An Introduction to Satisfiability Modulo Theories

Philipp Rümmer

University of Regensburg
Uppsala University

February 7, 2024

Outline

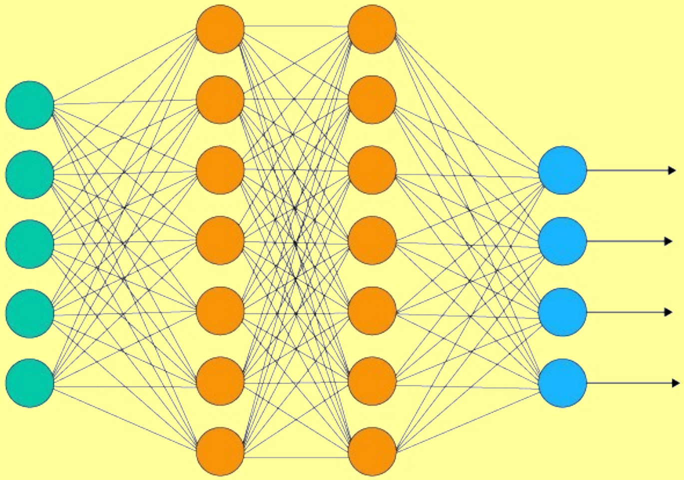
- From theory …
  - From DPLL to DPLL(T)
  - Slides courtesy of Alberto Griggio
- … to practice
  - SMT-LIB and some common theories
  - https://microsoft.github.io/z3guide/
  - https://cvc5.github.io/app/
  - https://eldarica.org/princess/

Typical Applications of SMT

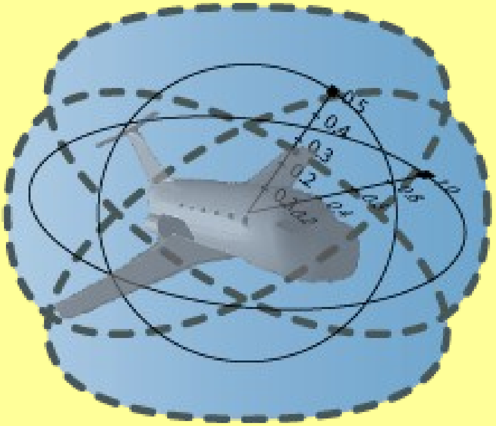
- Deductive program verification
    - Correctness of contracts, invariants
- Testing, symbolic execution
    - Path feasibility
- Bounded model checking
    - Reachability of errors within *k* steps
- Model checking
    - Computation of finite-state abstraction of programs

Broader Applications

$$
\begin{gathered}
(x > 3.0 \vee y < 2.0) \wedge \\
(x = y \vee x \neq y - 1.0) \wedge \\
y < 1.0
\end{gathered}
$$



```
i = 0;
x = j;
while (i < 50) {
    i++;
    x++;
}
if (j == 0)
    assert (x >= 50);
```



SAT and SMT

Def.: SAT Solver

Input: **Propositional** formula *C* in *n* variables

Output: ***C* sat** + satisfying assignment (model)
***C* unsat** [+ Proof]

Def.: SAT Modulo Theories Solver

Input: **First-order** formula *C* in *n* variables and **theories** $T_1, \ldots, T_m$

Output: ***C* sat** + satisfying assignment (model)
***C* unsat** [+ Proof]

SAT and SMT

Def.: SAT Solver

Input: **Propositional** formula *C* in *n* variables

Output: ***C* sat** + satisfying assignment (model)
***C* unsat** [+ Proof]

Also called a **solution**

Def.: SAT Modulo Theories Solver

Input: **First-order** formula *C* in *n* variables and **theories** $T_1, \ldots, T_m$

Output: ***C* sat** + satisfying assignment (model)
***C* unsat** [+ Proof]

Theories

Definition (theory)
A (first-order) theory T is specified by a signature Σ_T of operations (sorts, functions, predicates), and a class $\mathcal{S}_T$ of intended interpretations of the symbols in Σ_T.

- A theory is like a library:
  - Data-types
  - Operations on those data-types
- Various examples later

We know how to …

Solve **Boolean formulas** efficiently:

- DPLL, CDLL
- Implemented in SAT solvers

Solve **theory constraints** efficiently:

- Linear arithmetic: LP, ILP, MIP
- Finite domains: CP, local search
- *etc.*

We know how to ... ???

Solve **Boolean formulas** efficiently:

- DPLL, CDLL
- Implemented in SAT solvers

\+

Solve **theory constraints** efficiently:

- Linear arithmetic: LP, ILP, MIP
- Finite domains: CP, local search
- *etc.*

Example!

- **How can we solve this formula?**

Eager SMT

- Wide range of data-types can directly be encoded in propositional logic:
  - Bit-vectors/machine arithmetic
  - Equality logic
  - Integer arithmetic *(how?)*

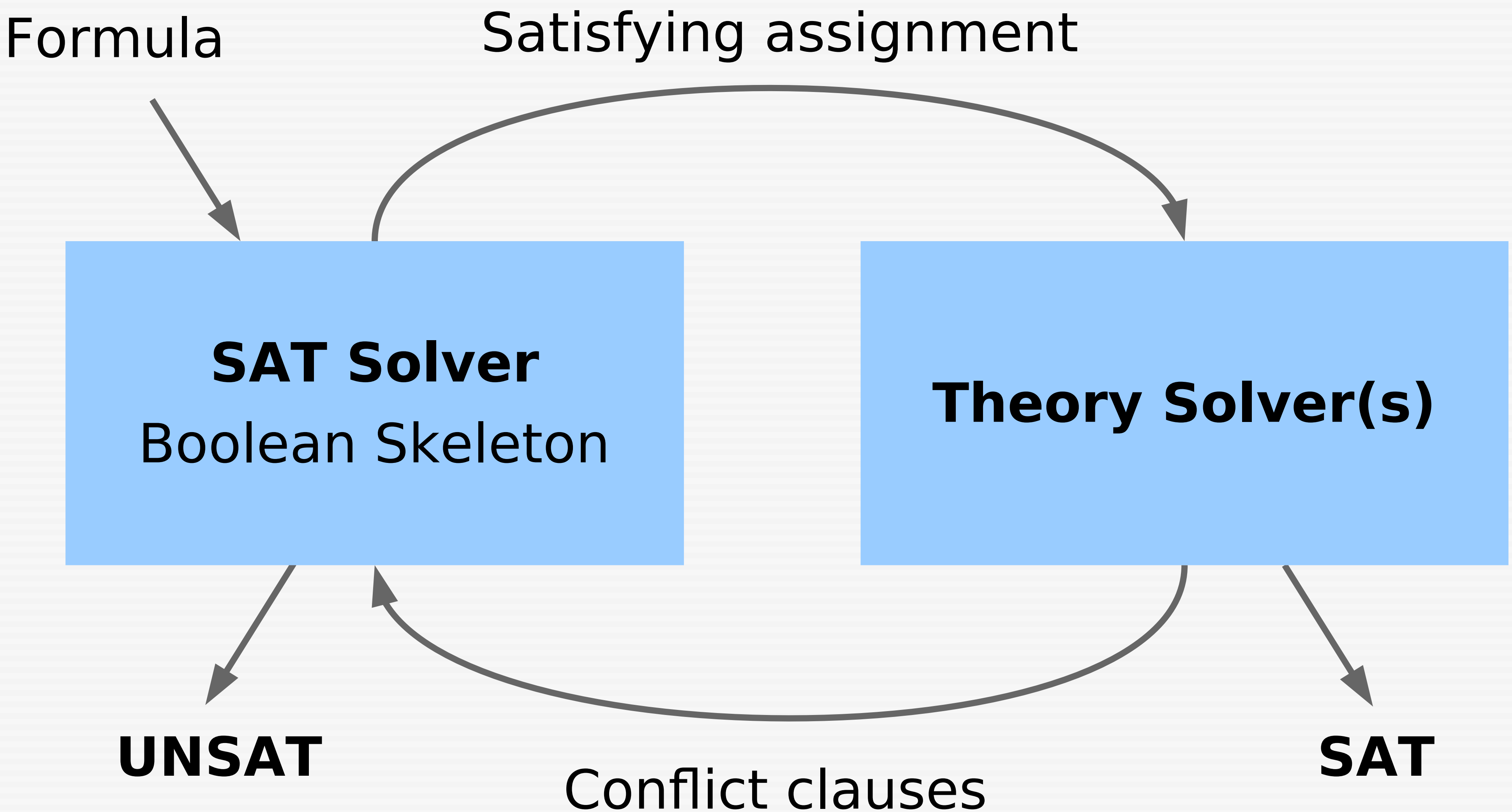
- Approach pioneered by UCLID (2004)
- Today mostly used for bit-vectors

Lazy, Offline SMT

- Construct a **Boolean** skeleton of a formula, and solve it using SAT

- **UNSAT** → Finished!
- **SAT** → Check consistency of assigned theory literals
  → Produce a **model** or **refine skeleton**

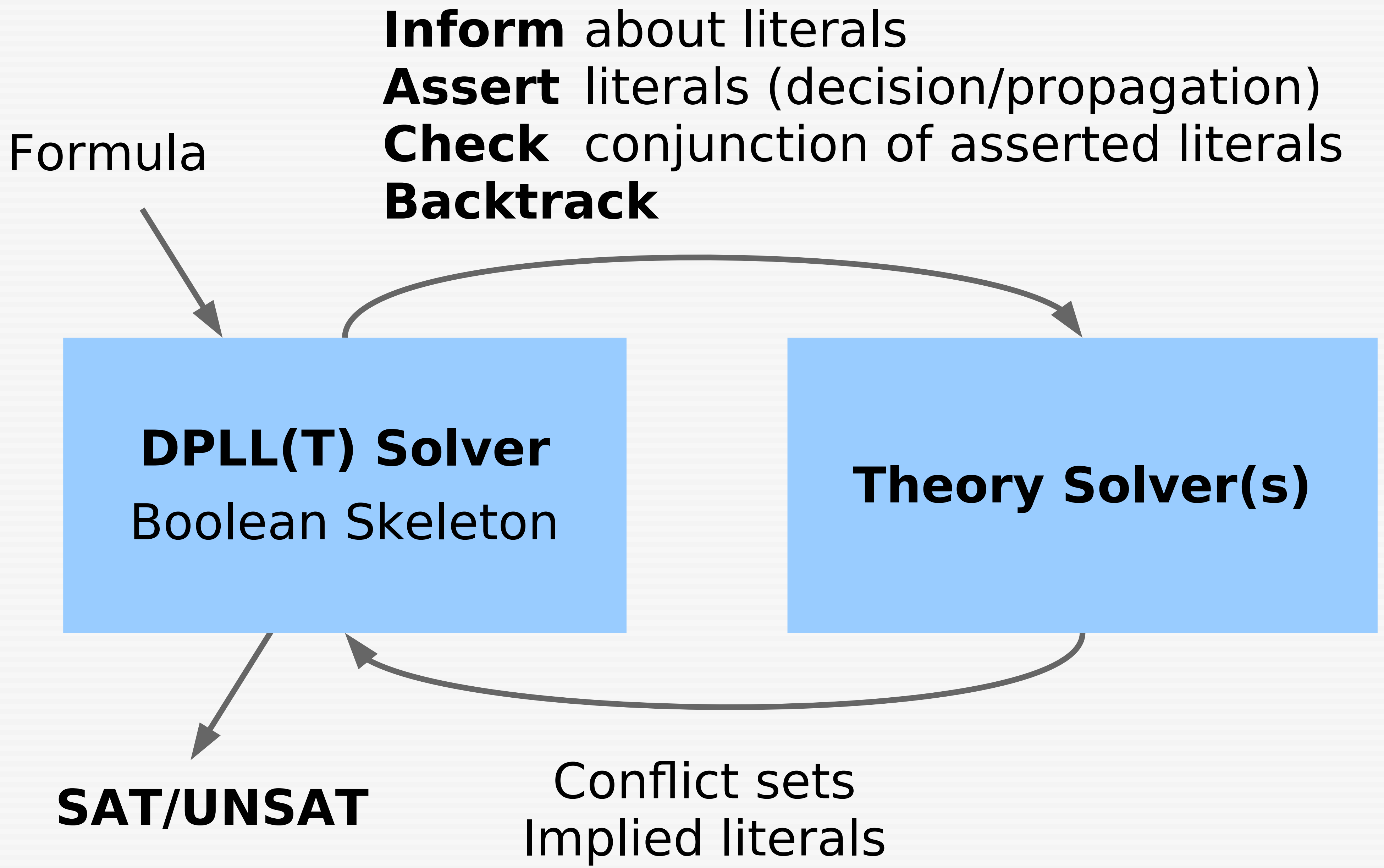
Lazy, Offline SMT



Lazy, **Online** SMT

- Tightly interleave/integrate **Boolean** and **theory** reasoning
  - SAT solver informs theory solvers each time a literal is asserted
    → **incremental** theory solving
  - Theory solver informs SAT solver when there is a **conflict**
  - + *Some further refinements*
- Formalised in the **DPLL(T)** algorithm
  [Nieuwenhuis, Oliveras, Tinelli, 2006]

The DPLL(T) Loop

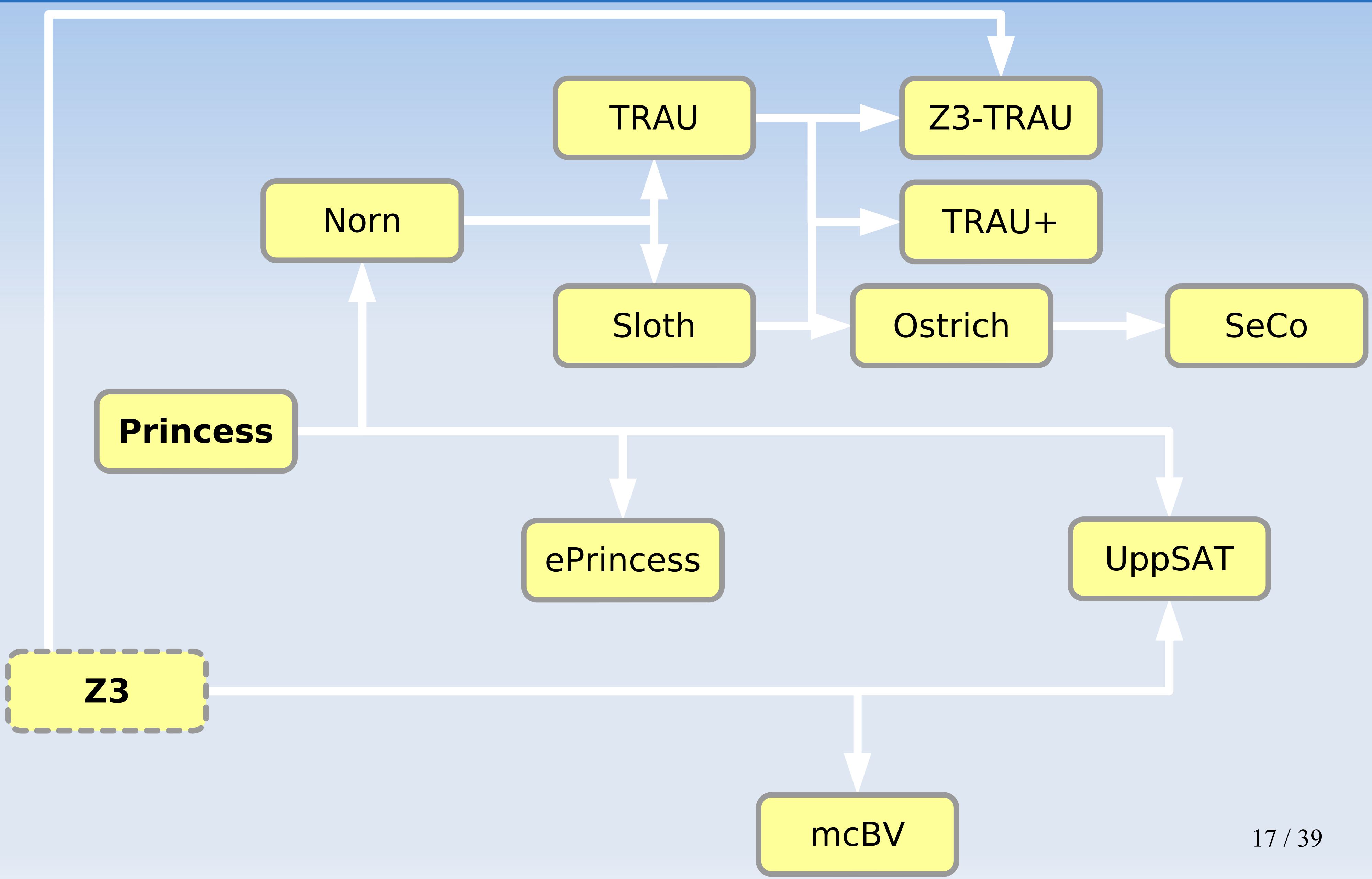


On to the other slide set ...

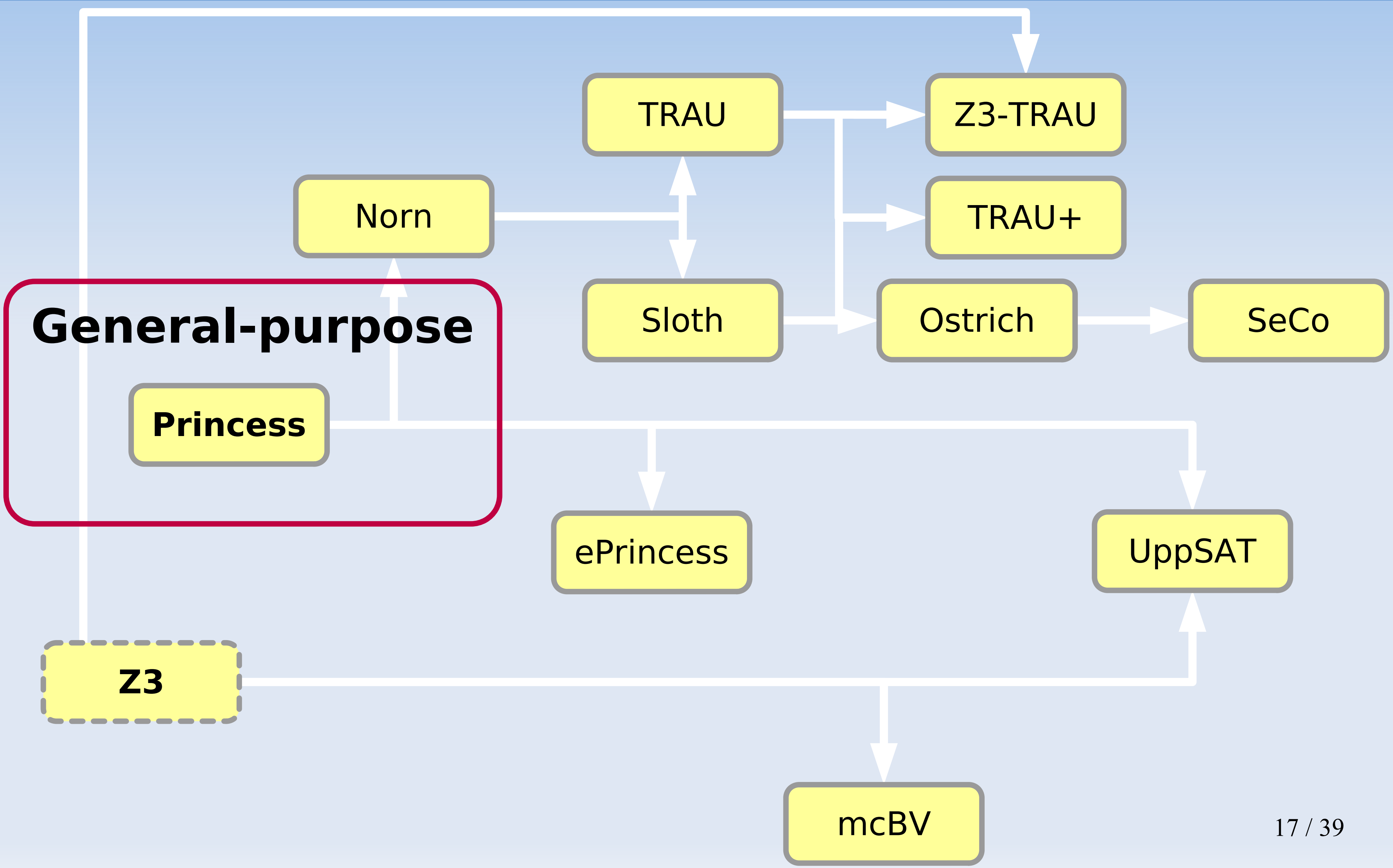
Some SMT solvers

- Z3
- cvc5
- MathSAT
- Yices
- OpenSMT
- Bitwuzla
- SMTInterpol

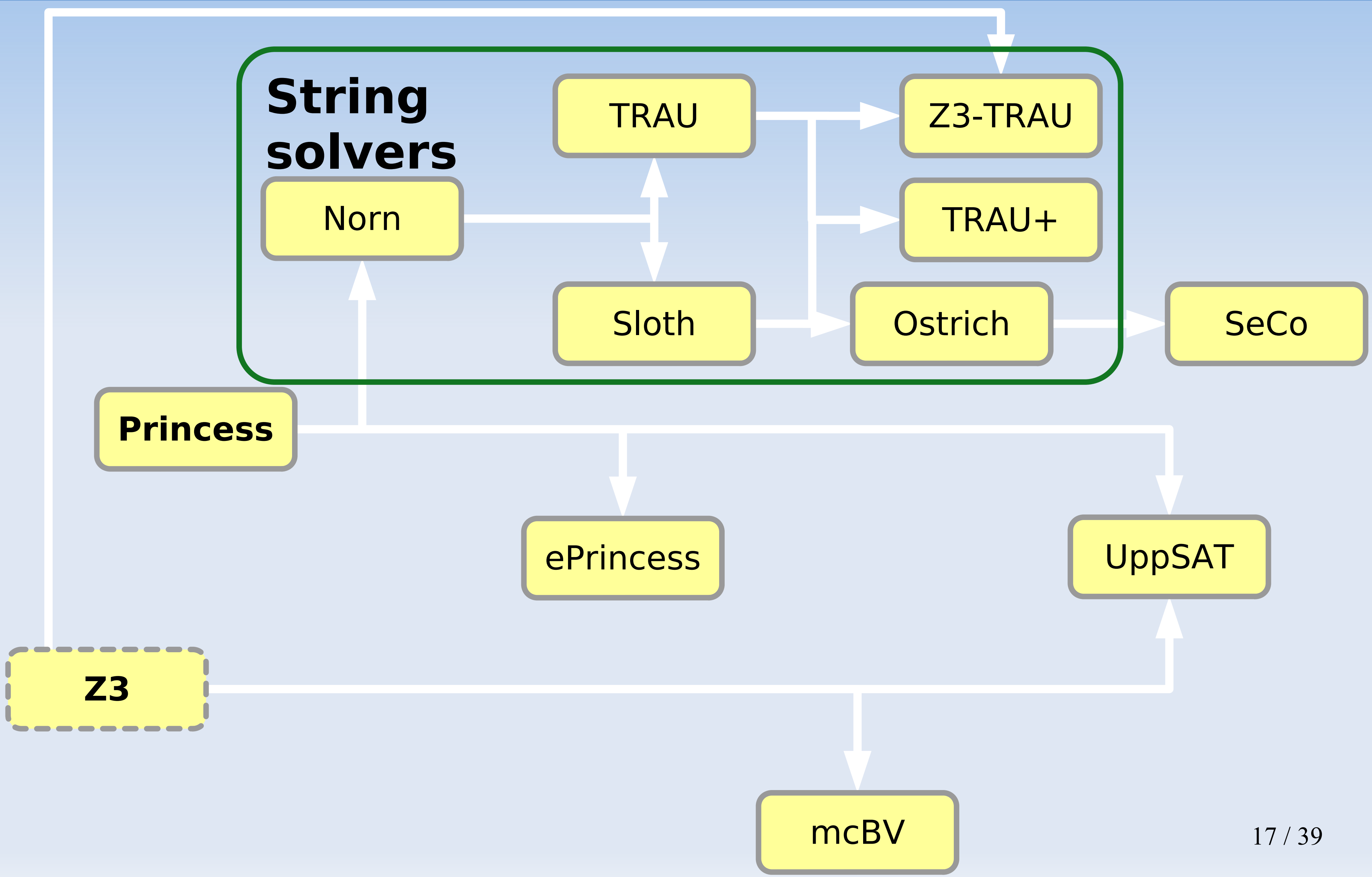
SMT in Uppsala / Regensburg



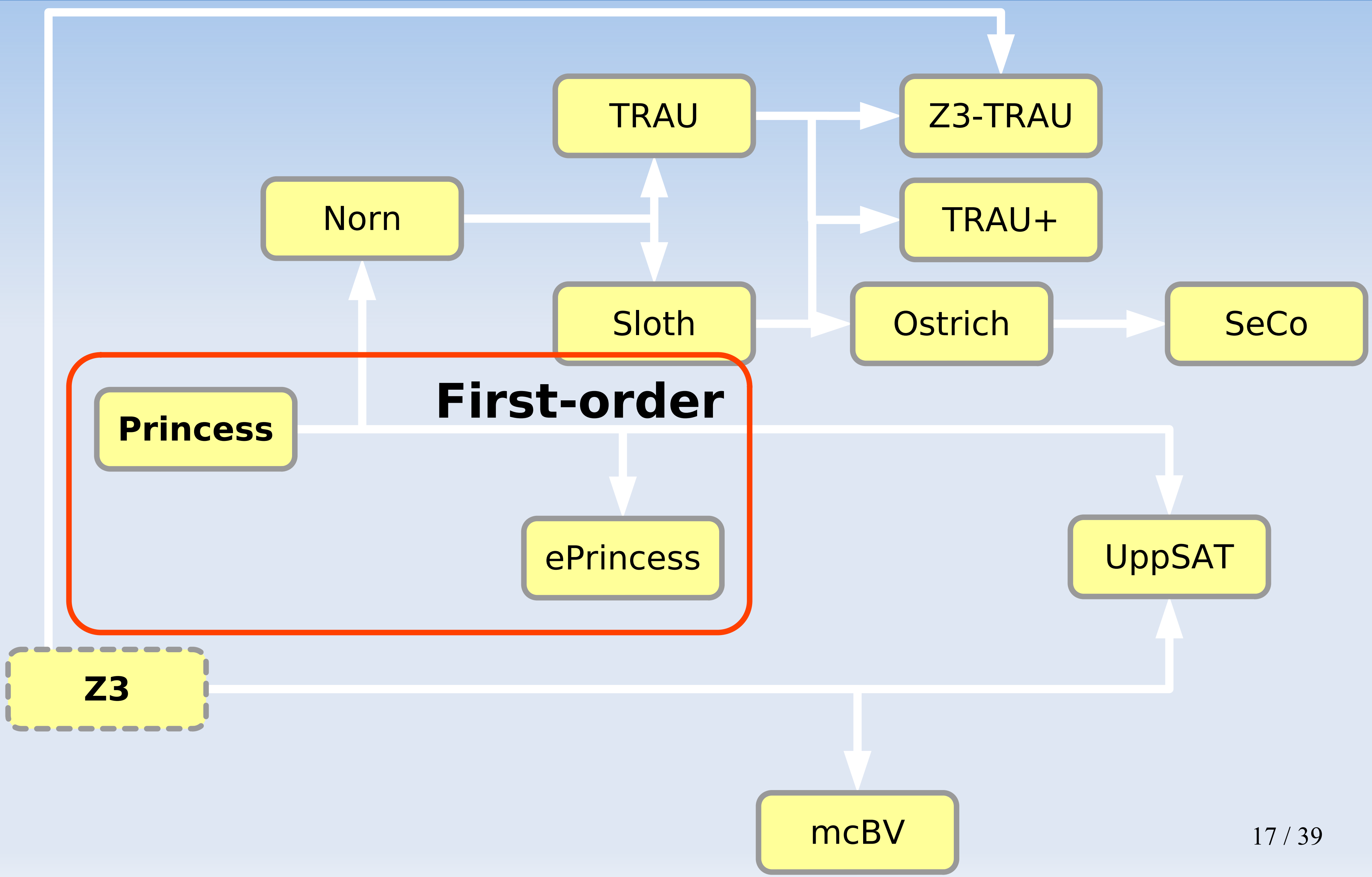
SMT in Uppsala / Regensburg



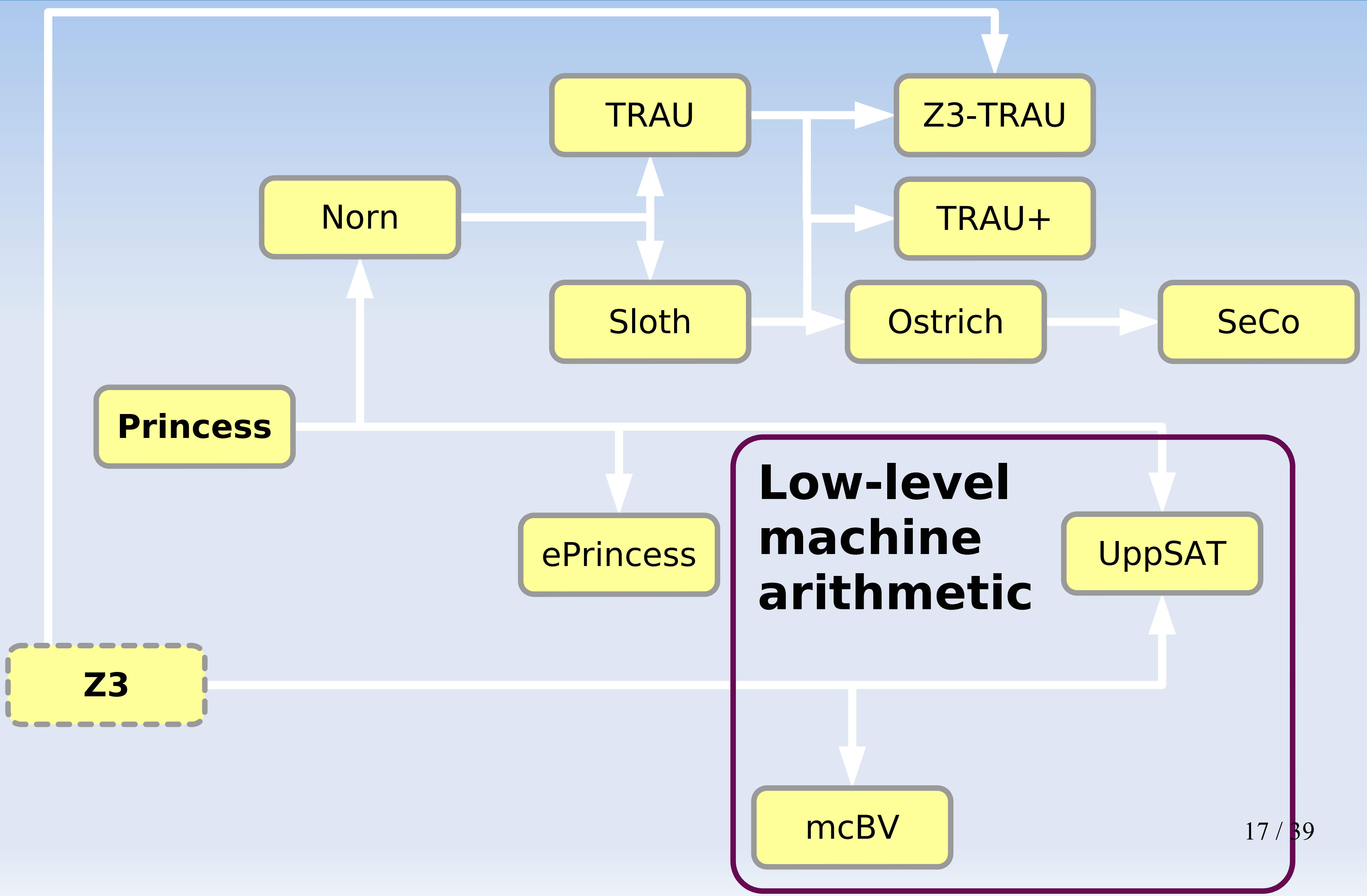
SMT in Uppsala / Regensburg



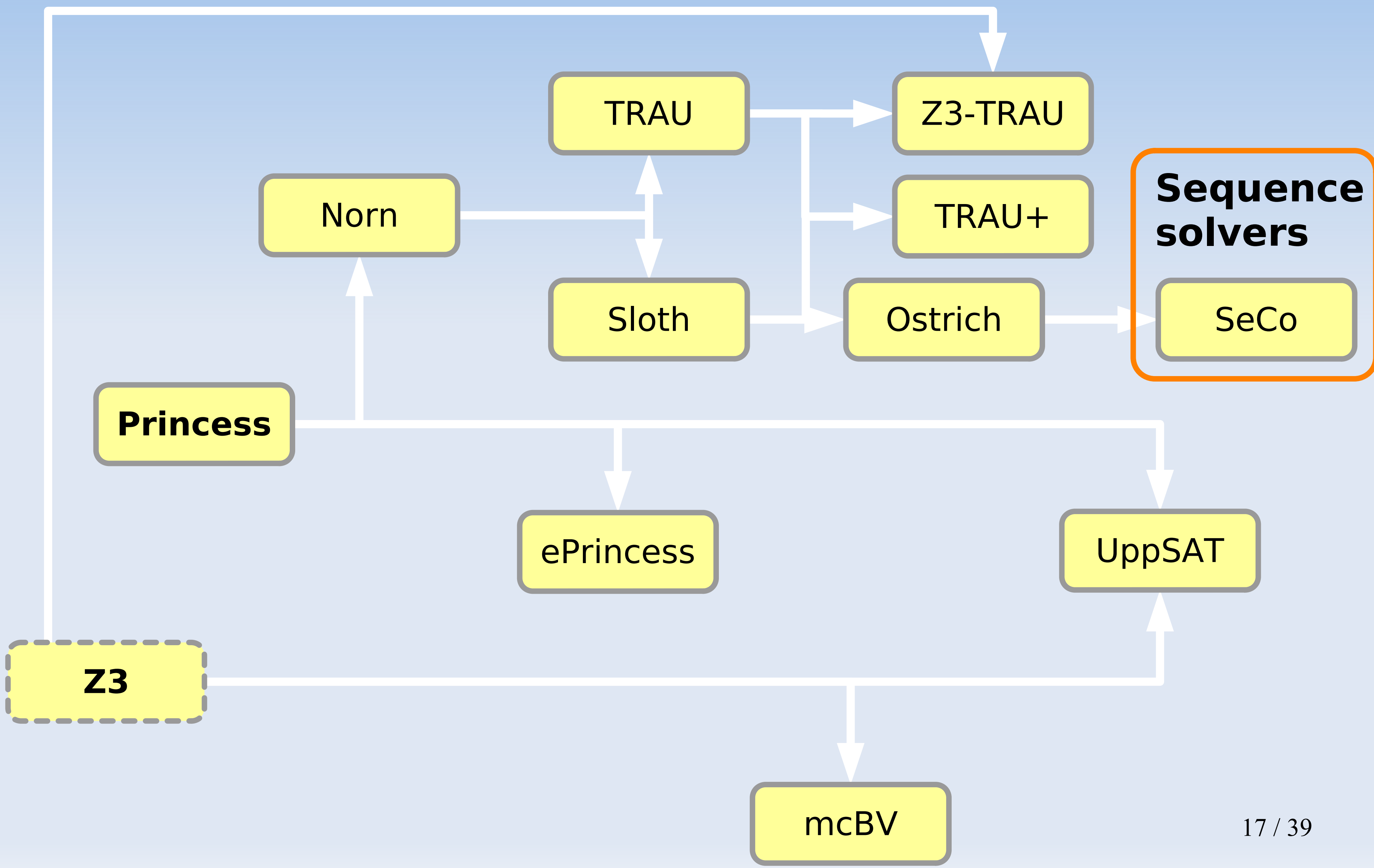
SMT in Uppsala / Regensburg



SMT in Uppsala / Regensburg



SMT in Uppsala / Regensburg



The SMT-LIB Standard

Version 2.6

Clark Barrett Pascal Fontaine Cesare Tinelli

Release: 2017-07-18

SMT-LIB

- Standardised interface for SMT solvers, supported by most tools
- Rich set of features, many theories
- Comes with a large library of benchmarks; annual competition SMT-COMP
- http://www.smtlib.org

Example 1

$$
\begin{aligned}
2y - z &> 2 \vee p \\
3x - z &> 6 \vee \neg p \\
2z - 4y &> 5 \vee p \\
y - z &\not> 6 \vee \neg p
\end{aligned}
$$

In SMT-LIB

```
(set-logic QF_LIA)

(declare-const p Bool)
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(assert (or (> (- (* 2 y) z) 2)       p))
(assert (or (> (- (* 3 x) z) 6)       (not p)))
(assert (or (> (- (* 2 z) (* 4 y)) 5) p))
(assert (or (not (> (- y z) 6))       (not p)))

(check-sat)
(get-model)
```

Permalink:
https://eldarica.org/princess/?ex=perma%2F1644243468_187318180

Important SMT-LIB commands

- `(set-logic QF_BV)`
  `(set-option …)`
- `(declare-const b (_ BitVec 8))`
  `(declare-fun f ((x (_ BitVec 2))) Bool)`
- `(assert (= b #b10100011))`
- `(check-sat)`
- `(get-value (b)), (get-model)`
- `(get-unsat-core)`
- `(push 1), (pop 1)`
- `(reset), (exit)`

Important SMT-LIB commands

- `(set-logic QF_BV)`
  `(set-option …)`

Z3, and many solvers don't care ...

- `(declare-const b (_ BitVec 8))`
  `(declare-fun f ((x (_ BitVec 2))) Bool)`
- `(assert (= b #b10100011))`
- `(check-sat)`
- `(get-value (b))`, `(get-model)`
- `(get-unsat-core)`
- `(push 1)`, `(pop 1)`
- `(reset)`, `(exit)`

Important SMT-LIB commands

- `(set-logic QF_BV)`
  `(set-option …)`
- `(declare-const b (_ BitVec 8))`
  `(declare-fun f ((x (_ BitVec 2))) Bool)`
- `(assert (= b #b10100011))`
- `(check-sat)`
- `(get-value (b)), (get-model)`
- `(get-unsat-core)`
- `(push 1), (pop 1)`
- `(reset), (exit)`

Z3, and many solvers don't care ...

In CP or MIP, this would be called `assume` or `constraint`

General SMT-LIB constructors

- `(and …), (or …), (not …), (=> …)`
- `(= b c)`
- `(ite (= b c) #b101 #b011)`
- `(let ((a #b001) (b #b010)) (= a b))`
- `(exists ((x (_ BitVec 2))) (= #b101 x))`
  `(forall …)`
- `(! (= b c) :named X)`

Example 2

Example 2

- Every 32bit number `x` that is a power of 2 has the property that
  `x & (x - 1) == 0`

  (and vice versa)

Quantifying Satisfaction?

- SAT/SMT solvers check **satisfiability**:

$$\phi[x, y, z] \text{ is sat} \quad \Leftrightarrow \quad \exists x, y.z.\ \phi[x, y, z] \text{ is sat}$$

- How to prove a universal property?

$$\forall x, y.z.\ \phi[x, y, z] \text{ is } \textbf{valid}?$$

Quantifying Satisfaction?

- SAT/SMT solvers check **satisfiability**:

$$\phi[x, y, z] \text{ is sat} \quad \Leftrightarrow \quad \exists x, y.z.\ \phi[x, y, z] \text{ is sat}$$

- How to prove a universal property?

$$\forall x, y.z.\ \phi[x, y, z] \text{ is } \textbf{valid}?$$

$$\begin{aligned} & \forall x, y.z.\ \phi[x, y, z] \text{ is } \textbf{valid} \\ \iff \quad & \neg\forall x, y.z.\ \phi[x, y, z] \text{ is } \textbf{unsat} \\ \iff \quad & \exists x, y.z.\ \neg\phi[x, y, z] \text{ is } \textbf{unsat} \\ \iff \quad & \neg\phi[x, y, z] \text{ is } \textbf{unsat} \end{aligned}$$

In SMT-LIB

```
(set-logic QF_BV)

(declare-const e (_ BitVec 32))
(declare-const x (_ BitVec 32))

(assert (= x (bvshl #x00000001 e)))
(assert (not (= (bvand x (bvsub x #x00000001)) #x00000000)))

(check-sat)
```

Permalink
https://eldarica.org/princess/?ex=perma%2F1644322914_843217432

Main SMT-LIB Bit-vector ops.

http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV

- `(_ BitVec 8)`
- `#b1010, #xff2a, (_ bv42 32)`
- `(= (concat #b1010 #b0011) #b10100011)`
- `(= ((_ extract 1 3) #b10100011) #b010)`
- *Unary:* `bvnot, bvneg`
- *Binary:* `bvand, bvor, bvadd, bvsub, bvmul,`
  `bvudiv, bvurem, bvshl, bvlshr`
- `(bvult #b0100 #b0110)`
- *And many more derived operators ...*

Example 3: Programs

```
int x, y;

x = x * x;
y = x + 1;

assert(y > 0);
```

Example 3: Programs

Z3-specific

```
int x, y;

x = x * x;
y = x + 1;

assert(y > 0);
```

```
(set-option :pp.bv-literals false)

(declare-const x0 (_ BitVec 32))
(declare-const y0 (_ BitVec 32))

(declare-const x1 (_ BitVec 32))
(declare-const y1 (_ BitVec 32))

(assert (= x1 (bvmul x0 x0)))
(assert (= y1 (bvadd x1 (_ bv1 32))))

(assert (not (bvsgt y1 (_ bv0 32))))

(check-sat)
(get-model)
```

Signed comparison

Example 3: Programs

```
int x, y;

x = x * x;
y = x + 1;

assert(y > 0);
```

Z3-specific

```
(set-option :pp.bv-literals false)

(declare-const x0 (_ BitVec 32))
(declare-const y0 (_ BitVec 32))

(declare-const x1 (_ BitVec 32))
(declare-const y1 (_ BitVec 32))

(assert (= x1 (bvmul x0 x0)))
(assert (= y1 (bvadd x1 (_ bv1 32))))

(assert (not (bvsgt y1 (_ bv0 32))))

(check-sat)
(get-model)
```

Signed comparison

Permalink:
https://eldarica.org/princess/?ex=perma%2F16442436

Modelling of Program Variables

- An SMT-LIB constant represents a **single** value
  - Just like mathematical variables
- Program variables can be reassigned … how to model computations?
- Main idea: every assignment creates a new “version” of a variable
  - `x0`/`y0` vs. `x1`/`y1` in example

Modelling of Program Variables

- An SMT-LIB constant represents a **single** value
  - Just like mathematical variables
- Program variables can be reassigned … how to model computations?

In compilers, this is called "Single Static Assignment" form (SSA)

- Main idea: every assignment creates a new "version" of a variable
  - `x0`/`y0` vs. `x1`/`y1` in example

Example 4: Conditionals

```
int x, y;

if (x > 0)
  y = x;
else
  y = -x;

assert(y >= 0);
```

Example 4: Conditionals

```
int x, y;

if (x > 0)
  y = x;
else
  y = -x;

assert(y >= 0);
```

```
(set-option :pp.bv-literals false)

(declare-const x0  (_ BitVec 32))
(declare-const y0  (_ BitVec 32))
(declare-const y1a (_ BitVec 32))
(declare-const y1b (_ BitVec 32))
(declare-const y2  (_ BitVec 32))
(declare-const b   Bool)

(assert (= b (bvsgt x0 (_ bv0 32))))
(assert (=> b       (= y1a x0)))
(assert (=> (not b) (= y1b (bvneg x0))))
(assert (= y2 (ite b y1a y1b)))

(assert (not (bvsge y2 (_ bv0 32))))

(check-sat)
(get-model)
```

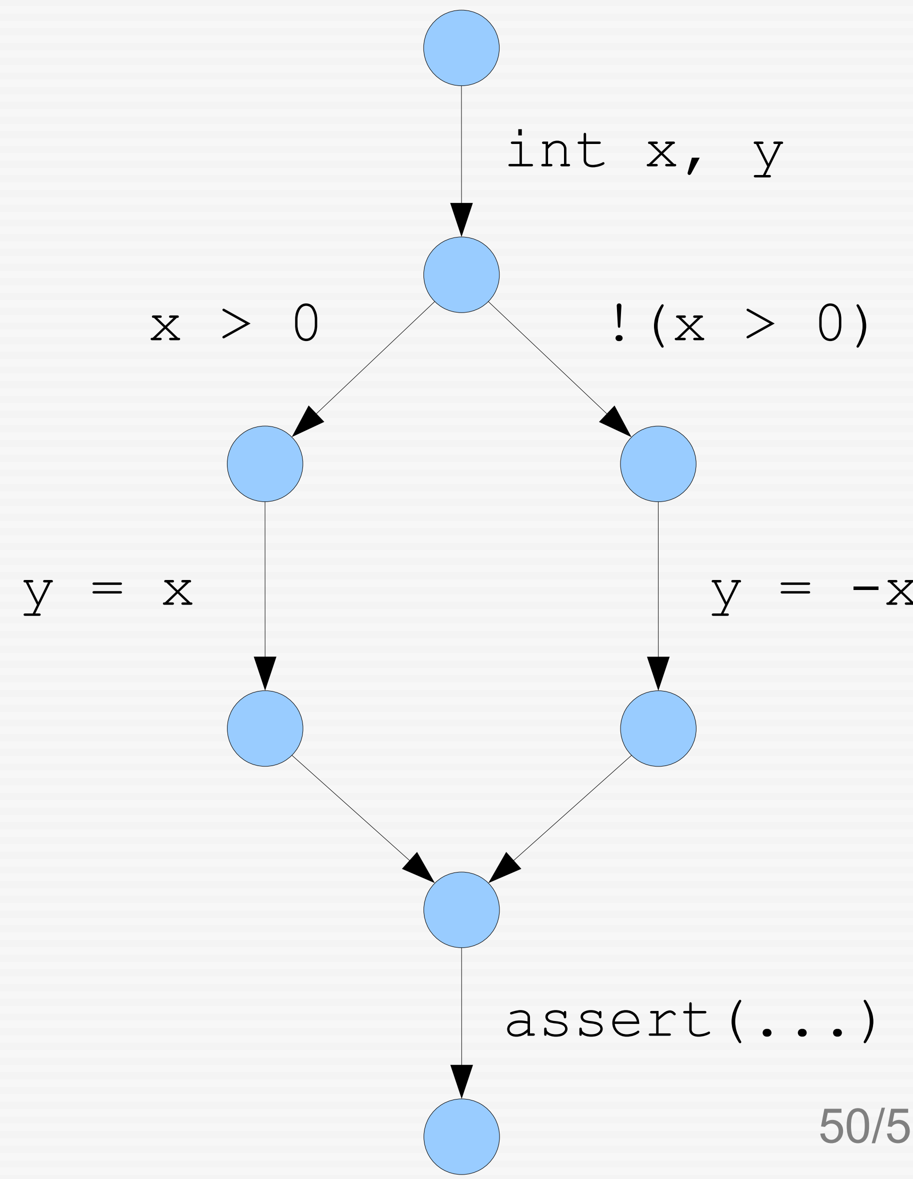
Example 4: Conditionals

```
int x, y;

if (x > 0)
  y = x;
else
  y = -x;

assert(y >= 0);
```

```
(set-option :pp.bv-literals false)

(declare-const x0  (_ BitVec 32))
(declare-const y0  (_ BitVec 32))
(declare-const y1a (_ BitVec 32))
(declare-const y1b (_ BitVec 32))
(declare-const y2  (_ BitVec 32))
(declare-const b   Bool)

(assert (= b (bvsgt x0 (_ bv0 32))))
(assert (=> b       (= y1a x0)))
(assert (=> (not b) (= y1b (bvneg x0))))
(assert (= y2 (ite b y1a y1b)))

(assert (not (bvsge y2 (_ bv0 32))))

(check-sat)
(get-model)
```

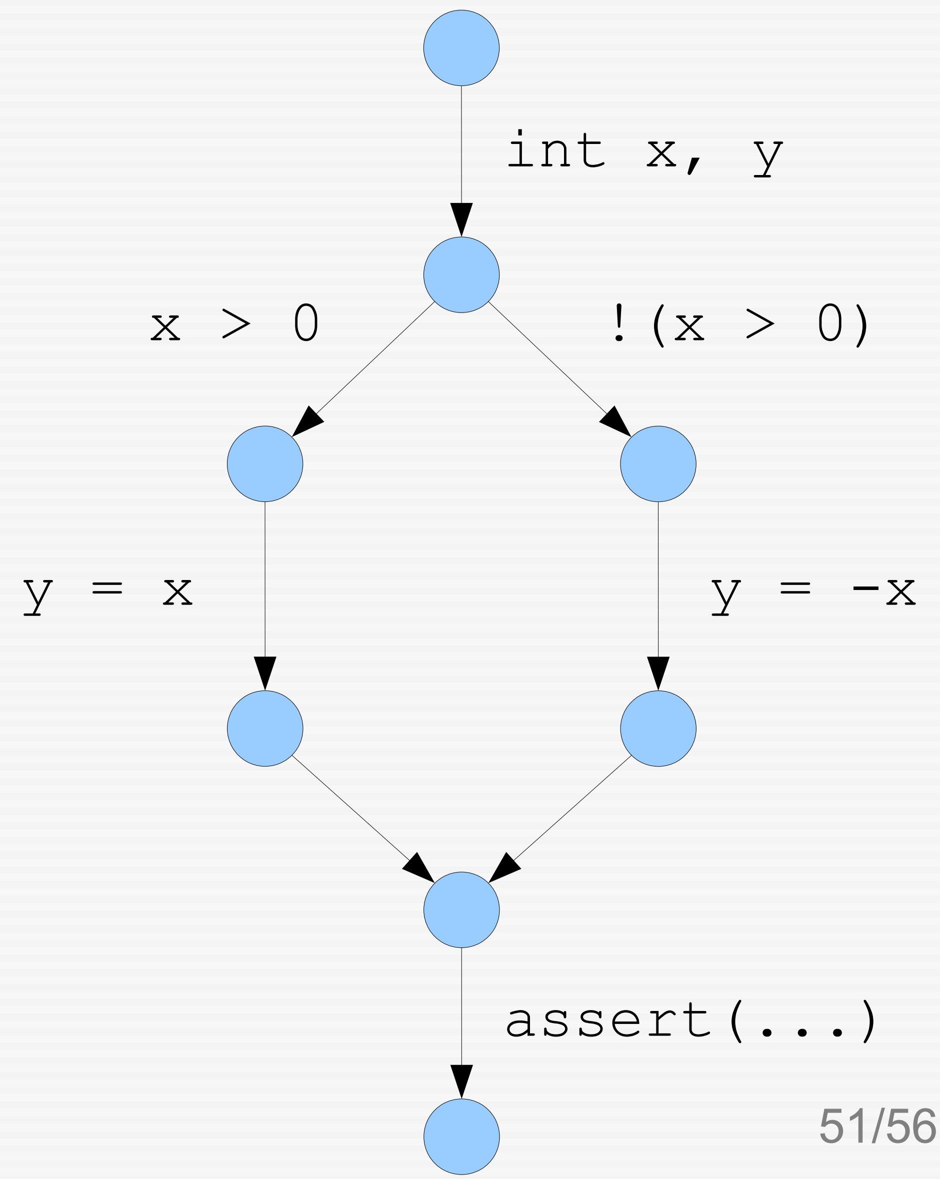
Permalink:
https://eldarica.org/princess/?ex=perma%2F1644243661_2074758304

Alternative method: path-wise exploration



Alternative method: path-wise exploration

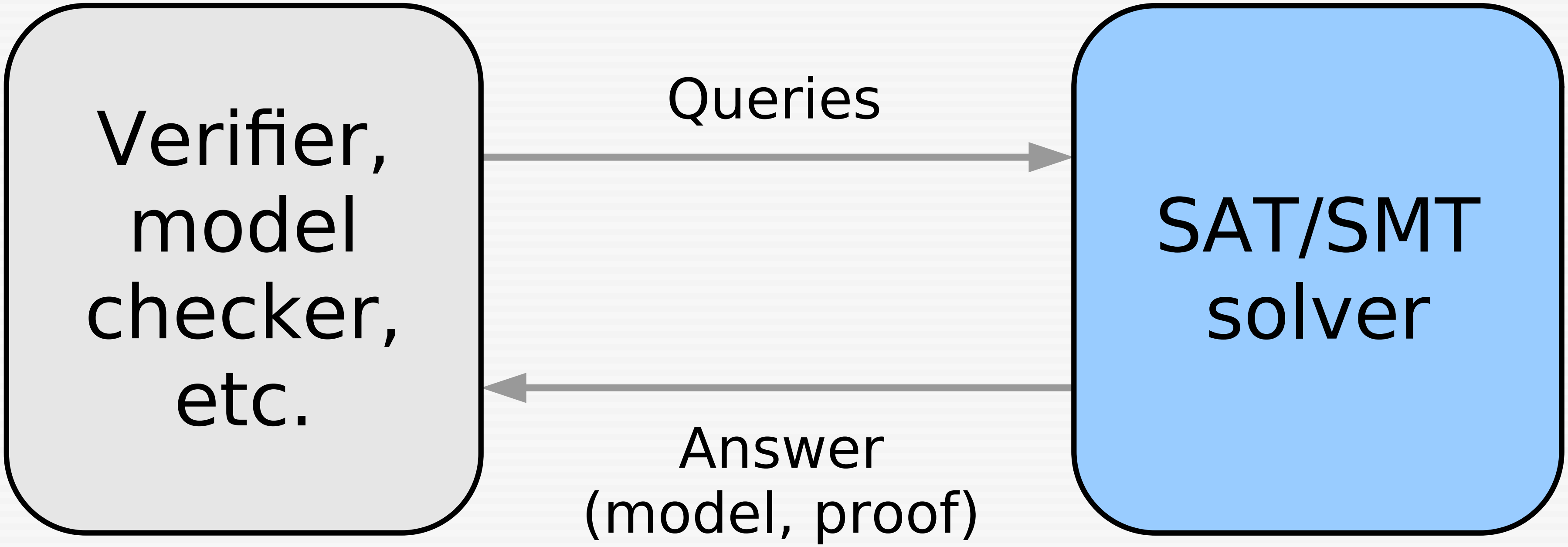
- Each query smaller, but possibly exponentially many paths
- Learning similar to CDCL can be used to avoid analysing all paths



The assertion stack

- Holds both assertions and declarations, but no options
- Important for incremental use of solver
- `(push` n`)` → add *n* new frames to the stack
- `(pop` n`)` → pop *n* frames from the stack

Typical Architecture



Example 5: Functions

- Every monotonic function

$$f : \{0, 1\} \to \{0, 1\}$$

is idempotent:

$$f \circ f = f$$

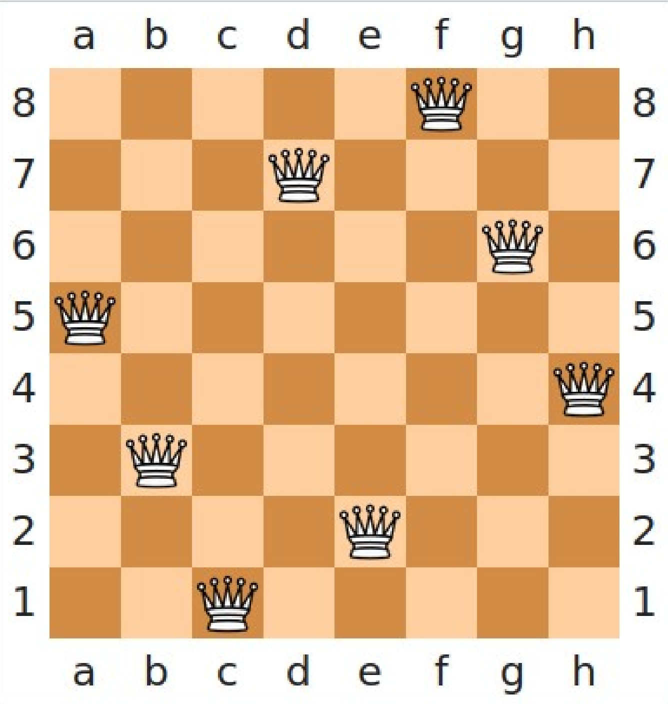
In SMT-LIB

```
(declare-fun f (Int) Int)

(assert (forall ((x Int))
            (=> (and (<= 0 x) (< x 2))
                (and (<= 0 (f x)) (< (f x) 2)))))
(assert (<= (f 0) (f 1)))

(assert (not (forall ((x Int))
            (=> (and (<= 0 x) (< x 2))
                (= (f (f x)) (f x)))))))

(check-sat)
```

Permalink:
https://eldarica.org/princess/?ex=perma%2F1668764094_519287413

Example 6

- *N*-queens problem using SMT



The only symmetrical solution to the eight queens puzzle (up to rotation and reflection)

[Wikipedia]

In SMT-LIB

```
(define-fun N () Int 4)

(declare-const q0 Int)
(declare-const q1 Int)
(declare-const q2 Int)
(declare-const q3 Int)

(assert (and (>= q0 0) (< q0 N)))
(assert (and (>= q1 0) (< q1 N)))
(assert (and (>= q2 0) (< q2 N)))
(assert (and (>= q3 0) (< q3 N)))

(assert (distinct q0 q1 q2 q3))
(assert (distinct (+ q0 0) (+ q1 1) (+ q2 2) (+ q3 3)))
(assert (distinct (- q0 0) (- q1 1) (- q2 2) (- q3 3)))

(check-sat)
```

Permalink:
https://eldarica.org/princess/?ex=perma%2F1668764522_925741057

Conclusions

- Most important idea in this lecture:
  Lazy encoding of formulas to SAT
- SMT solvers are ...
  - Usually optimised for verification:
    Good at proving unsat
  - Able to handle infinite domains:
    Arithmetic, arrays, strings, etc.
  - Side-effect: restricted set of operators:
    Capture *decidable* domains
  - Good at propositional reasoning

Conclusions

Compare to **relaxations**

- Most important idea in this lecture: Lazy encoding of formulas to SAT
- SMT solvers are ...
  - Usually optimised for verification: Good at proving unsat
  - Able to handle infinite domains: Arithmetic, arrays, strings, etc.
  - Side-effect: restricted set of operators: Capture *decidable* domains
  - Good at propositional reasoning

Outlook

- Various further topics:
  - More theories: ADTs, floats, strings, etc.
  - Handling of quantifiers
  - Fixed-point computation
  - MaxSAT/MaxSMT
  - Optimising SMT
- More lecture slides:
  - http://ssa-school-2016.it.uu.se/
  - https://sat-smt-ar-school.gitlab.io/www/2022/