CADET User Guide

Markus N. Rabe
University of California, Berkeley
rabe@berkeley.edu

February 9, 2017

CADET is a solver for quantified boolean formulas (QBFs). It is based on the Incremental Determinization algorithm [10] and currently only supports formulas with one quantifier alternation (2QBF).

Contents

1 Setup

1.1 Requirements

CADET comes with no requirements but those included in the package. A slightly extended version of PicoSAT (build 965) is used. (The modifications

to PicoSAT don't affect CADET's standard mode, but are necessary for the experimental CEGAR extension, for example.)

Any modern C compiler (clang or gcc) should be able to build CADET as of Feb 2017. The testing scripts require Python 2.7.

1.2 Building CADET

To compile the solver type:

```
./configure && make
```

To make sure the solver works correctly execute the test suite:

```
make test
```

One of the test cases will timeout as part of the testsuite and a number of tests will return with the result UNKNOWN, which is the intended result.

2 Usage

The most direct use case for the solver is to call it on a file.

```
./cadet file.qdimacs
```

You can also pipe an instance (in QDIMACS format) in the solver:

```
cat file.qdimacs | ./cadet
```

Note: CADET currently supports piping only for QDIMACS files.

3 Input formats

CADET reads files in both QDIMACS and AIGER format. Files can be zipped with gzip, but must then end with the file ending `gz` or `gzip`.

3.1 QDIMACS

QDIMACS is a popular format for QBF solvers (`http://www.qbflib.org/qdimacs.html`). Consider the following QDIMACS file:

```
p cnf 2 2
a 1 0
e 2 0
1 2 0
-1 -2 0
```

This QDIMACS file corresponds to the QBF $\forall x_1.\exists x_2.\ x_1 \neq x_2$. The header (`p cnf 2 2`) specifies that the file contains 2 variables and two clauses. The following two lines specify the variable indices that are in the universally quantified (`a`

`1 0`) and existentially quantified (`e 2 0`), respectively. The symbol `0` just ends the lines. The remaining lines each specify a clause (again, each terminated by `0`): `1 2 0` corresponds to the clause $x_1 \vee x_2$.

3.2 AIGER

AIGER is a popular format for circuits (`http://fmv.jku.at/aiger/`). CADET interprets AIGER circuits as 2QBF formulas. Internally, CADET transforms circuits into a CNF representation plus some additional constraints that may help the algorithm to solve the instance.

An AIGER circuit C is interpreted as follows. Let C have inputs i_1, ..., i_n, constraints c_1,..., c_m, bad signals b_1,..., b_k and remaining signals S. Further let the inputs be partitioned into controllable I_c and uncontrollable inputs I_u. (Controllable inputs are indicated by naming them with the prefix `pi_` in the AIGER file. This prefix can be changed by the command line option `--aiger_controllable_inputs [string]`.)

An AIGER circuit then represents the formula:

$$\forall I_u.\ \exists I_c, C, B, S.\ \ C(I_u, I_c)\ \wedge\ \left(\bigwedge_{c \in C} c\ \rightarrow\ \bigvee_{b \in B} b\right)$$

It is assumed that constraints refer only to the uncontrollable inputs and thus represent constraints on the universally quantified part of the problem. If that is not the case in the AIGER, the computation may fail. Bad signals typically depend on both, controllable and uncontrollable inputs.

3.2.1 How to encode a problem as an AIGER file?

This encoding is thought to be used for problems of the following common form:

$$\forall X.\ \mathit{constraints}(X)\ \Longrightarrow\ \exists Y.\ \mathit{properties}(X, Y)$$

With the interpretation detailed above both, the constraints and the properties, can be encoded naturally as an AIGER circuit without encoding the implication 'by hand'. The properties should be encoded as bad signals (after negation). The variables Y should be encoded as controllable inputs (see above). All other variables (e.g. those that are needed to encode the circuit) will naturally be quantified on the same level as the variables Y.

If you have a problem that does not fit this scheme, you can encode the negation of the property over X and Y that should hold as a single bad signal.

4 Certificates

CADET produces a certificate for true QBF formulas when it is run with the command line option `-c [file]`. You can either provide a file name for the

file of the certificate (ending in `aag` or `aig`) or you can specify `sdtout` to let CADET print the certificate on the terminal.

As soon as you work with the certificates you may want to install the AIGER tool set (`http://fmv.jku.at/aiger/aiger-1.9.4.tar.gz`) and the ABC (`https://people.eecs.berkeley.edu/~alanmi/abc/`)

4.1 Certcheck

By default CADET produces certificates that can be checked by Certcheck written by Leander Tentrup. Certcheck comes with the distribution of CAQE (`https://www.react.uni-saarland.de/tools/caqe/`).

For example use you can use the following command:

```
./cadet -c certificate.aag file.qdimacs
```

The distribution of CADET comes with several scripts that demonstrate how to generate, simplify, and check certificates.

4.2 QBFcert

To get certificates that are compatible with the QBFcert standard (`http://fmv.jku.at/qbfcert/`), add the `--qbfcert` option to the command line.

Note that QBFcert standard is only compatible with the ASCII format of the AIGER standard (so be sure that the certificate file name ends with `aag`). Also the QBFcert certificates cannot be minimized by ABC.

4.3 Certificates for AIGER files

The certificates CADET generates for AIGER input are relative to the CNF that is used internally. All variable names in the AIGER file are preserved, so it is possible to relate the certificate back to the input file. There is currently no tool to check the validity of certificates of AIGER files.

Certificates for AIGER files may contain unnecessary structure. In particular they contain the definitions of all signals specified in the original AIGER file. Post-processing and simplification with ABC is recommended.

5 FAQ and known problems

- **CADET crashes! Help!**

  Keep calm. A couple of crashes have been observed with the current version on OS X. They all seem to be caused by an extremely rare memory corruption in PicoSAT (which is reported). You can probably avoid this problem by compiling and running the solver in Ubuntu. If the problem persists you found a new bug. Please report to rabe@berkeley.edu!

The good new is that the correctness of your runs should never be affected through this bug and that you can check the certificates, if you don't trust me.

- **CADET does not terminate even though all other solvers terminate quickly on this instance!**

  This can happen since CADET implements a fundamentally different algorithm. There is an experimental option that you can activate with the command line option `--cegar` to let the solver do a little bit of CEGAR additionally to the normal solver mode. This should solve the problem for most classes of problems. Certification is currently incompatible with this feature (and CADET will complain if you try).

- **Will CADET solve problem A? How should I encode my problem such that CADET can solve my problem? How can I help CADET to solve my problem?**

  I have absolutely no idea, but I'm very interested in investigating these issues along concrete examples.

- **Should I apply preprocessing?**

  The short answer is "No." While most other QBF solver profit a lot from preprocessors like bloqqer [3], CADET typically works better without. Be aware that, after preprocessing, the certificates generated by CADET are relative to the preprocessed instance.

5.1 CADET in the context of other QBF solvers

CADET is the first and so far the only 2QBF solver implementing the Incremental Determinization algorithm [10]. There is a number of competitive QBF solvers based on QDPLL and CEGAR, including (in alphabetical order) CAQE [11], DepQBF [8], Qesto [6], RAReQS [5, 7], and sKizzo [1]. For certain classes of benchmarks also the solvers quantor [2], AIGSolve [9], and QuBE [4] can be interesting. Performance varies wildly among different classes of benchmarks and it is hard to say in advance which QBF solver works best on your formulas. Just try different solvers and use what works best for you.

The use cases in which you should definitely try using CADET are:

- Certificates: CADET certifies its results with barely any overhead. Unlike other solvers, CADET does not have to deactivate any optimizations for certification. Its certificates are also much smaller compared those generated by CAQE or DepQBF.

- Fast solving time: If you need an answer for simple queries quickly, CADET may be right for you. If CADET terminates it is on average 10 times faster than solvers that need bloqqer as a preprocessor. (Though this may vary by benchmark.)

- Overall solving power for 2QBF: With the experimental option `--cegar`, CADET significantly outperforms other solvers on most benchmarks. However, certification for true formulas (returning the result SAT) is not supported right now. So use with caution.

References

[1] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Proceedings of CADE*, 2005.

[2] Armin Biere. Resolve and expand. In *Proceedings of SAT*, 2004.

[3] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *Proceedings of CADE*, pages 101–115, 2011.

[4] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE: A system for deciding quantified boolean formulas satisfiability. In *Proceedings of IJCAR*, pages 364–369, 2001.

[5] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. In *Proceedings of SAT*, pages 114–128, 2012.

[6] Mikolás Janota and Joao Marques-Silva. Solving QBF by clause selection. In *Proceedings of IJCAI*, pages 325–331. AAAI Press, 2015.

[7] Mikolás Janota and João P. Marques Silva. Abstraction-based algorithm for 2QBF. In *Proceedings of SAT*, pages 230–244, 2011.

[8] Florian Lonsing and Armin Biere. DepQBF: A dependency-aware QBF solver. *JSAT*, 7(2-3):71–76, 2010.

[9] Florian Pigorsch and Christoph Scholl. An aig-based qbf-solver using sat for preprocessing. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 170–175, New York, NY, USA, 2010. ACM.

[10] Markus N. Rabe and Sanjit A. Seshia. Incremental determinization. In *Proceedings of SAT*, Berlin, Heidelberg, 2016. Springer-Verlag.

[11] Markus N Rabe and Leander Tentrup. CAQE: A certifying QBF solver. In *Proceedings of FMCAD*, pages 136–143, 2015.