Validating the Result of a Quantified Boolean Formula(QBF) Solver: Theory and Practice[*]

Yinlei Yu, Sharad Malik
Dept. of Electrical Engineering, Princeton University
Princeton, NJ, USA

Abstract—Despite the increasing use of QBF solvers, current QBF solvers do not provide for any mechanism to verify their results. This paper demonstrates a methodology for independently validating the results of a DLL based QBF solver using the traces generated during the solving process. It also presents a mechanism to extract small unsatisfiable subformulas, called cores, from unsatisfiable QBF instances.

I. Introduction

Recent advancements in Boolean Satisfiability (SAT) solving techniques have inspired significant research on Quantified Boolean Formula satisfiability (QBF) [3][8][10][17]. QBF evaluation is P-SPACE complete, which makes it unlikely that we will find an efficient algorithm[14]. Yet as many practical problems in Electronic Design Automation [2][9][11][15] and artificial intelligence like planning [13] and reasoning (such as [7]) can be naturally transformed into QBF. It is of interest to see if heuristics can be developed that work well on practical instances. Bryant *et al.*[2] use QBF solvers to check the convergence of bounded model checking on decidable fragments of first order logic and apply their methodology on pipeline processors; Gopalakrishnan and his colleagues [9][15] propose verification methodologies for the Intel Itanium memory structure by transforming problems into QBF and SAT. Mneimneh *et al.* transform the diameter problem into QBF but eventually convert QBF into SAT due to the limitation of current QBF solvers[11]. Thus there is practical motivation to solve these QBF instances.

Currently, most state-of-the-art QBF solvers extend DPLL based SAT solving techniques[5][6] to QBF solving[4]. Many of them apply conflict and satisfaction based backjumping as well as clause/cube learning by Q-Resolution [3] or long distance resolution [19].

To our knowledge, no QBF solver generates verifiable results. The current literature does not provide a theoretical framework to independently verify QBF solvers. The QBF Solver Evaluation 2004 Competition even relies on majority to determine the truth value of a QBF problem when discrepancies arise. This motivates the need for an independent QBF verifier to validate the results of a given QBF solver.

The propositional part of a QBF formula is often generated by the conjunction of many different constraints. When a QBF instance evaluates to false, it is desirable to find a smaller set from all the constraints that still results the problem being false. This is referred to as the unsatisfiable core of the problem. This is helpful in diagnosing the cause of unsatisfiability.

This paper builds on the work of Zhang and Malik[18] for verifying SAT solvers. For an unsatisfiable SAT instance, their proposed verifier compiles the traces from the SAT solving process and then derives a DP-like[6] resolution based proof. We propose a similar methodology for independently validating the result of a QBF solver, for both the cases of the result being true and false. Our verifier compiles the traces of a QBF solver to validate the outcome. For *false* QBF formulas, our QBF verifier can extract unsatisfiable cores. This methodology is implemented and evaluated on the yQuaffle QBF solver.

The organization of the rest of paper is as follows: Section II explains related basic terms and definitions. Section III describes the algorithms of the yQuaffle solver and its verifier. It also gives a brief proof for the verifier. Section IV describes an unsatisfiability core extractor for *false* QBF instances using the verifier. Section V provides experimental results. Section VI has some discussion and Section VII concluding remarks.

II. Preliminaries

A quantified Boolean formula is a propositional formula with a quantifier prefix, whose variables are quantified existentially or universally. A QBF formula has the form as $F = Q_1X_1 \ldots Q_nX_nP(X_1, \ldots, X_n)$, where X_1, …, X_n denote n mutually disjoint sets of variables that are quantified by $Q_1 \ldots Q_n$, respectively. $Q_1 \ldots Q_n$ alternate between universal($\forall$) and existential($\exists$) quantifiers. $P(X_1, \ldots, X_n)$ is the propositional part of the formula.[1] For example, $\forall xy\exists abc(x+y'+a)(x+y+b+c)$ is a QBF with $(x + y' + a)(x + y + b + c)$ as its propositional part, variables x and y are universally quantified and variables a, b and c are existentially quantified. Most recent QBF solvers only accept QBF problems with propositional parts in Conjunctive Normal Form (CNF), which is the conjunction of *clauses*, which in turn are disjunction of *literals* (variables in their true or complemented form). Any propositional formula can be transformed into CNF of linear size by introducing extra variables that are then existentially quantified with an innermost existential quantifier in the prefix. *Cubes* are the conjunction of literals. *Empty clauses* (cubes) has no literals, which means *false*(*true*).

In this paper, the subscripts of Q's are referred to as *quantification levels*. A variable $x \in X_k$ has the corresponding quantifier Q_k, so its quantification level is k. In the previous example, x and y have quantification level 1; a, b, and c have quantification level 2.

In this paper, we use *resolution* operation extensively[19]. The resolution of two clauses C_1, C_2 containing c and c' is the disjunction of all literals of C_1 and C_2 except c and c'. For example, the resolution of clauses $(a + c)$ and $(c' + d')$ is $(a + d')$. The resolution of two cubes is defined similarly. Multi-clauses (cubes) resolution is possible by applying binary resolutions iteratively.

[*]This work is supported by Semiconductor Research Corporation Award#: 2002-TJ-1025

[1]In some literature, a QBF formula may contain free variables, which is equivalent to quantifying these free variables existentially as the outermost quantifiers without change the satisfiability of the problem.

III. Algorithm and practical implementation of a QBF verifier

To demonstrate and evaluate our QBF verification methodology, we choose to verify yQuaffle (a new version of Quaffle with same algorithm)[16]. First, we describe the algorithm of yQuaffle:

A. *yQuaffle QBF solving algorithm*

yQuaffle is a typical extended-DPLL QBF solver. As in most QBF solvers, yQuaffle requires the propositional part of the QBF problem in CNF. The following description is based on Zhang and Malik's papers[17][19] and assumes some familiarity with QBF solvers. It is being provided here for review. Fig. 1 is the pseudo code for the yQuaffle algorithm.

```
preprocess();
while (true) {
  decide_next_branch();
  while(true) {
    status = deduce();
    if (status==CONFLICT) {
      blevel = analyze_conflict();
      if (blevel < 0)
        return is_clause_conflict ? UNSAT : SAT;
      else backtrack(blevel);
    } else if (status == SAT) {
      blevel = analyze_SAT();
      if (blevel < 0) return SAT;
      else backtrack(blevel);
    } else break;
  }
}
```

Fig. 1. yQuaffle algorithm in C-style pseudo code

yQuaffle preprocesses the clauses according to quantification rules (`preprocess()`). It deletes all the universal literals with higher quantification levels than any existential literal in the same clause without changing the truth value of the problem.

yQuaffle branches on variables using its decision heuristic(`decide_next_branch()`) and uses *Q-Implication*[4][17] rules to derive implications from unit clauses (`deduce()`).

If a clause (cube) conflict according to the conflict rules in papers[4][17] is found (a cube conflict is an assignment that can satisfy a cube, which in turn satisfy the whole problem): All existential (universal) literals evaluate to zero (one) and no universal (existential) literals are evaluated to one (zero), the conflicting clause (cube) is resolved iteratively with each antecedent clause (cube) of its most recent assigned literals until a Q-Unit clause (cube) is resolved. The resultant clause (cube) is added to the database and Q-implications are derived accordingly. *Long distance resolution* is used, in which if an empty clause (cube) is derived, the QBF evaluation will conclude with result *false* (*true*).

For a search based QBF solver like yQuaffle, the *decision level* of a variable assignment is the number of branches the solver has taken before the assignment has been made. A *Zero Decision Level Variable (ZDLV)* is a variable assigned without branches. It implies that the solver has a proof that having such assignment will not alter the truth value of the QBF formula. Once a satisfying assignment for all the clauses is found (while not satisfying any of the cubes), the solver derives a cube from the assignment to satisfy all clauses. A resolution process on this cube and previous cubes generates a new cube forcing a backtrack. The original propositional part of the QBF formula is augmented by ORing with the new cube. Since the new cube imply the satisfaction of the CNF, the truth value of the QBF is not altered. The new cube is recorded in the database[19] (`analyze_SAT()`).All the cubes forms the *Disjunctive Normal Form*(DNF) part of the augmented problem.

The correctness of the Quaffle algorithm has been shown in Zhang and Malik's paper[17].

B. *Verifying algorithm for QBF*

B.1 Instrumenting the yQuaffle solver for verification

We instrumented the solver as follows:

1. Each clause and cube, either in the original formula or generated in the solving process, is given a unique identification number (*id*).
2. When a new conflict clause (cube) is generated, the *id*'s of all the clauses (cubes) involved in generating the new clause by iterative resolutions are recorded. These clauses (cubes) are called the *antecedents* for the new clause/cube. For example, if clause 381 is built by iteratively resolving clauses 12, 67, 24 and 8, a line `CLR: 381 12 67 24 8` will be added into the trace.
3. When a satisfying assignment is reached, the partial assignment that satisfies all the original clauses is recorded. For example, if the solver find that assignments $x_1 = 1, x_3 = 0, x_8 = 1, x_{12} = 0$ can satisfy all original clauses, a line `SAT: +1 -3 +8 -12` will be added to the trace.
4. The last satisfying assignment/conflict clause (cube) that leads to resolve an empty clause (cube) is recorded. Along with that, all the ZDLV's are recorded with corresponding values and the literals of their corresponding antecedent clauses and cubes. The final clause (cube) may be empty.

The instrumentation of yQuaffle solver consists of about 300 lines of additional code. Most code deals with the special cases yQuaffle uses; for example, yQuaffle removes ZDLV's from newly added clauses and cubes, which forces the checker to verify the validity of the corresponding ZDLV's. The instrumentation should be easily applicable to most other search based solvers like QUBE++ and Semprop. It may be adapted to some other non-search based QBF solvers also. For example, the algorithm in [12] that uses a mixed searching and resolution approach can be checked with the methodology in this paper.

B.2 Depth First Search based verification algorithm

We propose to use a post-root depth-first search to construct a resolution based proof for a QBF problem:

If the final clause/cube is not available, we reconstruct it by resolution (`recursive_build()`). However, the clauses/cubes that are involved in generating the final clause/cube may be unavailable as well, in which case, further reconstruction is necessary. We reconstruct the final clause recursively from the original clauses. When reconstructing a clause, we apply the Q-Resolution process to resolve the clause from its antecedents. In the SAT case, we reconstruct the final cube recursively from resolving satisfying cubes in similar way.

All the literals in the final clause (cube) evaluate to zero (one) by ZDLV's. To derive a proof for the whole problem, we need to derive a proof for such ZDLV's. A proof for a ZDLV is the resolution process for a corresponding unit clause (cube) that can assert such assignment. Such a proof consists of proofs for the antecedents and the proof of the zero level assignments for all the other variables in its antecedent clauses/cubes.

The pseudo-code of the DFS algorithm is shown in Fig. 2 here, a `comp` is either a clause or a cube. `prove_QBF_DFS()` is the main function; it first obtains the final conflict clause/cube (`get_final_id()`). Then it recursively builds the final clause/cube (`recursive_build()`) and generates proof the assignments for every literals in the clause/cube (`prove_unit_lit()`). When building a clause/cube, the verifier recursively builds all the antecedent clauses/cubes and resolve them iteratively to form that clause/cube.

```
prove_QBF_DFS() {
  comp_id = get_final_id();
  comp = recursive_build(comp_id);
  foreach lit in comp
    if (comp.is_clause())
      prove_unit_lit(lit);
    else
      prove_unit_lit(!lit);
  return (no_error_exists());
}
recursive_build(comp_id) {
* if (is_orig_clause(comp_id) RECORD_CORE(comp_id);
  if (is_built(comp_id)) return comp(comp_id);
  if (is_cube(comp_id) && is_sat_cube(comp_id) {
    cb = get_sat_assignment(comp_id);
    check_sat_assignment(cb);
    return cb;
  }
  ante_id = get_first_ante_id();
  comp = recursive_build(ante_id);
  while (other_ante_exists()) {
    next_id = get_next_ante_id();
    next_comp = recursive_build(next_comp);
    comp = resolve(comp, next_comp);
  }
  return comp;
}
prove_unit_lit(lit) {
  ante_id = get_ante_id(variable(lit));
  ante_comp = recursive_build(ante_id);
  foreach ante_lit in ante_comp
    if (lit.is_enforcing_lit(ante_lit))
      if (ante_comp.is_clause())
        prove_unit_lit(lit);
      else
        prove_unit_lit(!lit);
}
```

Fig. 2. DFS based algorithm in C-style pseudo-code

B.3 BFS based algorithm

In a typical yQuaffle solving process, most interim clauses and cubes generated are later deleted to make sure that the solver does not use too much memory space. However, for the verifier, it requires all the related clauses and cubes to be stored in memory during the DFS search process. This may potentially lead to memory blow up problem. A solving process with an hour of CPU time can typically generate a trace file of several gigabytes in length. As the trace size may often be huge, the problem is real.

To deal with the problem, we propose a hard disk backed Breadth First Search (BFS) algorithm similar to the one for SAT verifiers in Zhang and Malik's paper[18].

We first run a preprocessing step to record the location in the log file of every clause and cube in a temporary file.

Then a depth-first search similar to the algorithm in Fig. 2 is used, but we only mark the clause/cube id's and ZDLV's that are necessary to build the whole proof in the temporary file instead of building the clauses and prove the ZDLV's. We record the latest usage of a clause/cube C in generating new clauses/cubes, after which, no other clause/cube is generated by resolving other clauses/cubes with C, so that C may be deleted from memory when the clause database grows too large.

The last step is building and verifying these clauses, cubes and ZDLV's in the sequence of their generation in the solution process. After the last use of a clause/cube in generating new clauses/cubes, it is deleted from memory to save space for newer clauses and cubes. This guarantees that only the same amount of memory as used in the solving process is needed. This is because the solver will only be able to use the clauses or cubes in its memory when generating new clauses or cubes; therefore the active clause/cube set in verifier will never exceed the clause/cube set in the QBF solver.

This approach uses less memory than the depth first search in previous section, yet is slower as it uses disk to store intermediate results. Theoretically, the verifier may still run out of memory in the clause and cube marking stage if the recursion is so deep that it may exhaust the memory space. (We assume we have unlimited hard disk space.) In this case we may need to store the backtracking stack into hard disk as well, which will further slower the verification speed. However, in practice this is never needed.

C. Proof of the correctness of the QBF verifier

In this section, we give a brief proof on the correctness of the QBF verifying algorithm.

Lemma 1. The truth value of a QBF problem is not affected by adding newly derived clauses and/or satisfaction cubes by resolution.

Lemma 2. Given the CNF (DNF) part of a QBF problem with a clause (cube) containing single existential (universal) literal clause, assigning to satisfy (unsatisfy) the clause (cube) does not change the truth value of the QBF problem.

Lemma 3. A universal (existential) literal in a clause (cube) can be removed if no existential (universal) literal with higher quantification level exists in the same clause (cube) without changing the evaluation result of the QBF formula.

The proof for Lemmas 1 to 3 can be found in [19].

Lemma 4. If an empty clause (cube) is derived from resolution, the whole QBF problem is *false* (*true*).

Brief Proof: Resolving an empty clause means the propositional part of the QBF formula implies *false*. That is equivalent to saying the propositional part is *false*, which means the whole problem is *false*. With similar reasoning, resolving a full cube means *true* can be disjuncted to the propositional part of the QBF problem, which make the whole propositional part *true*, in turn making the QBF formula *true*.

Lemma 5. (DAG lemma) The zero level assignments can form a partial order p on the variable set $v_1, v_2, \ldots, v_n$ such that if $p(v_1) \prec p(v_2)$, v_2 will not appear in the antecedence of v_1 that may enforce the Q-Implication (*i.e.*, for an existential variable v_1, any variables other than universal variables with higher quantification level of v_1 are considered as "may enforce Q-Implication".) Similarly for universal variables and their antecedent cubes.

Proof: We may use the chronological order of zero decision level assignments as the partial order. If v_1 was assigned earlier than v_2, it is lower in the partial order than v_2.

First let us consider the case of existential variables. According to the solving process, the antecedent clause of an existential variable v is the clause that a Q-Unit implication took place when v is assigned. Since it is a Q-Unit clause at that time, all the remaining variables where "may enforce Q-Implication" are assigned at decision level zero and evaluate to zero. Since they are already assigned, it is impossible for v to appear in their antecedence, because at that time,

v is not assigned. A similar argument applies to universal variables and their antecedent cubes.

Lemma 6. The proof for variable assignment generated by verifier is correct.

Proof: For a unate variable, *i.e.* a variable that only occur in its true form or only occur in its inverse form in the CNF part of QBF, if it is existential, the verifier has generated its proof of unateness in its preprocessing step. Since assigning to satisfy an existential unate literal and falsify a universal unate literal do not change the satisfiability of the QBF problem, the proof of unateness is a correct proof for its assignment. For non-unate variables we prove by induction:

Base case: Consider the first variable v in the solving process assigned on the zero decision level. If v is unate, the proof of its unateness is the proof of assignment of this variable. Otherwise, if it is existential and its antecedent clause does not contain any other variables that may enforce Q-Implication, by the Q-Resolution rule, the antecedent clause should not have any universal variables with higher quantification level either. So the antecedent caluse is a single literal clause. Therefore, by Lemma 3, the variable may be assigned to satisfy the clause to obtain an equivalent formula as the original one. So the verifier has generated a correct proof for such variable v. If v is universal, a similar argument can show the proof for its assignment is correct.

Induction step: Suppose the first to the k^{th} zero level assignments are proven by the verifier. Consider the $(k+1)^{th}$ zero level assignment v: If v is a unate variable, its unateness proof is the proof for v's assignment. Otherwise, consider v's antecedent clause C (suppose v is an existential variable). C must be a Q-Unit clause. For any existential literal v_1 other than v in C, v_1 must have been assigned before v; thus the assignment for v_1 is among the first to the kth zero level assignments. By the induction hypothesis, the assignment on v_1 has already been proven by the verifier. All the universal literals are not assigned to one in the first to the kth zero level assignments as well. By Lemma 3, having a proof for a literal assignment means all the occurrence of the inverse of the literal in a clause may be removed without changing the evaluation result. By removing such literals, the only literals left in C are unassigned universal literals with higher quantification level than v (otherwise C will not be a Q-Unit clause) and v itself. By Lemma 2, the assignment for variable v is proven. Similar arguments are applied on the proof for v's assignment if v is a universal variable.

Using the base case and the induction step, the proof for the variable assignment for all zero decision level variables is complete.

Theorem 1. The proof generated by the verifier is sound.

Proof: According to the process, the verifier can build a final clause (cube) through resolution. For each literal in the final clause (cube), the solver is able to find the proof for its assignment. The application of such assignments are proven to not affect the truth value of the QBF problem. By applying such assignments on the final clause (cube), an empty clause (cube) is constructed; therefore, the proof of the problem being *false* (*true*) is derived.

IV. Unsatisfiable Core Extraction for QBF problems

When a QBF problem instance evaluates to *false* and its propositional part is expressed in CNF form, we may remove certain clauses without affecting the truth value of the problem. The smaller problem is called an unsatisfying core.

To extract the unsatisfying core, we instrument the QBF verifier by recording all the original clauses involved in the proof. In Fig. 2, it is the line started with a star (`if (is_orig_clause(comp_id) RECORD_CORE(comp_id);`). Applying the original quantifier prefix Q on the resultant clause set, the resulting formula is the core QBF formula. We may run the solver and verifier iteratively to get a minimal unsatisfiable core for the given *false* QBF problem. Note that *minimal* here means the smallest possible core that can be obtained through iterating with yQuaffle and extracting the core by the verifier.

The correctness of the algorithm is obvious. Only the clauses in the unsatisfying core are involved in proving unsatisfiability; by using these clauses, the same proof for the unsatisfiability for the whole problem can be applied to the core.

V. Experimental Results

We implemented the QBF verifier and core extractor as described in the previous sections. The verifier code is written in C++ and is totally independent from the solver code. The experiments are run on a Linux machine with single Pentium 4 2.8GHz CPU with 1MB L2 Cache and 1GB main memory. The compiler is GCC 3.3.2 and the optimization options are set as `-O4 -fomit-frame-pointer`. The experimental results are reported here to evaluate the feasibility and efficiency of the verifiers. The benchmarks are constructed by Rintanen and included in QBF Evaluation 2004[1]. All cases that finish in 1,800 seconds CPU time are listed here.

Table I gives an overview of runtime by the solver and verifier. Column ORT is the runtime of the original yQuaffle solver in seconds. IRT is the runtime of the instrumented solver in seconds. The overhead ratio of the instrumentation is shown in column OH. The column 'LOG' is the size of trace files in bytes. The last column is the run time for the BFS verifier, in seconds too. Each row is a group of benchmarks. The number in parenthesis in the first column is the number of instances in each group. Each number shown in columns 2 to 6 are averages of all the instances in the corresponding group. The BLOCK instances incur a small 5.8% overhead and a small verification time, which corresponds to few satisfying assignments in the solving process. In contrast the CHAIN group has a 57.2% overhead in generating the trace due to its large number of satisfying assignments for the sub-spaces.

TABLE I
Results for verifying the QBF solver

name(#)	ORT	IRT	OH	LOG	BT
blocks(11)	37.28	37.63	0.058	3.1M	1.24
chain(7)	147.02	182.86	0.572	0.86G	464.53
impl(10)	0.01	0.01	0	198	0.01
bwlarge(4)	0.02	0.025	0.2	35K	0.02
toilet(6)	12.11	13.41	0.112	14M	4.22

To evaluate our proposed core extraction technique, we iteratively run the solver and verifier/core extractor on *false* QBF cases until either reaching a fix-point or a limit of iterations. Fig. 3 shows the reduction of clauses number for the QBF instance BLOCK4ii.7.2 in 30 iterations. The Y-axis is the number of clauses of the cores in each iteration and the X-axis is the number of the iterations. We can see the number of clauses drops very quickly in initial iterations, while later only few clauses are dropped in each iteration.

Table II shows the core size and number of iterations. The instances are the same instances that used in verifier evaluation, only *false* cases are used. In the results, most *false* QBF instances have small cores compared with their original size. The first column is

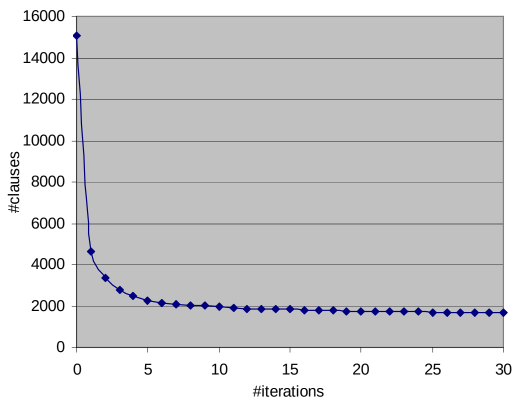

Fig. 3. Core size in iterations for BLOCK4ii.7.2

the name of each instance. Column 'OrigCL' shows the number of clauses in the original instance; column 'CoreCL' shows the number of clauses of the core when either fix-point or a given number of iterations is reached (here the number is 30). The column labeled 'Core%' is the percentage of core clauses over all the original clauses. The number of iterations when the fix-point is reached is shown in the last column (the last iteration that obtain the same size core size as previous one is not counted here). From the table we can see, in most cases, it takes only a few steps to reach fix point. The core size is much smaller than the original problem as well, but the reduction ratio varies greatly. For TOILET7.1.iv.13, the core size is 62.3%, while our algorithm finds a tiny core with 0.06% original clauses in lognBWLARGEB1.

TABLE II
EXTRACTED CORE SIZE FOR *false* QBF INSTANCES

Instance name	OrigCL	CoreCL	Core %	#Iter
BLOCKS3i.4.4	2,928	125	4.27	3
BLOCKS3i.5.3	2,892	406	14.04	13
BLOCK3ii.4.3	2,533	107	4.22	2
BLOCK3ii.5.2	2,707	161	5.95	7
BLOCK3iii.4	1,433	46	3.21	3
BLOCK4ii.6.3	15,061	340	2.26	11
BLOCK4ii.7.2	15,047	1,664	11.06	30*
BLOCKS4iii.6	9,661	203	2.10	4
lognBWLARGEA1	62,820	77	0.12	1
lognBWLARGEB1	178,750	120	0.06	1
TOILET2.1.iv.3	70	20	28.57	1
TOILET6.1.iv.11	1,046	626	59.85	5
TOILET7.1.iv.13	1,491	929	62.31	2
Average:	22,803	371.1	15.23	6.4

VI. Discussion

Some QBF solvers use solving techniques that may differ somewhat from those described in this paper. The verification methodology proposed in this paper can also be applied when other solving techniques are employed.

1. *Trivial Truth* is finding satisfying assignments with some variables unassigned. This does not change the verification paradigm here as they are still SAT cubes. The verifier can still work.
2. *Pure Literals* are variables that occur only in one phase in unsatisfied clauses. They may be assigned to satisfy or unsatisfy such occurrences depend on their quantifiers. The proposed method cannot directly be applied on pure literal deductions. However, if we plug in the proof for the ZDLV's that can satisfy all the original clauses containing the other phase of the variable, a proof for the whole problem can be built accordingly.

VII. Conclusion

This paper proposes an algorithm and methodology for independently verifying a QBF solver. By instrumenting the yQuaffle solver, and running our verifier, the yQuaffle solver is independently validated. Also, we propose methods to extract unsatisfiable core for *false* QBF instances that have been shown to be efficient and result in small cores in practice.

References

[1] QBF evaluation 2004 benchmarks, available from http://www.mrg.dist.unige.it/qbflib/benchmarks/04secondtestset.tar, 2004.
[2] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence testing in term-level bounded model checking. In *Proc. CHARME'03*, vol. 2860 of *LNCS*, pp. 348–362, 2003.
[3] H. K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
[4] M. Cadoli and M. Schaerf. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. In *Highlights of Satisfiability Research in 2000*, 2000.
[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Comm. ACM*, 5(7):394–397, July 1962.
[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.
[7] U. Egly, T. Eiter, H. Tompits, and S. Woltran. Solving advanced reasoning tasks using quantified Boolean formulas. In *Proc. AAAI'00 and IA*AI'00*, pp. 417–422, 2000.
[8] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *Proc. AAAI'02*, pp. 649–654, 2002.
[9] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Proc. CAV'04*, 2004.
[10] R. Letz. Lemma, model caching in decision procedures for quantified Boolean formulas. In *Proc. TABLEAUX'02*, vol. 2381 of *LNCS*, pp. 160–175, 2002.
[11] M. Mneimneh and K. A. Sakallah. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *Proc. SAT'03*, vol. 2919 of *LNCS*, pp. 411–425, 2003.
[12] D. A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Discrete Appl. Math.*, 130(2):291–328, 2003.
[13] J. Rintanen. Constructing conditional plans by a theorem-prover. *J. of Artificial Intelligence Research*, 10:322–352, 1999.
[14] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proc. 5th Annual ACM Symp. on Theory of Computing*, pp. 1–9, 1973.
[15] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proc. CHARME'03*, vol. 2860 of *LNCS*, pp. 81–95, 2003.
[16] Y. Yu and S. Malik. yQuaffle QBF solver. available from http://www.princeton.edu/ chaff/quaffle.html
[17] L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver.
[18] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications.
[19] L. Zhang and S. Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proc. CP'02*, vol. 2470 of *LNCS*, pp. 200–215, 2002.