Reasoning Engines for Rigorous System Engineering

Block 3: Quantified Boolean Formulas and DepQBF

1. DepQBF in Practice

Uwe Egly Florian Lonsing

Knowledge-Based Systems Group
Institute of Information Systems
Vienna University of Technology

This work is supported by the Austrian Science Fund (FWF) under grant S11409-N23.

Overview

- DepQBF: search-based, QCDCL solver.
- First release in February 2010, under active development.
- Approx. 17,000 lines of C code.
- Open source under GPL: `http://lonsing.github.io/depqbf/`

- "DepQBF": optional dependency analysis to relax the quantifier ordering.
- Design decision: allow for use as a library.
- No pre/inprocessing (yet).
- Trace generation for certificate generation.
- Based on PCNF, QDIMACS input format.
- Incremental solving: beneficial when solving sequences of closely related PCNFs.
- API to manipulate the input PCNF, configure the solver.
- New version about to be released.

Overview

- DepQBF: search-based, QCDCL solver.
- First release in February 2010, under active development.
- Approx. 17,000 lines of C code.
- Open source under GPL: `http://lonsing.github.io/depqbf/`

- "DepQBF": optional dependency analysis to relax the quantifier ordering.
- Design decision: allow for use as a library.
- No pre/inprocessing (yet).
- Trace generation for certificate generation.
- Based on PCNF, QDIMACS input format.
- Incremental solving: beneficial when solving sequences of closely related PCNFs.
- API to manipulate the input PCNF, configure the solver.
- New version about to be released.

QDIMACS:

- Extension of DIMACS format used in SAT solving.
- Easy to parse.
- Literals of variables encoded as signed integers.
- One quantifier block per line ("a" labels $\forall$, "e" labels $\exists$), terminated by zero.
- One clause per line, terminated by zero.

Example

$$\exists x_1, x_3, x_4 \forall y_5 \exists x_2.(\neg x_1 \vee x_2) \wedge (x_3 \vee y_5 \vee \neg x_2) \wedge (x_4 \vee \neg y_5 \vee \neg x_2) \wedge (\neg x_3 \vee \neg x_4)$$

Encode literals of variables x_i, y_i as signed integers i.

```
p cnf 5 4
e 1 3 4 0
a 5 0
e 2 0
-1 2 0
3 5 -2 0
4 -5 -2 0
-3 -4 0
```

Using DepQBF in Your Application

- Encode your problem in QDIMACS format: support for other formats?
- DepQBF is a standalone QBF solver and...
- ...provides a library with a API in C: add a formula, solve, ...
- Library use is more convenient: incremental calls.

- Compile DepQBF, which produces the library `libqdpll.a`.
- Include the header file `qdpll.h` in your source code.
- Compile and link against the solver library: `gcc your_code.c -L. -lqdpll`
- Call the solver API from your application.

Using DepQBF in Your Application

- Encode your problem in QDIMACS format: support for other formats?
- DepQBF is a standalone QBF solver and. . .
- . . . provides a library with a API in C: add a formula, solve, . . .
- Library use is more convenient: incremental calls.

- Compile DepQBF, which produces the library `libqdpll.a`.
- Include the header file `qdpll.h` in your source code.
- Compile and link against the solver library: `gcc your_code.c -L. -lqdpll`
- Call the solver API from your application.

API: Solver Object Generation

```
/* Create and initialize solver instance. */
QDPLL * qdpll_create (void);

/* Delete solver instance and release all memory. */
void qdpll_delete (QDPLL * qdpll);

/* Ensure variable table size to be at least 'num'. */
void qdpll_adjust_vars (QDPLL * qdpll, VarID num);
```

- No static data: generate multiple solver objects.
- DepQBF uses variable indices as given by the QDIMACS file to index a table of variable objects: keep indices compact in the encoding.

API: Solver Configuration

```
/* Configure solver instance via configuration string.
   Returns null pointer on success and error string otherwise. */
char * qdpll_configure (QDPLL * qdpll, char * configure_str);
```

Possible configuration strings:

- Call `./depqbf -h` for a partial listing of options.
- `--no-cdcl`: disable clause learning and backtrack chronologically from conflicts.
- `--no-sdcl`: disable cube learning backtrack chronologically from solutions.
- `--no-pure-literals`: disable pure literal detection.
- Various learning variants: long-distance resolution, lazy learning.
- Many more: heuristics,. . .

API: Manipulating the Input Formula

Prefix Manipulation:

- Add quantifier blocks of any type at any prefix position.
- Add new variables to quantifier blocks.
- No explicit deletion of blocks/variables: garbage collection.

CNF Manipulation:

- Add/delete clauses.
- No modifications of present clauses: must delete old and add new clause.

Stack-Based Clause Additions/Deletions:

- Push new clauses onto the clause stack.
- Pop most recently added clauses from the stack.

API: Manipulating the Input Formula

Prefix Manipulation:

- Add quantifier blocks of any type at any prefix position.
- Add new variables to quantifier blocks.
- No explicit deletion of blocks/variables: garbage collection.

CNF Manipulation:

- Add/delete clauses.
- No modifications of present clauses: must delete old and add new clause.

Stack-Based Clause Additions/Deletions:

- Push new clauses onto the clause stack.
- Pop most recently added clauses from the stack.

API: Manipulating the Input Formula

Prefix Manipulation:

- Add quantifier blocks of any type at any prefix position.
- Add new variables to quantifier blocks.
- No explicit deletion of blocks/variables: garbage collection.

CNF Manipulation:

- Add/delete clauses.
- No modifications of present clauses: must delete old and add new clause.

Stack-Based Clause Additions/Deletions:

- Push new clauses onto the clause stack.
- Pop most recently added clauses from the stack.

API: Prefix Manipulation (1/3)

```
enum QDPLLQuantifierType:
  QDPLL_QTYPE_EXISTS = -1
  QDPLL_QTYPE_UNDEF = 0
  QDPLL_QTYPE_FORALL = 1


typedef unsigned int Nesting;


/* Add new quantifier block with type 'qtype' at right end of prefix. */
Nesting qdpll_new_scope (QDPLL * qdpll, QDPLLQuantifierType qtype);


/* Add new quantifier block with type 'qtype' at level 'nesting'. */
Nesting qdpll_new_scope_at_nesting (QDPLL * qdpll,
                                    QDPLLQuantifierType qtype,
                                    Nesting nesting);
```

API: Prefix Manipulation (2/3)

```
typedef unsigned int VarID;

/* Add new variable 'id' to the block at level 'nesting'.
   Fails if a variable with 'id' already exists. */
void qdpll_add_var_to_scope (QDPLL * qdpll, VarID id, Nesting nesting);

typedef int LitID;

/* Add new variable 'id' to the current quantifier block
   opened by a previous call of 'qdpll_new_scope' or
   'qdpll_new_scope_at_nesting'.
   Adding '0' closes the current block.
   Fails if a variable with 'id' already exists. */
void qdpll_add (QDPLL * qdpll, LitID id);
```

API: Prefix Manipulation (3/3)

```
/* Returns the nesting level of the current rightmost block. */
Nesting qdpll_get_max_scope_nesting (QDPLL * qdpll);

/* Return largest declared variable ID. */
VarID qdpll_get_max_declared_var_id (QDPLL * qdpll);

/* Returns non-zero iff. variable 'id' has been added to the formula. */
int qdpll_is_var_declared (QDPLL * qdpll, VarID id);

/* Return nesting of block which contains variable 'id'. */
Nesting qdpll_get_nesting_of_var (QDPLL * qdpll, VarID id);

/* Return the type of the block at level 'nesting'.*/
QDPLLQuantifierType qdpll_get_scope_type (QDPLL *qdpll, Nesting nesting);
```

API: CNF Manipulation (1/2)

```
/* Add a literal 'id' to the current open clause.
   Adding '0' closes the clause. */
void qdpll_add (QDPLL * qdpll, LitID id);

/* Pretty-print PCNF to 'out' using QDIMACS format. */
void qdpll_print (QDPLL * qdpll, FILE * out);
```

- Note: `qdpll_add` is used to add variables to blocks and literals to clauses.
- Tautological input clauses are discarded.
- Superfluous literals (double occurrences) in clauses are discarded.
- Literals in input clauses are sorted by prefix order and universal-reduced.
- No free variables: if `id` in a clause is a literal of new variable, then that variable is put into a default existential quantifier block $\exists B_0$ at the left end of the prefix: $\exists B_0 Q_1 B_1 \ldots Q_n B_n . \phi$.
- In practice: first add the prefix, then the clauses.

API: CNF Manipulation (1/2)

```
/* Add a literal 'id' to the current open clause.
   Adding '0' closes the clause. */
void qdpll_add (QDPLL * qdpll, LitID id);

/* Pretty-print PCNF to 'out' using QDIMACS format. */
void qdpll_print (QDPLL * qdpll, FILE * out);
```

- Note: `qdpll_add` is used to add variables to blocks and literals to clauses.
- Tautological input clauses are discarded.
- Superfluous literals (double occurrences) in clauses are discarded.
- Literals in input clauses are sorted by prefix order and universal-reduced.
- No free variables: if `id` in a clause is a literal of new variable, then that variable is put into a default existential quantifier block $\exists B_0$ at the left end of the prefix: $\exists B_0 Q_1 B_1 \dots Q_n B_n . \phi$.
- In practice: first add the prefix, then the clauses.

API: CNF Manipulation (2/2)

```
/* Open a new top-most frame on the clause stack.
   Clauses added by 'qdpll_add' are added to the top-most frame. */
unsigned int qdpll_push (QDPLL * qdpll);

/* Pop the top-most frame from the clause stack.
   The clauses in that frame are considered deleted from the formula. */
unsigned int qdpll_pop (QDPLL * qdpll);

/* Enforce garbage collection of popped off clauses. */
void qdpll_gc (QDPLL * qdpll);
```

- Solver makes sure that incorrect learned clauses and cubes are discarded.
- Pushing is optional: without any push before, clauses are added to a default frame and cannot be removed.

API: Push and Pop

- Must configure by `--dep-man=simple`: use given linear quantifier ordering.
- Useful if a sequence of closely related PCNFs is solved.
- Example: encoding a transition relation for i steps, $i + 1$ steps,...
- No need to parse all the PCNFs from scratch, but only the new clauses.
- More important: solver tries to re-use learned clauses and cubes when solving other PCNFs in the sequence.

In Practice:

- Push and add clauses which are shared between the PCNFs first.
- Push clauses which have to be removed last, so that they can be deleted by a pop.

API: Push and Pop

- Must configure by `--dep-man=simple`: use given linear quantifier ordering.
- Useful if a sequence of closely related PCNFs is solved.
- Example: encoding a transition relation for i steps, $i + 1$ steps,...
- No need to parse all the PCNFs from scratch, but only the new clauses.
- More important: solver tries to re-use learned clauses and cubes when solving other PCNFs in the sequence.

In Practice:

- Push and add clauses which are shared between the PCNFs first.
- Push clauses which have to be removed last, so that they can be deleted by a pop.

API: Deletion of Clauses, Variables, and Quantifier Blocks

Clauses:

- No explicit deletion through API.
- A clause is considered deleted after its frame has been popped from the stack.
- Garbage collection triggered heuristically, or enforced by calling `qdpll_gc`.

Variables:

- No explicit deletion through API.
- A variable x is deleted if all the clauses where x occurs have been deleted.
- The IDs of deleted variables can be re-used: check with `qdpll_is_var_declared`.

Quantifier Blocks:

- No explicit deletion through API.
- A quantifier block is deleted if all of its variables have been deleted.

API: Deletion of Clauses, Variables, and Quantifier Blocks

Clauses:

- No explicit deletion through API.
- A clause is considered deleted after its frame has been popped from the stack.
- Garbage collection triggered heuristically, or enforced by calling `qdpll_gc`.

Variables:

- No explicit deletion through API.
- A variable x is deleted if all the clauses where x occurs have been deleted.
- The IDs of deleted variables can be re-used: check with `qdpll_is_var_declared`.

Quantifier Blocks:

- No explicit deletion through API.
- A quantifier block is deleted if all of its variables have been deleted.

API: Deletion of Clauses, Variables, and Quantifier Blocks

Clauses:

- No explicit deletion through API.
- A clause is considered deleted after its frame has been popped from the stack.
- Garbage collection triggered heuristically, or enforced by calling `qdpll_gc`.

Variables:

- No explicit deletion through API.
- A variable x is deleted if all the clauses where x occurs have been deleted.
- The IDs of deleted variables can be re-used: check with `qdpll_is_var_declared`.

Quantifier Blocks:

- No explicit deletion through API.
- A quantifier block is deleted if all of its variables have been deleted.

API: Solving (1/2)

```
enum QDPLLResult:
  QDPLL_RESULT_UNKNOWN = 0
  QDPLL_RESULT_SAT = 10
  QDPLL_RESULT_UNSAT = 20

/* Solve the given PCNF. */
QDPLLResult qdpll_sat (QDPLL * qdpll);

/* Reset internal solver state, but keep the PCNF and learned constraints.
void qdpll_reset (QDPLL * qdpll);

/* Discard all learned constraints. */
void qdpll_reset_learned_constraints (QDPLL * qdpll);
```

- `QDPLL_RESULT_UNKNOWN` returned only if formula not solved under imposed limits.
- `qdpll_reset` deletes the variable assignments.
- Incremental calls after reset: push, pop, add further clauses.
- For convenience: calling `qdpll_reset_learned_constraints` is **never** required for the correctness of incremental solving. The solver keeps track of learned constraints.

API: Solving (2/2)

```
typedef int QDPLLAssignment;
#define QDPLL_ASSIGNMENT_FALSE -1
#define QDPLL_ASSIGNMENT_UNDEF 0
#define QDPLL_ASSIGNMENT_TRUE 1


/* Get current assignment of variable. */
QDPLLAssignment qdpll_get_value (QDPLL * qdpll, VarID id);

/* Like 'qdpll_get_value' but print to standard output. */
void qdpll_print_qdimacs_output (QDPLL * qdpll);
```

- Call after `qdpll_sat` but before `qdpll_reset`.
- From the command line: `--qdo`
- Get partial certificates of (un)satisfiability as assignments to leftmost variables...
- ...if the PCNF $\exists B_1 \ldots, \phi$ is satisfiable.
- ...if the PCNF $\forall B_1 \ldots, \phi$ is unsatisfiable.
- In practice: useful for encodings of problems from the second level of the polynomial hierarchy with prefix $\forall\exists$ and $\exists\forall$.

API: Solving Under Assumptions

```
/* Assign a variable permanently in the next run (assumption).
   If 'id < 0' then assign variable 'id' to false.
   If 'id > 0' then assign variable 'id' to true. */
void qdpll_assume (QDPLL * qdpll, LitID id);

/* Returns an array of safe arguments to 'qdpll_assume'. */
LitID * qdpll_get_assumption_candidates (QDPLL * qdpll);

/* Returns the subset of assumptions used by the solver
   to determine the result. */
LitID * qdpll_get_relevant_assumptions (QDPLL * qdpll);
```

- Safe arguments to `qdpll_assume` are variables from the leftmost block (recursively).
- Assignments added by `qdpll_assume` are persistent in the next call of `qdpll_sat`.
- `qdpll_reset` removes assignments added by `qdpll_assume` before.
- Constraints learned under assumptions are correct independently.
- For convenience: calling `qdpll_reset_learned_constraints` is **never** required for the correctness of incremental solving. The solver keeps track of learned constraints.

API: Generating Traces and Certificates

```
/* Configure solver instance via configuration string.
   Returns null pointer on success and error string otherwise. */
char * qdpll_configure (QDPLL * qdpll, char * configure_str);
```

- Print the *full* resolution derivation in QRP format to standard output: can be huge!
- `--trace=qrp` (text format) or `--trace=bqrp` (binary format).
- QBFcert framework: `http://fmv.jku.at/qbfcert/`.
- Acknowledgments: Aina Niemetz and Mathias Preiner.
- Resolution proof checking by QRPcheck: `http://fmv.jku.at/qrpcheck/`.
- Certificate extraction (Skolem/Herbrand functions) by QRPcert: `http://fmv.jku.at/qrpcert/`.
- Skolemization/Herbrandization by CertCheck: `http://fmv.jku.at/certcheck/`.
- Checking skolemized/herbrandized formula using a SAT solver.

Remarks

- Please publish your benchmarks!
- Effective use of QBF solvers (sometimes) requires expert knowledge.
- Long-term goal: usability, integrated workflow

Example

C code: push/pop, assumptions.

Efficient QBCP (1/2)

- How to efficiently detect unit clauses, pure literals and conflicts/solutions?

Watching for Unit Literals: dual for cubes.

- In each clause, watch two unassigned literals l_1 and l_2 such that either (1) both l_1,l_2 are existential or (2) l_1 universal, l_2 existential and $l_1 < l_2$.
- If $\neg l_1 \notin A$ and $\neg l_2 \notin A$ then no work has to be done.
- Otherwise, find another unassigned literal to be watched, wrt. $<$ and quantifiers.
- Conflicting clause: no unassigned existential literal left.
- Unit clause: exactly one unassigned existential literal left, under UR.

Efficient QBCP (1/2)

- How to efficiently detect unit clauses, pure literals and conflicts/solutions?

Watching for Unit Literals: dual for cubes.

- In each clause, watch two unassigned literals l_1 and l_2 such that either (1) both l_1,l_2 are existential or (2) l_1 universal, l_2 existential and $l_1 < l_2$.
- If $\neg l_1 \notin A$ and $\neg l_2 \notin A$ then no work has to be done.
- Otherwise, find another unassigned literal to be watched, wrt. $<$ and quantifiers.
- Conflicting clause: no unassigned existential literal left.
- Unit clause: exactly one unassigned existential literal left, under UR.

Efficient QBCP (2/2)

Clause Watching for Pure Literals:

- For each variable x, watch two unsatisfied clauses C_x and $C_{\neg x}$ containing a positive and negative literal of x.
- When satisfied under A: find new C_x and $C_{\neg x}$
- Variable is pure if no new $C_x/C_{\neg x}$ can be found.
- Additional optimization: ignore learned clauses and cubes at the cost of spurious conflicts/solutions.