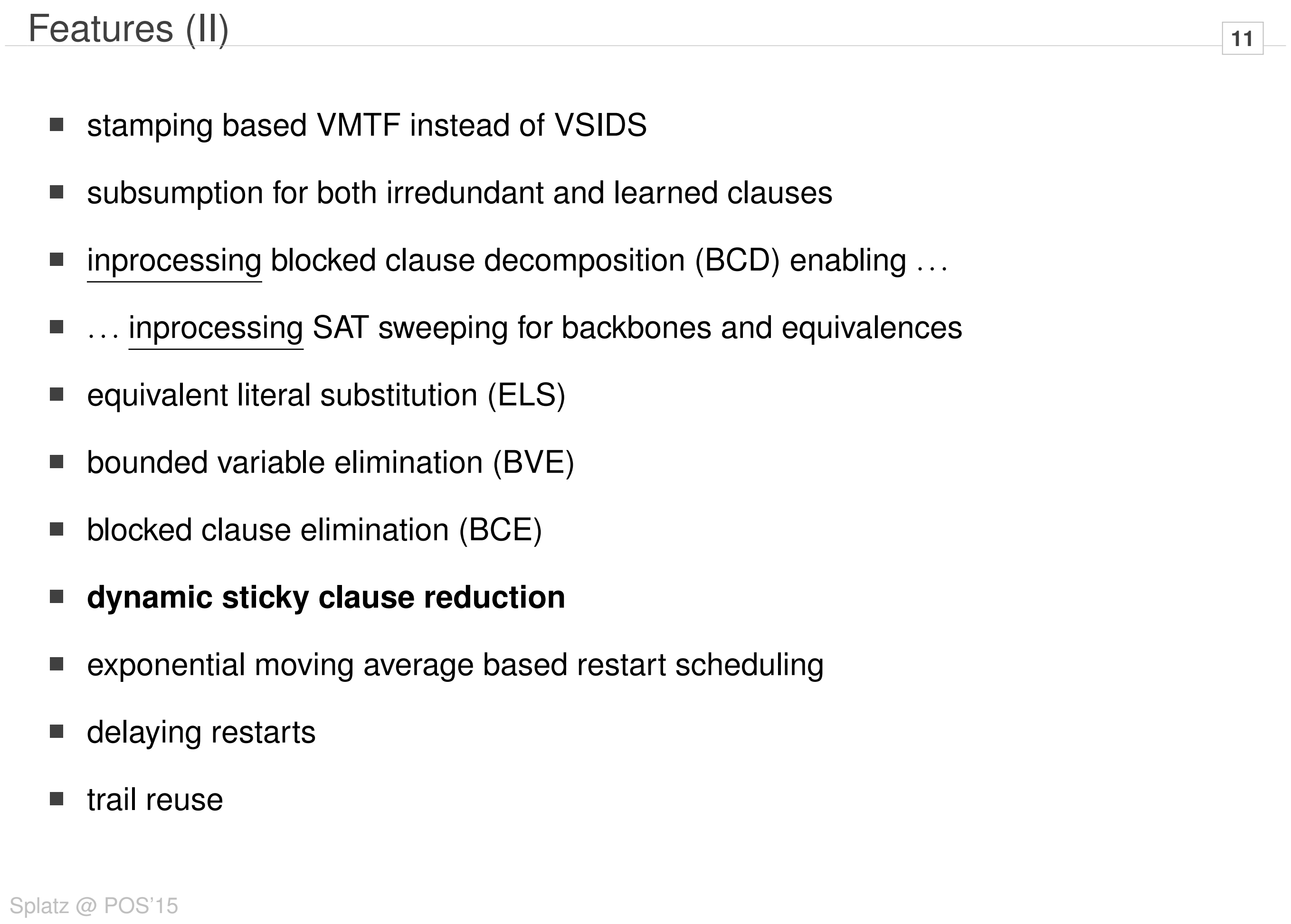
- arena based memory allocation for clauses and watchers
- blocking literals (BLIT)
- special handling of binary clause watches
- literal-move-to-front watch replacement (LMTF)
- learned clause minimization with poison
- on-the-fly hyper-binary resolution (HBR)
- learning additional units and binary 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

*Code only*

Thank you, Norbert & Mate!

**Slides by Armin Biere**



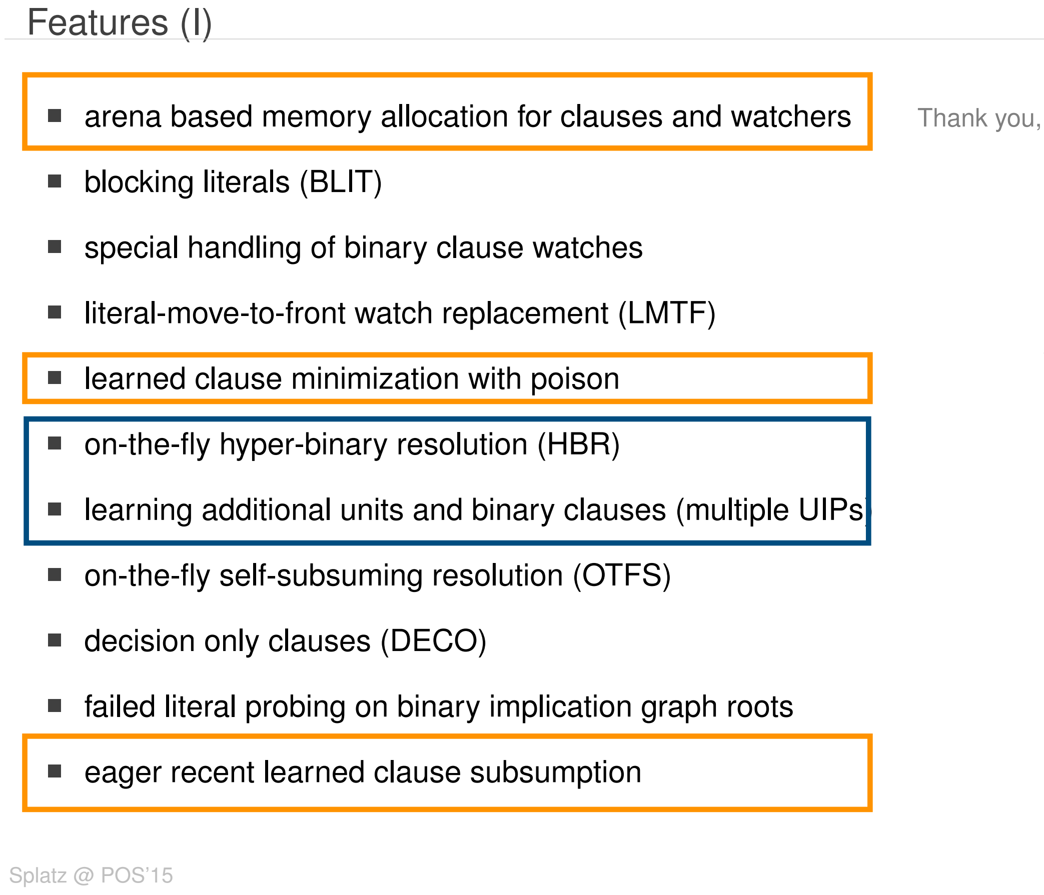
## Features (II)

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

# How much is missing?



## Features (I)

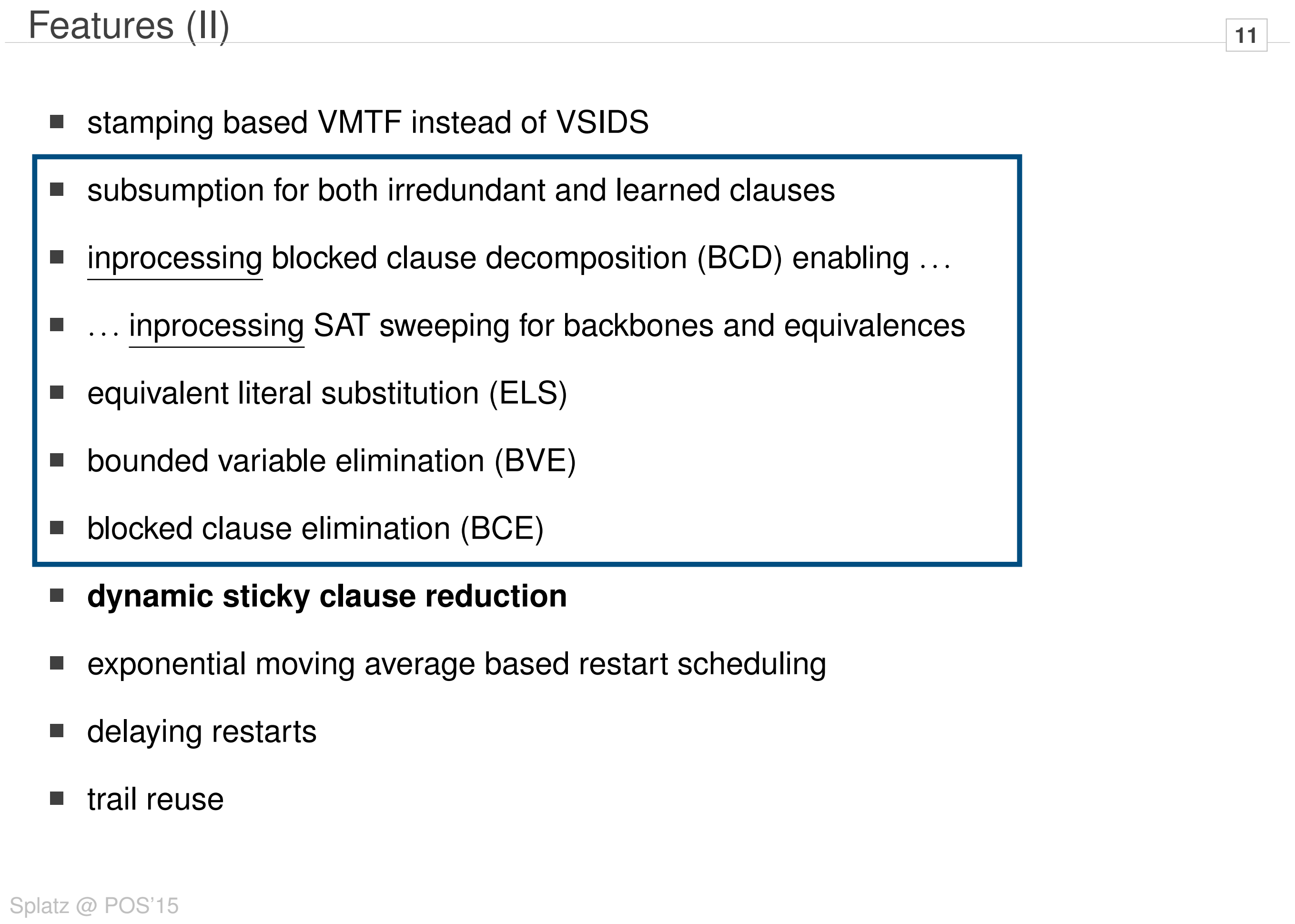
10

- 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*

*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

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*

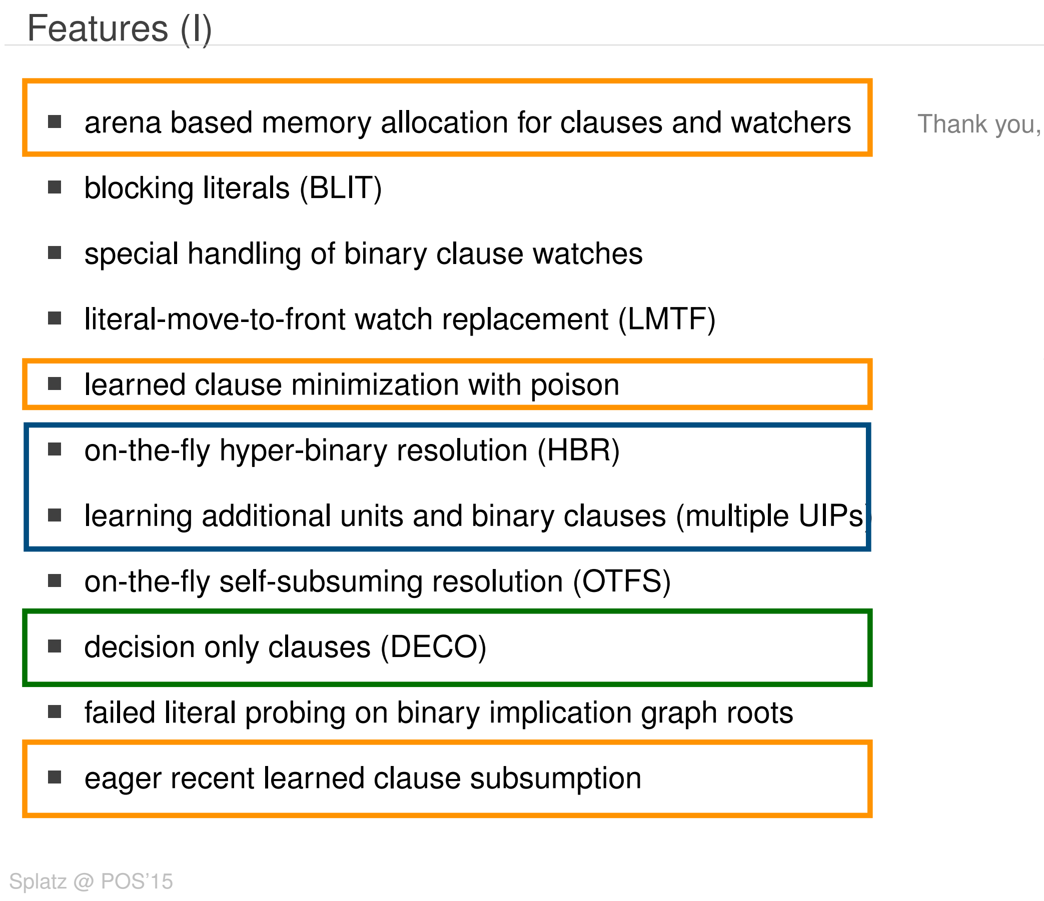
*Strengthening*

*Change CDCL*

# How much is missing?

**Slides by Armin Biere**

Thank you, Norbert & Mate!

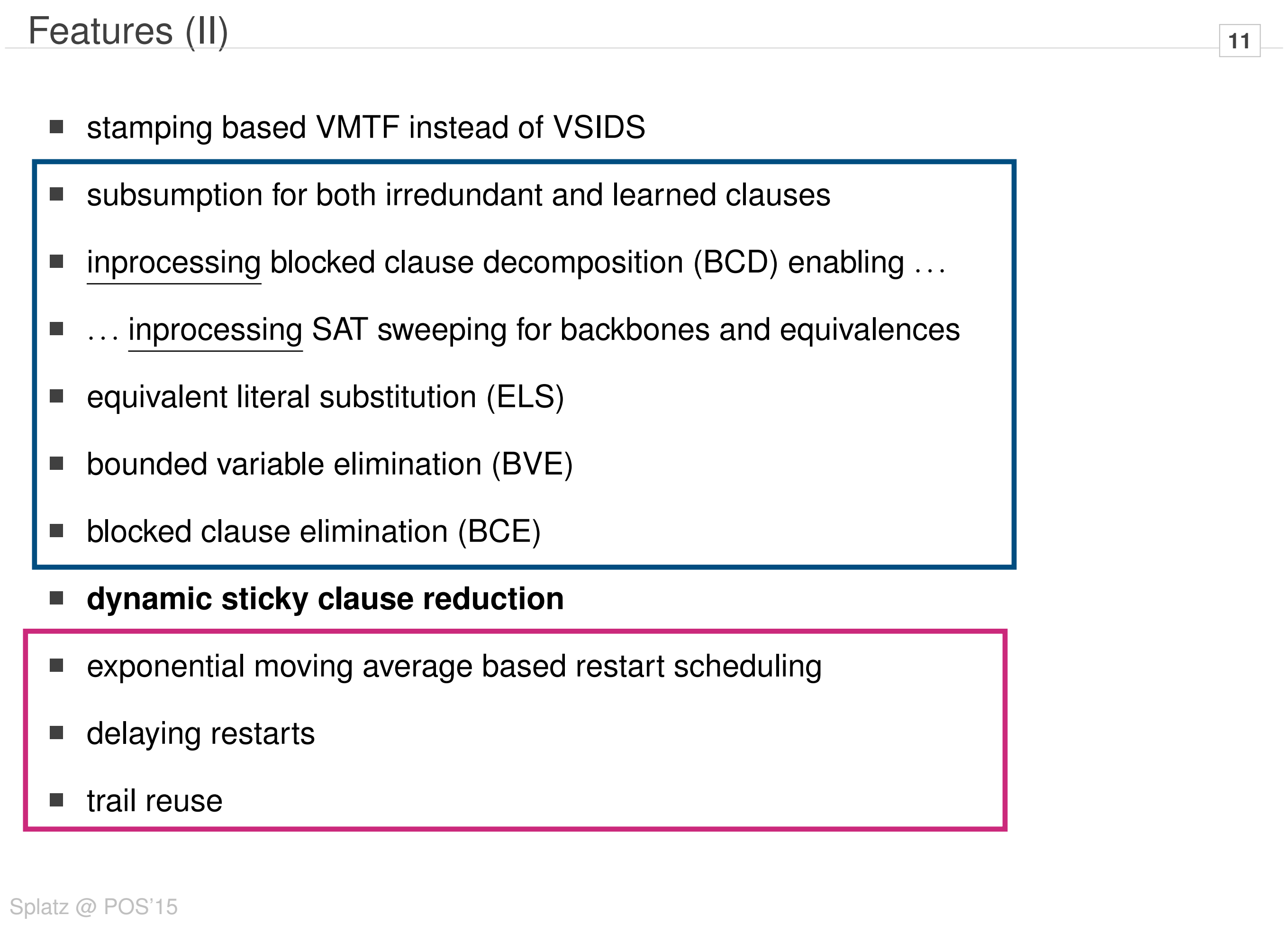


## Features (I)

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



## 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

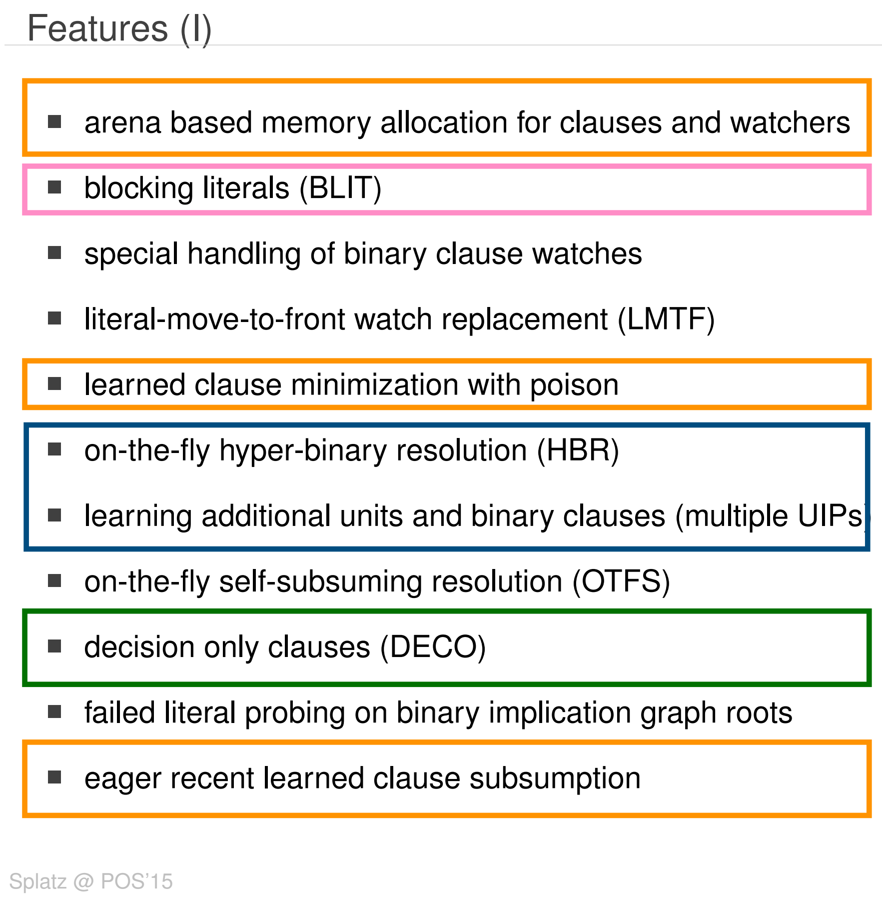
*Code only*

*Restarts (future)*

*Strengthening*

*Change CDCL*

# How much is missing?



## Features (I)

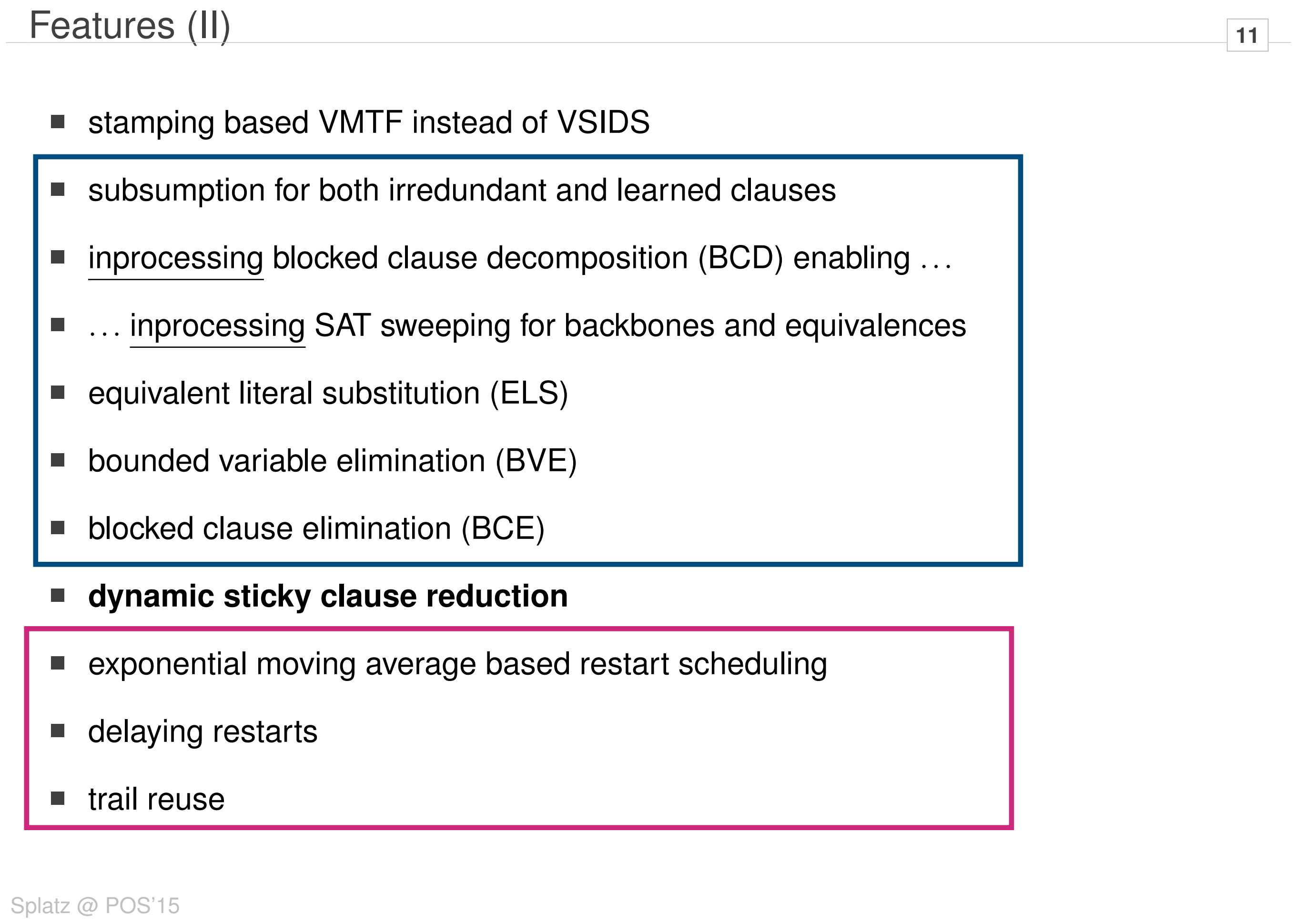
10

- 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

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*

*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

Thank you, Norbert & Mate!

Splatz @ POS'15

**Slides by Armin Biere**

Features (II)

11

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

- **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*

# What is under the carpet? (I)

```
code_printing constant nth_u_code' ⇀ (SML) "(fn/ ()/ =>/ Array.sub/ ((_),/ Word32.toInt (_)))"

code_printing constant nth_u64_code' ⇀ (SML) "(fn/ ()/ =>/ Array.sub/ ((_),/ Uint64.toFixedInt (_)))"

code_printing constant heap_array_set'_u' ⇀
  (SML) "(fn/ ()/ =>/ Array.update/ ((_),/ (Word32.toInt (_)),/ (_)))"

code_printing constant heap_array_set'_u64' ⇀
  (SML) "(fn/ ()/ =>/ Array.update/ ((_),/ (Word64.toInt (_)),/ (_)))"

code_printing constant two_uint32 ⇀ (SML) "(Word32.fromInt 2)"

code_printing constant length_u_code' ⇀ (SML_imp) "(fn/ ()/ =>/ Word32.fromInt (Array.length (_)))"

code_printing constant length_aa_u_code' ⇀ (SML_imp)
  "(fn/ ()/ =>/ Word32.fromInt (Array.length (Array.sub/ ((fn/ (a,b)/ =>/ a) (_),/
     IntInf.toInt (integer'_of'_nat (_))))))”

code_printing constant nth_raa_i_u64' ⇀ (SML_imp)
  "(fn/ ()/ =>/ Array.sub (Array.sub/ ((fn/ (a,b)/ =>/ a) (_),/
      IntInf.toInt (integer'_of'_nat (_))), Uint64.toFixedInt (_)))"

code_printing constant length_u64_code' ⇀ (SML_imp) "(fn/ ()/ =>/ Uint64.fromFixedInt (Array.length (_)))"

code_printing constant arl_get_u ⇀ (SML) "(fn/ ()/ =>/ Array.sub/ ((fn/ (a,b)/ =>/ a) (_),/ Word32.toInt (_)))"
```

# What is under the carpet? (I)

```
)/ =>/ Array.sub/ ((_),/ Word32.toInt (_)))"

n/ ()/ =>/ Array.sub/ ((_),/ Uint64.toFixedInt (_)))"


Int (_)),/ (_)))"


Int (_)),/ (_)))"

l32.fromInt 2)"

p) "(fn/ ()/ =>/ Word32.fromInt (Array.length (_)))"
```

VRIJE
UNIVERSITEIT
AMSTERDAM

max planck institut
informatik

# What is under the carpet? (II)

```
lemma append_aa_hnr[sepref_fr_rules]:
  fixes R ::  ‹'a ⇒ 'b :: {heap, default} ⇒ assn›
  assumes p: ‹is_pure R›
  shows
    ‹(uncurry2 append_el_aa, uncurry2 (RETURN ∘∘∘ append_ll)) ∈
     [λ((l,i),x). i < length l]a (arrayO_assn (arl_assn R))d *a nat_assnk *a Rk → (arrayO_assn (arl_assn R))›
proof -
  obtain R' where R: ‹the_pure R = R'› and R': ‹R = pure R'›
    using p by fastforce
  have [simp]: ‹(∃Ax. arrayO_assn (arl_assn R) a ai * R x r * true * ↑ (x = a ! ba ! b)) =
     (arrayO_assn (arl_assn R) a ai * R (a ! ba ! b) r * true)› for a ai ba b r
    by (auto simp: ex_assn_def)
  show ?thesis — ‹TODO tune proof›
    apply sepref_to_hoare
    apply (sep_auto simp: append_el_aa_def)
     apply (simp add: arrayO_except_assn_def)
     apply (rule sep_auto_is_stupid[OF p])
    apply (sep_auto simp: array_assn_def is_array_def append_ll_def)
    apply (simp add: arrayO_except_assn_array0[symmetric] arrayO_except_assn_def)
    apply (subst_tac (2) i = ba in heap_list_all_nth_remove1)
     apply (solves ‹simp›)
    apply (simp add: array_assn_def is_array_def)
    apply (rule_tac x=‹p[ba := (ab, bc)]› in ent_ex_postI)
    apply (subst_tac (2)xs'=a and ys'=p in heap_list_all_nth_cong)
      apply (solves ‹auto›)[2]
    apply (auto simp: star_aci)
    done
qed
```

# Conclusion

## Concrete outcome

- Watched literals optimisation
- Verified executable SAT solver

## Methodology

- Refinement using the Refinement Framework
- No proof of heuristics (w.r.t. standard)

## Future work

- Restarts (ongoing)
- Use SAT solver in IsaFoR