Supplementary material: An efficient solver for ASP(Q) [*]

WOLFGANG FABER[1]

GIUSEPPE MAZZOTTA[2]

FRANCESCO RICCA[2]

[1] *Alpen-Adria Universität Klagenfurt, Austria*
[2] *University of Calabria, Rende, Italy*

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

1 Additional experimental data

This section reports some more data on the Experiments described in Section **??**, reported here to provide a more detailed view on our results for the reviewers.

Table 1 shows the PAR2 score for all the compared systems. Recall that the PAR-2 score of a solver is defined as the sum of all execution times for solved instances and 2 times the timeout for unsolved ones. The lower the score, the better the performance.

Table 2 shows the average memory usage for all the compared systems. Memory usage (measured in MB) is aggregated for instances solved within the time limit (Complete), instances that exceeded the time limit (Timeout) and over all the instances (Total). The lower the memory usage, the better the performance.

Table 3 shows, for each system variant, the number of solved instances, timeouts and memory out for each benchmark and also the total number of solved instances overall.

Table 4 reports the comparison with QASP and ST-UNST implementation (respectively 4a 4b). We considered the best variants of PYQASP against the other systems, QASP with supported back-end solvers and ST-UNST. For each of them, the number of solved instances for each benchmark and the overall number of solved instances is reported.

2 Implementation Details

This section reports a more detailed description of the process used by PYQASP to evaluate an ASP(Q) program.

2.1 Base solver

PyQASP has been entirely developed in Python and it is made by different modules that we will describe in this section. The evaluation of an ASP(Q) program is, basically, done

[*] This work was partially supported by MUR under PRIN project PINPOINT Prot. 2020FNEB27, CUP H23C22000280006, and PNRR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1.

Solver	Par	Arg.Cohe.	Minmax Cli.	QBF
QASP^{DEPS}	771,235.84	321,279.33	36,427.46	546,292.46
PYQASP^{DEPS}	645,221.78	348,019.17	43,217.22	654,649.81
PYQASP$^{DEPS}_{WF}$	589,804.20	299,694.65	21,414.65	619,824.33
PYQASP$^{DEPS}_{WF+GC}$	588,632.92	314,896.34	1,322.85	**513,758.82**
QASP^{QBS}	822,400.00	369,190.24	553.76	768,291.40
PYQASP^{QBS}	822,400.00	382,814.62	587.20	795,651.07
PYQASP$^{QBS}_{WF}$	822,400.00	362,715.16	**458.76**	749,836.74
PYQASP$^{QBS}_{WF+GC}$	822,400.00	481,788.14	524.75	749,905.75
QASP^{RQS}	190,769.32	231,845.74	38,109.56	1,054,759.98
PYQASP^{RQS}	239,202.25	261,084.28	41,908.61	1,091,812.72
PYQASP$^{RQS}_{WF}$	180,695.99	**218,876.82**	36,427.19	1,067,281.23
PYQASP$^{RQS}_{WF+GC}$	**180,692.43**	499,799.74	8,876.10	1,059,731.92

Table 1: PAR-2 score in seconds for each system variants on: (i) Paracoherent ASP (Par. Comp, Par. Rand.); (ii) Argumentation Coherence (Arg. Cohe.); (iii) Minmax Clique (Minmax Cli.); (iv) Quantified Boolean Formula (QBF)

Solver	Complete	Timeout	Total
QASP^{DEPS}	359.06	1281.35	856.37
PYQASP^{DEPS}	108.41	657.65	394.71
PYQASP$^{DEPS}_{WF}$	168.15	607.33	**374.97**
PYQASP$^{DEPS}_{WF+GC}$	170.56	789.57	445.17
QASP^{QBS}	553.15	1892.40	1701.98
PYQASP^{QBS}	272.02	1022.99	**874.03**
PYQASP$^{QBS}_{WF}$	343.78	1232.27	1014.53
PYQASP$^{QBS}_{WF+GC}$	379.94	1304.55	1077.01
QASP^{RQS}	356.44	1409.87	847.78
PYQASP^{RQS}	103.73	701.44	398.42
PYQASP$^{RQS}_{WF}$	152.70	634.67	**380.00**
PYQASP$^{RQS}_{WF+GC}$	177.27	632.06	432.74

Table 2: Average memory consumption in megabyte for each system variant

in two steps that are encoding and solving. In the encoding phase an ASP(Q) program is parsed, identifying ASP programs enclosed under the quantifiers' scope. Then, each ASP subprogram P_i passes through the following pipeline:

1. **Rewriting Module**. This modules is designed to compute syntactical properties of P_i, in order to check whether it is a Guess&Check or trivial subprogram and sub-

sequently apply the appropriate rewriting techniques described in this paper. First of all, P_i is rewritten, taking into account a previous Guess&Check subprogram P_j with $j < i$, if any exists. Then, if the resulting program is trivial, this module returns atoms defined at the current level with an empty program. Otherwise, if it is a Guess&Check program and it is universally quantified then P_i is split into G_{P_i} and C_{P_i}. Moreover, the module computes the result of the transformation τ, introducing a fresh propositional atom u_i, that will be used for rewriting the following levels. As a result, it returns the atoms defined in the guess split of the current subprogram with an empty program. In all the other cases this module returns the current program together with symbols defined at the current level.

2. **Well-founded Module**. This module computes the well-founded model together with the residual program by means of `DLV2` as a back-end system and stores the truth values of literals belonging to the well-founded model.
3. **CNF Encoder Module**. This module takes as input the residual program produced by the well-founded module and encodes it into a CNF formula. In particular, if the residual program is incoherent then it is encoded as the empty clause that is equivalent to $\bot$ and then it breaks the pipeline. Otherwise, if the residual program is empty it is encoded as an empty CNF. In all the other cases the residual program is encoded into a CNF by means of ASPTOOLS.
4. **QBF Builder**. This module produces the final QBF formula by associating the symbols produced by **Rewriting Module** with the respective quantifiers and joining the CNFs produced by the previous module in the final conjunction.

The last step of the encoding phase is combining previous CNFs into the formula ϕ_c. As a result, a QBF formula in `QCIR` format is obtained. The solving step is mainly performed by the solver module, which is a wrapper module for the various QBF solvers. In order to use a solver, the first step in the wrapper is to convert the QCIR formula into an equivalent formula in the solver's input format. Then, the external QBF solver

Table 3: Comparison of system variants on Paracoherent ASP (PAR), Argumentation Coherence (AC), Minmax Clique (MMC), QBF and overall (TOTAL).

Solver	PAR			AC			MMC			QBF			TOTAL
	#SO	#MO	#TO	#SO	#MO	#TO	#SO	#MO	#TO	#SO	#MO	#TO	#SO
QASP^{DEPS}	37	0	477	133	0	193	25	0	20	682	1	309	877
PYQASP^{DEPS}	157	0	357	117	0	209	20	0	25	626	1	365	920
PYQASP$^{DEPS}_{WF}$	189	0	325	148	0	178	35	0	10	648	1	343	869
PYQASP$^{DEPS}_{WF+GC}$	189	0	325	130	0	196	**45**	0	0	699	1	292	918
QASP^{QBS}	0	6	508	102	26	198	**45**	0	0	530	20	442	677
PYQASP^{QBS}	0	0	514	96	0	230	**45**	0	0	518	19	455	659
PYQASP$^{QBS}_{WF}$	0	0	514	107	0	219	**45**	0	0	546	19	427	698
PYQASP$^{QBS}_{WF+GC}$	0	0	514	26	0	300	**45**	0	0	547	13	432	618
QASP^{RQS}	**442**	0	72	197	0	129	23	0	22	350	1	641	1012
PYQASP^{RQS}	**442**	0	72	176	0	150	22	0	23	331	1	660	971
PYQASP$^{RQS}_{WF}$	**442**	0	72	205	0	121	24	0	21	345	1	646	1016
PYQASP$^{RQS}_{WF+GC}$	**442**	0	72	14	0	312	42	0	3	351	1	640	849
PYQASP^{Auto}	**442**	0	72	**222**	0	104	44	0	1	**718**	7	267	**1426**

is executed on the converted formula and the final outcome is computed. In the current implementation we provide the following solver wrappers:

- `QuabsWapper`. It uses the QBF solver `quabs` and doesn't require any format conversion since the solver directly accepts QCIR formulas.
- `RareqsWrapper` This wrapper uses the QBF solver `rareqs` whose input format is `gq`. The conversion from `QCIR` to *gp* is implemented by the external module `qcir-conv` provided by (ref to qcir-conv).
- `DepqbfWrapper` It uses the QBF solver `depqbf` equipped with the QBF pre-processor `bloqqer`. This solver takes as input formulas in *QDIMACS* format and so, translation to CNF is required. In particular, if all universally quantified subprograms were Guess&Check then we know that the produced formula is, indeed, in CNF. So a direct mapping into QDIMACS format exists, just reporting quantifiers and clauses of intermediate CNFs. Otherwise, the external module `qcir-conv` combined with `fmla` is used in order to translate the input formula into an equivalent QDIMACS one. Note that this translation could introduce extra symbols and clauses leading to a bigger formula.

2.2 Automatic selection of the back-end

The automatic back-end selection has been realized by exploiting machine learning models that have been trained on dataset reporting syntactical properties of benchmarks proposed for ASP(Q). For this task we extended our system by adding a module (ASPSTATS) that analyzes ground programs during encoding phase and then, a Random Forest Classifier is used to predict the back-end solver to be used. In order to train the employed model we considered a dataset containing instances from all our benchmarks: Argumentation Coherence, Paracoherent ASP, Minmax Clique, Point of No Return, QBF and 2QBF. In particular, for each instance the features reported in Table 5 have been computed by using ASPSTATS module. As required by the ME-ASP methodology, training set has been constructed by considering only those instances that have been solved exactly by one back-end solver that indeed is the target label, and considered the best oracles available

Table 4: Comparison with QASP and ST-UNST: Solved instances.

Solver	PAR	AC	MMC	QBF	TOTAL
QASP^{DEPS}	37	133	25	682	877
PYQASP$^{DEPS}_{WF+GC}$	189	130	**45**	699	1063
QASP^{QBS}	0	102	**45**	530	677
PYQASP$^{QBS}_{WF}$	0	107	**45**	546	698
QASP^{RQS}	**442**	197	23	350	1012
PYQASP$^{RQS}_{WF}$	**442**	205	24	345	1016
PYQASP^{Auto}	**442**	**222**	44	**718**	**1426**

(a) Comparison with QASP.

Solver	PAR	PONR	2-QBF	TOTAL
ST-UNST	60	30	1416	1506
PYQASP$^{DEPS}_{WF+GC}$	189	0	**2048**	2237
PYQASP$^{QBS}_{WF}$	0	63	346	409
PYQASP$^{RQS}_{WF}$	**442**	**94**	0	536
PYQASP^{Auto}	**442**	88	**2048**	**2578**

(b) Comparison with ST-UNST.

R	Rule count
A	Number of atoms
(R/A)	Ratio between rules count and atoms count
$(R/A)^2$	Squared ratio between rules count and atoms count
$(R/A)^3$	Cube ratio between rules count and atoms count
(A/R)	Ratio between atoms count and rules count
$(A/R)^2$	Squared ratio between atoms count and rules count
$(A/R)^3$	Cube ratio between atoms count and rules count
$R1$	Rule with body of length 1
$R2$	Rule with body of length 2
$R3$	Rule with body of length 3
PR	Positive rule count
F	Normal facts count
DF	Disjunctive facts count
NR	Normal rule count
NC	Constraint count
VF	Universal atoms count
VE	Existantial atoms count
QF	Universial levels count
QE	Existantial levels count
QL	Quantification levels count

Table 5: ASPSTATS features

as labels for multinomial classification. Regarding training phase we used a Random Forest Classifier made of 100 trees that have been trained by using Gini impurity criterion and bootstrap sampling technique.

3 Examples encoding of qasp program into qbf formula

Consider a ASP(Q) program Π of the form: $\exists P_1 \forall P_2 : C$, where

P_1	P_2	C
$\{a; b\} \leftarrow$	$c \leftarrow not\ a,\ not\ b$	$\leftarrow e,\ c$
$\leftarrow a,\ not\ b$	$d \leftarrow a,\ b$	$\leftarrow e,\ d$
	$\{e\} \leftarrow$	

The first step of the encoding produces the following programs by adding interface from previous levels:

G_1	G_2	G_3
$\{a; b\} \leftarrow$	$c \leftarrow not\ a,\ not\ b$	$\leftarrow e,\ c$
$\leftarrow a,\ not\ b$	$d \leftarrow a,\ b$	$\leftarrow e,\ d$
	$\{e\} \leftarrow$	$\{a; b; c; d; e\} \leftarrow$
	$\{a; b\} \leftarrow$	

The resulting CNF encodings are the following:

$$\begin{array}{ll}
CNF(G_1): & (b \lor aux_1) \land (-b \lor -aux1) \land (a \lor aux2) \land (-a \lor -aux2) \land (-a \lor b) \\
CNF(G_2): & (a \lor aux_3) \land (-a \lor -aux_3) \land (b \lor aux_4) \land (-b \lor -aux_4) \land \\
 & (d \lor -b \lor -a) \land (-d \lor b) \land (-d \lor a) \land \\
 & (c \lor a \lor b) \land (-c \lor -b) \land (-c \lor -a) \land \\
 & (e \lor aux_5) \land (-e \lor -aux_5) \\
CNF(G_3): & (a \lor aux_6) \land (-a \lor -aux_6) \land (b \lor aux_7) \land (-b \lor -aux_7) \land \\
 & (c \lor aux_8) \land (-c \lor -aux_8) \land (d \lor aux_9) \land (-d \lor -aux_9) \land \\
 & (e \lor aux_{10}) \land (-e \lor -aux_{10}) \land \\
 & (-e \lor -d) \land (-e \lor -c)
\end{array}$$

where aux_i are hidden atoms are fresh propositional variables introduced by translation The final qbf formula $\Phi(\Pi)$:

$$\begin{array}{l}
\exists\ a,\ b,\ aux_1,\ aux_2 \\
\forall\ c,\ d,\ e,\ aux_3,\ aux_4,\ aux_5 \\
\exists\ aux_6,\ aux_7,\ aux_8,\ aux_9,\ aux_{10} \\
((\phi_1 \leftrightarrow CNF(G_1)) \land (\phi_2 \leftrightarrow CNF(G_2)) \land (\phi_3 \leftrightarrow CNF(G_3))) \land \\
(\phi_1 \land (\phi_2 \lor \phi_3))
\end{array}$$

4 Example of Guess&Check rewriting procedure

Consider a ASP(Q) program Π of the form: $\forall P_1 \exists P_2 : C$, where C is empty and

$$\begin{array}{ll}
P_1 & P_2 \\
\{a(1); a(2)\} \leftarrow & b(1) \leftarrow \\
\leftarrow a(1),\ a(2) & b(2) \leftarrow \\
 & c(1) \leftarrow b(1) \\
 & c(2) \leftarrow b(2)
\end{array}$$

Program P_1 is a guess&check program and so Π it can be rewritten as $\forall P_1' \exists P_2' : C'$:

$$\begin{array}{lll}
P_1' & P_2' & C' = \emptyset \\
\{a(1); a(2)\} \leftarrow & b(1) \leftarrow unsat & \\
 & b(2) \leftarrow unsat & \\
 & c(1) \leftarrow b(1),\ unsat & \\
 & c(2) \leftarrow b(2),\ unsat & \\
 & unsat \leftarrow a(1),\ a(2) &
\end{array}$$

The resulting program contains only one universal level, that is a trivial program and so it can be directly encoded in a QBF formula in CNF. However, well-founded optimization can be further applied but this is a corner case in which the combination of well-founded and guess check optimization results in larger programs. Once we compute the well-founded of P2' (with the interface from previous level) we are unable to derive new knowledge, and all the rules of P2' are kept. On the other hand, if we only apply the the well-founded simplification to P2 (with the interface from previous level) we derive b(1),b(2),c(1),c(2), and we are able to simplify all the rules.