Istituto per la Ricerca Scientifica e Tecnologica (IRST)
Via Sommarive 18, 38055 Povo, Trento, Italy

User Manual

(Manual v0.4, documenting solver v0.8-beta, rev. 257)

Marco Benedetti
mabene@gmail.com

20th October 2005

Contents

1 Introduction

sKizzo is a QBF solver, i.e. a program designed to decide whether or not a given *Quantified Boolean Formula* has at least one model. It works with formulas in *prenex conjunctive normal form* represented in the DIMACS 1.1 format. The simplest possible usage of sKizzo (given a `myInstace.qdimacs` file containing a QBF of interest) consists in issuing the command:

```
sKizzo myInstance.qdimacs
```

and wait for a TRUE/FALSE answer (see Section 2). Advanced users can:

- Customize the behavior of the solver using the options documented in Section 3.
- Use sKizzo in conjunction with the companion tools[1] ozziKs and libQBM to certify the (un)satisfiability of formulas, extract unsatisfiable cores, manage and query stand-alone certificates of satisfiability (models) for QBFs.

This manual is organized as follows:

- Section 2 gives details on the input/output behavior of the solver (commandline syntax and exit values).
- In Section 3 we present and discuss some options useful to customize the behavior of the solver.
- Section 4 shows how to make the solver process a set of instances instead of just one.
- Section 5 contains notes about the run-time behavior of the solver and its customization.
- In Section 6 we give a brief introduction to the data structures and algorithms used by sKizzo, with the aim to make the user able to understand the meaning of the commandline options.

1.1 Contact and feedback

At present, sKizzo (version 0.8-beta, revision 257) is in beta testing. Feedback from users is welcome on both the solver and its documentation. Please send an e-mail to

mabene@gmail.com

mentioning "sKizzo" in the subject in case you:

- are able to make the solver crash under specific, reproducible circumstances.
- have problems solving some "reasonbably-sized" instance you were expecting to be able to solve with sKizzo.
- find errors in the documentation.
- observe any unexpected behavior of the solver.
- need hints on personality tuning (Section 5.1).
- miss some feature you would like to see in future releases of the software.
- encounter any circumstance or are able to give any suggestion that could help to improve the solver or its documentation.

[1] Not yet available for download.

2 How to launch sKizzo

To launch sKizzo, use the following syntax.

```
sKizzo [OPTS] (FILE|DIR) [TIMEOUT]
```

where you have to provide:

1. An optional list of space-separated options `OPTS`. Such options are used to modify the solving process and the input/output behavior of the solver. They are discussed in Section 3.
2. A mandatory `DIR` or `FILE` path:
   (a) `FILE`, the full name (possibly with a path) of a DIMACS file containing the QBF instance to be solved. The extension of this file has to be one of the following: "`.qdimacs`", "`.dimacs`", "`.qcnf`".
   (b) `DIR`, the root of a directory subtree which is to be recursively visited. When you give a directory as argument, the solver works in *batch mode*. See Section 4.
3. An optional `TIMEOUT` (in seconds). When a timeout is given, sKizzo works for no longer than the specified amount of time, then gives up (exit code 30, see below).

Possible **exit values** for the command—returned to the program that called sKizzo or to the shell from which the solver was launched—are:

10 on TRUE instances

20 on FALSE instances

30 on timeout

40 on a "unable to decide" condition

-1 on unrecoverable internal error

-2 on I/O error or file not found.

-3 on commandline parse error

-4 on DIMACS parse error

-5 on a SIGBUS/SIGSEV crash

-6 on unmanaged out-of-mem conditions

These return codes are valid when one single instance is given as argument. For the return codes in batch mode see Section 4.2.

3 Command-line options

Each option is identified by a small string beginning with the minus "-" sign. Some options are followed by one or more optional or mandatory arguments. The order of options is not relevant.

3.1 Options that modify the personality of the solver

`-style STYLE` selects a solving *personality* (see Section 6 and Section 5.1), where `STYLE` is a string containing one or more of the following switches:

- `Q`, to enable ground QBF reasoning.
- `S`, to enable incomplete symbolic reasoning.
- `R`, to enable complete symbolic reasoning.
- `B`, to enable DPLL-like branching reasoning.
- `G`, to enable SAT-based reasoning.

By default, all the switches are enabled (equivalent to `-style QSRBG`). So, for example, "`-style BS`" disables q-resolution based preprocessing, prevents the SAT solver from being called, and forces complete symbolic rules not to be used.

Notice that—besides all the considerations related to the hybridization of different evaluation algorithms—the `Q` and `S` styles are not able to solve all the instances (they adopt a refutationally incomplete inference rule set). The `G` style is complete in principle, but often its application cannot even be commenced (without at least a `Q,S`-based "preprocessing") due to exponential space requirements. Finally, the `R` style is complete, but a pure `R`-based personality may fail to evaluate the instance due to mem-out problems. Hence, depending on the combination of styles (personality) chosen and on the particular instance at hand, a *unable to decide* outcome may result. By combining the different styles, $\sum_{i=1}^{5} \left(\begin{array}{c} 5 \\ i \end{array} \right) = 31$ personalities can be obtained.

`-BS COOP` controls the way the `S` style cooperates with the `B` style. In particular, this option decides which symbolic rules (among those you have made active with the `-use`/`-dontuse` switches) will be applied after each existential branching step.

`COOP` is one of the following strings:

`none` to disable all the symbolic rules, hence performing a pure branching reasoning.

`ucp` to leave SUCP as the only active rule.

`lazy` to behave like `ucp`, but limiting the time spent in SUCP w.r.t. the time required by branching.

`nobin` to leave all the rules active but the binary ones (SER,SHBR).

`full` to apply all the active symbolic rules after each existential branching step.

By default, the `ucp` style is adopted (`-BS ucp`). The support for all the cooperation styles but `none` and `ucp` is experimental.

-SG WHEN controls the condition under which the transition S→G occurs. Such control is of interest when the current formula is such that both styles could in principle be used (and the user has some good idea about which one is the best).

WHEN is one of the following strings:

asap to make the transition happen *as soon as* it is affordable to generate and solve the propositional expansion—even if further symbolic simplifications would be possible.

fix to prevent the transition from happening until every active symbolic rule in the S state has reached the *fixpoint*[2]—even if the propositional expansion of the current formula is already small enough for the G style.

auto to leave sKizzo decide when to switch. The heuristics used for this decision is as follows:

1. if the ground expansion is not affordable (not enough memory), the possibility of a switch is not taken into account at all;
2. if the ground expansion becomes "very small" the switch is triggered immediately;
3. if the ground expansion is affordable but it is not not very small, the symbolic reasoning is continued, but only so long as it is shrinking further the size of the ground expansion in a "fast" way.

By default, the auto condition is adopted (-SG auto).

3.2 Options to tune the Q style (qbf reasoning)

-linear prevents quantifier tree reconstruction from being performed. The solver thus works with the linear prefix of the input DIMACS instance. The effects of tree reconstruction VS a linear prefix are discussed in [1, 5]. In general, there is no good reason to disable tree reconstruction. By default, tree reconstruction is enabled.

-use RULE$_1$ [RULE$_2$ RULE$_3$...] disables all the inference rules but RULE$_1$, RULE$_2$, etc., where RULE$_i$ is one of the following:

QUCP for Quantified Unit Clause Propagation.

QPLE for Quantified Pure Literal Elimination.

QRES for Q-resolution.

Q* for the QUCP, QRES, QPLE rules.

By default, all the rules are enabled (equivalent to "-use Q*").

-dontuse RULE$_1$ [RULE$_2$ RULE$_3$...] enables all the inference rules but RULE$_1$, RULE$_2$, etc., where RULE$_i$ is like in the -use option.

[2]When using the fix switch, you'll probably want to disable the R style, and possibly the SRES rule either. If you don't, you have to consider a slightly unnatural meaning for the word "fixpoint". Indeed, if you disable neither R nor SRES the fixpoint condition is only reached when the formula is decided (SRES is refutationally complete), hence the ground expansion never takes place unless an out-of-memory condition makes it necessary to abandon the R style in favor of the G one. If you disable R but not SRES, the fixpoint is reached when all the refutationally incomplete rules have reached their fixpoint and the SRES rule—in the *bounded* version applied during S–is not able to shrink the formula any more. See Section 6.3 and Section 5.2 for more details.

3.3 Options to tune the S style (incomplete symbolic reasoning)

-use RULE$_1$ [RULE$_2$ RULE$_3$...] disables all the inference rules but RULE$_1$, RULE$_2$, etc., where RULE$_i$ is one of the following:

- **SUCP** for Symbolic Unit Clause Propagation.
- **SPLE** for Symbolic Pure Literal Elimination.
- **SPLEi** for Incremental Symbolic Pure Literal Elimination.
- **SSUB** for Symbolic Subsumption.
- **SER** for Symbolic Equivalence Reasoning.
- **SHBR** for Symbolic Hyper Binary Reasoning.
- **SRES** for Symbolic Directional Resolution.
- **S*** for the SUCP, SPLE(i), SER, SHBR, SRES, SSUB rules.

By default, all the rules are enabled (equivalent to "-use S*").

-dontuse RULE$_1$ [RULE$_2$ RULE$_3$...] enables all the inference rules but RULE$_1$, RULE$_2$, etc., where RULE$_i$ is like in the -use option.

-pre X$_1$:Y$_1$ [X$_2$:Y$_2$ ...] constraints the order in which symbolic rules may be executed. Each X$_i$ and each Y$_i$ is the name of a rule (among SUCP, SPLE, SPLEi, SSUB, SER, SHBR, SRES). The switch -pre X:Y prevents the application of rule Y so long as the rule X has not reached its fixpoint. Put it another way: The fixpoint of X is a precondition to apply Y. For example, -pre SUCP:SER allows to perform symbolic equivalence reasoning only after all the symbolic unit clauses have been propagated. You can set multiple precedences (in one single -pre statement), but be aware that no loop occurrence check is performed, so you may drive the solver into a deadlock.

When no precondition is given, rules are applied regardless of their relative fixpoint status. In general, this enables a more flexible behavior. Sometimes, a specific order of application yields (far) better results than others.

By default, only one precedence is enforced, namely that symbolic hyper-binary reasoning is not attempted so long as there is some unit clause around (equivalent to "-pre SUCP:SHBR").

-nopartial avoids splitting clauses by partial subsumption. When a symbolic clause is only partially subsumed during SSUB or by any symbolic assignment, just do nothing. The non-splitting behavior increases the redundancy in the formula (the ground expansion contains replicated clauses), but keeps the symbolic representation more compact. The tradeoff seems generally in favor of splitting clauses. By default, partial subsumption splits clauses.

-hbdd ['0'..'7' ':'] '0'..'19' is used to select both the heuristics for the initial variable ordering in the BDD manager, and the heuristics for dynamic reordering. The syntax -hbdd S:R gives both the initial ordering rule S $\in \{0, \ldots, 7\}$ and the reordering style R $\in \{0, \ldots, 19\}$ to be used. If you only provide one argument (e.g.: -hbdd R) you are selecting the reordering heuristics, while the initial ordering rule defaults to the value 0 (this is for backward compatibility with the syntax of the solver up to version $0.7.1$, where the initial ordering was not configurable).

sKizzo uses the CUDD package [10] as a core tool to manage binary decision diagrams. The CUDD implements a number of reordering heuristics, and the R value just tells sKizzo

which heuristics is to be selected in the CUDD. An actual reorder is explicitly triggered by sKizzo on certain occasions, and left to the auto-reordering CUDD's facility in other cases. The integer R $\in \{0, \ldots, 19\}$ selects a dynamic reordering heuristics according to the following table:

0 dynamic reordering is disabled
1 CUDD_REORDER_RANDOM
2 CUDD_REORDER_RANDOM_PIVOT
3 CUDD_REORDER_SIFT
4 CUDD_REORDER_SIFT_CONVERGE
5 CUDD_REORDER_SYMM_SIFT
6 CUDD_REORDER_SYMM_SIFT_CONV
7 CUDD_REORDER_WINDOW2
8 CUDD_REORDER_WINDOW3
9 CUDD_REORDER_WINDOW4
10 CUDD_REORDER_WINDOW2_CONV
11 CUDD_REORDER_WINDOW3_CONV
12 CUDD_REORDER_WINDOW4_CONV
13 CUDD_REORDER_GROUP_SIFT
14 CUDD_REORDER_GROUP_SIFT_CONV
15 CUDD_REORDER_ANNEALING
16 CUDD_REORDER_GENETIC
17 CUDD_REORDER_LINEAR_CONVERGE
18 CUDD_REORDER_LAZY_SIFT
19 CUDD_REORDER_EXACT

See the CUDD documentation [10] for the modus operandi of each heuristics. The initial ordering (which is maintained during the whole evaluation if dynamic reordering is disabled) is selected by the value of the S $\in \{0, \ldots, 7\}$ argument. Each value selects a different rule for placing universal variables at different decision levels of the BDD manager. Such rules operate by analyzing either the quantifier tree or the matrix. In particular:

0 (BDD_ORDERING_LEFT_TO_RIGHT), universal variables are ordered as they show up by reading the prefix in a left-to-right way (the first variable is placed at level 0, the second at level 1, and so on).
1 (BDD_ORDERING_RIGHT_TO_LEFT), universal variables are ordered as they show up by reading the prefix in a right-to-left way (the last variable at level 0, the last but one at level 1, and so on).
2 (BDD_ORDERING_INCREASING_U_DEPTH), universal variables are ordered according to their universal depth in the quantifier tree (the higher the depth, the higher the decision level). Variables with the same universal depth are left in the same relative order as in the prefix.
3 (BDD_ORDERING_DECREASING_U_DEPTH), like the previous rule, but the higher the depth, the lower the decision level.
4 (BDD_ORDERING_INCREASING_SPLITBALANCE), universal variables are ordered according to how well balanced is the number of clauses containing the positive literal v (let us call $N^+(v)$ such number) against the number of clauses containing the negative literal $\neg v$ ($N^-(v)$). The higher the value of $0 \leq \frac{\min(N^+(v),N^-(v))}{\max(N^+(v),N^-(v))} \leq 1$ the lower the level of v.

5 (`BDD_ORDERING_DECREASING_SPLITBALANCE`), like the previous rule, but here the more balanced a variable is, the higher the level it takes.

6 (`BDD_ORDERING_INCREASING_WIGHTED1_SPLITBALANCE`), the higher the value of $(N^+(v) \times N^-(v))$ the lower the level of v.

7 (`BDD_ORDERING_DECREASING_WIGHTED1_SPLITBALANCE`), the higher the value of $(N^+(v) \times N^-(v))$ the higher the level of v.

In most cases, reordering makes it possible to solve an otherwise unaffordable instance. The best reordering heuristics depends on the instance at hand. However, reordering may sometimes consume more time than it saves. By default, `CUDD_REORDER_GROUP_SIFT` is applied for dynamic reordering, and `BDD_ORDERING_LEFT_TO_RIGHT` for the initial ordering (equivalent to `-hbdd 0:13`).

`-hucp [0..10]` selects an ordering heuristics to be used during symbolic unit clause propagation (SUCP). Given a symbolic formula with more than one symbolic unit clause—on which SUCP is about to operate—this heuristics is responsible for choosing the next one to be propagated. The integer value immediately following the **`-hucp`** option selects a heuristics according to the following table:

0 (`SYMB_CNF_UCP_HEURISTICS_RND`), follow a random order

1 (`SYMB_CNF_UCP_HEURISTICS_RECENT`), focus on recently generated unit clauses

2 (`SYMB_CNF_UCP_HEURISTICS_RIGHT_TO_LEFT`), follow the right-to-left order of existential variables in the prefix

3 (`SYMB_CNF_UCP_HEURISTICS_LEFT_TO_RIGHT`), follow the left-to-right order of existential variables in the prefix

4 (`SYMB_CNF_UCP_HEURISTICS_MAX_U_DEPTH`), choose the unit clause with the maximal universal depth

5 (`SYMB_CNF_UCP_HEURISTICS_MIN_U_DEPTH`), choose the unit clause with the minimal universal depth

6 (`SYMB_CNF_UCP_HEURISTICS_MAX_GROUND_SIZE`), choose the symbolic unit clause that represents the maximal number of ground unit clauses

7 (`SYMB_CNF_UCP_HEURISTICS_MIN_GROUND_SIZE`), choose the symbolic unit clause that represents the minimal number of ground unit clauses

8 (`SYMB_CNF_UCP_HEURISTICS_MAX_BDD_SIZE`), choose the symbolic unit clause with the largest representation for the BDD component.

9 (`SYMB_CNF_UCP_HEURISTICS_MAX_BDD_SIZE`), choose the symbolic unit clause with the smallest representation for the BDD component.

10 (`SYMB_CNF_UCP_HEURISTICS_MIN_RESOLVED_CLAUSE_LENGTH`), choose the unit clause for which is minimal the maximal existential length of the symbolic clauses which will be resolved.

Notice that in the purely existential case (SAT) the issue of selecting a suited ordering heuristics for unit clauses is rarely if ever raised, as (1) unit clause propagation is a confluent process, (2) watched-literal based data structures support very fast inferences, and (3) the order of propagation makes no significant difference to performance. Conversely, in the symbolic framework watched literals are not used because the bottleneck is usually in BDD-based computation. Moreover, intermediate results may considerably differ in size, depending on the ordering chosen for unit clause elimination (even if the process stays propositionally confluent). This makes the selection of a good heuristics a sensible choice on certain families (those on which most solving time is spent in SUCP).

By default, `SYMB_CNF_UCP_HEURISTICS_LEFT_TO_RIGHT` is applied (equivalent to `-hucp 3`).

Note: Previous versions of the solver (up to v. 0.6.1) did not expose this switch. They used to have `SYMB_CNF_UCP_HEURISTICS_MINIMIZE_RESOLVING_CLAUSE_LIST_LENGTH` as an internal default value for this option (`-hucp 10`).

3.4 Options to tune the R style (complete symbolic reasoning)

`-hdres HEUR` selects a heuristics to be used during symbolic directional resolution (SRES) to choose the ordering of variable elimination. **`HEUR`** is one of the following:

rnd to eliminate variables in a random order

resolvents to eliminate the variable that generates the minimal number of resolvent clauses (greedy evaluation)

udepth to eliminate the variable that causes the maximal reduction in the average universal depth of the set of resolvent clauses

By default, the number of resolvent clauses is minimized (`-hdres resolvents`).

3.5 Options to tune the B style (branching reasoning)

`-learning L` adjusts the maximal size for the set of learned symbolic clauses. The symbolic clause learning mechanism used in the B style learns new clauses on closed branches. The maximal number of learned clauses managed at once is equal to $L \cdot |F|$, where L is an integer (<=10) and $|F|$ is the number of clauses in the original QBF formula. For example, a `-learning 2` directive allows the reasoning engine to retain up to 200 learned clauses while solving a 100-clause formula. When the limit on learned clauses is reached and there is a new learned clause to be added, the least *relevant* in the current set is discarded. The inverse of the number of literals in each clause is assumed as a measure of relevance.

A `-learning 0` directive completely disables symbolic learning: It is equivalent to the former `-nolearning` switch (up to v. 0.6.1). Conflict-directed backjumping is unaffected (it cannot be disabled at present). By default, L is equal to 1.

3.6 Options to tune the G style (SAT-based reasoning)

`-solver NAME` selects the SAT solver to be used in the G style. In the present release NAME can be one of the following:

`auto` for solver auto-selection (see below).

`minisat` to select the solver `minisat`, v.1.14.

`siege` to select the solver `siege`, v.4.

`zchaff` to select the solver `zChaff`, v. 2004.5.13.

Besides the solving strengths and weaknesses of each solver, notice the following differences among the three options:

minisat (v. 1.14) is distributed in source format. It is statically linked[3] against sKizzo, so you need nothing particular to execute it. The extraction of an unsatisfiable core is not directly supported by minisat, so when the G style is reached from the B style a "blind" backtrack is performed[4]. This problem does not arise when the G style is reached from either Q or S / R.

siege (v. 4) is distributed in executable format only [9]. It is used as an external, balck-box solver, and is executed in a separate process. `Siege` is not distributed with sKizzo. So, you have to download that solver on your own (from [9]), then make sure that the "`siege_v4`" executable is reachable through the PATH environment variable. The `siege` solver does not extract unsatisfiable cores, and this might impact on the backtracking effectiveness of the branching engine when the G style is reached from the B style (conversely, when G is reached from previous styles, only one SAT-equivalent instance is generated, and the core extraction technique plays no role). No Mac/OsX version of the `siege` executable is distributed, so the `-solver siege` option cannot be used on such platform.

zChaff (v. 2004.5.13) is distributed in source format. It is statically linked[5] against sKizzo, so you need nothing particular to execute it. When the G style is reached from the B style, an unsatisfiable core is extracted (through the statically linked "`dverify`" tool that comes with `zChaff`) at each contradictory branch. The analysis of this core is used to help the backtracking symbolic engine to perform smarter choices.

By default, the `auto` setting is used. This amounts to use `minisat` in all the occasions but when the B style is reached and it needs splitting over an existential variable before G is visited. In the latter case, the auto-selection mechanism switches to `zChaff` for the rest of the process. The reason for this is that the cooperative conflict analysis (made possible by the unsatisfiable core extraction) is only effective after at least one existential split has been performed: in all the other cases the very first UNSAT answer of the SAT solver decides the QBF instance.

3.7 Options to control memory usage

`-mem STYLE` selects a memory exploitation style. STYLE is one of the following:

`nocheck` to disable memory consumption check.

`light` to enable a memory consumption check style that uses all the free memory as reported by `vmstat`.

`heavy` to enable a memory consumption check style that uses all the free memory and minimize the inactive memory as reported by `vmstat`.

By default, the `light` style is adopted. Notice that:

- When memory consumption check is disabled (`nocheck` option) sKizzo allocates as much memory as required by the current solving style and inference rule, regardless of the amount of RAM memory *physically* available and currently free. The VM module of the operating system is made

[3] Minisat's license allows users to do so, provided they report a suited copyright notice. We honor the request in Appendix A.

[4] The next release of sKizzo will address this issue.

[5] Permission granted by zChaff's author, see Appendix A.

responsible for swapping to/from disk when physical memory is exhausted. As solvers keep on referring to the whole allocated memory, the `nocheck` option may result in trashing conditions, with very low CPU usage and high I/O throughput. In addition, those transitions between inference styles that are triggered by detecting low-memory conditions (for example, the transition from R to B) are prevented.

- The `light` option makes sKizzo worry about memory consumption on behalf of the OS. As soon as the OS reports that all the physical memory has been allocated (no *free* pages left), sKizzo stops requiring further memory allocation. This entails that (1) trashing conditions are avoided, and (2) if the current rule/style cannot execute anymore as a consequence of the enforced memory restrictions, a transition is triggered (the R→B transition is a prototypical example). In essence, this option attempts to use the available memory at its best, shifting from memory-intensive computation to CPU-intensive computation when necessary. However, it may fail to work because of the way free memory is managed by the OS. In particular, two conditions under which this option may (heavily) underestimate the amount of memory that can be safely allocated are: (1) sKizzo is running on a machine with "few" memory (0.5GB or less), (2) memory-eager processes other than sKizzo are running (even if inactive).
- The `heavy` option should be used when one or both the conditions above occur. The `heavy` option cause sKizzo to stop allocating memory only when there are no more *inactive* pages associated to other processes. This leads to a more intensive memory exploitation, but under many circumstances it may require a great deal of I/O work.

3.8 Options to make sKizzo stop on certain occurrences

`-dontdescend` prevents recursive processing of subdirectories in batch mode. See Section 4.1.

`-giveup` instructs the solver to give up after the first instance in a family is found unsolvable (for batch mode only). See Section 4.3.

`-timeout T` makes the solver work for no longer than T seconds on each instance. This option is redundant in the single-instance processing mode, as it is equivalent the last command-line parameter of Section 2 (the latter being the standard way to give a timeout to most solvers). However, this switch is useful in the batch mode to select different timeouts for different families using a configuration file (see Section 4.3).

`-treeonly` prevents the solver from solving the instance: sKizzo just reconstructs the quantifier tree and prints a brief report to `stdout`, then exits (thus acting mostly like the `qTree` tool [2]). By default, the whole solving process is enabled.

3.9 Options to control verbosity, dumping, logging

`-dimacsout` disables the standard output behaviour of the solver, and makes it comply with the DIMACS 1.1 output specification (notice: *partial certificates* are not dumped at present). By default, the solver uses its own output format. `-dimacsout` disables the `-v` switch.

`-dump DATA` dumps to file the information "DATA" (obtained at the beginning, during or at the end of the solving process), where DATA is a string selecting what to dump according to the following table:

tree dumps a DOT representation of the initial quantifier tree. Leaves are house-shaped. They do not list all the clauses, but just report how many clauses are there.

trees works like "tree", but beyond the initial quantifier tree also dumps (in separate files) the sequence of trees obtained by ground QBF pre-processing (if enabled).

TREE works like "tree", but attaches a detailed clause list at the leaves (to be used for small formulas only).

TREES works like "trees", but attaches a detailed clause list at the leaves (to be used for small formulas only).

dimacs dumps to file in DIMACS format each SAT instance generated and solved (if any) in the G status.

report dumps reports during a batch processing. See Section 4.6.

flatreport dumps reports during a batch processing. See Section 4.6.

The names of the files used to dump information are chosen as follows.

- The files used to dump trees and DIMACS instances have the same name and position in the file system as the ".qdimacs" file they refer to, with just an additional ".qtree.dot" extension for quantifier trees, and ".dimacs" for SAT instances. If multiple ".qtree.dot" and/or ".dimacs" files are generated from the same instance, a progressive sequence number is appended between the original file name and the .qtree.dot or .dimacs extension.

By default, no dumping to file is performed.

-log [TYPE] instructs the solver to produce an inference log recording the whole solution process. The log is to be used by the ozziKs companion application. TYPE is an optional argument. It can be:

text to record the log in a purely textual format.

bin to record the log in a mixed textual/binary format.

The former option produces a human-readable format, and as such is useful for debug purposes, the latter create logs that require much less space. ozziKs is able to distinguish the two formats by inspecting the log. By default, no inference log is written. With no argument, -log generates an inference log in textual format.

-v [0..9] controls the output verbosity. The value "0" makes the solver absolutely silent, so that the only output is the return value. The value "1" makes the solver give just a TRUE/FALSE feedback for each solved instance. This is the default. Higher values cause sKizzo to expose a part of its internal status as the evaluation goes on.

3.10 Miscellaneous options

-copyright prints copyright and license notes (see Appendix A).

-help OPT gives you help on the command-line option OPT (a short version of the information you find in this manual).

-version prints sKizzo's version.

4 Batch mode

When you launch sKizzo giving a subdirectory as an entry point (rather than a single QDIMACS file), the solver operates in *batch mode*. It traverses the whole directory subtree rooted at the given entry point, processing each QDIMACS file encountered during the visit.

There are a few behaviors specific to the batch mode, and some differences w.r.t. the single-instance mode. They are discussed in the following subsections.

4.1 Order of processing

The order in which multiple instances are processed complies with following rules.

- Directories are traversed recursively, in a depth-first way.
- The solution process takes place in post-order (before visiting nested directories).
- Instances in each directory are first sorted according to the number of variables they contain (through a pre-parsing of all the DIMACS files, and regardless of their file names). Then, they are solved in a smallest-to-greatest order.

The recursive descent into deeper directories can be prevented by using the **-dontdescent** option. Only the instances in the root of the directory tree given as entry point are processed.

4.2 Return values

In batch mode, sKizzo returns the number of instances successfully solved (i.e. those solving which no timeout or error occurred). Some error codes are also used. The full list of possibilities is the following.

-1 on unrecoverable internal error

-2 on I/O error or file not found.

-3 on commandline parse error

n (≥ 0): the number of instances successfully solved

4.3 Timeouts

When sKizzo is required to traverse a directory subtree and a timeout is specified, it works for no longer than the specified amount of time *on each instance*, then moves on to the next instance.

Additionally, the **-giveup** option can be used to skip one entire family when a timeout occurs. More precisely, **-giveup** instructs the solver to give up after the first instance in a family is found unsolvable:

- Each directory is supposed to contain a set of increasingly complex instances from a parametrically scalable family.
- As usual, such instances are first sorted according to the number of variables they contain, then solved in order.
- The very first failure (timeout, memout, crash, etc.) occurring during the solution process causes the rest of the sequence in the current directory to be skipped.
- Solving is then re-started from the next directory.

This option—used in conjunction with a timeout—is mainly useful in limiting the time taken by benchmarking, under the reasonable assumption that—fixed the structure of the family—the larger the instance the longer the solving time.

By default, instance solving is attempted regardeless of previous outcomes over instances in the same directory (family).

4.4 Interaction among solution processes

Interactions among different solution processes in the same batch execution are made as small as possible, with the aim to avoid unwanted interferences that might negatively affect performance. Launching the solver in batch mode should not give significantly different results w.r.t. executing it on each instance separately.

The effects of possible memory leaking, improper object deallocation, bad re-initializations etc. are limited by executing one small, carefully designed core process as a controller. It forks in one (almost completely) fresh solver at each encountered instance.

4.5 Commandline parameters

If any commandline parameter is specified, it affects the solution process of every single instance.

You may override this default behavior by placing a *configuration file* named "`sKizzo.conf`" in a directory. Things work as follows.

- The options specified in the `sKizzo.conf` file contained in the directory D are applied (only) to the instances (immediately) within D.
- There is no inheritance among configuration files. Each one is applied on top of the default configuration. Conversely, the commandline options specified by the user while launching the batch procedure are applied to all (and only) the instances contained in directories without a configuration file.
- Each configuration file is made up of (one or more) lines. Each line specifies a setting to be tried. Each setting is used against all the instances in the current directory.
- Every line in a configuration file has to contain a list of valid, space-separated options, just as if they were commandline options. Lines exhibiting a syntax that does not comply with the one given in Section 3 are skipped.

For example, suppose that the directory D contains three instances and a configuration file:

```
adder-2-sat.qdimacs
adder-4-sat.qdimacs
adder-8-sat.qdimacs
sKizzo.conf
```

...and that the configuration files is made up of two lines:

```
-dontuse SPLEi SPLE -hbdd 9
-dontuse SPLEi SPLE SER -hbdd 1:0
```

sKizzo solves the three instances working as if the `-dontuse SPLEi SPLE -hbdd 9` options were given at commandline, then solves again the three instances, this time working as if the `-dontuse SPLEi SPLE SER -hbdd 1:0` options were given.

4.6 Reporting

The **report** and **flatreport** arguments of the **-dump** switch enable the production of *report file(s)* for the batch mode. The difference between the two arguments is as follows.

- The **-dump report** switch activates the dumping of *one textual report file for each (sub) directory traversed*. Each report file contains one line for each solved instance in the directory it refers to. Each line of a report file contains the following tab-separated information on the solved instance:

  1. name
  2. number of existential variables
  3. number of universal variables
  4. prefix shape
  5. number of alternations
  6. time taken to solve (or: TIMEOUT)
  7. memory required to solve (or: MEMOUT)
  8. outcome (TRUE/FALSE/CRASH)

  The files containing reports are placed in the directory containing the instances they refer to, and are named after the directory itself, with an additional "`.txt`" extension. If a directory contains a valid `sKizzo.conf` file, then the report file for that directory will contain one additional heading line (for each line in the configuration file) summarizing the options under which the subsequent instances have been solved.

- The **-dump flatreport** switch is similar to the **report** one, except that it produces *only one textual report file* containing the report lines of every solved instance. Such file is always named "`sKizzo_report.txt`" and is placed in the root of the directory subtree which the user requested to traverse. To distinguish among instances belonging to different families/directories, the position in the directory subtree is written in the report before solving the set of instances in that position. Some additional heading and tailing information is added. A sample "flatreport" file is the following:

```
# sKizzo_0.8 batch processing report file
# started on Wed Oct 12 19:26:10 2005
# options:  -dump flatreport -timeout 10 ./sample_report/

# Processing: "./sample_report/FALSE/Logn"
lognBWLARGEA1   1099    62820   E[270]AE[828]   2   3.07    23.5    FALSE
lognBWLARGEB1   1871    178750  E[396]AE[1474]  2   10      0.0     TIMEOUT

# Processing: "./sample_report/FALSE/s27"
s27_d3_u    165 254 E[42]A[23]E[52]     2   0.65    4.9 FALSE
s27_d4_u    271 366 E[55]A[36]E[78]     2   1.20    5.0 FALSE
s27_d5_u    403 478 E[68]A[49]E[104]    2   2.01    4.8 FALSE

# Processing: "./sample_report/FALSE/toilet"
TOILET6.1.iv.11 294 1046    E[77]A[3]E[214]     2   0.68    5.5 FALSE
TOILET7.1.iv.13 399 1491    E[104]A[3]E[292]    2   1.27    5.7 FALSE

# Processing: "./sample_report/TRUE/counter"
cnt06   266 691 E[39]AE[39]AE[39]AE[39]AE[39]AE[39]AE[26]   12  0.40    5.8 TRUE
cnt06e  286 751 E[42]AE[42]AE[42]AE[42]AE[42]AE[42]AE[28]   12  4.23    6.6 TRUE
cnt06r  286 745 E[42]AE[42]AE[42]AE[42]AE[42]AE[42]AE[28]   12  2.09    6.0 TRUE
cnt06re 306 805 E[45]AE[45]AE[45]AE[45]AE[45]AE[45]AE[30]   12  5.89    6.2 TRUE

# Processing: "./sample_report/TRUE/mutex"
mutex-2-s   559     127     A[8]E[96]       1   0.01    0.5 TRUE
mutex-16-s  3961    1779    A[64]E[1314]    1   0.10    1.0 TRUE
mutex-64-s  15625   7443    A[256]E[5490]   1   0.40    2.8 TRUE
```

```
# Processing: "./sample_report/TRUE/s499"
# Specific options: -dontuse SPLE SPLEi SHBR SSUB -hbdd 9
s499_d2_s   1213    2665    E[328]A[131]E[491]  2   0.43    6.1 TRUE
s499_d3_s   2545    4816    E[481]A[284]E[982]  2   1.40    7.5 TRUE
s499_d4_s   4368    6967    E[634]A[437]E[1473] 2   4.25    8.9 TRUE

# batch processing finished
# time now: Wed Oct 12 19:27:02 2005
# 16/17 instances successfully solved.
```

5 Notes on run-time behavior, memory consumption, solving personalities

5.1 Tuning your solving personality

sKizzo is a hybrid solver incorporating a number of evaluation algorithms (see Section 6). Furthermore, some inference styles (e.g. S, R) exploit many individually controllable inference rules. Heuristics also play a great role in some cases. As a consequence, you have a lot of freedom in customizing the way sKizzo attacks your QBF instances (*solving personality*). This is a good news, as long as either the solver (*automatically*), or you (*manually*) are able to find out a suited configuration.

Commandline options give you means to fine control the solving personality. You don't need to use such options as long as the default behavior of the solver is ok for your instances. To a certain extent, such default behavior is flexible and automatically adapts to different instance structures. For example, the inference state machine briefly introduced in Section 6 allows the solver to *hybridize* inference styles in a way that depends directly (e.g. number of "cheaply" removable variables, etc.) and indirectly (e.g. memory requirement of resolution-based reasoning, etc.) on the particular instance (see also Section 5.2). Furthermore, a learning approach is employed to try to guess the optimal switch point between inference styles (e.g. from B to G), the optimal resource allocation (e.g. relative execution time for each symbolic rule), and more.

When all this fails, you should resort to manual adjustment[6]: In our experience, *a failure (timeout) with the default configuration doesn't imply that the solver is unable to solve the instance at hand*.

The degrees of freedom of the auto-adjusting capability of the solver are limited to the switch between inference styles and the resource allocation among rules. The manual counterpart to these auto-adjusting features are the `-style`, `-use`, and `-dontuse` options. However, some modifications to the solving behavior can be obtained only manually. For example, those controlled by the `-learning`, `-hbdd`, `-hdres`, `-hucp`, and `-mem` options. A "`-style QSBG -dontuse SRES`" configuration causes the solver to behave quite differently from a "`-style QR`" customization, which is in turn different from "`-style SR -dontuse SHBR SER`".

Finally, notice that even if the automatic settings work, you might be able to find out a configuration of the parameters that yields better results on your instances (due at least to the removed overhead of some trial-and-error behaviors of the automatic engine).

5.2 Memory consumption

To perform at its best in different execution environments, sKizzo tries to adapt to the available computational resources (CPU speed and free memory available). It keeps on measuring these quantities, then takes some (important) decisions on the basis of what has been measured. As a side effect, it is to be taken into account that:

- No two runs of the solver are exactly the same (nor even on the very same instance/machine). The algorithm is deterministic, but it takes as inputs unpredictable and fluctuating values from the computational environment, thus exhibiting a *non-deterministic* behavior.
- Other processes in execution might interfere with sKizzo in unpredictable manners. Their CPU and memory occupation affects the behavior of the solver. For example, an external process P that allocates much memory may cause symbolic directional resolution to run out of memory (and, consequently, it may cause sKizzo to switch to branching reasoning), whereas symbolic directional resolution would have decided the instance if P were not running. Similar effects might (noticeably) affect performance.

[6] Future versions of this documentation will contain guidelines on how to attempt personality tuning.

- Counterintuitive effects sometimes arise due to suboptimal resource management: An instance in a parametrically scalable family of instances might be solved in less time than a smaller instance of the same family.

5.3 Shell commands

sKizzo relies on the existence of a `/bin/sh` shell to launch a few commands (`ps`, `vmstat`, and so on) used to perform self-monitoring. Failing to launch such commands might prevent sKizzo from working properly.

5.4 Multiprocessing

sKizzo forks into two processes at the very beginning, one performing the real job, the other measuring mem/time usage, catching crashes and recursively traversing directories and selecting instances to solve if required. This is the reason why possible crashes/segfaults are dealt with as "regular" errors, and also the reason why killing one sKizzo's process may not stop the whole thing.

6 sKizzo's internals

In this section we give a very brief introduction to the data structures and algorithms employed by sKizzo. We aim at providing the user with the information necessary to understand some peculiar commanline options of the solver, such as the `-style` and `-use` options.

For a more in-depth presentation of the matter, we refer the reader to:

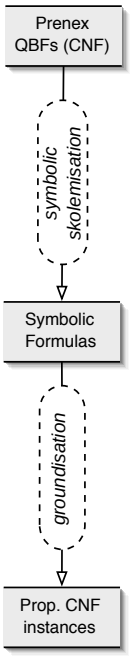
- [1] for a description of an early version of the solver, with algorithms and data structures.
- [2] for experimental results and up-to-date software releases.
- [3] for a presentation of the symbolic skolemization technique.
- [4] for the approach to QBF satisfiability certification.
- [5] for quantifier tree reconstruction.
- [6] for a discussion of how the solver hybridizes different solving styles.

6.1 Overview

At the hearth of sKizzo stays a *symbolic* representation for clauses and formulas, based on *Binary Decision Diagrams* (BDDs)[7]. As opposed to other BDD-based approaches to propositional logic, sKizzo's one employs a two-level data structure [1] purposely designed to take advantage of the distinguishing features of quantified propositional formulas. A symbolic formula to be managed and solved is obtained from the input QBF through the symbolic skolemization technique [3]. Such symbolic representation coexists with other kinds of representations and data structures within sKizzo, such as quantifier trees [5]. They are briefly discussed in Section 6.2.

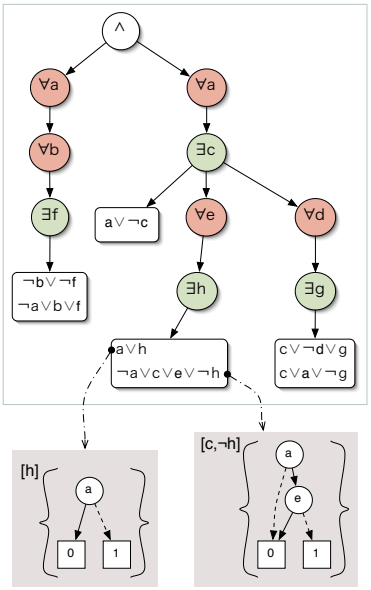
Besides allowing for a purposely designed kind of *symbolic* reasoning, our symbolic representation makes it possible to unify within a coherent framework many other approaches to QBF-satisfiability proposed so far. Namely: DPLL-like branching reasoning, q-resolution based algorithms, and compilation-to-SAT techniques. The role of these evaluation strategies within sKizzo is briefly addressed in Section 6.3.

6.2 Problem Representation



Three representation spaces for QBFs coexist within sKizzo. They are interconnected by two satisfiability-preserving trasformations (applied one-way), as reported in the picture aside. The first transformation leverages *outer skolemization* to map any (prenex CNF) instance $F \in QBFs$ onto a *symbolic* formula $\mathcal{F} = SymbSk(F)$, which is said to be *symbolic* as it couples list-based and BDD-based data structures to compactly represent a (possibly) exponentially less succinct propositional formula. The sentence $\mathcal{F}$ encodes the definability of a set of Skolem functions that capture a model (if any) of the original instance, according to the *symbolic skolemization* technique presented in [3]. A formal semantics is associated to symbolic formulas in such a way that $F \stackrel{sat}{\equiv} SymbSk(F)$ for every F. The other transformation—called *groundisation*—translates a symbolic formula $\mathcal{F}$ into a purely existential CNF propositional instance $Prop(\mathcal{F})$ (a SAT problem) such that $F \stackrel{sat}{\equiv} SymbSk(F) \stackrel{sat}{\equiv} Prop(SymbSk(F))$. The role of these representations is as follows: Plain QBFs are handled in a pre-processing phase. Then, sKizzo moves to the symbolic representation and performs most of its work thereon. Zero or more CNF instances are generated/solved during the whole process.

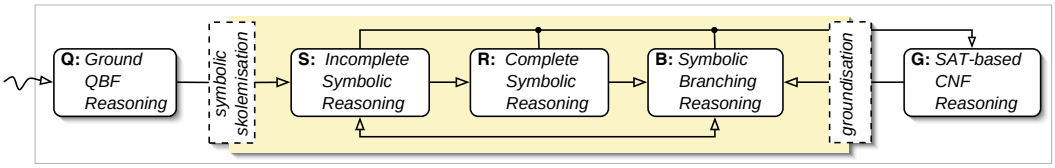
[7] We employ the CUDD package [10], version 2.4.0, and the DDDMP package [8], version 2.0.3.



Symbolic skolemization (and most of the processes introduced below) relies on the existence of a *quantifier tree* stating which existential variables are in the scope of which universal variables. Such tree-shaped structures are extracted out of the flat prenex input according to [5]. They replace linear prefixes so to more closely reflect the intrinsic dependencies in the matrix. A sample quantifier tree for the QBF $\forall a \forall b \exists c \forall d \forall e \exists f \exists g \exists h.(a \vee \neg c) \wedge (a \vee h) \wedge (c \vee \neg d \vee g) \wedge (\neg a \vee b \vee f) \wedge (\neg a \vee c \vee e \vee \neg h) \wedge (\neg b \vee \neg f) \wedge (a \vee c \vee \neg g)$ is depicted aside. The symbolic representation is designed to allow for efficient forms of *symbolic reasoning*, where universal reasoning is taken apart form existential reasoning (ROBDDs conveniently deal with the former, list-based representations with the latter). A symbolic formula is made up by symbolic clauses. During symbolic skolemization, one symbolic clause is extracted out of each QBF clause. The two major components of a symbolic clause $\Gamma_{\mathcal{I}}$ are a list Γ of existential literals and an index-set $\mathcal{I}$ represented via a ROBDD whose support set is the set of universal variables dominating the existential node at which the clause is attached in the quantifier tree. For example, the symbolic clauses $[h]_{\{00,01\}}$ and $[c, \neg h]_{\{10\}}$ are extracted out of $a \vee h$ and $\neg a \vee c \vee e \vee \neg h$ respectively (see the picture). Each symbolic clause $\Gamma_{\mathcal{I}}$ compactly represents a set $Prop(\Gamma_{\mathcal{I}})$ (with cardinality $|\mathcal{I}|$) of ground propositional clauses, in such a way that F is sat iff $Prop(\mathcal{F})$ is sat. For example, $Prop([c, g]_{\{01,10\}}) = \{c_0 \vee g_{01}, c_1 \vee g_{10}\}$. For details, see [1, 3].

6.3 Inference Strategy

The *inference strategy* followed by sKizzo changes as the solution process goes on. Its evolution is described by a finite state machine whose inference states $S^{inf} = \{\mathsf{G}, \mathsf{S}, \mathsf{R}, \mathsf{B}, \mathsf{G}\}$ are traversed.



Each state in S^{inf} is associated to the application of an *inference style*. Each transition $x \rightarrow y$ in the picture, $x, y \in S^{inf}$, is labeled by a condition that triggers the shift from the style x to y (possibly requiring a satisfiability-preserving transformation). The conditions under which transitions between one state and another are triggered is documented in [6]. The `-style` option can be used to forbid the visit of certain states, thus customizing the *personality* of the solver.

Q: Ground QBF Reasoning. In the Q-state sKizzo works in the original QBF space. The quantified form of three simple (incomplete) inference rules—*unit clause propagation* (QUCP), *pure literal elimination* (QPLE)), and *forall-reduction*—is applied until fixpoint. Then, a quantifier tree for the current formula is reconstructed. Finally, *q-resolution* (QRES) is applied to eliminate the "*cheap*" existentially quantified variables in the deepest existential scope of each branch. The cycle is repeated until no cheap existential variable exists.

S: Incomplete Symbolic Reasoning. The instance is attacked by means of a set of (incomplete) *symbolic inference rules*, designed to perform efficient symbolic deductions.

SUCP (Symbolic Unit Clause Propagation). Unit clauses are symbolically computed and assigned all at once.

SPLE (Symbolic Pure Literal Elimination). A symbolic representation for the set of pure literals is computed, and the formula is accordingly simplified.

SSUB (Symbolic SUBsumption). This rules removes all the symbolic clauses that are subsumed by other clauses (*forward subsumption*). This rule complements the *backward subsumption* mechanism which is applied on-the-fly at each clause insertion.

SHBR (Symbolic Hyper Binary Resolution). This rules enumerates all the resolution chains of binary symbolic clauses in the formula, looking for contradictions, hence for implied symbolic literals.

SER (Symbolic Equivalency Reasoning). This rules look for non-empty strongly connected components in the *symbolic binary implication graph*[1] of the formula. Each such component determines a symbolic equivalence which is applied to simplify the formula.

The `-use` and `-dontuse` options can be used to disable some of these rules. Notice that the above set of rules is not refutationally complete. One additional, refutationally complete rule is applied in this state (in a limited form), as explained in the R state.

R: Complete Symbolic Reasoning. This state is similar to S, with one major exception: (the full form of) a refutationally complete rule is inserted in the pool of symbolic rules exercised at each inference round. Namely:

SDR (Symbolic Directional Resolution). This rules eliminates one symbolic variable per step by substituting the set of resolving clauses with the set of their symbolically computed resolvents.

The relation between S and R is a bit more complicated than it seems: **SDR** is used also in S, but in a *limited* form. In fact, the execution of **SDR** within S is performed in a *controlled environment* that involves a rollback whenever the rule is unable to shrink the formula without exceeding certain time/memory resources. Such limitation is removed in the R state.

B: Branching Reasoning. In this status, a recursive search-based branching decision procedures extending the DPLL approach to the quantified case is applied. Both universal and existential splits are performed symbolically. The partial order induced by the internal structure of the quantifier tree is substituted for the left-to-right order of variables in the prefix. Either symbolic reasoning or ground reasoning (see the G status) are leveraged as look-ahead tools to decide base cases of the recursion. A conflict-analysis machinery is employed in the event of inconsistent partial assignment to perform a conflict-directed backjumping. The B, S, and G states share a common conflict-analysis engine. A symbolic learning mechanism extracts symbolic clauses out of contradictions (it can be disabled via the `-learning 0` option). Branching heuristics are enrolled (MOMS, VSDIS).

G: SAT-based Ground Reasoning. In the G-state we explicitly construct $Prop(SymbSk(F))$ and solve it via some state-of-the-art SAT solver.

References

[1] M. Benedetti. sKizzo: a QBF Decision Procedure based on Propositional Skolemization and Symbolic Reasoning, Tech.Rep. 04-11-03, ITC-irst, 2004.

[2] M. Benedetti. `sKizzo`'s web site, `sra.itc.it/people/benedetti/sKizzo`, 2004.

[3] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proc. of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR04)*, number 3452 in LNCS. Springer, 2005.

[4] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proc. of 9th International Joint Conference on Artificial Intelligence (IJCAI05)*, 2005.

[5] M. Benedetti. Quantifier Trees for QBFs. In *Proc. of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT05)*, 2005.

[6] M. Benedetti. sKizzo: a Suite to Evaluate and Certify QBFs. In *Proc. of 20th International Conference on Automated Deduction (CADE05)*, 2005.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *proceedings of the 38th Design Automation Conference*, 2001.

[8] S. Quer and G. Cabodi. `DDDMP`'s web page, `fmgroup.polito.it/quer/research/tool/tool.htm`.

[9] Lawrence Ryan. The `siege` satisfiability solver, web page, `www.cs.sfu.ca/ loryan/personal`.

[10] Fabio Somenzi. Colorado University Binary Decision Diagrams, `vlsi.colorado.edu/~fabio/CUDD`, 1995.

Appendix A: Copyright and License

Copyright

Definitions

- By "SOFTWARE" we mean the software "sKizzo" and the associated documentation files, which are offered under the terms of this License.

- By "AUTHOR" we mean Marco Benedetti, i.e. the individual who created the SOFTWARE, and who offers it under the terms of this License.

- By "USER" we mean an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the SOFTWARE, or who has received express permission from the AUTHOR to exercise rights under this License despite a previous violation.

- By "NONCOMMERCIAL USE" we mean use for research, evaluation, or development for the purpose of advancing knowledge, teaching, learning, or customizing the technology for personal use. NONCOMMERCIAL USE expressly excludes use or distribution for direct or indirect commercial (including strategic) gain or advantage.

License

By using the SOFTWARE the USER indicates that he or she has read, understood and will comply with the following:

- The AUTHOR hereby grants USER nonexclusive permission to use the SOFTWARE for NONCOMMERCIAL USE only.

- Permission to copy and redistribute the SOFTWARE is granted so long as no fee is charged, and so long as the the present unmodified copyright notice (including the disclaimer below) appear in all the copies made.

- For any other permission (including–but not limited to–the permission to use the SOFTWARE for commercial purposes, the permission to create/distribute derivative or modified works, etc.) please contact the AUTHOR at `mabene@gmail.com`.

Disclaimer

This SOFTWARE is provided "as is". The AUTHOR makes no representations or warranties, express or implied, including those of merchantability or fitness for any purpose. The AUTHOR shall not be liable under any circumstances for any direct, indirect, special, incidental, or consequential damages with respect to any claim by USER or any third party on account of or arising from the use, or inability to use, the SOFTWARE.

Copyrights and Licenses for Third Party Software Distributed with the SOFTWARE

The SOFTWARE contains compiled code written by third parties. Such pieces of software have additional or alternate copyrights, licenses, and/or restrictions. Namely, the SOFTWARE is statically linked against:

1. The CUDD package, version 2.4.0, by Fabio Somenzi (Department of Electrical and Computer Engineering, University of Colorado at Boulder). The CUDD package is copyright of the University of Colorado at Boulder. The autoritative source of information on the CUDD is:
   `http://vlsi.colorado.edu/~fabio/CUDD/`

2. The DDDMP-2.0 package, version 2.0.3, by Gianpiero Cabodi and Stefano Quer. The DDDMP package is Copyright (c) 2002 by Politecnico di Torino. The autoritative source of information on DDDMP is: `http://staff.polito.it/stefano.quer/research/tool/tool.htm`

3. zChaff, version 2004.5.13, a search-based SAT solver by the SAT Research Group at the Princeton University. zChaff is Copyright 2000-2004, Princeton University, with all rights reserved. zChaff is for non-commercial purposes only. No commercial use of zChaff is allowed withouth written permission from Princeton University. Please contact Sharad Malik (malik@ee.princeton.edu) for details. The autoritative source of information on zChaff is:
   `http://www.princeton.edu/~chaff/software.html`

4. minisat, version 1.14, a SAT solver by Niklas Een and Niklas Sorensson. In compliance with minisat's license, we include the following notes taken from the minisat official distibution (in the quoted text below the word "Software" refers to minisat):

   > MiniSat – Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson
   >
   > Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
   >
   > The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
   >
   > THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

   The autoritative source for information on minisat is:
   `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`