Journal on Satisfiability, Boolean Modeling and Computation 7 (2010) 65–70

AQME'10

SYSTEM DESCRIPTION

Luca Pulina Luca.Pulina@unige.it
Armando Tacchella Armando.Tacchella@unige.it
DIST, Università di Genova
Viale Causa, 13 – 16145 Genova
Italy

Abstract

In this paper we describe AQME'10, the version of the Adaptive QBF Multi-Engine submitted to QBFEVAL'10.

KEYWORDS: *Self-adaptive multi-engine solver, quantified Boolean formulas, AQME*

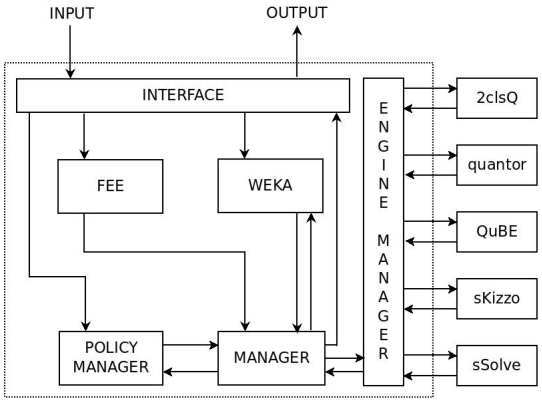
Submitted March 2010; revised May 2010; published July 2010

1. Introduction

The problem of evaluating quantified Boolean formulas (QBFs) is one of the cornerstones of Complexity Theory. In its most general form, it is the prototypical PSPACE-complete problem, also known as QSAT [13]. It has been shown in literature that QBFs can provide compact propositional encodings in many automated reasoning tasks (see, e.g., [1]). To cope with such tasks successfully, QBF solvers ought to be robust, i.e., able to perform well across different problem classes. The results of the QBF solvers competitions (QBFEVAL) show, on the contrary, that QBF solvers are rather brittle. This is to be expected, since every heuristic algorithm will occasionally find problem instances that are exceptionally hard to solve, while the same instances can easily be tackled by resorting to another algorithm, or by using a different heuristic (see, e.g., [7]).

In [9] we studied the problem of engineering a robust QBF solver, i.e., a tool that can efficiently solve formulas across different problem domains without the need for domain-specific tuning. The solver resulting from the above is a multi-engine solver, i.e., a tool that can choose among its engines the one which is more likely to yield optimal results with respect to the current state of the art. It was called AQME, for Adaptive QBF Multi-Engine. In [10] we studied a way to enhance the performances of AQME. We designed an adaptation schema that we called *retraining*, and we applied it to the engine selection-policies whenever they fail to give good predictions. The rewarding results obtained during the last QBF solvers competitions, and the results in [8], in which AQME was used for minimal module extraction from DL-Lite ontologies, witness the effectiveness of such mechanisms.

In this paper we present AQME'10, the version of our self-adaptive multi-engine solver submitted to QBFEVAL'10. In Section 2 we show both its architecture and its core algorithm. The peculiarity of AQME'10 is that its pool of engines is composed only by versions of solvers that participated to QBFEVAL'06. One of the motivations of this choice is that, as organizers of QBFEVAL'10, we wanted a baseline to compare the current progress in the




```
AQME(φ, μ, τ, Σ)
 1  i ← 0,   r ← FAIL
 2  while τ > 0 and r = FAIL do
 3    σ ← APPLY(μ, φ)
 4    Σ′ ← REMOVE(Σ, σ)
 5    τ′ ← GAUGEPREDICTED(τ, Σ, i)
 6    r ← EXEC(σ, φ, τ′),   τ ← τ - τ′
 7    while r = FAIL and Σ′ is not empty do
 8      ⟨σ, Σ′⟩ ← REMOVEFIRST(Σ′)
 9      τ′ ← GAUGEALTERNATIVE(τ, Σ, i)
10      r := EXEC(σ, φ, τ′),   τ ← τ - τ′
11      if r ≠ FAIL then
12        μ ← UPDATE(μ, φ, σ)
13    i ← i + 1
14  return ⟨r, μ⟩
```

Figure 1. The architecture of AQME'10 (left), and its main loop featuring the retraining algorithm (right). The dotted box represents the whole system and, inside it, each solid box represent its modules. Arrows denote functional connections between modules.

development of QBF solvers. Despite the usage of engines dating back to 2006, in Section 3 we show in our experiments that AQME'10 is competitive with the current state of the art.

2. The structure of AQME

Figure 1 (left) presents AQME'10[1] architecture. AQME'10 is written in JAVA, which also makes for an easier interfacing with the WEKA library [6]. Looking at Figure 1, we can see that AQME'10 is composed by six modules:

`INTERFACE`: This module manages both the input received by the user and the output of the whole system. It also dispatches the input data to the remaining modules, as denoted by the outgoing arrows. In particular, `INTERFACE` collects (*i*) the QBF in QDIMACS format; (*ii*) the classifier type and its base inductive model; and (*iii*) the policies related to the time granted to each engine, and the order in which engines are invoked.

`FEE`: Feature Extraction Engine. This module extracts the syntactic features from the input QBF, e.g., number of variables and quantifier alternations, as detailed in [10]. The cost to extract such features is negligible.

`WEKA`: This module is built on top of the WEKA library. `WEKA` is devoted to the prediction of the engine to run. It implements four different inductive models, namely 1-nearest-neighbor, Decision Trees, Decision Rules, and Logistic Regression. `WEKA` receives as input both the classifier type and its base inductive model (from `INTERFACE`) and a vector of features (from `MANAGER`). It returns to `MANAGER` the name of the predicted solver.

`POLICY MANAGER`: This module is devoted to compute all the parameters required by the retraining procedure. It implements the policies related to the time to grant to each

1. AQME'10 is available for download at http://www.mind-lab.it/aqme.

engine, and the order in which solvers are invoked. Further details about these policies can be found in [10]. `POLICY MANAGER` receives from `INTERFACE` the policy parameters passed by the user. It is also called by `MANAGER` whenever retraining occurs.

`ENGINE MANAGER`: This module manages the interaction with the engines. It receives from `MANAGER` informations about the engine to fire, and the time granted to it. At the end of the engine computation, `ENGINE MANAGER` returns to `MANAGER` the satisfiability result.

`MANAGER`: This module works as a coordinator of AQME modules, and it also provides the final result to `INTERFACE`.

The engines of AQME'10, as depicted in Figure 1 (the rightmost boxes) are five state of the art QBF solvers in QBFEVAL'06, namely 2CLSQ [12], QUANTOR2.11 [3], QUBE3.1 [5], SKIZZO [2], and SSOLVE [4]. They are used as "black-boxes", in the sense that AQME interacts with them by means of system calls only.

In Figure 1 (right) we present the main loop of AQME in pseudo-code format. The function AQME in Figure 1 takes as input four parameters:

- φ is the QBF to be solved;
- μ is an extended inductive model comprised of (i) a classifier that predicts the solver to be run on unseen QBFs, and (ii) a training set, i.e. a set of feature vectors corresponding to QBFs, where each vector is labeled by the best engine on that QBF;
- τ is the maximum amount of CPU time granted to solve a single QBF; and
- Σ is a list that contains the basic engines of AQME arranged in a specific order.

The return value of the function is comprised of the result r, and a – possibly updated – model μ. The result can be one of FAIL, SAT or UNSAT according to whether φ could not be solved, was determined to be satisfiable or unsatisfiable, respectively. The algorithm in Figure 1 works as follows:

- An iteration counter i and the result r are initially set to 0 and FAIL, respectively (line 1);
- The outermost **while** loop (lines 2 to 13) ends only when either the resources are exhausted, i.e. $\tau = 0$, or when some engine is successful, i.e., $r \neq$ FAIL.
- Inside the main loop, AQME leverages the model μ to predict the best engine σ to be run on the input QBF φ; this is the task of the function APPLY (line 3).
- The solver σ is removed from the list Σ by the order-preserving function REMOVE (line 4) to obtain the list of alternative solvers Σ'; the function GAUGEPREDICTED computes a specific time limit τ', possibly considering the amount of resources left τ, the list of engines Σ, and the current iteration i (line 5);
- The solver σ is fired on φ with time limit τ' (function EXEC), and the amount of available resources is updated (line 6).

- After the call to EXEC, if the result r is either SAT or UNSAT, then the outermost **while** loop (line 2) ends; otherwise an innermost **while** loop (lines 7 to 12) starts, and it goes on until either the result r is not FAIL, or there are no more alternative solvers to try, i.e., Σ' becomes empty; the innermost loop works as follows:
  - The first solver to try is picked from Σ' (line 8), and a time limit for such solver is computed by the function GAUGEALTERNATIVE (line 9), which works analogously to GAUGEPREDICTED.
  - The alternative solver σ is fired on the input QBF φ with time limit τ' (function EXEC), and the amount of available resources is updated (line 10);
  - If the result of the above call is not FAIL, then a call to UPDATE returns an updated model μ (line 11-12); notice that UPDATE must (*i*) add the feature vector corresponding to φ labeled by the currently selected engine σ to the training set stored in μ, and then (*ii*) swap the classifier stored in μ with a new one obtained considering the updated training set.
- AQME returns a pair consisting of the result r and a model μ (line 14); the model μ is unchanged with respect to the input one either when the first predicted engine is successful, or when all the alternatives are unsuccessful.

Concerning AQME'10, the classifier in μ is 1-nearest-neighbor, while the implementations of GAUGEPREDICTED and GAUGEALTERNATIVE is based on the policy "Trust the Predicted Engine" (TPE). Briefly, it works by allowing a fixed short amount of time to all the engines during the first iteration. Afterwards, should the predicted engine and all the alternative ones be unsuccessful, the whole amount of resources left is granted to the engine predicted in the first place. To see how this works, consider the algorithm in Figure 1 and let L be the initial value of the parameter τ: the function GAUGEPREDICTED and GAUGEALTERNATIVE should return a value T such that $0 < T < L$ when $i = 0$; this can be done because GAUGEPREDICTED "knows" the full amount of resources available when $i = 0$; during iteration $i = 1$, if necessary, GAUGEPREDICTED simply returns $\tau' = \tau$, i.e., the full amount of resources left. In AQME'10, we set $T = 120$s, where the limit L was 1200 seconds[2]. Finally, concerning the order in which alternative engines are probed, in AQME'10 the content of Σ is sorted according according to the QBFEVAL'06 ranking (RAW policy in [10]).

3. Experimental analysis

Table 1 presents the results of AQME'10[3] in QBFEVAL'10, considering all tracks wherein AQME'10 participated. Looking at the table, we can see that AQME'10 was the best solver in both MAIN and RND tracks, while it ranked second best in the remaining ones.

In Figure 2 we report the pie charts related to the total amount of calls to each engine, considering the pool of AQME'10 solved formulas for each QBFEVAL'10 track. Looking at the leftmost plot, related to the MAIN track, we can see that the three most called solver give about the same contribution: QUBE3.1 is used to solve 33% of formulas, while

2. Except for the SH track, for which the CPU time limit was 12 hours.
3. Our empirical analysis is obtained on a family of identical Linux workstations comprised of 10 Intel Core 2 Duo 2.13 GHz PCs with 4GB of RAM running Linux `Debian 2.6.18.5`.

Table 1. Results of QBFEVAL'10 for all prenex CNF solvers. For each track, we report the number of formulas solved within the time limit ("#") and the total CPU time ("Time") spent on the solved instances. Results in boldface are those of the best solver in each track – according to number of problems solved and total time only. NA means that the solver did not participate in a track.

Solver	MAIN		2QBF		SH		RND	
	#	Time	#	Time	#	Time	#	Time
AIGSolve	329	22786.60	NA	NA	**37**	1140.01	NA	NA
AQME	**434**	33346.60	128	2323.11	11	30132.40	**407**	20078.90
DEPQBF	370	21515.30	24	690.42	4	41448.00	342	12895.10
DEPQBF-PRE	356	18995.90	51	877.02	4	33371.90	343	9438.62
NENOFEX	225	13786.90	50	3545.65	3	30194.20	149	34502.80
QMAIGA	361	43058.10	NA	NA	NA	NA	NA	NA
QUANTOR3.1	205	6711.37	48	3689.30	5	57960.90	134	2830.97
StruQS'10	240	32839.70	**132**	1399.30	5	26257.30	117	15480.40

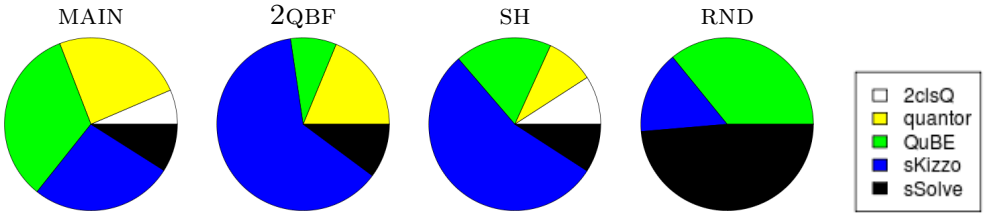


Figure 2. Pie charts representing the total amount of formulas solved by each engine. The figure is organized in four columns, the labels of which denote the related track of QBFEVAL'10.

sKizzo and QUANTOR3.1 are called 27% and 24%, respectively. Both ssolve and 2clsQ are used for less than 10% of times, respectively. However, their contribution is very important: for instance, the formula `C6288.blif_0.10_1.00_0_1_inp_exact` (Family `C6288`, Suite `Scholl-Becker`) was uniquely solved by AQME'10 by using ssolve. Finally, we report that AQME'10 executed 22 retraining procedures.

Considering the plot related to the 2QBF track, we can see that sKizzo is clearly the engine called most of the times (63%). On the other hand, 2clsQ is never called by AQME'10. We also notice that the retraining procedure was called 3 times only. Looking now at the next plot (SH track), we can see that also in this case sKizzo is the most called engine (55%). Finally, looking at the rightmost plot, only three engines were used by AQME'10, namely ssolve (49%), QuBE3.1 (36%), and sKizzo (15%). We also report that AQME'10 called the retraining procedure 15 times.

In conclusion, our experiments show the effectiveness of the multi-engine approach w.r.t. state-of-the-art QBF solvers, considering that the engines of AQME'10 date back 2006. We also report that the performance of AQME is "limited" to the one obtained by the state-of-the-art solver, i.e., the oracle that always fares the best time among a pool of engines. In order to overwhelm such limitation, our current work focuses on the integration between different algorithms, not black-box engines, and some preliminary results are available in [11].

References

[1] C. Ansotegui, C.P. Gomes, and B. Selman. Achille's heel of QBF. In *Proc. of AAAI*, pages 275–281, 2005.

[2] M. Benedetti. sKizzo: a suite to evaluate and certify QBFs. In *20th International Conference on Automated Deduction (CADE)*, **3632** of *Lecture Notes in Computer Science*, pages 369–376. Springer, 2005.

[3] A. Biere. Resolve and Expand. In *Seventh Intl. Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, **3542** of *LNCS*, pages 59–70. Springer Verlag, 2005.

[4] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified boolean formulae. In *Proceedings of the Seventeenth National Conference in Artificial Intelligence (AAAI'00)*, pages 285–290. AAAI Press / The MIT Press, 2000.

[5] E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE++: an Efficient QBF Solver. In *5th Formal Methods in Computer Aided Design conference (FMCAD 2004)*, Lecture Notes in Computer Science. Springer Verlag, 2004.

[6] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, **11**(1):10–18, 2009.

[7] B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, **3**, 1997.

[8] R. Kontchakov, L. Pulina, U. Sattler, T. Schneider, P. Selmer, F. Wolter, and M. Zakharyaschev. Minimal Module Extraction from DL-Lite Ontologies using QBF Solvers. In *Proc. of IJCAI 2009*, pages 836–841, 2009.

[9] L. Pulina and A. Tacchella. A multi-engine solver for quantified boolean formulas. In *13th Conference on Principles and Practice of Constraint Programming (CP 2007)*, **4741** of *LNCS*, pages 574–589. Springer Verlag, 2007.

[10] L. Pulina and A. Tacchella. A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints*, **14**(1):80–116, 2009.

[11] L. Pulina and A. Tacchella. Learning to integrate deduction and search in QBF reasoning. In *Seventh International Symposium on Frontiers of Combining Systems (FROCOS'09)*, **5749** of *LNAI*, pages 350–365. Springer, 2009.

[12] H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. In *Nineth International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*, **4121** of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2006.

[13] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the Fith Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973.