What is the State-of-the-Art in DQBF solving?

Gergely Kovásznai

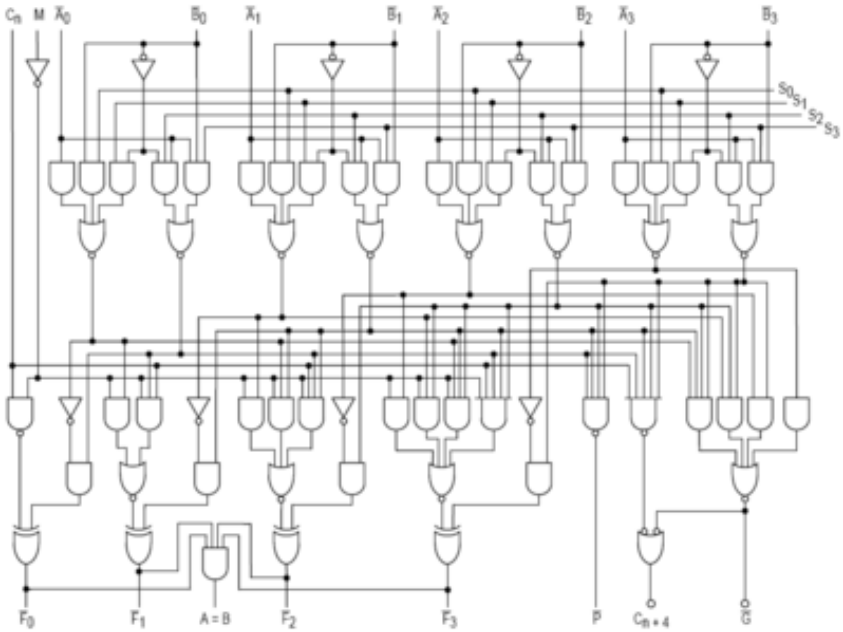
IoT Research Institute,
Eszterhazy Karoly University of Applied Sciences,
Eger, Hungary
kovasznai.gergely@iot.uni-eger.hu

MACS 2016
May 21, 2016
Eger, Hungary

What is this all about?

Formal (a priori) verification of systems.

- Hardware = logical circuits



- Software

```
vate void CollectAllRules(EngineMode engineMode)

var rules = GetRuleCollection(engineMode);
rules.Clear();

foreach (var companion in AllCompanions)
{
    var typeInfo = companion.GetType().GetTypeInfo();
    foreach (var methodInfo in typeInfo.DeclaredMethods)
    {
        foreach (var attribute in methodInfo.GetCustomAt
        {
            var lhs = attribute as LhsAttribute;
            if (lhs != null && lhs.EngineMode == engineM
            {
                var rule = AddOrGetRule(engineMode, lhs.
                rule.Companion = companion;
                rule.Salience = lhs.Salience;
```

Let's start with SAT solving!

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

SAT technology - Applied in our life [le Berre et al., 2011]:

- Intel Core i7 CPUs designed with the help of SAT solvers
- Windows device drivers verified on SAT bases (SMT solver, Z3)

Millions of variables and clauses.

Regular SAT-related competitions:

- SAT Competition 2016, SAT Race 2015
- SMT Competition 2016
- Hardware Model Checking Competition 2015

Let's start with SAT solving!

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

SAT technology - Applied in our life [le Berre et al., 2011]:

- Intel Core i7 CPUs designed with the help of SAT solvers
- Windows device drivers verified on SAT bases (SMT solver, Z3)

Millions of variables and clauses.

Regular SAT-related competitions:

- SAT Competition 2016, SAT Race 2015
- SMT Competition 2016
- Hardware Model Checking Competition 2015

Let's start with SAT solving!

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

SAT technology - Applied in our life [le Berre et al., 2011]:

- Intel Core i7 CPUs designed with the help of SAT solvers
- Windows device drivers verified on SAT bases (SMT solver, Z3)

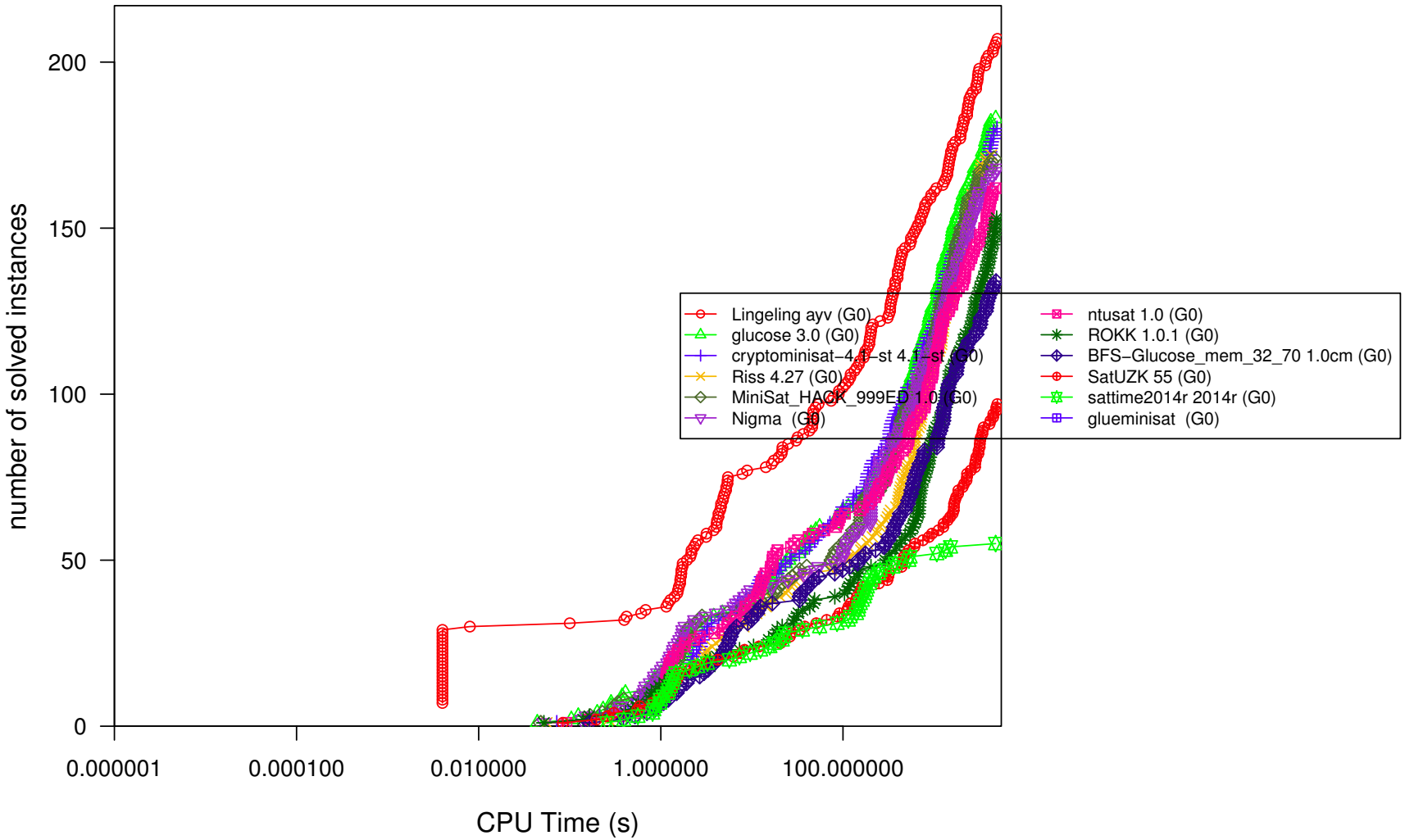
Millions of variables and clauses.

Regular SAT-related competitions:

- SAT Competition 2016, SAT Race 2015
- SMT Competition 2016
- Hardware Model Checking Competition 2015

SAT Competition

Number of solved instances within a given amount of CPU time



QBF – Generalizing SAT

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Quantified Boolean Formulas:

$$\exists e \forall u \; . \; (e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Applications: model checking, formal verification or synthesis, etc.

QBF is not yet widely applied in industry, but is evolving rapidly.

- Several solvers such as Quantor, DepQBF, RAReQS, etc.
- Preprocessors such as Bloqqer, QxBF, Qprocessor
- Proof extractors and validators such as QBFcert

Ten thousands of variables and hundred thousands of clauses.

QBF competitions: QBFEVAL 2016, QBF Gallery 2014

QBF – Generalizing SAT

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Quantified Boolean Formulas:

$$\exists e \forall u \; . \; (e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Applications: model checking, formal verification or synthesis, etc.

QBF is not yet widely applied in industry, but is evolving rapidly.

- Several solvers such as Quantor, DepQBF, RAReQS, etc.
- Preprocessors such as Bloqqer, QxBF, Qprocessor
- Proof extractors and validators such as QBFcert

Ten thousands of variables and hundred thousands of clauses.

QBF competitions: QBFEVAL 2016, QBF Gallery 2014

QBF – Generalizing SAT

Boolean formulas, SAT:

$$(e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Quantified Boolean Formulas:

$$\exists e \forall u \; . \; (e \vee \overline{u}) \wedge (\overline{e} \vee u)$$

Applications: model checking, formal verification or synthesis, etc.

QBF is not yet widely applied in industry, but is evolving rapidly.

- Several solvers such as Quantor, DepQBF, RAReQS, etc.
- Preprocessors such as Bloqqer, QxBF, Qprocessor
- Proof extractors and validators such as QBFcert

Ten thousands of variables and hundred thousands of clauses.

QBF competitions: QBFEVAL 2016, QBF Gallery 2014

DQBF – Generalizing QBF

Henkin quantifiers are used to express the "independence" of variables from each other.

In QBF: An existential variable depends on *all* the universal variables to the left in the quantifier prefix.

$$\forall u_1, u_2\ \exists e\ \forall u_3\ \exists f\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

In DQBF: Variable dependencies are *explicitly* given.

$$\forall u_1, u_2, u_3\ \exists e(\mathbf{u_1}, \mathbf{u_3}), f(\mathbf{u_2}, \mathbf{u_3})\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

Fundamental application: partial/imperfect information games.

Higher complexity:

- SAT – NP-complete
- QBF – PSPACE-complete
- DQBF – NEXPTIME-complete

DQBF – Generalizing QBF

Henkin quantifiers are used to express the "independence" of variables from each other.

In QBF: An existential variable depends on *all* the universal variables to the left in the quantifier prefix.

$$\forall u_1, u_2\ \exists e\ \forall u_3\ \exists f\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

In DQBF: Variable dependencies are *explicitly* given.

$$\forall u_1, u_2, u_3\ \exists e(\mathbf{u_1}, \mathbf{u_3}), f(\mathbf{u_2}, \mathbf{u_3})\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

Fundamental application: partial/imperfect information games.

Higher complexity:

- SAT – NP-complete
- QBF – PSPACE-complete
- DQBF – NEXPTIME-complete

DQBF – Generalizing QBF

Henkin quantifiers are used to express the "independence" of variables from each other.

In QBF: An existential variable depends on *all* the universal variables to the left in the quantifier prefix.

$$\forall u_1, u_2\ \exists e\ \forall u_3\ \exists f\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

In DQBF: Variable dependencies are *explicitly* given.

$$\forall u_1, u_2, u_3\ \exists e(\mathbf{u_1}, \mathbf{u_3}), f(\mathbf{u_2}, \mathbf{u_3})\ .\ (u_2 \vee \overline{u}_3 \vee e) \wedge (u_1 \vee \overline{u}_2 \vee \overline{e} \vee f)$$

Fundamental application: partial/imperfect information games.

Higher complexity:

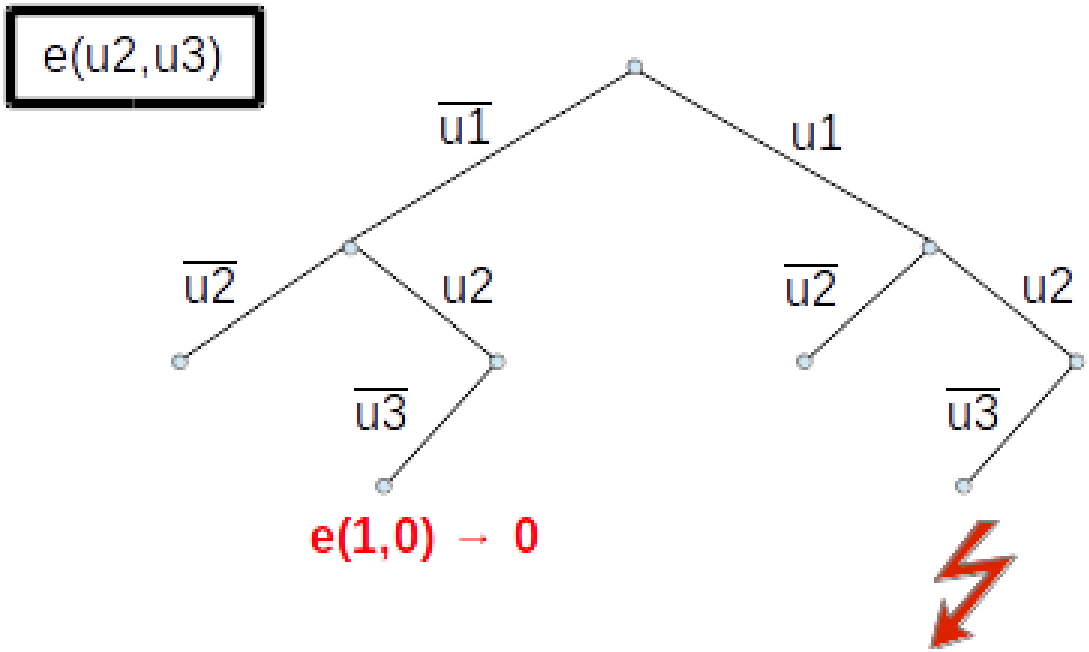
- SAT – NP-complete
- QBF – PSPACE-complete
- DQBF – NEXPTIME-complete

1st solving approach – DQDPLL

[Fröhlich, Kovásznai, Biere, 2012]

Adaptation of QDPLL from QBF to DQBF: e.g., unit propagation, clause learning, universal reduction, watched literals, etc.

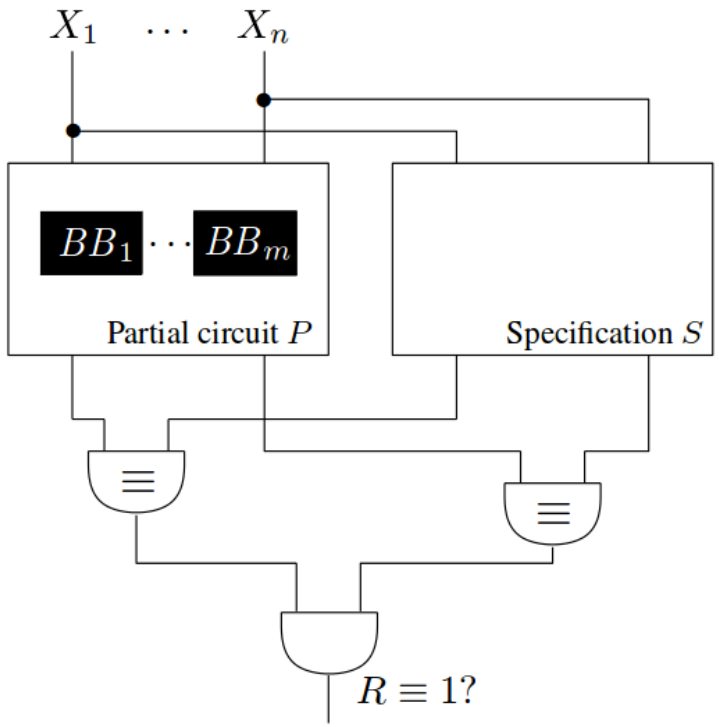
Implemented, but slow. Why?



1st "killer" application

[Gitina, Reimer, Sauer et al., 2013]

"Killer" app: partial equivalence checking (PEC) of circuits



Black boxes = Existential DQBF vars

Inputs = Universal vars

The goal: To *synthesize* Skolem-functions for the existential vars.

A DQBF solver!

[Finkbeiner, Tentrup, 2014]

Unrolling the DQBF to QBF. Given a bound $k \geq 1$,

- Use k *copies* of all variables and the matrix
- Solve the QBF

$$\forall u_1^1, \ldots, u_m^k \; \exists e_1^1, \ldots, e_n^k \; .$$
$$\mathit{consistent}(e_1, k) \wedge \cdots \wedge \mathit{consistent}(e_n, k) \;\Rightarrow\; \bigvee_{1 \leq i \leq k} \neg \phi^k$$

- Ackermann constraints as a *guard*:

$$\mathit{consistent}(e, k) := \bigwedge_{1 \leq i,j \leq k} \left(\bigwedge_{u \in \mathit{deps}_e} u^i = u^j \;\Rightarrow\; e^i = e^j \right)$$

In practice, it can solve only *UNSAT* problems.

1st publicly available and complete solver – iDQ

[Fröhlich, Kovásznai, Biere, 2014]
Adapts and extends the *Inst-Gen* approach to DQBF.

Inst-Gen:

- The solving approach for EPR logic
  - The $\exists^\star\forall^\star.\phi$ fragment of 1st-order logic
  - Has the same complexity as DQBF
- The core of iProver, the most successful EPR-solver
  - Regular theorem proving competition – CASC 2015

iDQ solves SAT and UNSAT instances, quite quickly.

1st publicly available and complete solver – iDQ

[Fröhlich, Kovásznai, Biere, 2014]
Adapts and extends the *Inst-Gen* approach to DQBF.

Inst-Gen:

- The solving approach for EPR logic
  - The $\exists^\star\forall^\star.\phi$ fragment of 1st-order logic
  - Has the same complexity as DQBF
- The core of iProver, the most successful EPR-solver
  - Regular theorem proving competition – CASC 2015

iDQ solves SAT and UNSAT instances, quite quickly.

A new solver – HQS

[Gitina, Wimmer, Reimer et al., 2015]
An improved expansion-based solver:

- Expands DQBF to QBF
  - Eliminates (universal and existential) variables

$$\forall u_1, u_2 \exists e(u_1) \,.\, \phi \;\longrightarrow\; \forall u_2 \exists e, e' \,.\, \phi[0/u_1] \wedge \phi[1/u_1][e'/e]$$

- Eliminates the *minimum set* of variables that cause non-linear dependencies
  - Expressed as a partial MaxSAT problem

Publicly not available.

A new solver – HQS

DQBF PEC benchmarks

	#(sat/uns)	TO/MO	time	#(sat/uns)	TO/MO	time
	adder			bitcell		
HQS	300 (42/258)	0/0	9.7	300 (7/293)	0/0	11.3
iDQ	216 (3/213)	84/0	89828	190 (2/188)	110/0	78107
	lookahead			pec_xor		
HQS	300 (10/290)	0/0	23.2	200 (24/176)	0/0	33.6
iDQ	273 (4/269)	27/0	39540	200 (24/176)	0/0	181.6
	z4			comp		
HQS	240 (72/168)	0/0	4.9	155 (39/116)	9/76	17.8
iDQ	111 (8/103)	129/0	41626	25 (0/25)	180/35	11.6
	C432					
HQS	60 (19/41)	0/180	1333			
iDQ	20 (0/20)	85/135	0.2			

TO = timeout

MO = memory out

Advances of DQBF

- Preprocessing [Wimmer, Gitina, Hist et al., 2015]
- Certification [Balabanov, Chiang, Jiang, 2014]
- Resolution calculi [Beyersdorff, Chew, Schmidt, Suda, 2016]
- etc.

Future work

To find new applications for DQBF. Some candidates:

- Synthesis of protocols from existing ones by enforcing some given "template"
- Solving bit-vector formulas (as part of SMT)
- Inference with partial information of network topology
- etc.