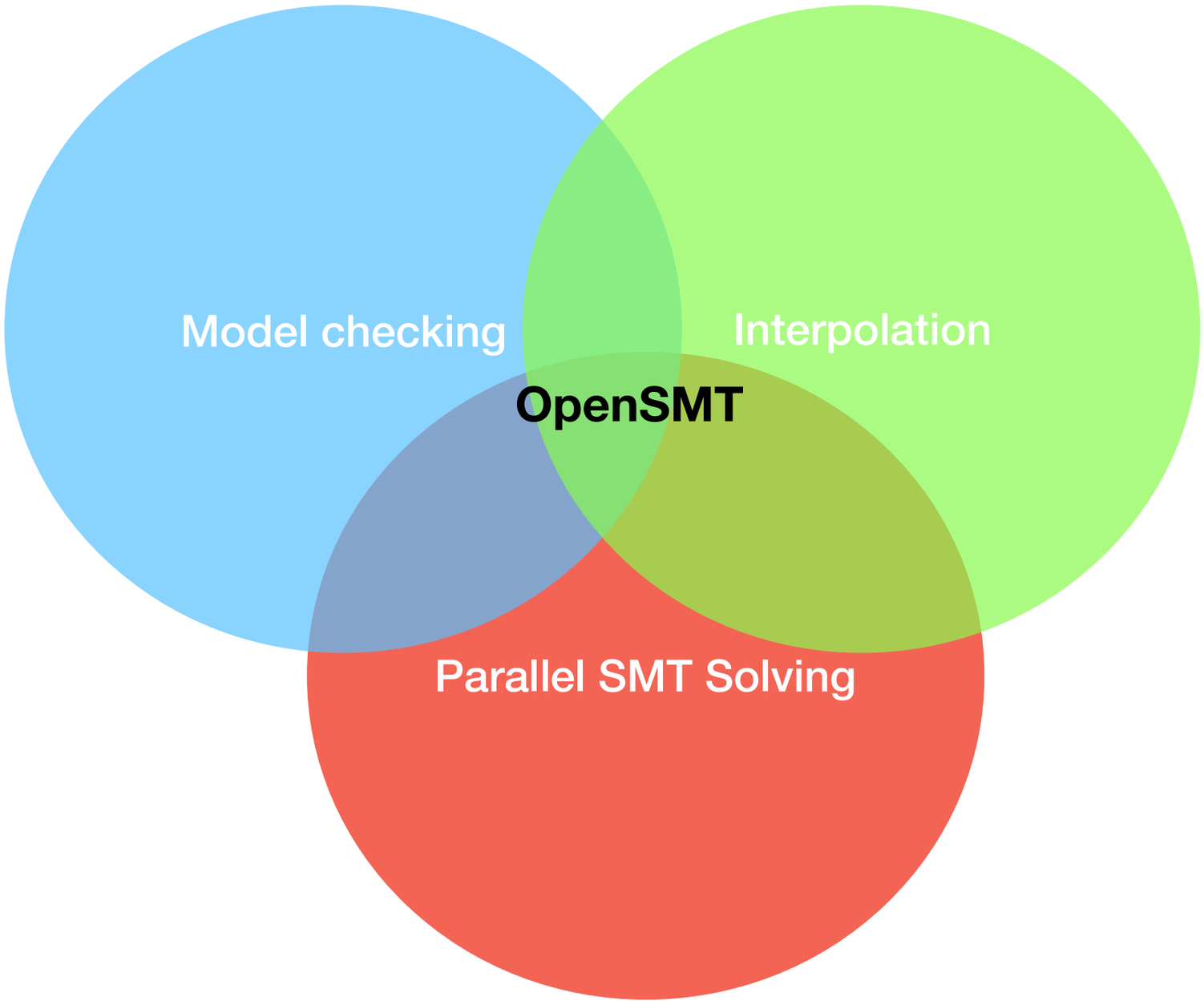
OpenSMT2

A Parallel, Interpolating SMT Solver

Antti Hyvärinen, Matteo Marescotti, Leonardo Alt, Sepideh Asadi, and Natasha Sharygina

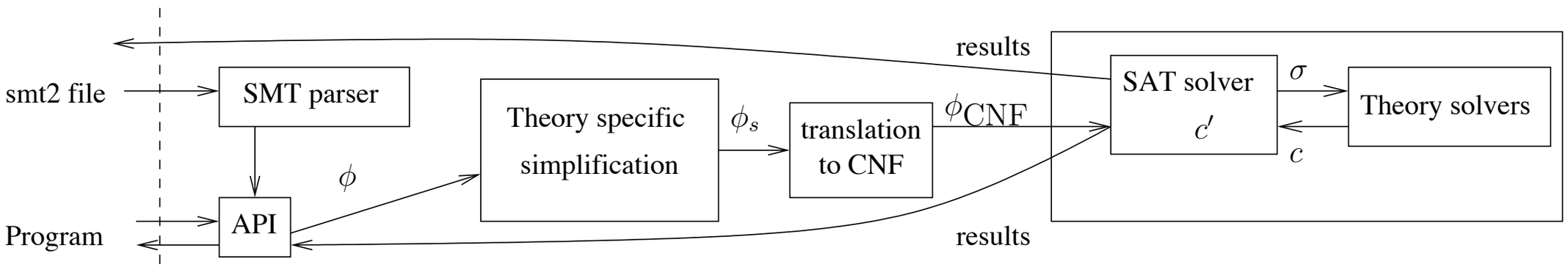
Why another SMT solver?



OpenSMT2

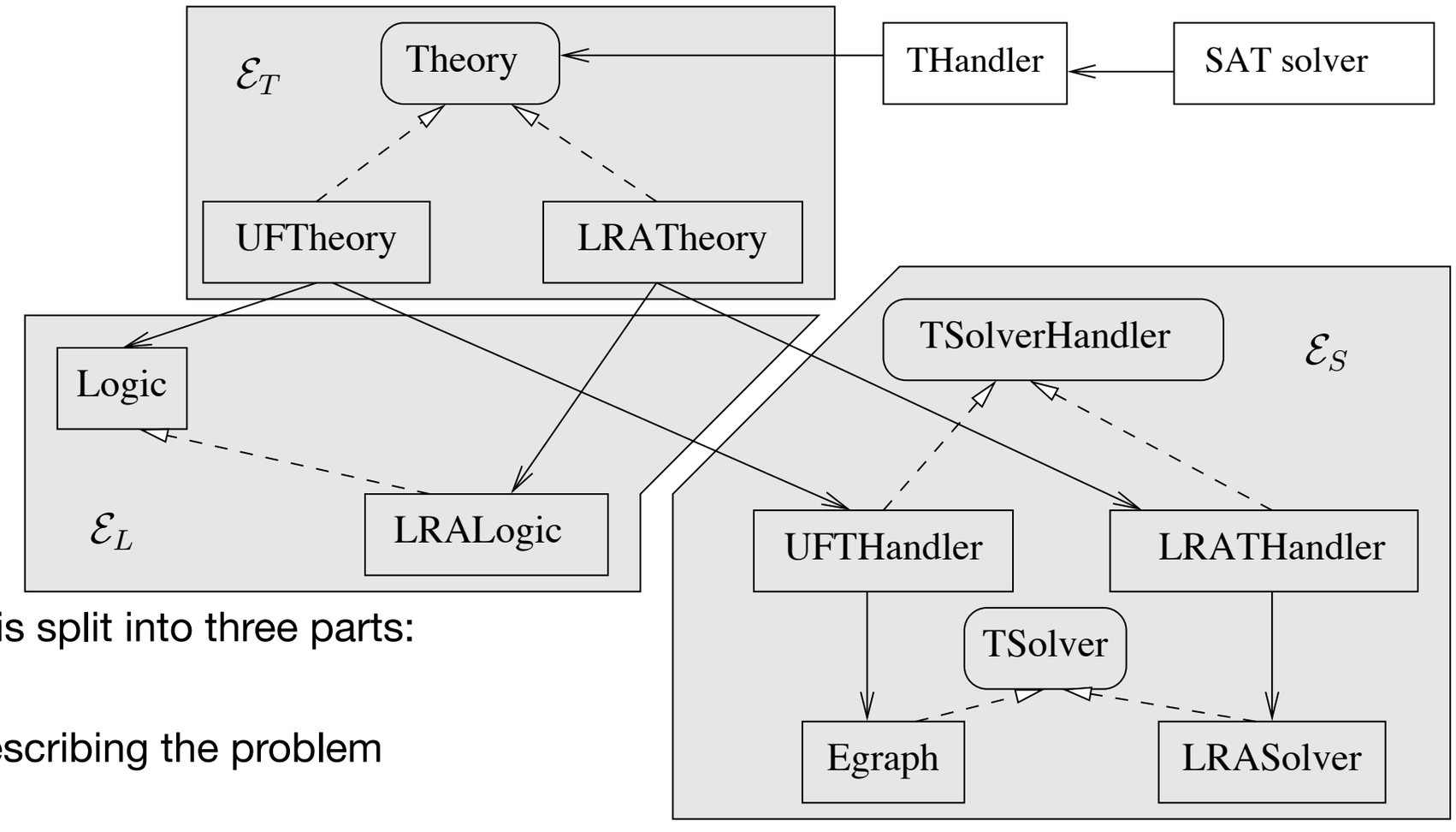
- MIT-licensed SMT solver written in C++, from Università della Svizzera italiana,
- Available from http://verify.inf.usi.ch/opensmt
- Currently supports QF_UF, QF_LRA, and to some extent QF_BV
- Labeled interpolation on propositional logic, QF_UF, and QF_LRA, with proof compression
- Cluster-scale parallelisation
- Relatively compact (approx. 55,000 LoC)
- An object-oriented design to help extensions for the theories

The DPLL(T) architecture in OpenSMT2



- The solver consists of a SAT solver and theory solvers
- API for C++
- A more limited API for C and Python

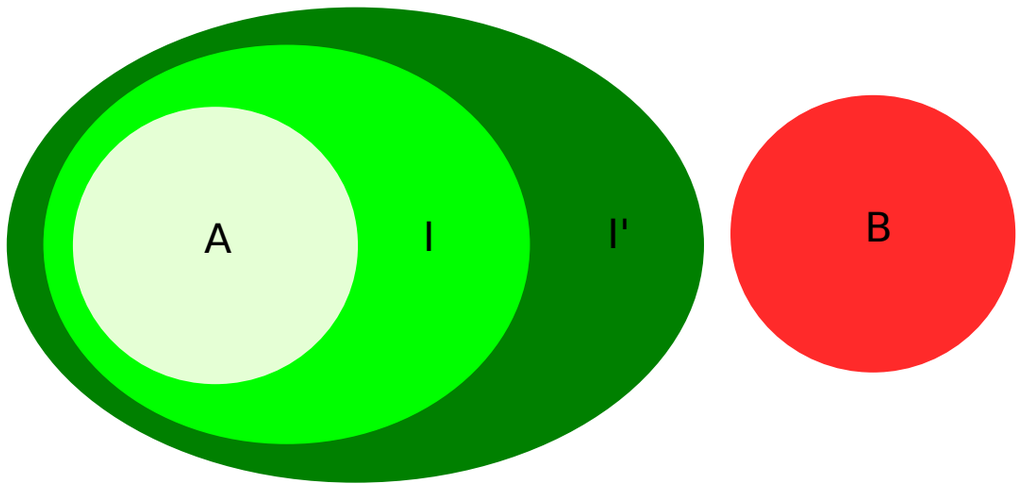
The Theory Solver Architecture



- Theory Solver architecture is split into three parts:
  - Logic framework for describing the problem
  - Solver framework for solving the problem
  - Theory framework connecting the solver to its logic
- SAT solver accesses the framework by making queries on the Theory

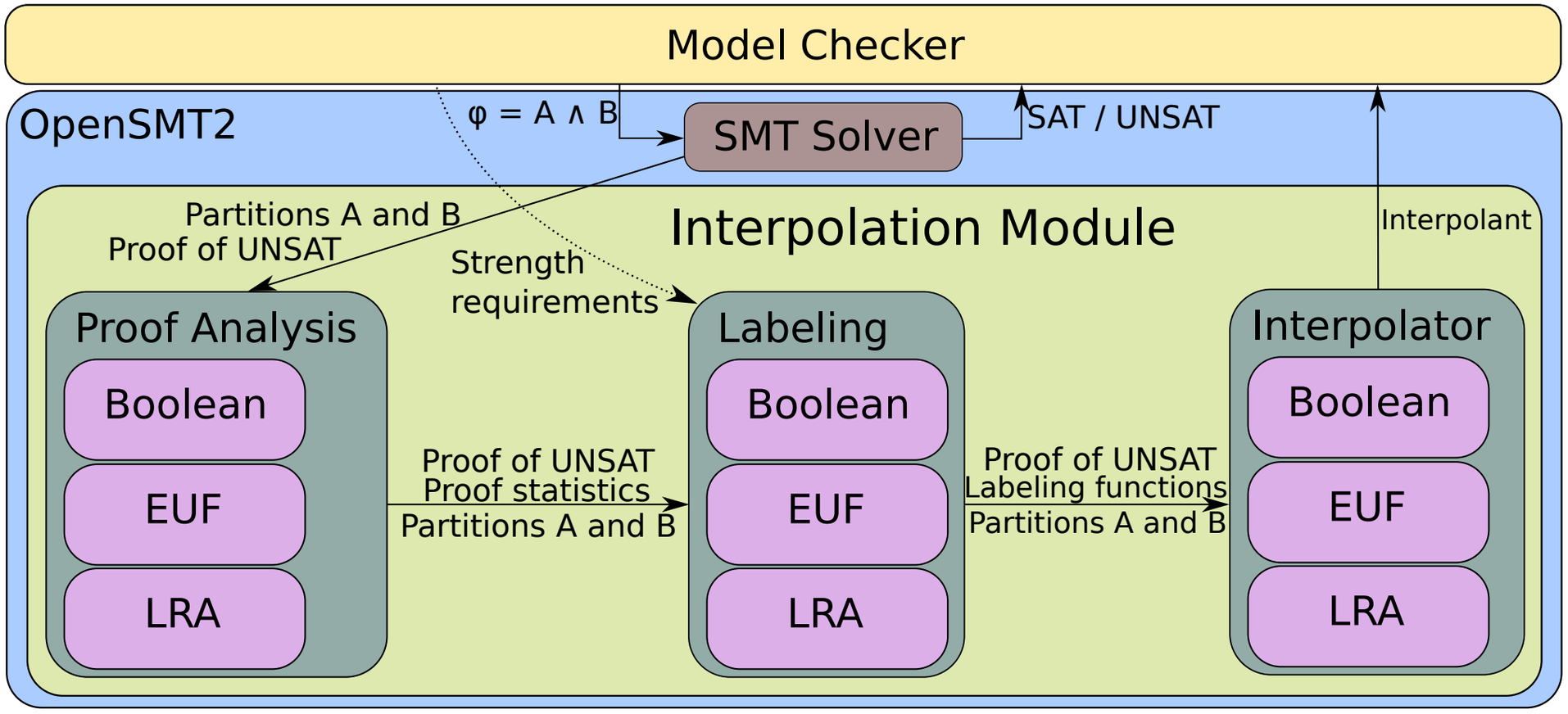
Interpolation

Interpolation



- Interpolation is a way to over-approximate states in a safe way
  - Given an unsatisfiable formula $A \wedge B$, compute an interpolant I such that $A \rightarrow I$ and $I \wedge B$ is unsat
- OpenSMT supports interpolation for
  - propositional logic,
  - uninterpreted functions with equality, and
  - linear real arithmetics

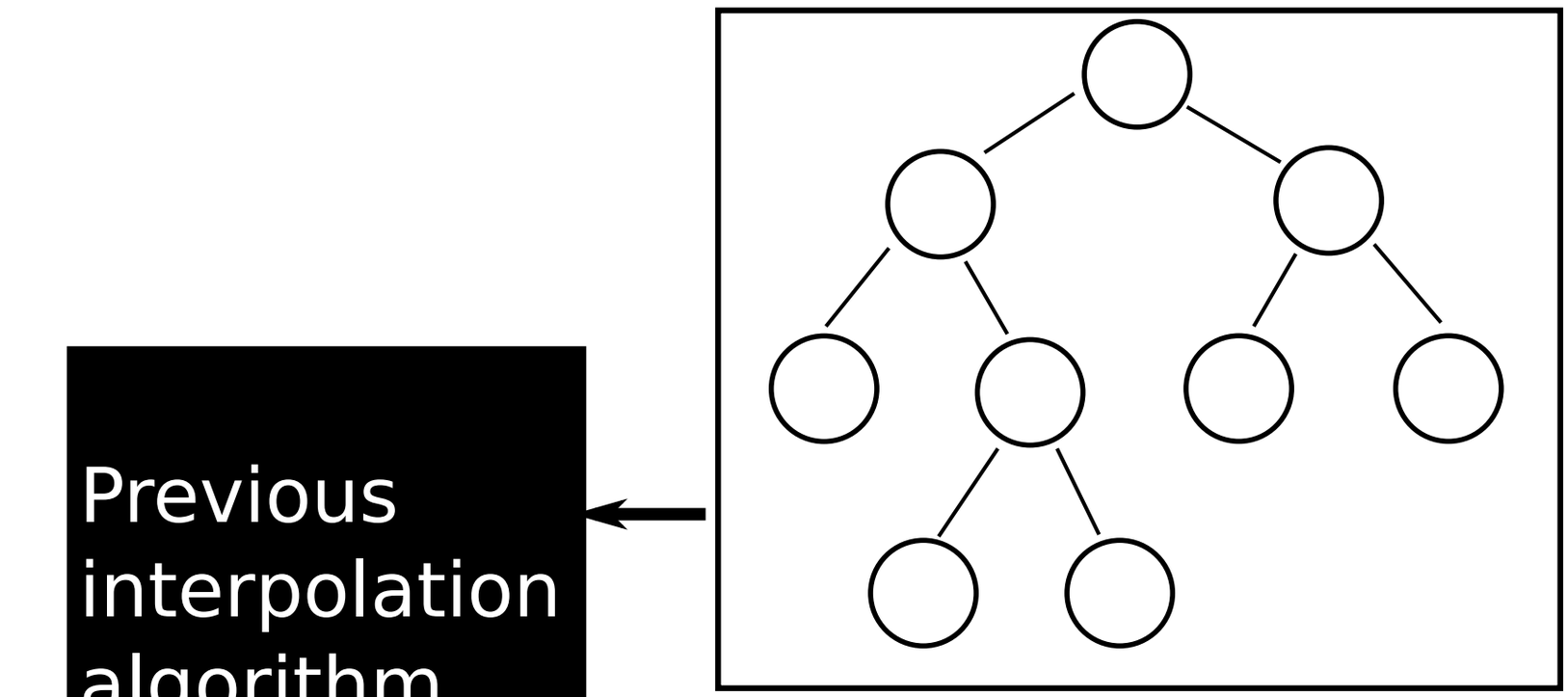
Interpolation in OpenSMT

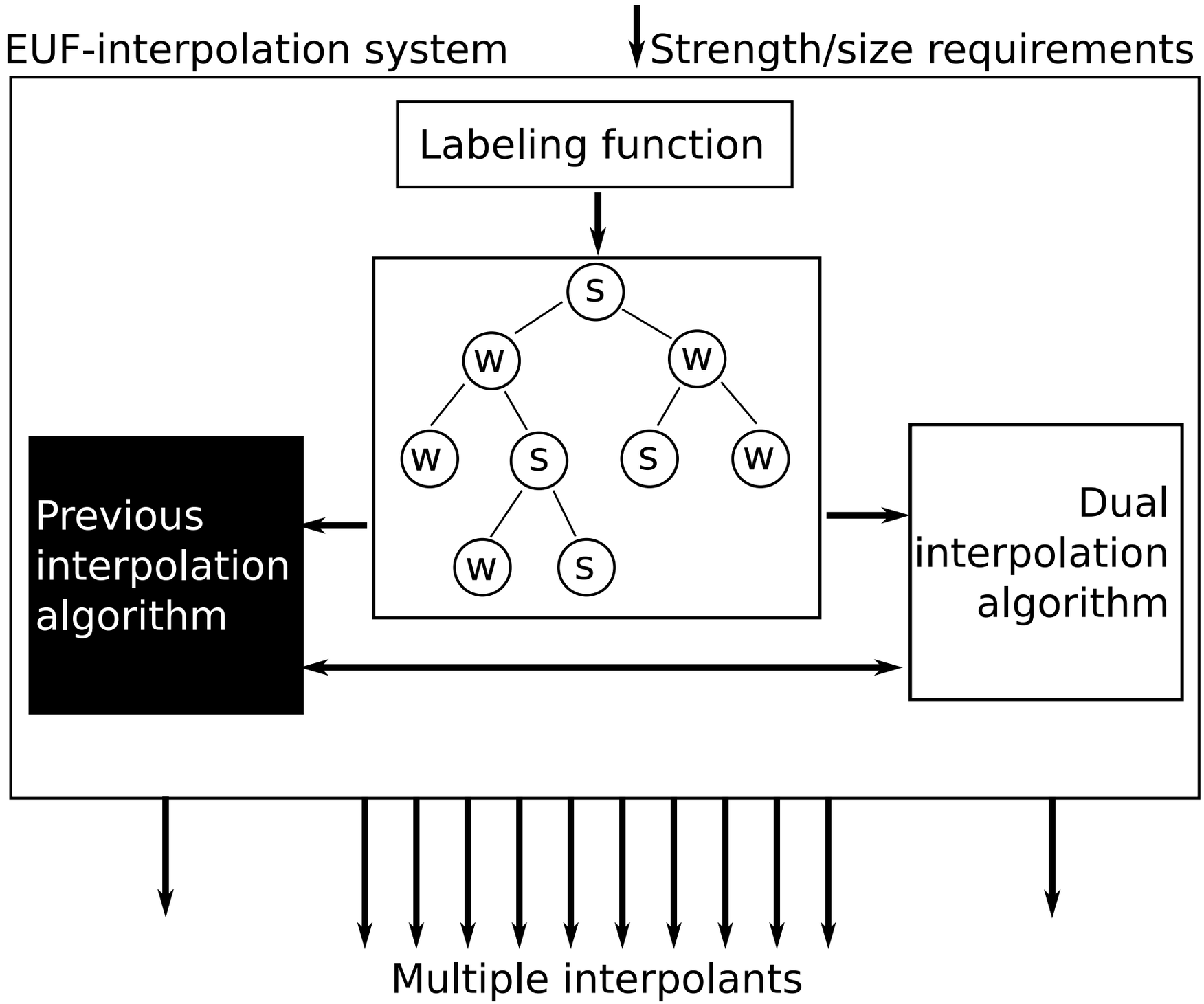


- The OpenSMT API provides the application with a full control over the interpolant generation
- Many of the more routine tasks are implemented efficiently inside OpenSMT so that the user does not need to take care of such details.
- The system makes it comfortable to construct and experiment with new labelling functions

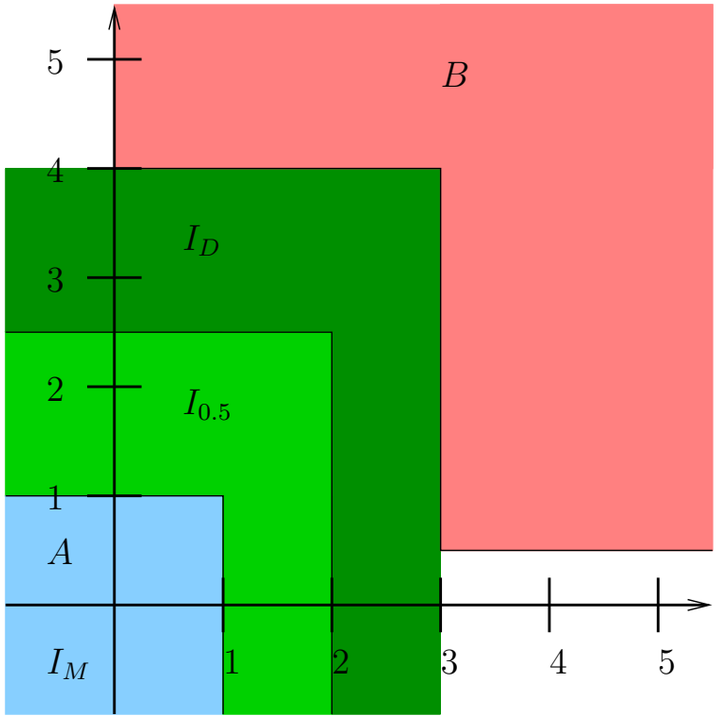
Interpolation with Equality of Uninterpreted Functions

- Interpolants have the duality property:
  - Let $I(A, B)$ be an interpolant over-approximating A.
    Then also $\neg I(B, A)$ is an interpolant over-approximating A
  - Often we can show that $I(A, B) \rightarrow \neg I(B, A)$
- The duality can be used to construct new interpolants with different interpolant strength


Previous interpolation algorithm
Single interpolant


EUF-interpolation system
Strength/size requirements
Labeling function
S
W
W
W
S
S
W
W
S
Previous interpolation algorithm
Dual interpolation algorithm
Multiple interpolants

Interpolation for LRA

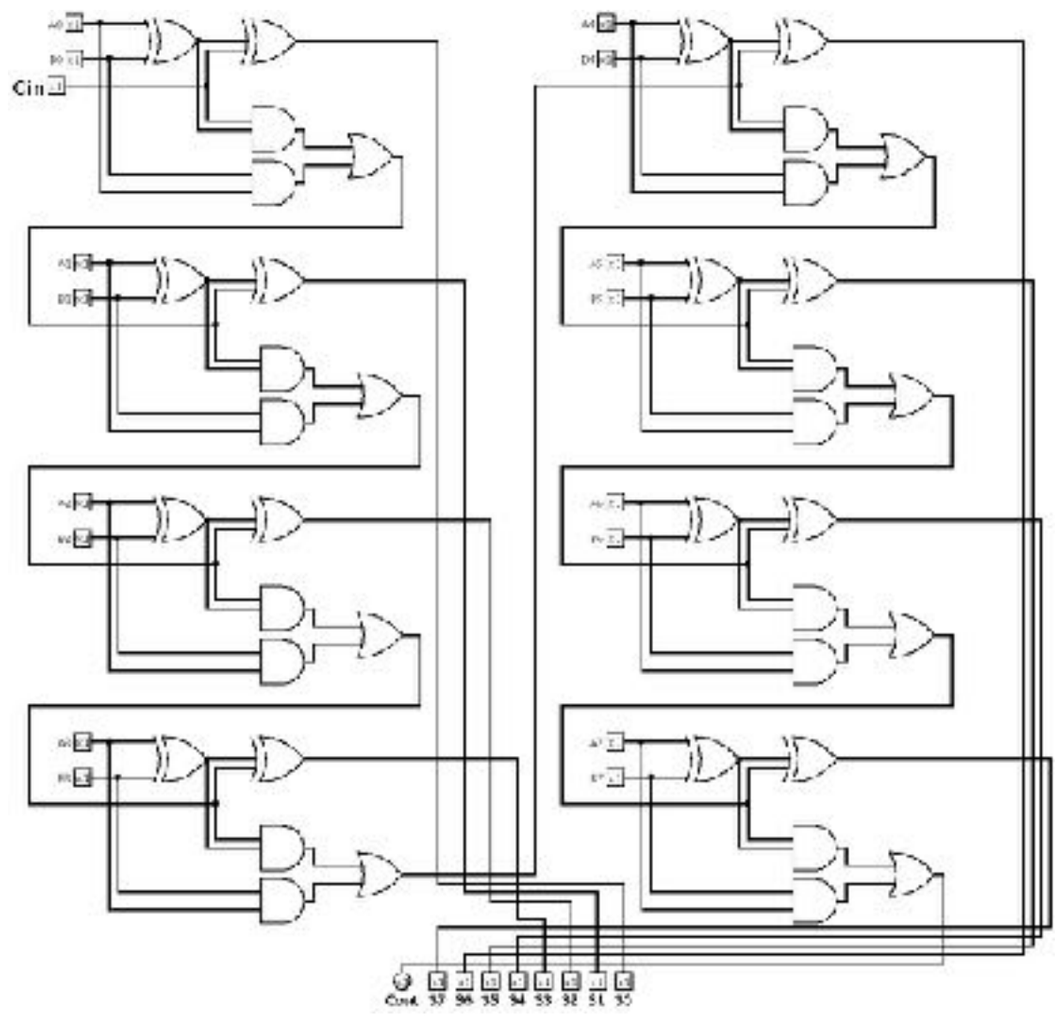


- We provide also LRA interpolation in OpenSMT
  - Based similarly on Duality
  - New interpolants are constructed by shifting the constraints with a constant

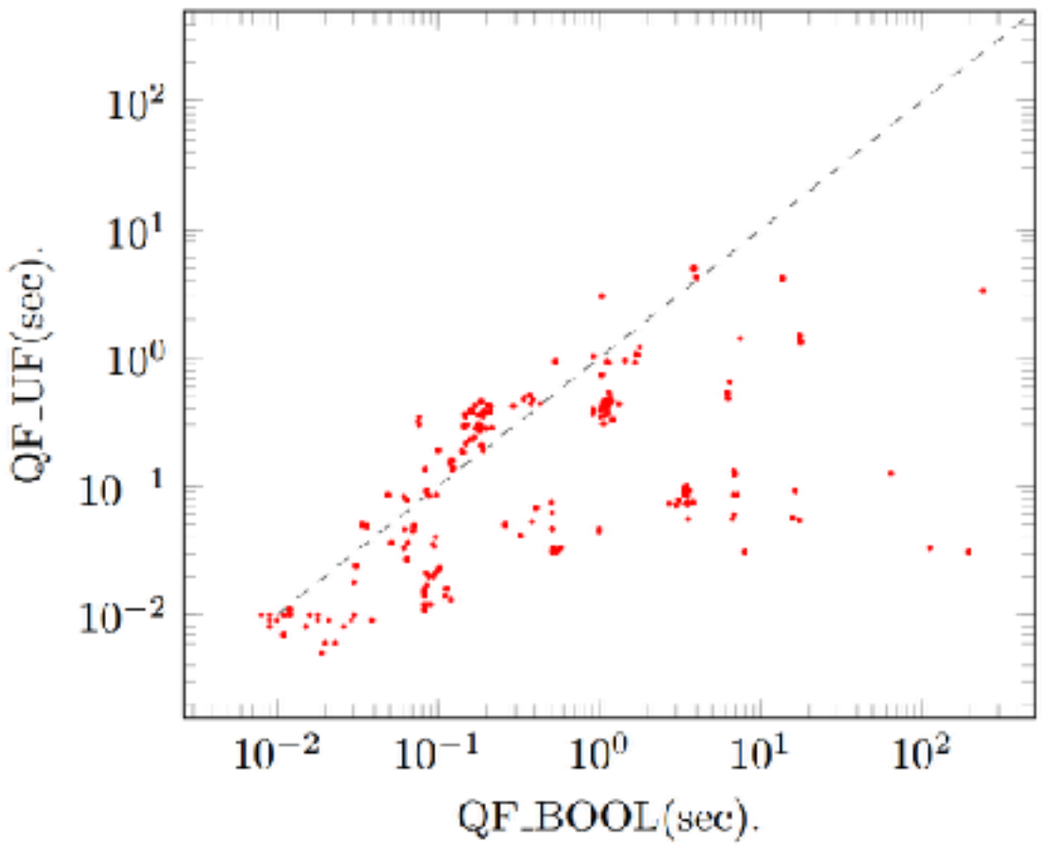
Model Checking

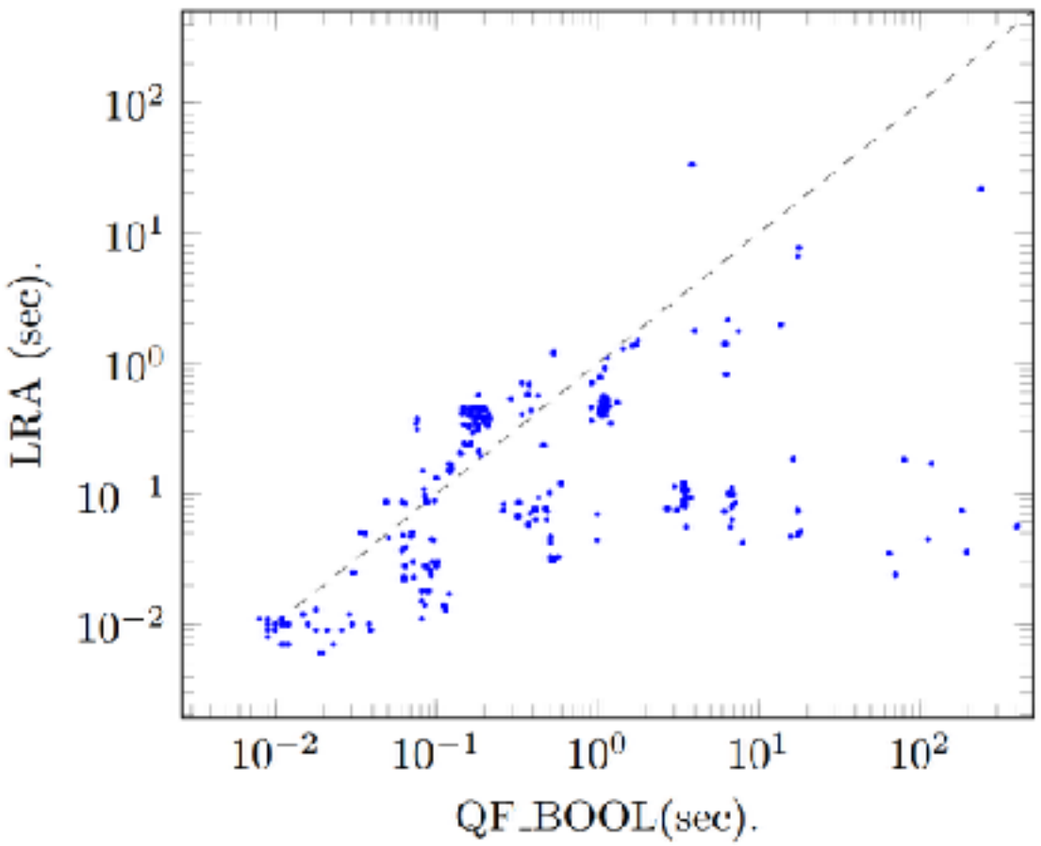
Modelling with Theories

```c
int nondet();

int inc(int x)
{
    x = x + 1;
    return x;
}

void
main()
{
    int y = nondet();
    if(y >= 0 && y <= 10)
    {
        int z = inc(y);
        assert(z > 0 && z <= 20);
    }
}
```

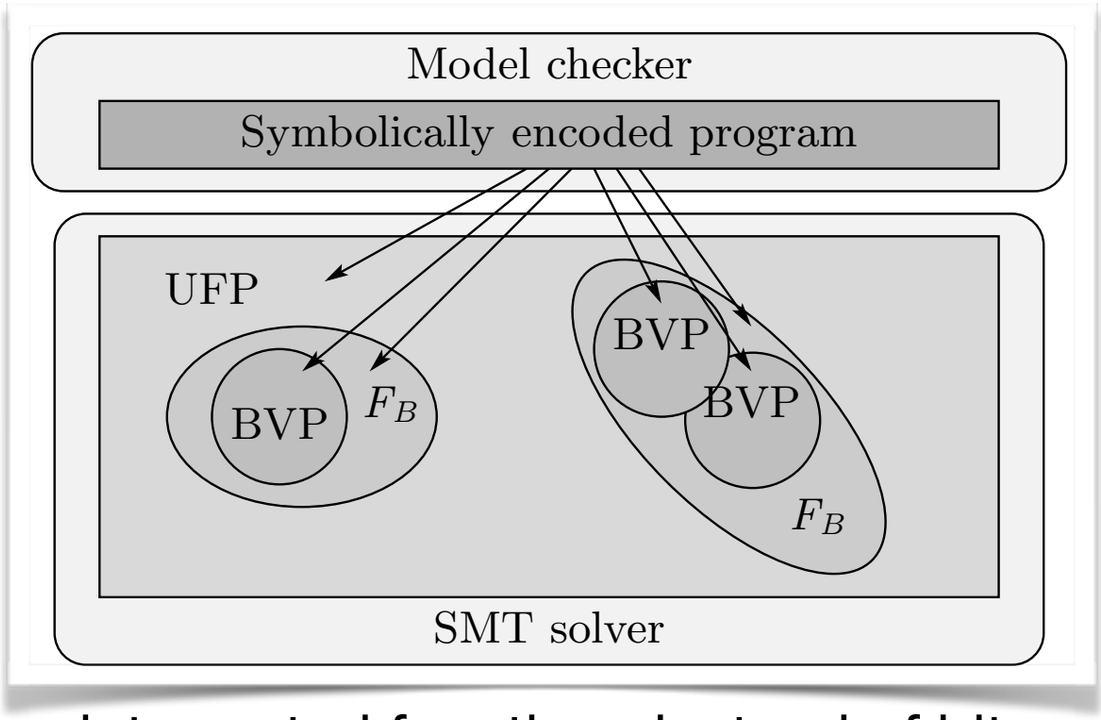


$$(inc(y_0) = y_0 + 1) \wedge ((y_0 \geq 0 \wedge y_0 < 10) \rightarrow ((z_0 = inc(y_0)) \wedge \neg(z_0 > 0 \wedge z_0 \leq 20)))$$


QF_UF(sec).
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-2}
10^{-1}
10^0
10^1
10^2
QF_BOOL(sec).


LRA (sec).
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-2}
10^{-1}
10^0
10^1
10^2
QF_BOOL(sec).

Theory Refinement



- Model a problem with uninterpreted functions instead of bit-precise semantics

- Use counter-examples to guide the refinement from uninterpreted functions to bit-precise circuits

- Add theory-combination-style clauses to connect inputs from uninterpreted functions to the circuits

Algorithm 1: The Counterexample-Guided Theory Refinement Algorithm

input : $P = \{(x_1 = t_1), \ldots, (x_n = t_n)\}$: a program, and t: a safety property
output: $\langle$**Safe**, $\bot\rangle$ or $\langle$**Unsafe**, $CE^b\rangle$

```
1  For all 1 ≤ i ≤ n initialize ρ[x_i = t_i] ← [x_i = t_i]^u
2  ρ[t] ← [t]^u
3  F_B ← ⊤
4  while true do
5      Query ← ρ[x_1 = t_1] ∧ ... ∧ ρ[x_n = t_n] ∧ ¬ρ[t] ∧ F_B
6      ⟨result, CE⟩ ← checkSAT(Query)
7      if result is UnSAT then
8          return ⟨Safe, ⊥⟩
9      end
10     CE^b ← getValues(CE)
11     foreach s ∈ P ∪ {t} s.t. ρ[s] ⊭ [s]^b do
12         ⟨result, _⟩ ← checkSAT([s]^b ∧ CE^b)
13         if result is UnSAT then
14             ρ[s] ← refine^s(ρ[s])
15             F_B ← computeBinding(ρ)
16             break
17         end
18     end
19     if No s was refined at line 14 then
20         return ⟨Unsafe, CE^b⟩
21     end
22 end
```

Construct a query

Get a counterexample

Refine

From Hyvärinen & al.: Theory Refinement for Program Verification, to appear at SAT 2017

Parallel Solving

Parallel SMT Solving

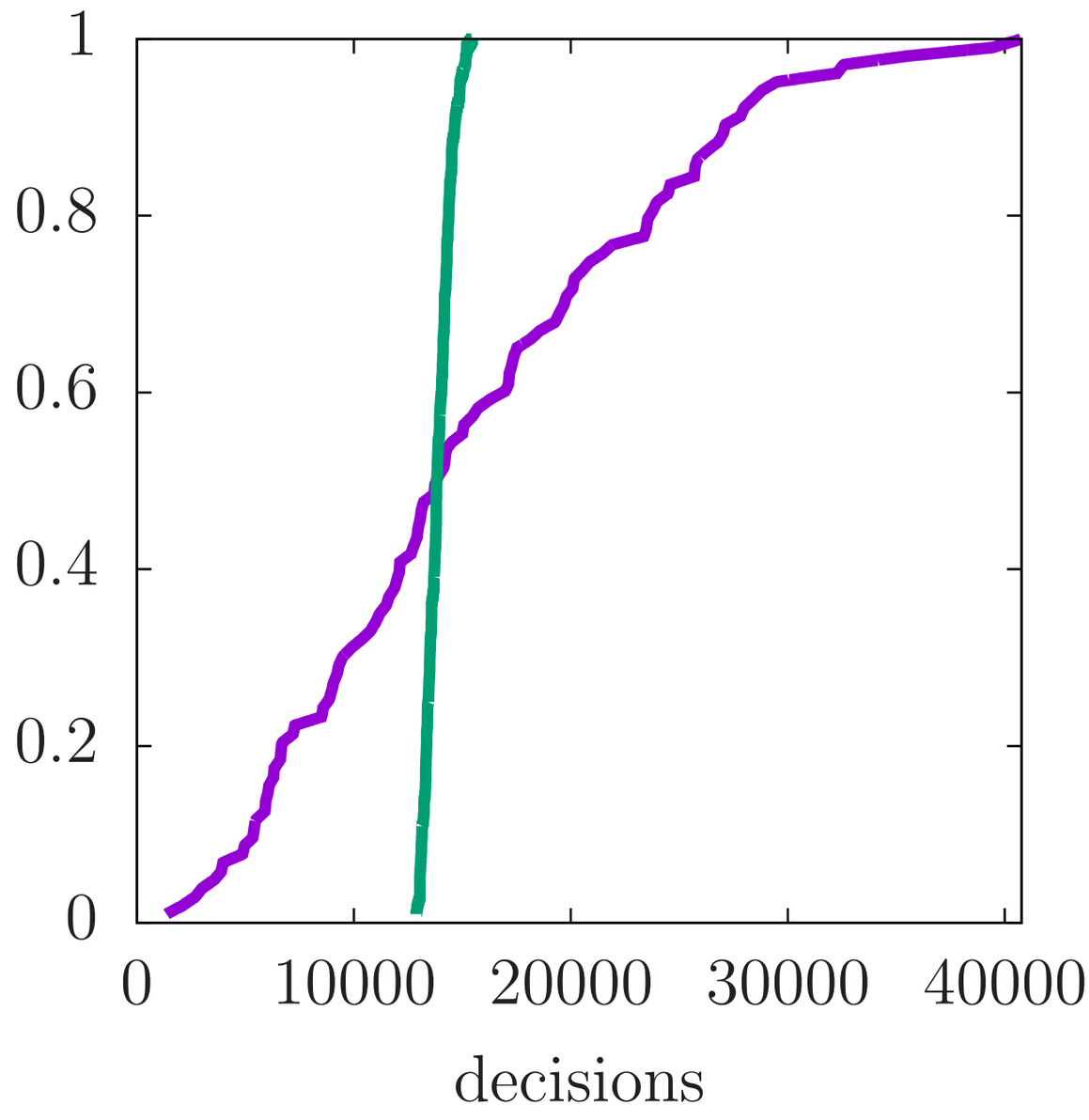
- OpenSMT uses the parallelisation tree as the basis for distributing work

- The idea is to combine divide-and-conquer approaches with algorithm portfolios

- In particular, the framework allows simultaneous partitioning in several different ways

Algorithm portfolios

- Using inherent randomness in runtimes to obtain speed-up
- The speed-up depends on the instance
  - Green instance probably provides little speed-up
  - Magenta instance has much more potential



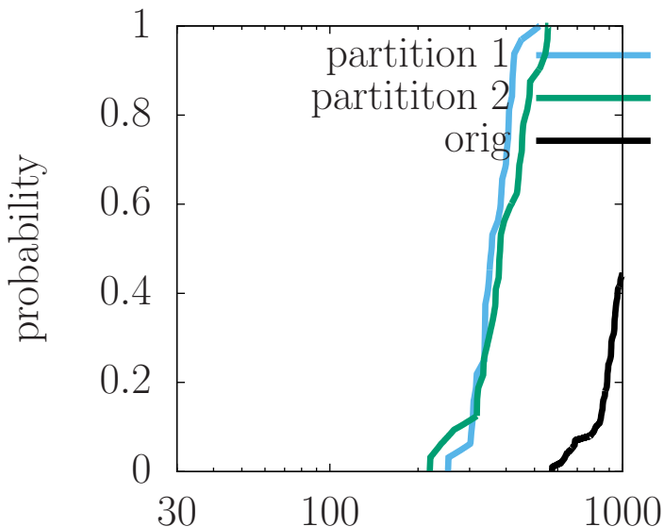
Divide-and-conquer: partitioning the search space to obtain speed-up

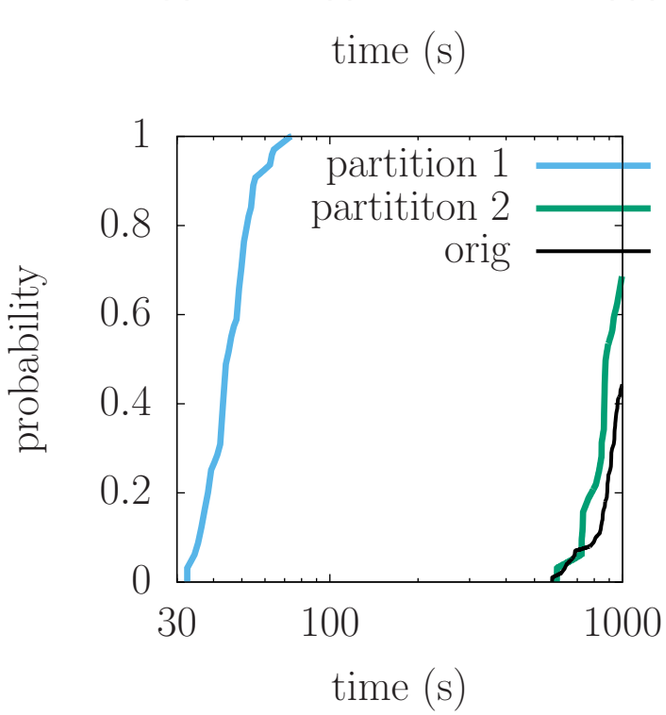
Use a partitioning function to produce from an instance F a set of instances $F_1, \ldots, F_n$ such that $F \equiv F_1 \vee \ldots \vee F_n$

Solve each F_i separately in parallel

The efficiency also depends significantly on the instance

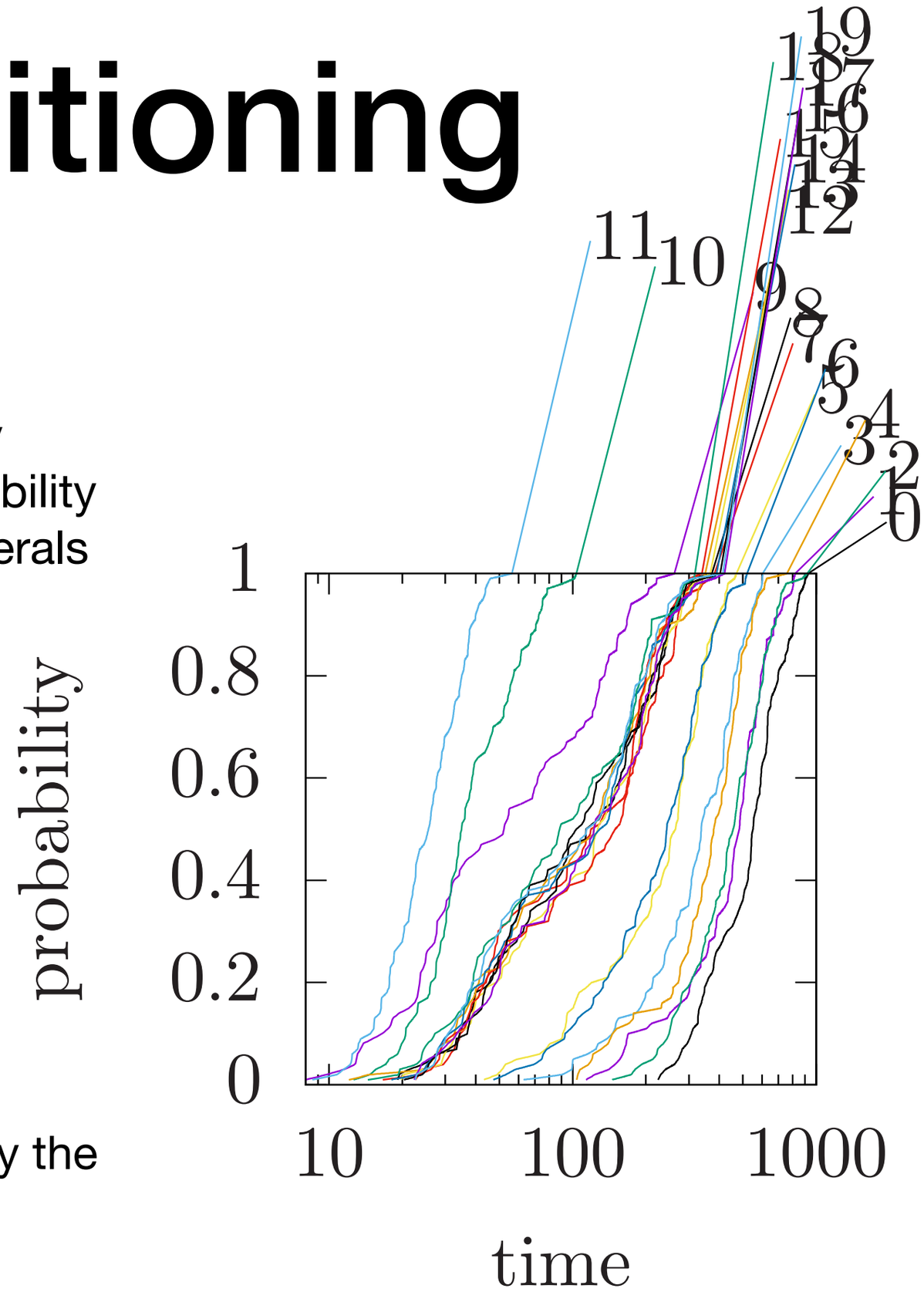
- The above partitioning results in two very even instances
- The partitioning below has one very easy and one equally hard instance





Partitioning

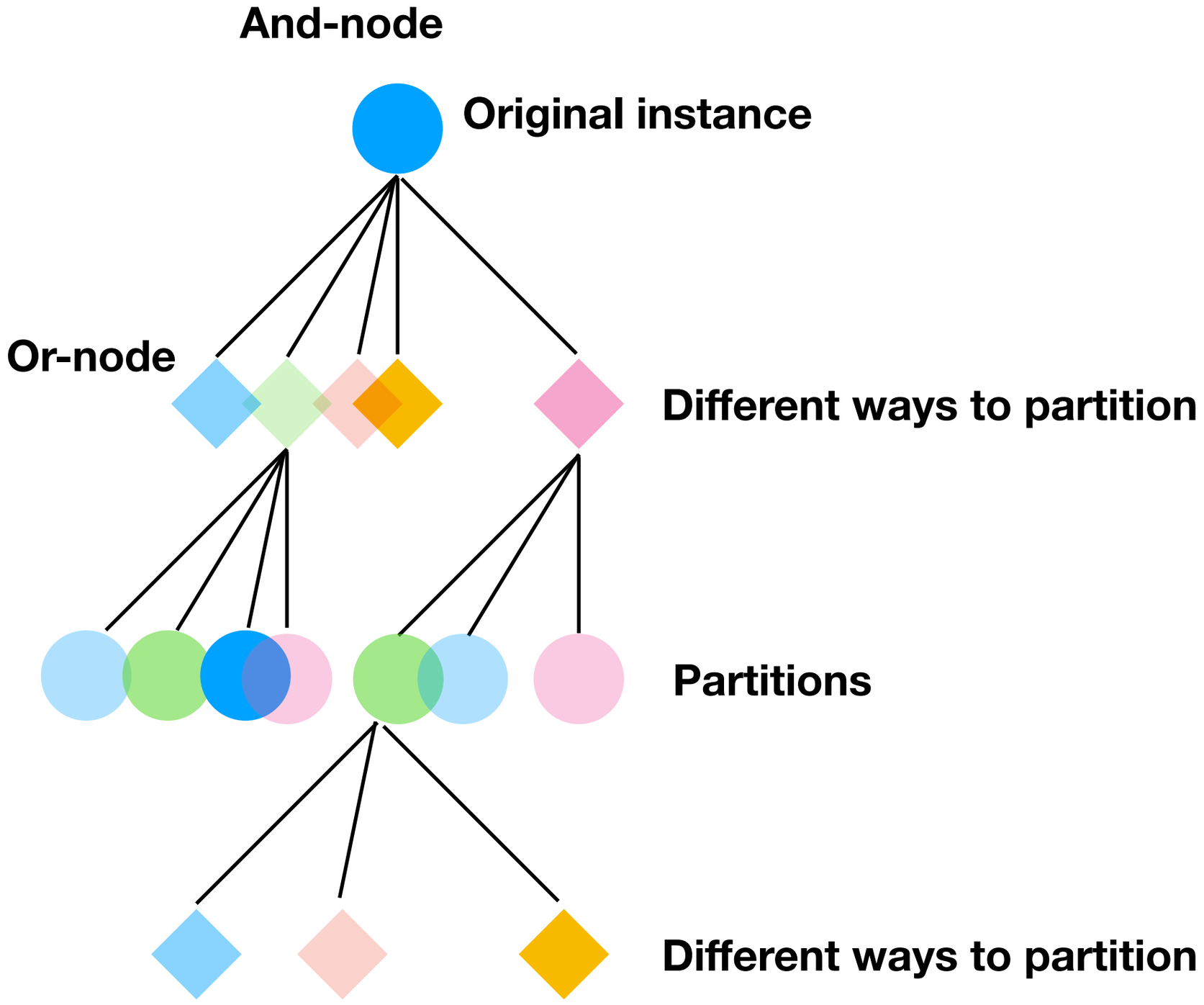
Divide-and-conquer can be tricky,
the efficiency depends on the theory
The figure shows the run-time probability
for an LRA instance when fixing n literals



Initially the instance changes little,
after a while takes a big jump,
and finally gets harder (as presumably the
other instances will get easier)
and stagnates

Parallelisation Tree

- Present under a single framework the ways portfolio and divide-and-conquer can be combined
- Based on an and-or-tree
    - Each and-node correspond to an SMT instance and has or-nodes as children
    - Or-nodes corresponds to applications of partitioning function and have and-nodes as children
    - The root of the tree is an and-node
    - The and-nodes can have solvers solving the contained SMT instance
    - Tree rooted at an and-node is
        - satisfiable, if the contained instance is shows satisfiable by an SMT solver or one of the subtrees is shown satisfiable
        - unsatisfiable, if the contained instance is shown unsatisfiable by an SMT solver or one of the subtrees is shown unsatisfiable
    - A tree rooted at an or-node is
        - satisfiable, if one of its children is satisfiable
        - unsatisfiable, if all of its children are unsatisfiable


And-node
Original instance
Or-node
Different ways to partition
Partitions
Different ways to partition

Conclusions

- A relatively compact SMT solver
- Supports the core SMT theories and incrementality
- Relatively efficient
- Supports embedding into tools through a library interface
- Supports interpolation
- Scales to cloud computing

http://verify.inf.usi.ch/opensmt