# The OpenSMT Solver

Roberto Bruttomesso[1], Edgar Pek[2],
Natasha Sharygina[1], and Aliaksei Tsitovich[1]

1 Formal Verification and Security Group, University of Lugano, Switzerland
2 University of Urbana Champaign, Illinois, USA

# Outline of the talk

- Motivations and Introduction
- The Developer's perspective
  - Architecture
  - Extending OpenSMT
- The User's perspective
  - Using OpenSMT over API
  - Getting witnesses (models/proofs)
  - Getting interpolants

# Motivation

- Satisfiability Modulo Theory (SMT) solvers are key engines of several verification approaches
- Most solvers available are proprietary

  (Z3, Yices, Barcelogic, MathSAT… )
- OpenSMT is an effort of providing a simple and extensible infrastructure, and efficient at the same time.
- Currently, the following logics are supported: QF_UF, QF_LRA, QF_IDL, QF_RDL, QF_BV and several theory combinations (comparison is available at SMT competition web-site).

# Introduction

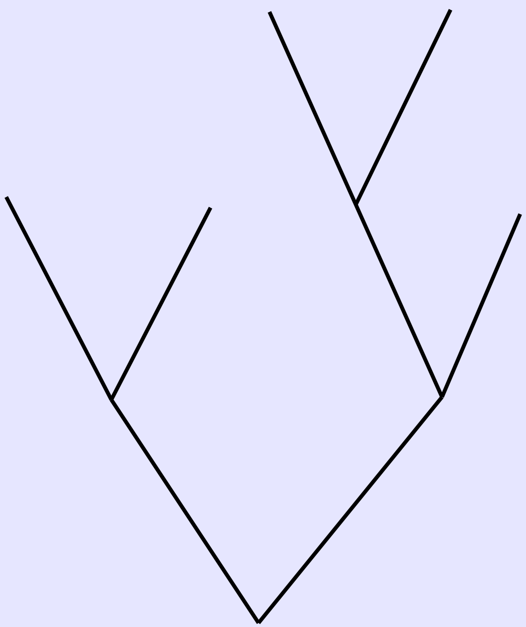
- Satisfiability Modulo Theories combines the efficiency of SAT and theory-specific decision procedures

$$a \wedge (\underbrace{(x + y \leq 0)}_{c} \vee \neg a) \wedge (\underbrace{(x = 1)}_{d} \vee b)$$

# We need to reason about Boolean combinations of atoms in a theory T (LRA for instance)

**SAT (Boolean)**

(e.g. DPLL)



**Dec. Proc. for LRA**

(e.g. Simplex)

$$a_1x_1 + \ldots a_nx_n + b \leq 0$$

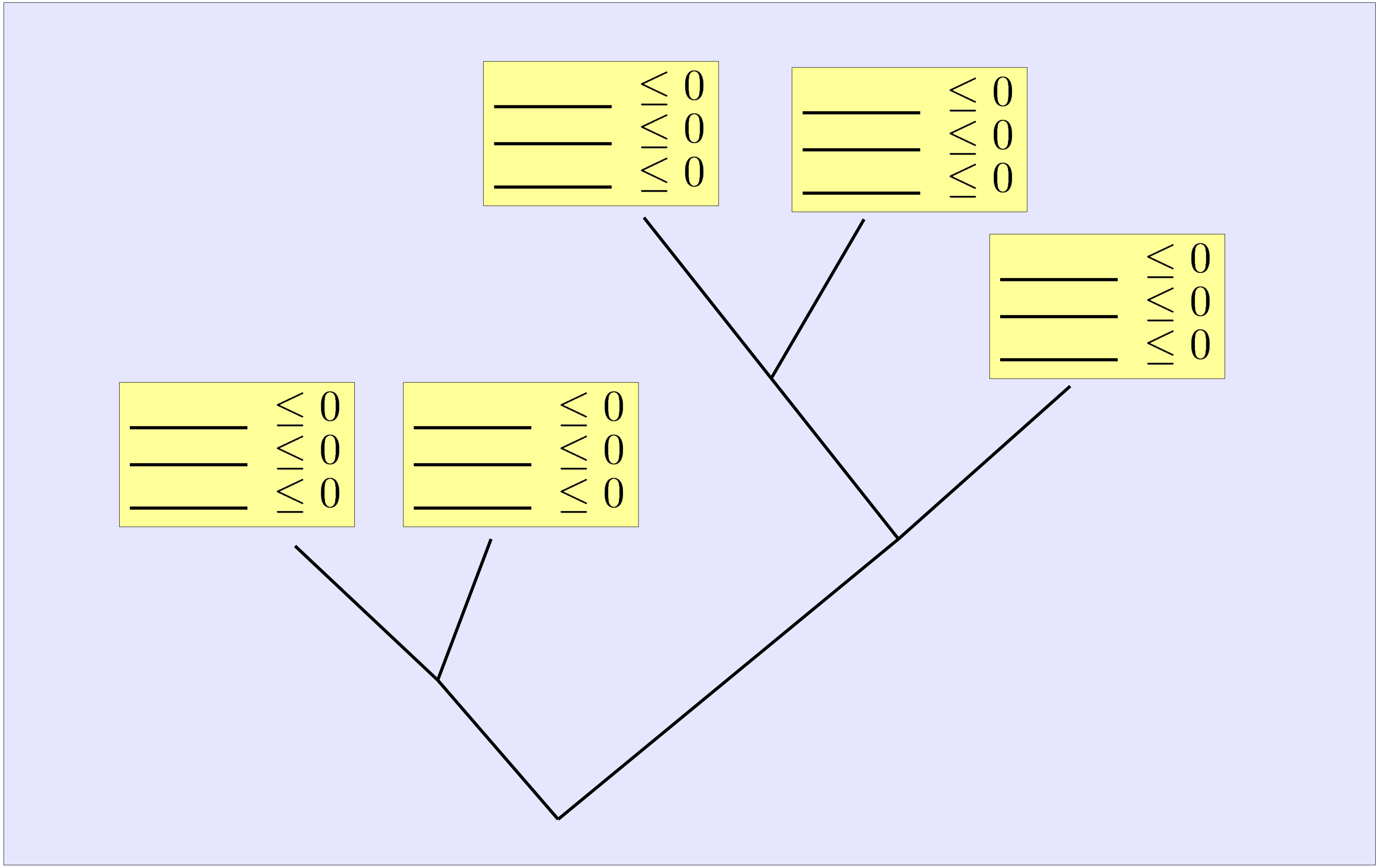
$$\_\_\_\_\_\_\_\_\_\_\_\_ \leq 0$$

$$\_\_\_\_\_\_\_\_\_\_\_\_ \leq 0$$

$$\_\_\_\_\_\_\_\_\_\_\_\_ \leq 0$$

$$\_\_\_\_\_\_\_\_\_\_\_\_ \leq 0$$

# DPLL+LRA $\Longrightarrow$ DPLL(LRA)



- DPLL(LRA) framework seems easy to achieve
  - Let DPLL enumerate a Boolean model
  - Check the LRA part with Simplex

- However it is not enough to connect an efficient DPLL solver and Simplex to get an efficient DPLL(LRA)
  - Theory Propagation
  - Don't wait for a complete Boolean model
  - Preprocessing
  - Conversion in CNF
  - Theory layering, etc.

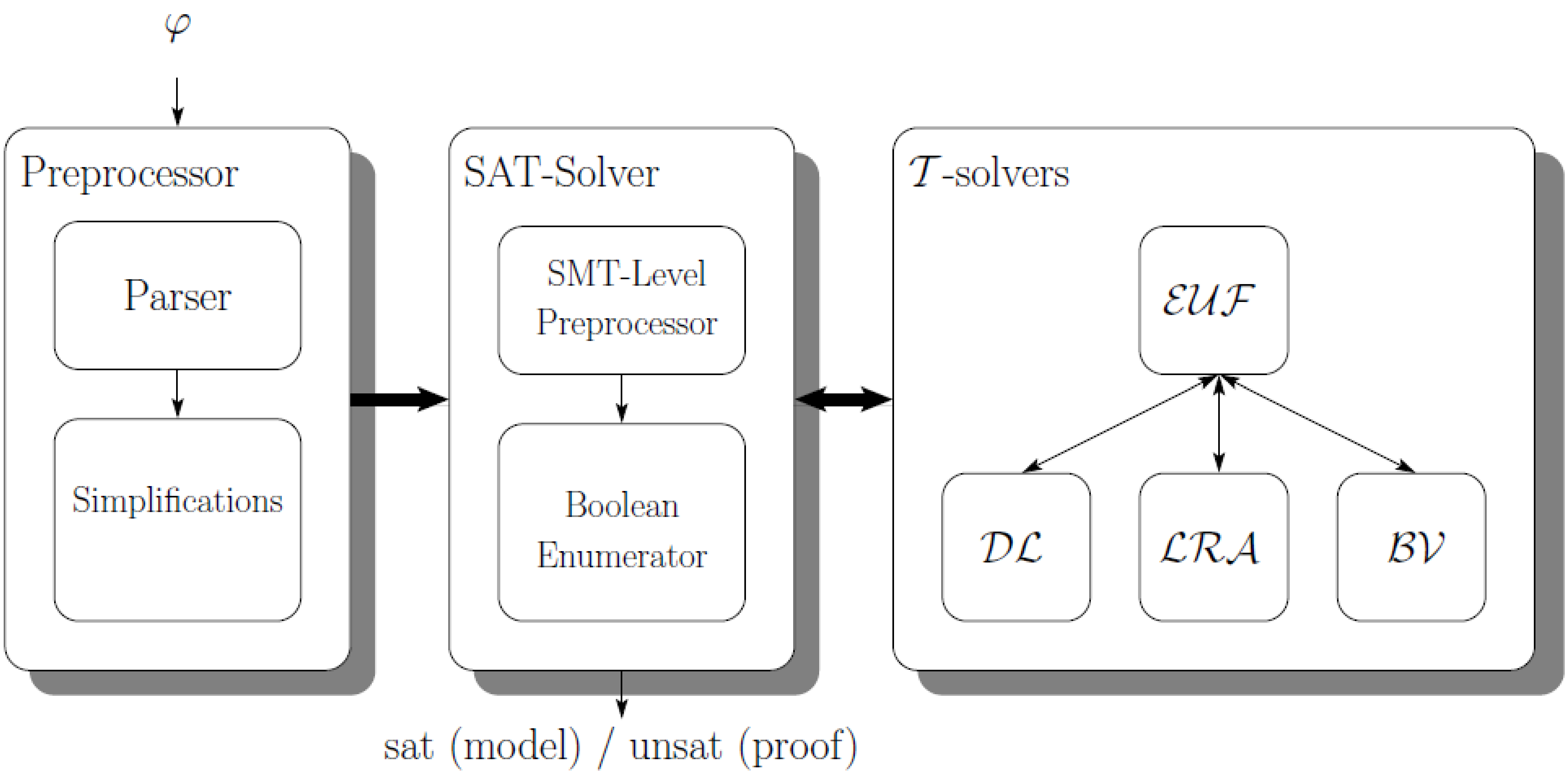
$$e(DPLL(T)) = e(DPLL)+e(T)+e(COMM)$$


OpenSMT provides you with e(DPLL) and e(COMM)

# Outline of the talk

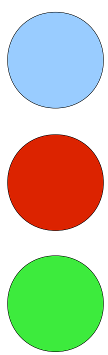
- Introduction and Motivations
- The Developer's perspective
  - Architecture
  - Extending OpenSMT
- The User's perspective
  - Programming with API
  - Getting witnesses (models/proofs)
  - Getting interpolants

# Architecture



# Architecture

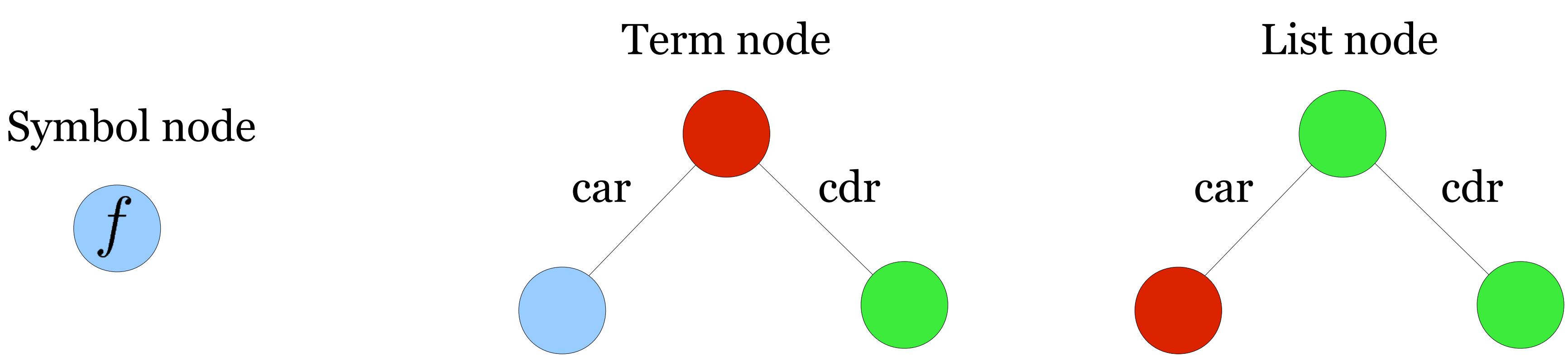
- Written in C++
- Based on MiniSAT2 SAT-Solver
- Enode data structure (borrowed from Simplify)



Symbol nodes: store the signature (name) of the operator

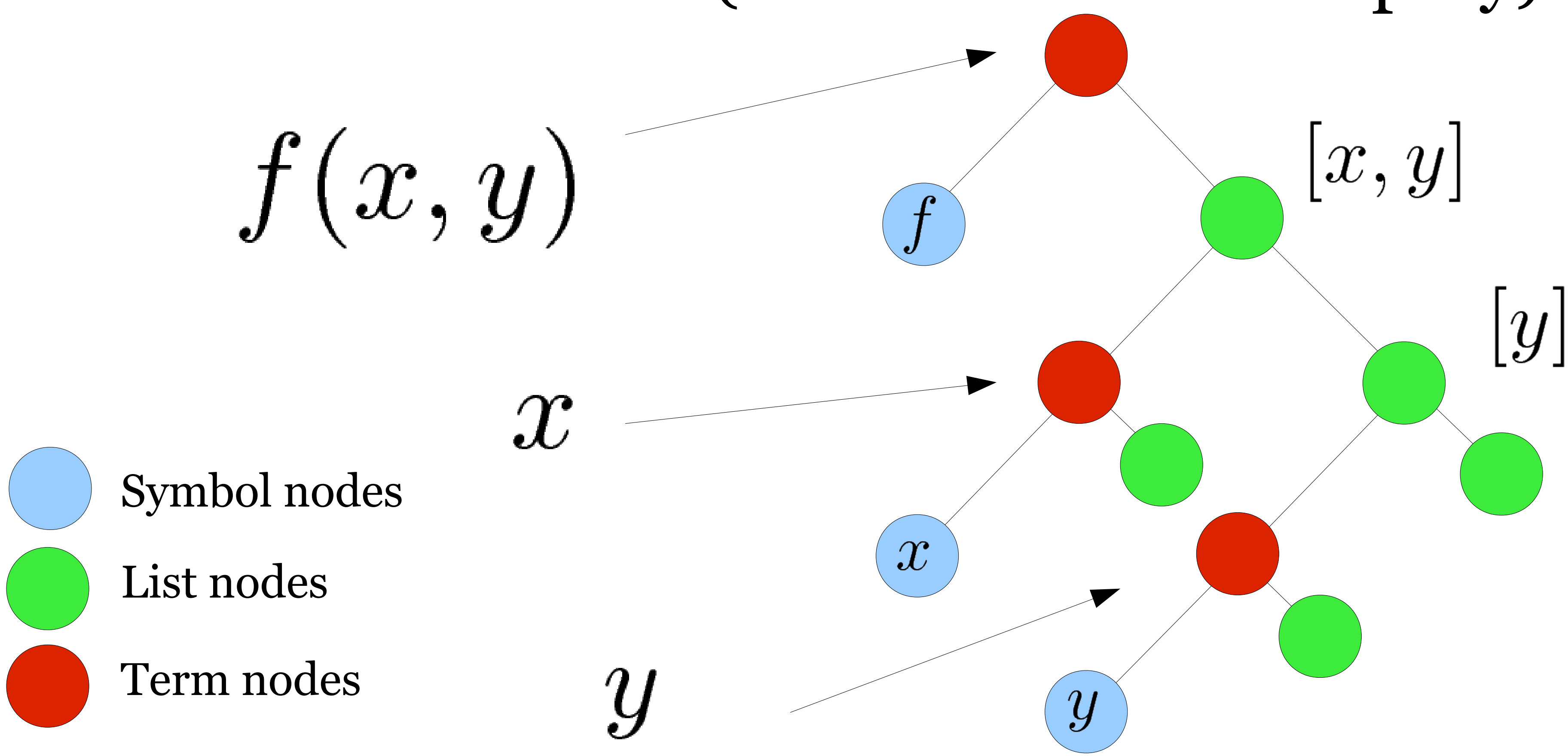
Term nodes: store a term of the formula

List nodes: store a list of terms



# Architecture

- Written in C++
- Based on MiniSAT2 SAT-Solver
- Enode data structure (borrowed from Simplify)



# Extending OpenSMT

- To create an empty template for a new theory solver use script `create_tsolver.sh`
  - Creates a new directory with basic class files
  - Creates/Sets up Makefile for compilation
  - Adds a new logic
  - Integrates the new solver with the core
  - Basically, it creates an incomplete solver
  - (demo)

# Extending OpenSMT

```
class Tsolver
{
  void inform          ( Enode * );
  bool assertLit       ( Enode * );
  bool check           ( bool );
  void pushBckPoint    ( );
  void popBckPoint     ( );
  bool belongsToT      ( Enode * );

  [...]

  vector< Enode * > & explanation;
  vector< Enode * > & deductions;
  vector< Enode * > & suggestions;
}
```

# Outline of the talk

- Introduction and Motivations
- The Developer's perspective
  - Architecture
  - Extending OpenSMT
- The User's perspective
  - Programming with API
  - Getting witnesses (models/proofs)
  - Getting interpolants

# The user's perspective

- APIs allow quick integration of the SMT-solver's facilities by linking to a library
- OpenSMT APIs are inspired to Yices
- E.g.
  - Create a context `ctx` of QF_LRA solver

    ```
    opensmt_context ctx = opensmt_mk_context( qf_lra );
    ```

  - Create a variable `i` of type integer

    ```
    char var[32]="i";
    opensmt_expr i = opensmt_mk_int_var( ctx, var );
    ```

# The user's perspective

- Example: encode the following simple loop symbolically (api_example/example2.c)

```
1 int i=0;
2 while (i<10)
3    i++;
4 assert( i==11 );
```

---

```
opensmt_context ctx = opensmt_mk_context( qf_idl );
opensmt_expr int_i = opensmt_mk_int_var( ctx, “i@0” );
opensmt_expr zero = opensmt_mk_num_from_string( ctx, "0" );
opensmt_expr eq = opensmt_mk_eq( ctx, int_i, zero );
opensmt_assert( ctx, eq );
```

# The user's perspective

- Example: encode the following simple loop symbolically (api_example/example2.c)

```
int i=0;
while (i<10)
    i++;
assert( i==11 );
```

---

```
opensmt_expr expr_list[2];
expr_list[0] = int_i;
opensmt_expr one = opensmt_mk_num_from_string( ctx, "1" );
expr_list[1] = one;
opensmt_expr plus = opensmt_mk_plus( ctx, expr_list, 2 );
opensmt_expr int_i_prime = opensmt_mk_int_var( ctx, “i@1” );
eq = opensmt_mk_eq( ctx, int_i_prime, plus );
opensmt_assert( ctx, eq );
```

# Getting witnesses - model

- Enable flag

  `print_model 1`

  in .opensmtrc to compute the assignment in case SAT

- Model is printed in SMT-LIB syntax

  ```
  (= x 1)
  (not b)
  ```

  (so it is handy to check – just put in conjunction with the original formula)

# Getting witnesses - proof/cores

- Enable flag `print_proof 1` in .opensmtrc
- Resolution Proof format
  - Leaf clauses `(let cls_1 (or a (not b))`
  - Res. Step `(let cls_3 (res cls_1 cls_2 a))`

$$\frac{a \lor \neg b \quad b \lor c}{a \lor c}\, b$$

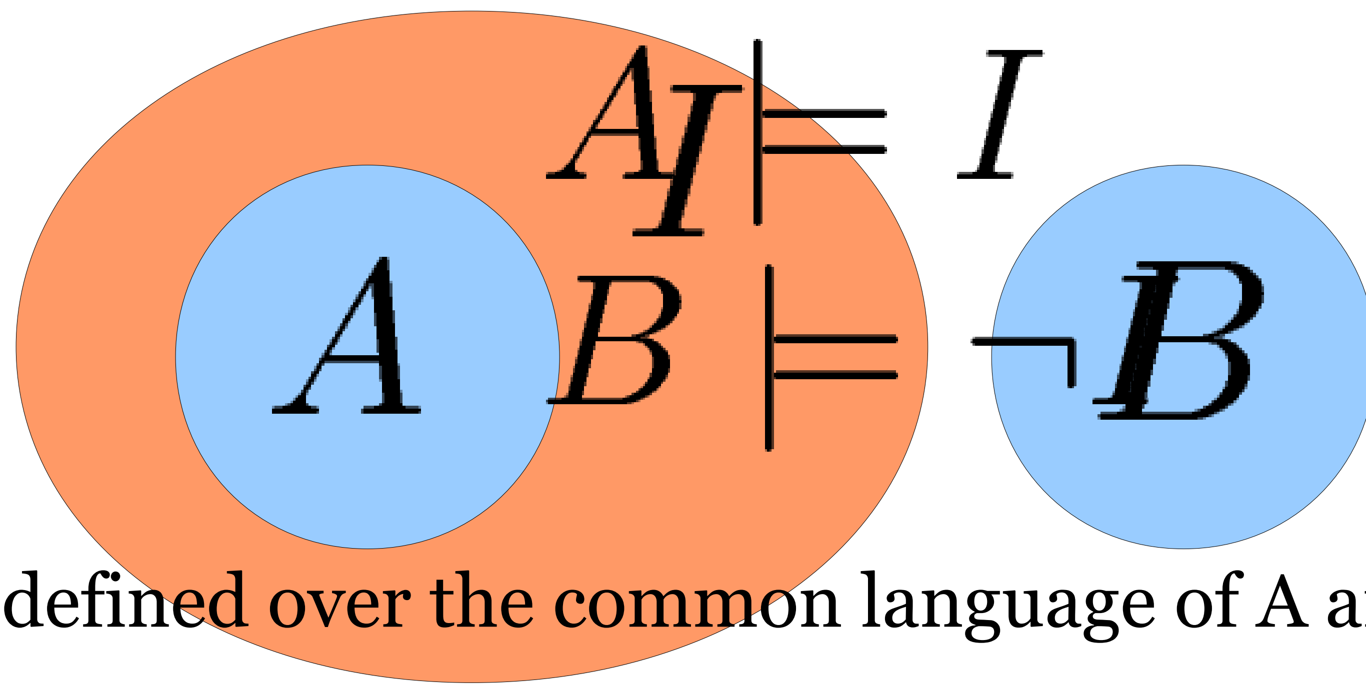
```
(let cls_1 (or a (not b))

(let cls_2 (or b c))

(let cls_3 (res cls_1 cls_2 b))
```

# Interpolation

- Interpolants are widely used in SAT-based Model Checking, for instance to compute over-approximations




$$A \models I$$

$$B \models \neg I$$

I defined over the common language of A and B

# Interpolation

- OpenSMT computes the general interp. form:
  - given an unsat conjunction $A_1, A_2, \ldots, A_n$
  - computes $I_0, I_1, \ldots, I_n$
  - such that

$$
\begin{aligned}
&I_i \wedge A_{i+1} \models I_{i+1} \\
&I_0 = \top \\
&I_n = \bot
\end{aligned}
$$

$$I_0 \wedge A_1 \models I_1$$

$$I_1 \wedge A_2 \models I_2$$

$$I_2 \wedge A_3 \models I_3$$

$$I_0 = \top$$

$$I_3 = \bot$$

| | | | | |
|---|---|---|---|---|
| $\top$ | $\wedge$ | $a \vee b$<br>$\neg a \vee b$ | $\models b$ | $(I_1)$ |
| $b$ | $\wedge$ | $\neg b \vee c \vee d$<br>$\neg b \vee \neg c \vee d$ | $\models d$ | $(I_2)$ |
| $d$ | $\wedge$ | $\neg d \vee e$<br>$\neg d \vee \neg e$ | $\models \bot$ | $(I_3)$ |

# Conclusion

- OpenSMT is an open, efficient, and extensible SMT-Solver
- Provides a framework to experiment with decision procedures
- Features API, witnesses generation, interpolant generation

# Availability

- Available at `verify.inf.usi.ch/opensmt`
- Discussion group
  `groups.google.com/group/opensmt`
- Demo/more details on request