Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
BP 6759 – 45067, University of Orléans, France

User Manual

(Manual v0.2, documenting ozziKs v0.3-beta, rev. 393)

Marco Benedetti
mabene@gmail.com

November 10, 2006

Contents

1 Introduction

ozziKs is the implementation of an algorithm that:

1. builds a *certificate of satisfiability* $C(F)$—a.k.a. *strategy* or *policy* or *quantified model*—for a given TRUE *Quantified Boolean Formula* F for which a suited inference log is available (at present, only the QBF solver sKizzo [4, 2] produces a suited log);
2. verifies $C(F)$ against F, thus certifying in a solver-independent way the validity of F;
3. evaluates user-provided expressions of various kind over $C(F)$;
4. writes to file in different formats $C(F)$ and/or the result of the evaluation of the above mentioned expressions.

For more information on the creation, representation, and usage of certificates, see Appendix A.

As we mentioned, ozziKs is not designed as a stand-alone tool. It works in tandem with the QBF solver sKizzo, in a two-step process that will be described in subsequent sections.
This manual is organized as follows:

- Section 2 gives details on the input/output behavior of the software, on its commandline syntax, and on exit values.
- Section 3 lists the commandline options useful to customize the behavior of the software.
- Section 4 introduces syntax and semantics of *expressions over certificates*.
- Section 5 shows how to make the software process a set of instances instead of just one.
- Section 6 provides several examples of usage.
- Appendix A contains introductory notions about the representation, reconstruction, and verification of QBF certificates.
- Appendix B provides details on the content and format of input and output files.

1.1 Contact and feedback

At present, ozziKs (version 0.3, revision 393) is in beta testing (check [1] for updates). Feedback from users is welcome on both the certificate reconstructor and its documentation. Please send an e-mail to

mabene@gmail.com

mentioning "ozziKs" in the subject in case you:

- are able to make the reconstructor crash under specific, reproducible circumstances;
- have problems building certificates for some "reasonbably-sized" instance you were able to solve with sKizzo;
- find errors in the documentation;
- observe any unexpected behavior of the software;
- 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 ozziKs or its documentation.

2 The ozziKs software

The ozziKs software is currently distributed at [1] for three platforms:

- *linux - i386*, as a stand-alone statically linked executable;
- *OS X*, as a universal binary that runs natively on both Intel and PowerPC machines;
- *win32*, as a cygwin[1] application (requires a cygwin installation or at least the `cygwin1.dll` library).

These three versions are functionally equivalent, and no platform-related disctinction is made through this manual.

2.1 How to execute it

To launch ozziKs, use the following syntax.

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

where we have to provide:

1. An optional list of space-separated options `OPTS`. Such options are used to modify the reconstruction process and/or the input/output behavior of the program. They are discussed in Section 3.
2. A mandatory `FILE` path or, alternatively, a path `DIR` to a directory, where:
   (a) `FILE`, is the full name (possibly with a path) of an inference log file to be processed (extension `.sKizzo.log`), or, alternatively, the full name (possibly with a path) of a certificate (extension `.qbm`) to be verified.
   (b) `DIR` is the root of a directory subtree which is to be recursively visited. When we specify a directory as an entry point, the certifier works in *batch mode*. See Section 5.
3. An optional `TIMEOUT` (in seconds). When a timeout is given, ozziKs 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 ozziKs or to the shell from which the certifier was launched—are:

10 on a successfully completed reconstruction yielding a *valid* certificate

20 on a successfully completed reconstruction yielding an *invalid* certificate

30 on timeout

-1 on unrecoverable internal error

-2 on commandline parse error

These return codes are only valid when a single log/certificate is given as argument. For the return codes in batch mode see Section 5.2. For codes in expression-evaluation mode see Section 4.1.

[1] Cygwin is a Linux-like environment for Windows which acts as a Linux API emulation layer providing substantial Linux API functionality from inside a win32 installation. For details see `www.cygwin.com`.

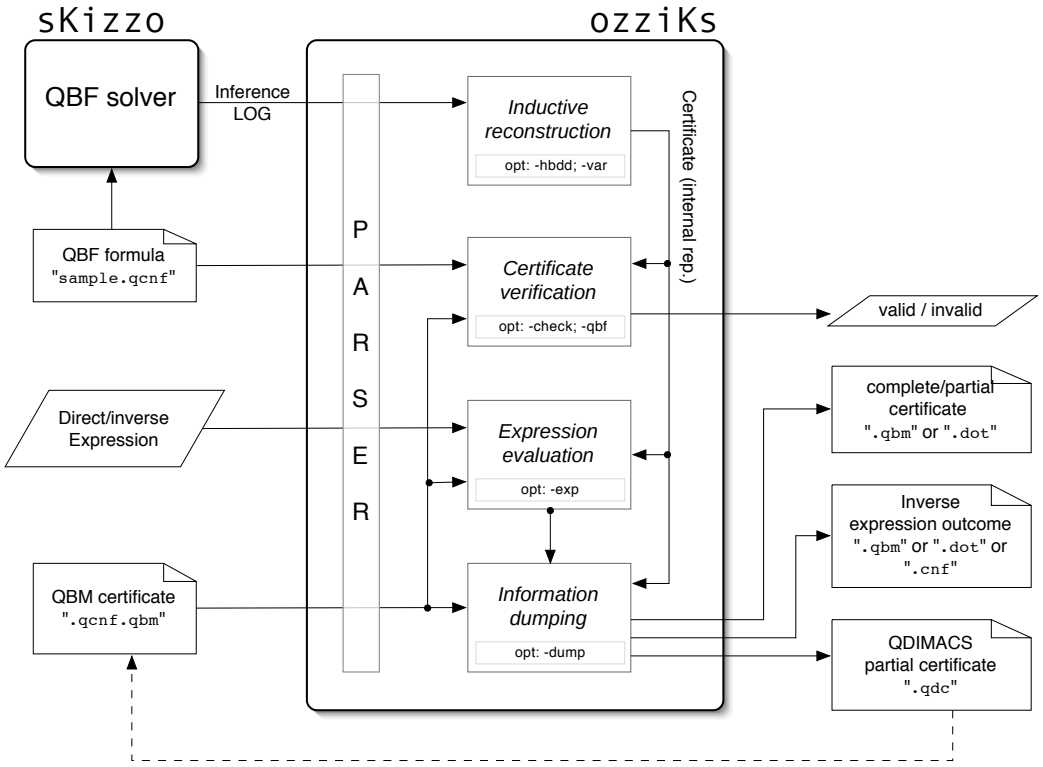


Figure 1: The internal structure and the input/output behaviour of ozziKs.

2.2 Input/output behaviour

The overall input/output behaviour of the solver is depicted in Figure 1. The four engines that constiute ozziKs have the following function and input/output behaviour:

Inductive reconstruction. Takes as input an inference log (produced by sKizzo during the evaluation of a true QBF F) and builds (an internal representation for) a sat-certificate $C(F)$ for F. Commandline options that impact on this stage are documented in Section 3.1.

Certificate verification. Takes as input a QBF F and (the internal representation of) a sat-certificate $C(F)$ for F (obtained either by the inductive reconstruction engine or by parsing an external previously produced certificate), certifies its validity (see Appendix A), and outputs a `valid` / `invalid` answer. Commandline options that impact on this stage are documented in Section 3.2.

Expression evaluation. Takes as input a sat-certificate $C(F)$ and an expression over $C(F)$ and outputs (the internal representation of) the result of the evaluation of the expression. The commandline option used to provide expressions to be evaluated is documented in Section 3.3, while their syntax and semantics is described in Section 4.

Information dump. Is responsible for dumping to file on request (a) the complete (or partial) certificate in `dot` or `qbm` format (see Appendix B), (b) the result of the evaluation of an expression in `dot`, or `bdd`, or `cnf` format, and (c) a partial certificate in QDIMACS format. What exactly is dumped depends on arguments provided to the `-dump` switch, which are listed in Section 3.4.

The default I/O behaviour of ozziKs depends on the commandline parameters provided (refer to Section 3 for the meaning of the switches mentioned below):

- When an inference log is provided as input, the default behaviour is to

  1. perform inductive certificate reconstruction;
  2. check the internal consistency of the certificate;
  3. verify the certificate (against the QBF referenced in the log itself, if no other QBF is explicitly provided via `-qbf`);
  4. output a valid/invalid answer;
  5. dump to file the QBM representation of the certificate (taking into account the arguments of the `-var` option, if it is present).

  The kind of information dumped in the last step can be altered via the `-dump` switch.

- When a (complete) QBM certificate is provided as input and the evaluation of no expression is requested, the default behaviour is to

  1. load the certificate;
  2. check the internal consistency of the certificate;
  3. verify the certificate (against the QBF referenced in the certificate itself, if no other QBF is explicitly provided via `-qbf`);
  4. output a valid/invalid answer;

  Notice that in this case, unless explicitly requested via the `-dump` switch, no dump to file is performed. If a dump to file is requested via `-dump` and variables are pruned via `-var`, the (complete) certificate will be overwritten by the requested partial version.

- When a (complete) QBM certificate is provided as input and the evaluation of one or more expressions is required (`-eval` switch present), the default behaviour is to

  1. load the certificate;
  2. check the internal consistency of the certificate;
  3. evaluate the expression over the certificate;
  4. (for inverse expresions only) dump to file a `bdd` representation of the result of the evaluation of the expression.

  Notice that in this case the verification stage is skipped, unless explicitly requested via the `-check` option. The kind of information dumped in the last step can be altered via the `-dump` switch.

- When a log is provided and the `-eval` switch is present, the default behaviour is to

  1. reconstruct and check the certificate;
  2. evaluate the expression over the certificate;
  3. dump to file the QBM representation of the certificate (taking into account the arguments of the `-var` option, if it is present);
  4. dump to file a `bdd` representation of the result of the evaluation of the expression.

- When a directory is provided as an entry point, the behaviour is as described in Section 5.

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 for the "Inductive reconstruction" engine

`-hbdd '0'..'19' [':' '0'..'19']` is used to select both the heuristics for the dynamic variable reordering in the BDD manager during the inductive certificate reconstruction, and the heuristics for the final reordering performed just before dumping the information to file.

The syntax `-hbdd S:R` gives both the reconstruction reordering rule $S \in \{0, \ldots, 19\}$ and the final reordering style $R \in \{0, \ldots, 19\}$ to be used. If we only provide one argument (e.g.: `-hbdd R`) we instruct the software to use the same heuristics for both cases.

ozziKs uses the CUDD package [7] as a core tool to manage binary decision diagrams. The CUDD implements a number of reordering heuristics, and the `R` value just tells ozziKs which heuristics is to be selected in the CUDD. 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 [7] for the modus operandi of each heuristics. Notice that:

- This option does not affect the semantics of the certificate, but just its representation. Indeed, a certificate is a forest of BDDs over the universal support, so there are as many ways of writing a certificate as possible orderings for universal variables.
- Unless we completely disable reordering with the option:

  ```
  -hbdd 0
  ```

  ...the actual universal order we find in the dumped certificate is the contingent ordering the BDD manager eventually reaches after performing the whole inductive model reconstruction process (during which multiple reordering steps might have been automatically triggered).
- Conversely, if reordering is disabled the whole work is done (and the final certificate dumped) using the left-to-right order of variables in the prefix of the original formula.
- By specifying two different heuristics we can make the solver "refine" the certificate by some costly heuristics just before dumping it, while the whole reconstruction process has benefited from some cheaper rule. For example, with

```
-hbdd 13:19
```

...we obtain the smallest possible version of a certificate which has been built using intermediate larger (but still somehow optimized) representations[2]. If the `-var` option is used to select a subset of interesting existential variables, the final reordering is performed after pruning the non-interesting ones.

`-var A1[-B1] [,A2[-B2] ,...]` makes the specified subset of the functions for existential variables to appear in the certificate, leaving all the others out. The set of interest is specified by a comma-separated list of intervals (with no space in between), each interval of the form $[A, B]$ being written as A-B, where A and B, $A \leq B$ are two positive integers naming the extremes of the interval. Singleton intervals may be written as the name of the single variable they contain. For example:

```
-var 10-20,1-5,30
```

makes the functions associated to the 17 variables in $\{1, 2, ..., 5, 10, 11, ..., 20, 30\}$ appear in the certificate (we suppose those variables are actually existentially quantified), while all the others (if any) are skipped. Some usage notices:

- A certificate that does not contain the interpretation of every skolem function is not a certificate, in fact. Let us call it a *partial certificate*. A partial certificate cannot be in general used to certify the validity of a formula.
- When requested to produce a partial certificate ozziKs will first produce a complete certificate against which the formula is checked, then it will drop the variables not requested before dumping information to files.
- A partial certificate may still be very useful if one knows, for example, that the dumped variables are the only *functionally independent* ones (all the rest can be computed given the universal scenario and the partial certificate). In this case, a partial certificate contains all the relevant information of a complete certificate while being (possibly) much smaller than that. For example, in a deterministic game a few variables dumped in the partial certificate may represent the moves of the existential player, while all the other variables just encode the state of the game (a thing which deterministically follows from the universal/existential moves).

3.2 Options for the "Certificate verification" engine

`-qbf QDIMACS_FILE_NAME` lets ozziKs know explicitly the name of the file containing the instance against which we want to verify the certificate. This option is not meant to be used in the batch mode. In the single-instance mode this option is often unnecessary, because the log file and the certificate both contain a pointer to the instance from which the log/certificate has been obtained, which is supposed to be the same against which we want to perform certification. If the instance and the log/certificate have a different relative position in the file system w.r.t. when they were solved/created, ozziKs may be unable to locate the qdimacs file, and this option can be used to supply it with a proper path.

`-check` requests explicitly to ozziKs to verify the validity of the certificate against the formula it refers to. This option is unnecessary in all the cases in which validity is examined by default, namely (1) when an inference log is given as input, and (2) when a certificate is provided as input an no expression is to be evaluated. When a certificate is loaded and

[2]The exact reordering is only time-feasible for very small formulas. Or, for big formulas with few universal variables and where most of the existential variables have been pruned away via the `-var` option.

some expression is given, validity check is skipped by default, and can the forced by this option. Also, in batch mode the default behaviour is to skip certificates and only process log files. With this options ozziKs is requested to check the certificates it encounters (while certificates just produced are checked anyway).

3.3 Options for the "Expression evaluation" engine

-eval "EXP" requests the evaluation of the (double-quote delimited) *direct* or *inverse* expression `EXP`. ozziKs distinguishes the two cases by their syntax. For details on the syntax and semantics of expressions see Section 4.

This switch also accepts a path to a (text) file instead of an expression. The file is opened, parsed line-by-line, and to each line starting by a (syntactically and semantically) valid expression over the current certificate, a tail is added where the result of the evaluation is provided (see Examples 6-7 in Section 6).

3.4 Options for the "Information dump" engine

-dump INFO enables the dumping to file of specific pieces of information. `INFO` is a space-separated list of arguments. Each argument is a menmonic string that specifies what has to be dumped. Possibilities are as follows:

qbm=bdd requests to dump the whole certificate as a forest of BDDs in a ozziKs format based on the DDDDMP-2.0 format (see Appendix B). This dump is enabled by default when an inference log is given as input, and disabled otherwise. The format produced is amenable to be re-parsed by ozziKs itself, or to be load and processed by some other application. For a logfile obtained from `instance.qcnf`, the name of the file containing the BDD certificate is `instance.qcnf.qbm`.

qbm=dot requests to dump the whole certificate as direct acyclic graph representing the BDD forest in the DOT format[3] (see Appendix B). This dump is disabled by default. When `qbm=dot` is specified, the dump to BDD format is disabled. The name of the file containing the DOT certificate is `instance.qcnf.qbm.dot`.

qbm requests to dump both the BDD and the DOT representation of the certificate.

exp=bdd requests to dump the result of the evaluation of an inverse expression or a set of expressions provided via the `-eval` switch (see Section 4.3) in a BDD-based format. This type of dump is enabled by default when `-eval` is used. The name of the file containing the result is `instance.qcnf.qbm.exp.bdd`. If multiple expressions are evaluated within a single `-eval` argument, this file contains a forest with a root (or two) for each expression.

exp=bdds works like `exp=bdd`, but if multiple expressions are evaluated in a single `-eval` argument, separate files are dumped, one for each of them. The name of the files containing the representation of the result of each expression is `instance.qcnf.qbm.`N`.exp.bdd`, where N is progressive counter starting from 1.

exp=dot requests to dump the result of the evaluation of an inverse expression as a DOT file. This type of dump is disabled by default, and when activated it disables the BDD dump. The name of the file containing the DOT representation of the result is `instance.qcnf.qbm.exp.dot`. If multiple expressions are evaluated within a single `-eval` argument, this file contains a forest with a root (or two) for each expression. See Section 4.3 for the content of the DOT file.

[3]The *DOT* format is a standard language to describe graphs. It is amenable to be parsed by automatic graph drawing programs like "*graphviz*" which visualize them. Both the rendering software and the language description can be found at `www.graphviz.org`.

exp=dots works like `exp=dot`, but if multiple expressions are evaluated within a single `-eval` argument, separate files are dumped, one for each of them. The name of the files containing the result of each expression is `instance.qcnf.qbm.`N`.exp.dot`, where N is a progressive counter starting from 1.

exp=cnf requests to dump the result of the evaluation of an inverse expression as a (couple of) CNF instances in the DIMACS format. This type of dump is disabled by default, and when activated it disables the BDD dump. Only one instance per expression is dumped if no DONT-CARE condition exists for such expression. The name of the file containing the CNF representation is `instance.qcnf.qbm.exp.cnf` if one single instance is evaluated, or `instance.qcnf.qbm.`N`.exp.cnf` for multiple instances, where N is as described before. If DONT-CARE conditions exist, two instances are dumped for each expression evaluation: a `instance.qcnf.qbm.true.exp.cnf` instance capturing scenarios in which the expression is true, and one `instance.qcnf.qbm.false.exp.cnf` instance for false scenarios. The `instance.qcnf.qbm.`N`.(true|false).exp.cnf` format is used to dump multiple evaluations of expressions with DONT-CARE conditions.

exp requests to dump all the three formats for the results of inverse expressions.

direxp enables the dump to a file named `instance.qcnf.qbm.exp.direct` of the direct expression(s) in input followed by the result(s) of their evaluation(s).

qbc requests to dump to a file named `instance.qcnf.qdc` a DIMACS1.1 compliant version of a partial certificate for the formula, i.e. a valid assignment to the existential variables in the outermost scope (`qdc` stands for *Quantified Dimacs Certificate*).

all requests to dump all the previously described information.

3.5 Other options

-remove (log|qbm) instructs ozziKs to remove logs/certificates after they have been processed (certificates are removed after they have been checked, logs after the reconstruction process is completed). This option is meant to free automatically the possibly huge amount of disk space occupied by such files, if the interesting piece of information one seeks is just the certification and not the certificate itself.

-report enables the production of a textual report about the certificates produced in a batch-mode run. See Section 5.4 for details.

-v [0..5] controls the output verbosity. The value 0 makes ozziKs absolutely silent, so that the only output is the return value. The value 1 makes it report just a VALID/INVALID feedback for each instance. Higher values cause ozziKs to expose a part of its internal status as its activities go on. The default value is 3 in single-instance mode (which prints on-screen a percentage completion value for each major step), and 1 in batch mode. With a 5 value we additionally obtain an indication of when ozziKs is performing BDD reordering.

-giveup makes ozziKs skip some instances in batch mode. See Section 5.3 for details.

-autogiveup makes ozziKs skip some instances in batch mode. See Section 5.3 for details.

-version prints ozziKs's version.

-copyright prints copyright and license notes.

4 Expressions over certificates and their evaluation

A sat-certificate $C(F)$ for a true QBF F contains lots of useful information about F. The usage of $C(F)$ as a mean to certify the validity of F is just a first step in the exploitation of such information. Once we are sure that $C(F)$ is indeed a valid sat-certificate for F, we may *query* $C(F)$ to obtain *explicit* knowledge about the theory *implicitly* described by F.

We do this by defining two kinds of *expressions* that can be evaluated over a certificate: *direct* and *inverse* expressions. In both cases, the user provides a (boolean) expression E over existential literals as input (the simplest case being just one existential literal). Then:

- In *direct expressions*, the user also provides some (partial) assignment A over the universal variables (hereafter called *scenario*). The result of the evaluation is the truth value of E under the assignment A (in a multi-valued logic which we describe below). Such expressions are named "direct" as they match the input-output behavior of skolem functions: They are meant to compute the truth value of existential variables (or compositions thereof) as a function of user-provided truth values for universal variables.

- In *inverse expressions*, the user does not specify any scenario. Rather, he asks for some representation of all the scenarios in which E holds. Such representation is provided as either a BDD or a CNF on the relevant universal variables. In this case the expressions are named "inverse" for a reason dual to the one above.

So, direct expression evaluations map an existential expression and a universal assignment onto a truth value, inverse ones map an existential expression over a set of assignments.

4.1 Simple direct expressions

The basic syntax to request the evaluation of an existential variable e given a (partial) universal assignment represented as a set of literals $\{\psi_1, \ldots, \psi_m\}$ is $e(\psi_1, \ldots, \psi_m)$, where variables are named by their numeric codes. For example, to know the truth value the certificate assigns to the existential variable with code 10 once the universal variables 2, 3, and 5 have been assigned to FALSE, TRUE, and FALSE respectively, we write

```
-eval "10(-2,3,-5)"
```

The evaluation of direct expressions fails in the following circumstances:

- the variable e in $e(\psi_1, \ldots, \psi_m)$ is not an existential variable;

- e is if fact existentially quantified but it was never mentioned in the matrix of the formula from which the queried certificate originates;

- e is existentially quantified and was mentioned in the matrix, but information about it has been striped off by means of `-var`;

- some of the universal literals in ψ_i were not universally quantified.

It is not requested that the literals $\psi_1, \ldots \psi_m$ constitute a total assignment over the universal variables dominating e. They may just specify a partial assignment, or (pointlessly) assign some universal variable not dominating e, or even assign no variable at all.

For this reason, the result of a successful evaluation for e can in general be more complex than a true/false answer. The full set of possibilities is as follows.

Outcome	Symbol	Meaning	Return code
Q_FALSE	F	e is FALSE, in the sense that in ALL the universal scenarios consistent with the inputs provided to the query the existential variable e takes the value FALSE according to its skolem function	0
Q_TRUE	T	e is TRUE, in the sense that in ALL the universal scenarios consistent with the inputs provided to the query the existential variable e takes the value TRUE according to its skolem function	1
Q_WEAK_FALSE	wF	e is weakly FALSE, in the sense that in SOME of the universal scenarios consistent with the inputs provided to the query, the bit has to be FALSE, while in SOME others it is a DONT-CARE condition; so, it is safe to consider it as FALSE though more determined input conditions may turn the answer for this variable into a Q_DC (see below)	2
Q_WEAK_TRUE	wT	e is weakly TRUE, in the sense that in SOME of the universal scenarios consistent with the inputs provided to the query, the bit has to be TRUE, while in SOME others it is a DONT-CARE condition; so, it is safe to consider it as TRUE though more determined input conditions may turn the answer for this variable into a Q_DC (see below)	3
Q_DC	DC	e is in a DONT-CARE condition, in the sense that there is no universal scenario consistent with the inputs of the query in which any specific assignment to this variable (to either true or false) may result in a contradiction	4
Q_DEPENDS	?	With the (universal) inputs that have been provided it is not possible to fix a value for e, as in some of the cases consistent with the input such variable has to be FALSE, while in some other cases it has to be TRUE	5
Q_ERROR	err	Some error occurred.	6

The result of the evaluation of a direct expression is communicated to the user by:

- Writing out the expression and the "symbol" outcome in verbose mode. For example:

```
10(-2,3,-5) = wT
```

- Returning to the calling script or shell the exit code mentioned in the last column of the above table;
- Dumping the whole "`10(-2,3,-5) = wT`" to the file `instance.qdimacs.qbm.exp`, if this dump has been requested through the "`-dump direxp`" option.

4.2 Compound direct expressions

Rather than asking for the truth value of an existential literal, we may compose existential literals into boolean expressions and ask for the truth value of such expressions (in some scenario).

The syntax of compound direct expressions is as follows.

- if e is (the numeric code of) an existential variable, then e and $-e$ are expressions.
- if arg is an expression, `not(`arg`)` and `dc(`arg`)` are expressions[4].
- if arg_1 and arg_2 are expressions, so are `and(`arg_1`,`arg_2`)`, `or(`arg_1`,`arg_2`)`, `xor(`arg_1`,` arg_2), and `implies(`arg_1`,`arg_2`)`[5].

[4] Parentheses can be omitted for these two unary operators, so we can write "`not` arg" and "`dc` arg". The "`not`" operator can be equivalently written as "`!`" (with or without braces).

[5] Parentheses cannot be omitted here, and operators are only binary at present.

A compound expression E evaluated in the universal scenario $\psi_1 \ldots, \psi_m$ is noted $< E >$ $(\psi_1 \ldots, \psi_m)$. For example:

```
-eval "<and(-20,or(24,25))>(-2,3)"
```

Like simple expressions, direct compound expressions evaluate to one of the six truth values described in the previous section (disregarding error conditions). The result of the evaluation of direct compound expressions is first defined for total universal assignment, then extended to partial ones:

- Given an existential literal e and an assignment $\psi_1, \ldots, \psi_m$ that is total over the set of universal variables dominating e, the expression $e(\psi_1, \ldots, \psi_m)$ cannot evaluate to an arbitrary value in the table, but only to one in {T,F,DC}. Namely, for a certificate C in which the couple of skolem interpretations associated to e is $\langle \mathcal{E}^+, \mathcal{E}^- \rangle$ (see Appendix A), it is
  - $e(\psi_1, \ldots, \psi_m) =$ T if $\langle \psi_1, \ldots, \psi_m \rangle \in \mathcal{E}^+$;
  - $e(\psi_1, \ldots, \psi_m) =$ F if $\langle \psi_1, \ldots, \psi_m \rangle \in \mathcal{E}^-$;
  - $e(\psi_1, \ldots, \psi_m) =$ DC if $\langle \psi_1, \ldots, \psi_m \rangle \notin \mathcal{E}^+ \cup \mathcal{E}^-$.

  From existential literals, this evaluation is extended to expressions (considering as "total for an expression" an assignment which is total on the dominating set of the deepest existential literal in the expression) according to the truth tables:

and	F	T	DC
F	F	F	F
T	F	T	DC
DC	F	DC	DC

not	
T	F
F	T
DC	DC

dc	
T	F
F	F
DC	T

  ... and by posing or(arg_1,arg_2)=!and(!arg_1,!arg_2), implies(arg_1,arg_2)= or(!arg_1,arg_2), xor(arg_1,arg_2)=or(and(!arg_1,arg_2),and(arg_1,!arg_2)).

- For any universal assignment U which is not total for an expression E, we consider each total assignment extending U which is total for E. For each such assignment, the expression E evaluates to a value in {T,F,DC}, according to the rule given at previous item, hence to a value in {T,wT,F,wF,DC,?} according to the definitions given in Section 4.1.

The feedback provided to the user after the evaluation of compound expressions is the same as in the case of simple expressions. For example, we may read on-screen the answer:

```
<and(-20,or(24,25))>(-2,3) = wT
```

4.3 Inverse expressions.

The syntax for inverse expressions is similar to the syntax for direct expressions, but arguments (and angular braces) are dropped. For example, we write:

```
-eval "and(-20,or(24,25))"
```

Rather than evaluating to a specific truth value, this expression evaluates to a set of scenarios. Namely, it represents all the scenarios in which and(-20,or(24,25)) is true, i.e. all the universal assignments under which the skolem interpretations given in the certificate for the existential variables 20, 24, and 25 are such that 20 evaluates to F, and at the same time at least one out of 24 and 25 evaluates to T.

A set of scenarios requires a suited representation to be delivered to the user: ozziKs uses either BDDs, or DOT representations, or CNF formulas (all these three formats are dumped to file). The type of dump is controlled via the `-dump` switch as discussed in Section 3.4. In the case of a CNF answer, for example, a propositional instance on the relevant universal variables is provided whose models are all and only the scenarios that meet the required condition. Such instance is written to file in the DIMACS format for the user to process. For detailed examples of inverse expression evaluations see Example 7.

The set of scenarios in which the expression provided evaluates to true is not enumerated in the on-screen feedback. Rather, an indication is given on the percentage of assignments (over the set of all the possible total universal assignments) that makes the expression evaluate to true. For example, we may see an answer like:

```
and(-20,or(24,25)) = 25%
```

which means that in 1 out of 4 (total) scenarios the expression we provided evaluates to `T`, while in the other 3 it evaluates to `F`. The special strings `always` and `never` are provided as answers in the 100% and 0% cases respectively.

There is one last thing to take into account: Compound existential expressions evaluated over total universal assignments may as well evaluate to `DC`. This means that once an expression like `and(-20,or(24,25))` is provided, we expect (1) a representation of the scenarios in which it evaluates to `T` *and* (2) a second representation of the scenarios in which it evaluates to `F`, and this because there might be cases that do no fall in either set: the `DC` scenarios.

The on-screen feedback for expressions in which DONT-CARE conditions exist is like this:

```
and(-20,or(24,25)) = [25%:50%]
```

... which means that in 25% of the scenarios the expression is true, in 50% it is false, and in the remaining 25% it is a DONT-CARE. The dump to file is also slightly affected by the existence of DONT-CAREs: For BDD and DOT representations, two roots exist in the diagram instead of just one: the root of the BDD (or DOT diagram) representing the arguments on which the expression evaluates to `T`, and the one in which it evaluates to `F`. For CNF representations, two separate CNFs are dumped to capture the two sets (see Section 3.4 and Example 7 for details).

4.4 Multiple expressions.

A syntax is provided to request the evaluation of more than one existential expression at once on the same universal input. To request the evaluation of the direct expressions $E_1, \ldots, E_n$ under the same assignment $\psi_1, \ldots, \psi_m$, we write $< E_1, \ldots, E_n > (\psi_1, \ldots, \psi_m)$. The answer is itself a tuple of size n enclosed in angular parentheses. For example:

```
<10,15,20,21>(1,-2,3,-5,8) = <T,F,wF,DC>
```

In case of tuple answers the return code outcome is not used, so the user may know the answer either on-screen, or requesting its dump to file. For inverse expressions the syntax is similar:

```
<24,xor(25,10),or(dc 21, dc 25)> = <50%,always,[25%:25%]>
```

5 Batch mode

When we launch ozziKs giving a subdirectory as an entry point (rather than a single log/certificate file), the solver operates in *batch mode*. It traverses the whole directory subtree rooted at the given entry point, processing each log file encountered during the visit. If the `-check` option is provided, QBM certificates are looked for and processed instead of log files.

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.

5.1 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 recursive visit takes place in post-order (log/certificates are processed before visiting nested directories).
- Log/certificates in each directory are first sorted according to a case-insensitive natural lexicographic ordering[6] of the file names, then processed in such order.

5.2 Return values

In batch mode, ozziKs returns the number of instances successfully processed (i.e. those for which the certificate has been successfully verified). 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 valid certificates built.

5.3 Timeouts

When ozziKs 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 log/certificate*, then moves on to the next one.

Additionally, the **-giveup** and **-autogiveup** options can be used to skip one entire family when a timeout or a problem occurs. More precisely, these options instruct the reconstructor to give up after the first log/certificate in a familiy cannot be produced or successfully verified. See the manual of sKizzo [1] for further details.

[6] This is the kind of ordering implemented by the PHP `strnatcasecmp` function, which is based on the C implementation by Martin Pool we find at `http://sourcefrog.net/projects/natsort/`. In this ordering, numeric substrings without enough leading zeros do not disrupt the human-friendly sequence (e.g. `adder9.qdimacs` comes before `adder10.qdimacs`), because any string is "exploded" into numeric and non-numeric components which are evaluated left-to-right according to a lexicographic comparison (non-numeric components) or an arithmetic comparison (numeric components).

5.4 Reporting

The **-report** option enables the production of a textual report about the certificates produced in a batch-mode run. A textual report is placed in each directory traversed, containing information about certificates extracted from instances in that directory. The report is named after the directory itself and has a `.qbm.txt` extension.

A report contains one line for each processed log file. Each line contains the following tab-separated information on the processed instance:

1. name of the instance
2. number of variables
3. number of clauses
4. prefix shape
5. number of alternations
6. time taken to solve (secs)
7. time taken to reconstruct the certificate (secs)
8. time taken to verify the certificate (secs)
9. Size of the inference log (number of steps)
10. Size of the model (number of internal nodes in the BDD forest)
11. Outcome (either "VALID" for a successful reconstruction/verification, or "INVALID" in case something went wrong)

See Example 5 for a case of batch processing with report construction.

6 Examples of usage

Example 1: *Basic certificate reconstruction and verification.*

In this first example we show how to use ozziKs in conjunction with the QBF solver sKizzo for a basic task and on a small instance. Both these softwares have to be available for you to reproduce the sample certificate extraction and check discussed below.

We consider a small publically available (TRUE) QBF instance[7], named "`s27_d2_s.qcnf`". It is a formula containing 142 clauses, whose prefix has the shape: $\exists E_1 \forall A \exists E_2$. The three disjoint sets of variables E_1, A, and E_2 contain 29, 10, and 26 variables respectively. As a sat-certificate for this formula we want to exhibit:

1. For every variable in E_1, an assignment to a truth value in $\{$`T, F, DC`$\}$, and
2. For every variable in E_2, a function from the set of truth assignments over A (a set of size $2^{10} = 1024$) into $\{$`T,F, DC`$\}$.

We start by issuing the command:

```
sKizzo -log s27_d2_s.qcnf
```

Notice the presence of the switch "`-log`", which causes the solver to produce an *inference log* of the solution process (see Appendix A and Appendix B for details).

After being prompted with a response like:

```
sKizzo v0.8.2-beta, (revision 3xx, compiled Mar 16 2006)
1.Processing "./s27_d2_s.qcnf"...  [OK, TRUE, 6.95Mb, 0.17s]
1/1 instance successfully solved.
```

... we know the instance is TRUE (by deduction), and we obtain a new file containg the inference log. It is named after the instance, with just an additional "`.sKizzo.log`" suffix:

```
s27_d2_s.qcnf.sKizzo.log
```

At this point, we ask ozziKs to process the inference log, by issuing the command:

```
ozziKs s27_d2_s.qcnf.sKizzo.log
```

As a result, we obtain something like this:

```
ozziKs v0.3-beta, (revision 392, compiled Nov  7 2006)

1.Processing "s27_d2_s.qcnf.sKizzo.log"...
  a.Model reconstruction and certification.
    1.Inductive reconstruction [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.02s]
    2.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    3.Validity check           [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Certificate is VALID, 0.02s, 0.00% in BDD]
  b.Dump to file:
    1.Additional reordering [OK, 0.00s]
    2.QBM representation: "s27_d2_s.qcnf.qbm" [OK, 25 nodes, 1.0 KBytes, 0.00s]
    [OK, 0.00s]
  [OK, VALID, 0.03s]
```

... where:

[7] This instance encodes a sequential depth computation problem (at depth 2) for the "s27" circuit in the ISCAS'89 benchmarks. It has been produced by Maher Mneimneh, and is available from `www.qbflib.org`.

- Row `1.a.1` documents the advancement of the inductive model reconstruction process that ozziKs performs (inductive reconstruction engine). On small instances (like our sample one), such process is almost instantaneous. On more complex instances, we see a gradual advancement of the completion bar: Certificate reconstruction may require a sensible amount of time/memory[8].
- Row `1.a.2` documents the consistency check that ozziKs performs on the certificate[9].
- Row `1.a.3` reports on checking the matrix of the QBF in `s27_d2_s.qcnf` against the certificate (certificate verification engine). So, the answer "`Certificate is VALID`" means that a solver-independent, evaluation-based approach confirms that the formula is indeed TRUE, ruling definitely out the possibility of an unsound answer by the solver (see Appendix A).
- Row `1.b.2` witnesses the dumping to file of a QBM representation for the certificate. This means that a file named after the original instance is created (with an additional `.qbm` extension):

  ```
  s27_d2_s.qcnf.qbm
  ```

  This file contains a BDD-based representation for the set of dependencies between universal and existential variables we were looking for (see Appendix B). Notice that the number of nodes in the forest as a whole, and the file size are also reported.

  It is possible to give such file back to ozziKs as input, in case we are interested in manipulating/querying it without being forced to reconstruct it from scratch (see next example).

Example 2: *How to dump more information.*

We may give ozziKs some additional commandline options to trigger further information dump (see Section 3.4). For example, by issuing

```
ozziKs -dump qdc qbm=dot s27_d2_s.qcnf.sKizzo.log
```

we ask the dump engine to produce a DOT representation of the certificate instead of a QBM one, and to dump a DIMACS1.1 partial certificate. We may as well provide as input the QBM certificate already constructed. To do this, we issue:

```
ozziKs -dump qdc qbm=dot s27_d2_s.qcnf.qbm
```

ozziKs's reply becomes like this:

```
ozziKs v0.9-beta, (revision 152, compiled Nov  7 2006)

1.Processing "s27_d2_s.qcnf.qbm"...
  a.Certification
    1.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Verification skipped as requested, 0.01s]
  b.Dump to file:
    1.Additional reordering [OK, 0.00s]
    2.DOT representation: "s27_d2_s.qcnf.qbm.dot" [OK, 0.00s]
    3.QDC representation: "s27_d2_s.qcnf.qdc" [OK, 0.00s]
    [OK, 0.00s]
  [OK, VALID, 0.01s]
```

[8] Occasionally, it might come out to be more (computationally) expensive than the solution process itself.

[9] The representation used for certificates (see Appendix A) is general enough to represent objects that are not valid certificates (in that they require a variable to take both truth values at the same time). So, this test is mainly intended to exclude that any problem leading to the generation of an invalid certificate arose during the reconstruction.

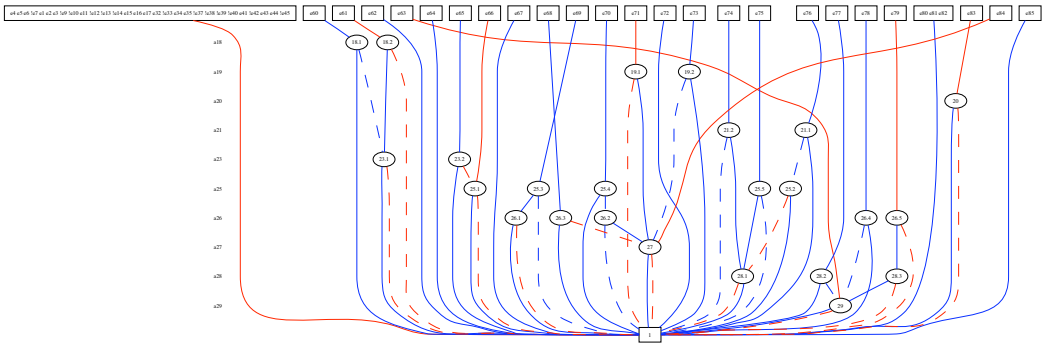


Figure 2: DOT representation of a (complete) sat-certificate for "`s27_d2_s.qcnf`".

... where:

- Only an internal consistency check is performed, as expected (the actual validity check may be forced by means of `-check`).
- Row `1.b.2` documents the dump triggered by the `qbm=dot` argument of `-dump`. The "`s27_d2_s.qcnf.qbm.dot`" file produced contains the very same information we find in "`s27_d2_s.qcnf.qbm`", but in a format amenable to be rendered graphically[10]. The result of such rendering is depicted in Figure 2. To know how to read it, please see Appendix B.3. Notice that the DOT format is an *output-only* format that cannot be parsed again by ozziKs.
- Row `1.b.3` documents the dump triggered by the `qdc` argument. The "`s27_d2_s.qcnf.qdc`" file produced contains a subset of all the universal/existential dependencies contained in the certificate, namely those concerning the outhermost existential scope (whose variables do not depend on any universal, hence are associated to constant functions). A validity-preserving assignment to the variables in such outermost scope is hence dumped in the DIMACS1.1 output format[11]. In our case, its content is like this:

```
c Solver: sKizzo v0.8.2-beta, (revision 3XX, compiled Mar 23 2006)
c Instance: "s27_d2_s.qcnf"
s cnf 1 66 [unknown]
v -4 -5 7 -1 -2 -3 9 10 -11 12 13 14 -15 -16 -17
v -32 33 -34 37 38 39 40 -41 42 -43 -44 45
```

  In the last rows of this file we recognize the required assignment to the outermost existential scope. By adding to the `s27_d2_s.qcnf` instance the unit clauses -4, -5, 7, etc. we obtain a true formula with a universal outermost scope[12].

[10] The complexity of the graphical representation is such that the rendering can be actually accomplished only for very small formulas/certificates (or, for certificates heavily pruned via `-var`), so this feature is mainly for exemplification and/or debug purposes.

[11] A specification of this format is given at `http://www.qbflib.org/qdimacs.html`. The DIMACS 1.1 file is built by ozziKs by extracting a few relevant information out of the qbm file (i.e. the zero-arity functions). Nevertheless, the whole certificate is built, which could be (much) harder than what is actually required to just compute a valid outermost assignment. This means that an optimized version of `ozziKs` that knows it has only been requested to produce the outermost assignment could be much faster. This optimization is currently not implemented.

[12] The DIMACS 1.1 specification requires that a solver dumps (i) the outermost existential assignment for TRUE instances, OR (ii) an outermost universal witness of inconsistency for FALSE instances. Unfortunately, the request number (ii) strongly relies on the hypothesis that we are working with a search-based solver. Right before answering FALSE, these solvers find themselves on a universal branch that shows how the cofactored existential formula fails to be true. They have an easy time in dumping such information. Conversely, semantics-based approaches just suppose that TRUE formulae have models (and `ozziKs` gives us a piece of that model in the case (i)), while FALSE formulae do not (`ozziKs` gives us nothing in this case).

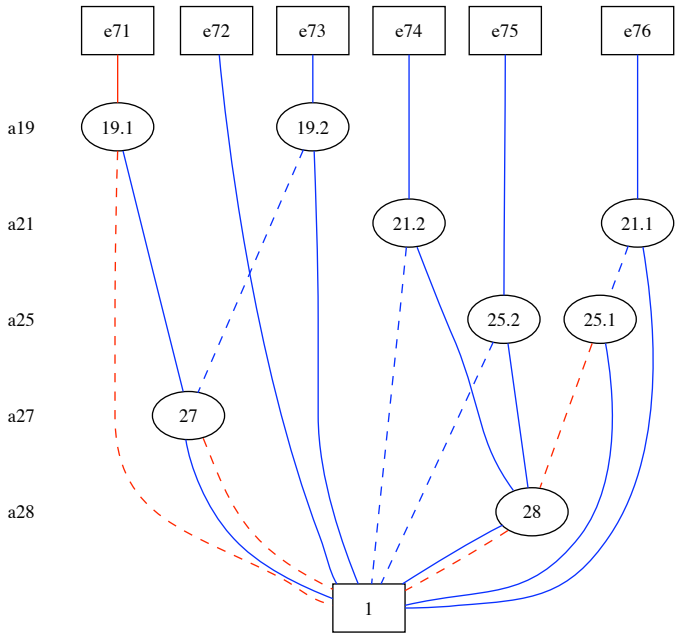


Figure 3: DOT representation of a (partial) sat-certificate for "`s27_d2_s.qcnf`". Only the skolem interpretations for variables in the interval $[71-76]$ are preserved.

Example 3: *Partial certificates.*

Suppose we are only interested in the behaviour of variables $[71-76]$ of the formula $s27\_d2\_s.qcnf$ and we want a graphical representation of such behaviour.

By issuing the command:

```
ozziKs -dump qbm=dot -var 71-76 s27_d2_s.qcnf.qbm
```

... we obtain the BDD forest in Figure 3.

Notice that the QBM certificate is unaffected by the above operation. If what we desire is to strip variables not in $[71-76]$ from the very QBM certificate (thus turning it in a *partial* certificate), we can either issue:

```
ozziKs -dump qbm=bdd -var 71-76 s27_d2_s.qcnf.qbm
```

or we can provide the `-var` switch to the reconstruction engine in the first place:

```
ozziKs -var 71-76 s27_d2_s.qcnf.sKizzo.log
```

Example 4: *"Don't care" conditions.*

In Figure 2 and Figure 3, each existential variable is associated to only one root in the forest. This happens because no DONT-CARE condition occurs, so it is enough to know the positive function associated to that variable (as the negative one is just its complement). However, suppose we consider the QBF instance

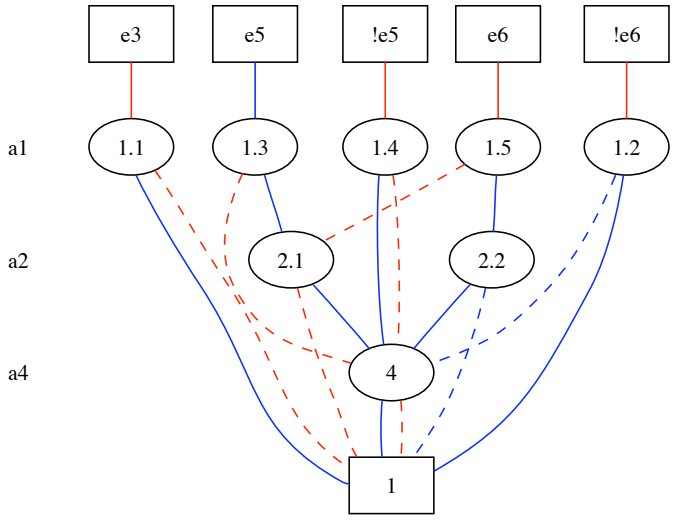


Figure 4: DOT representation of a (total) sat-certificate of a QBF with DONT-CARE conditions.

$$\forall a \forall b \exists c \forall d \exists e \exists f.\ (\neg b \vee e \vee f) \wedge (a \vee c \vee f) \wedge (a \vee d \vee e) \wedge (\neg a \vee \neg b \vee \neg d \vee e) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg c \vee \neg f) \wedge (a \vee \neg d \vee \neg e) \wedge (\neg a \vee d \vee \neg e) \wedge (a \vee \neg e \vee \neg f).$$

... which is encoded in the standard QDIMACS format by choosing an arbitrary correspondence between variable names and positive integer values, such as

$$a \leftrightarrow 1 \quad b \leftrightarrow 2 \quad c \leftrightarrow 3 \quad d \leftrightarrow 4 \quad e \leftrightarrow 5 \quad f \leftrightarrow 6$$

...we obtain a QDIMACS instance which we write in a `example.qdimacs` file as follows:

```
p cnf 6 9
a 1 2 0
e 3 0
a 4 5 6 0
-1 2 -3 0
-1 -2 -4 5 0
-1 4 -5 0
1 3 6 0
1 -4 -5 0
1 4 5 0
-1 -3 -6 0
-2 5 6 0
1 -5 -6 0
```

By processing this instance with sKizzo/ozziKs:

```
sKizzo -log example.qdimacs
ozziKs -dump qbm=dot example.qdimacs.qbm
```

... we obtain for its sat-certificate the DOT representation depicted in Figure 4. We notice that while variable 3 (i.e. variable c) is associated to a skolem function with no DONT-CARE condition, variables 5 and 6 have some DONT-CARE conditions: They indeed appear as a positive literals (labeling the root of diagrams representing conditions under which they evaluate to true) and negative literals (labeling the root of diagrams representing conditions under which they evaluate to false). Cases captured by none of the two diagrams are DONT-CAREs.

Example 5: *Batch processing.*

Suppose we want to

- process all the inference logs in the directory `/path/to/my/instances/` (and subdirectory thereof) so to produce the respective certificates and dump them to file, and
- verify such certificates against their originating QBFs, and
- dump for each reconstruction/verification a line of information into the textual report `/path/to/my/instances/instances.qbm.txt`

We just issue:

```
ozziKs -report /path/to/my/instances/
```

For example, if we solve by sKizzo all the instances in the directory `/path/to/qshifter`[13] with the `-log` option enabled:

```
sKizzo -log /path/to/qshifter
```

... and then we issue the command:

```
ozziKs -report /path/to/qshifter
```

... all the certificates are reconstructed and verified, and a report file named `qshifter.qbm.txt` is creaded inside `/path/to/qshifter` with approximately the following content:

```
qshifter_3  19    128 A[11]E[8]    1 0.00 0.00 0.00  10   33 VALID
qshifter_4  36    512 A[20]E[16]   1 0.00 0.00 0.01  18   81 VALID
qshifter_5  69   2048 A[37]E[32]   1 0.03 0.02 0.01  34  193 VALID
qshifter_6 134   8192 A[70]E[64]   1 0.19 0.09 0.07  66  449 VALID
qshifter_7 263  32768 A[135]E[128] 1 1.10 0.45 0.33 130 1025 VALID
qshifter_8 520 131072 A[264]E[256] 1 8.40 2.32 1.54 258 2305 VALID
```

If we desire to process all the inference logs in the directory `/path/to/my/instances/` (and subdirectory thereof) so to produce the respective certificates, in such a way that after each certificate is dumped to file its originating log gets erased, and we also prefer that if a certificate cannot be reconstructed within 10 seconds, then ozziKs has to give up and continue with the next log, we issue:

```
ozziKs -remove log /path/to/my/instances/ 10
```

Notice that logs for which certificate reconstruction times out are not erased.

[13] We suppose this directory happens to contain the 6 QBF instances of the `q-shifter` family as encoded by Pan, and as available from `www.qbflib.org`.

Example 6: *Evaluation of direct expressions .*

Let us consider the QBF and the certificate we introduced in Example 4. Suppose we want to know the truth value of variable 5 under the scenario $\{-1, 4\}$. We issue:

```
ozziKs -eval "5(-1,4)" example.qcnf.qbm
```

... and we obtain:

```
1.Processing "sat.qdimacs.qbm"...
  a.Certification
    1.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Verification skipped as requested, 0.01s]
  b.Evaluation (direct)
    1. 5(-1,4) = F [OK, 0.00s]
    [OK, 0.00s]
  c.Dump to file: [OK, 0.00s]
  [OK, VALID, 0.01s]
```

where the answer we look for is provided at line 1.b.1 (we see that the answer is consistent with Figure 4). Now we ask the evaluation of multiple existential literals over the same scenario, and we also request a dump to file of the result:

```
ozziKs -eval "<3,5,6>(-1,4)" -dump direxp example.qcnf.qbm

ozziKs v0.3-beta, (revision 372, compiled Nov  7 2006)

1.Processing "sat.qdimacs.qbm"...
  a.Certification
    1.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Verification skipped as requested, 0.01s]
  b.Evaluation (direct)
    1. 3(-1,4) = T [OK, 0.00s]
    2. 5(-1,4) = F [OK, 0.00s]
    3. 6(-1,4) = wT [OK, 0.00s]
    [OK, 0.00s]
  c.Dump to file:
    1.Direct expression result: "example.qcnf.qbm.direct" [OK, 0.00s]
    [OK, 0.00s]
  [OK, VALID, 0.01s]
```

The file "`example.qcnf.qbm.direct`" now contains a single line of text:

```
<3,5,6>(-1,4) = <T,F,wT>
```

We could have requested the evaluation of expression(s) rather than single variables. For example, by issuing:

```
ozziKs -eval "<3, -6, xor(3,-6), and(-3,implies(-5,6))>(1)"      \
       -dump direxp example.qcnf.qbm
```

we request to evaluate the four expressions 3, -6, $xor(3, -6)$, and $and(-3, implies(-5, 6))$ over 1, i.e. over all the scenarios in which 1 is true. In the dump file we read the answer:

```
<3,-6,xor(3,-6),and(-3,implies(-5,6))>(1) = <F,wF,wF,wT>
```

It is possible to collect several expressions to be evaluated inside some textual file, and then request to evaluate them all. For example, we can issue the command:

```
ozziKs -eval "./myExpressions.txt" example.qcnf.qbm
```

...where "`myExpressions.txt`" is a pure-text file whose content is shown in Figure 5. When ozziKs terminates, a new file "`myExpressions.txt.result`" exists, whose content is depicted in Figure 6: The two files are identical, but the results of each expression evaluation have been inserted at proper places.

```
*****************************************************************
* This is a test file for ozziKs' expression evaluation engine *
*****************************************************************

The formula whose certificate we query is:

\forall 1,2 \exists 3 \forall 4 \exists 5,6
(-1 2 -3) and (-1 -2 -4 5) and (-1 4 -5) and
(1 3 6) and (1 -4 -5) and (1 4 5) and
(-1 -3 -6) and (-2 5 6) and (1 -5 -6)

The empty universal assignment:
<3,5,6>()  <--- here we expect that no variable can get
                a definite value (as the outermost scope
                is universal)

All the partial assignment of size 1:
<3,5,6>( 1)
<3,5,6>(-1)
<3,5,6>( 2)
<3,5,6>(-2)
<3,5,6>( 4)
<3,5,6>(-4)

All the partial assignment of size 2:
<3,5,6>( 1, 2)
<3,5,6>( 1,-2)
<3,5,6>(-1, 2)
<3,5,6>(-1,-2)
<3,5,6>( 1, 4)
<3,5,6>( 1,-4)
<3,5,6>(-1, 4)
<3,5,6>(-1,-4)
<3,5,6>( 2, 4)
<3,5,6>( 2,-4)
<3,5,6>(-2, 4)
<3,5,6>(-2,-4)

All the partial assignment of size 3:
<3,5,6>( 1, 2, 4) // here it can only be T,F or DC
<3,5,6>( 1, 2,-4) // here it can only be T,F or DC
<3,5,6>( 1,-2, 4) // here it can only be T,F or DC
<3,5,6>( 1,-2,-4) // here it can only be T,F or DC
<3,5,6>(-1, 2, 4) // here it can only be T,F or DC
<3,5,6>(-1, 2,-4) // here it can only be T,F or DC
<3,5,6>(-1,-2, 4) // here it can only be T,F or DC
<3,5,6>(-1,-2,-4) // here it can only be T,F or DC
```

Figure 5: A text file containing direct expressions to be evaluated over `example.qcnf.qbm`

```
************************************************************
* This is a test file for ozziKs' expression evaluation engine *
************************************************************

The formula whose certificate we query is:

\forall 1,2 \exists 3 \forall 4 \exists 5,6
(-1 2 -3) and (-1 -2 -4 5) and (-1 4 -5) and
(1 3 6) and (1 -4 -5) and (1 4 5) and
(-1 -3 -6) and (-2 5 6) and (1 -5 -6)

The empty universal assignment:
<3,5,6>() = <?,?,?>   <--- here we expect that no variable can get
                           a definite value (as the outermost scope
                           is universal)

All the partial assignment of size 1:
<3,5,6>( 1) = <F,?,wT>
<3,5,6>(-1) = <T,?,?>
<3,5,6>( 2) = <?,?,?>
<3,5,6>(-2) = <?,?,wF>
<3,5,6>( 4) = <?,?,wT>
<3,5,6>(-4) = <?,?,?>

All the partial assignment of size 2:
<3,5,6>( 1, 2) = <F,?,wT>
<3,5,6>( 1,-2) = <F,wF,DC>
<3,5,6>(-1, 2) = <T,?,?>
<3,5,6>(-1,-2) = <T,?,wF>
<3,5,6>( 1, 4) = <F,wT,DC>
<3,5,6>( 1,-4) = <F,F,wT>
<3,5,6>(-1, 4) = <T,F,wT>
<3,5,6>(-1,-4) = <T,T,F>
<3,5,6>( 2, 4) = <?,?,wT>
<3,5,6>( 2,-4) = <?,?,?>
<3,5,6>(-2, 4) = <?,wF,DC>
<3,5,6>(-2,-4) = <?,?,wF>

All the partial assignment of size 3:
<3,5,6>( 1, 2, 4) = <F,T,DC>  // here it can only be T,F or DC
<3,5,6>( 1, 2,-4) = <F,F,T>  // here it can only be T,F or DC
<3,5,6>( 1,-2, 4) = <F,DC,DC>  // here it can only be T,F or DC
<3,5,6>( 1,-2,-4) = <F,F,DC>  // here it can only be T,F or DC
<3,5,6>(-1, 2, 4) = <T,F,T>  // here it can only be T,F or DC
<3,5,6>(-1, 2,-4) = <T,T,F>  // here it can only be T,F or DC
<3,5,6>(-1,-2, 4) = <T,F,DC>  // here it can only be T,F or DC
<3,5,6>(-1,-2,-4) = <T,T,F>  // here it can only be T,F or DC
```

Figure 6: A text file containing direct expressions and their evaluation outcome (over `example.qcnf.qbm`)

Example 7: *Evaluation of inverse expressions.*

Suppose we want a graphical representation of the universal scenarios in which we observe a DONT-CARE condition for at least one of the existential variables 5 and 6 in the sat-certificate for `example.qcnf` we extracted at Example 4. We issue:

```
ozziKs -eval "or(dc 5, dc 6)" -dump exp=dot example.qcnf.qbm
```

From the feedback we get:

```
1.Processing "sat.qdimacs.qbm"...
  a.Certification
    1.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Verification skipped as requested, 0.01s]
  b.Evaluation (inverse)
    1. or(dc(5),dc(6)) = 50% [OK, 0.00s]
    [OK, 0.00s]
  c.Dump to file:
    1.DOT representation: "example.qcnf.qbm.exp.dot" [OK, 0.00s]
    [OK, 0.00s]
  [OK, VALID, 0.01s]
```

... we know (line 1.b.1) that in half the scenarios this will be the case. We are also told (line 1.c.1) that the file "`example.qcnf.qbm.exp.dot`" contains a DOT representation of all such cases, which once rendered via *graphviz* appears like in Figure 7.

If the evaluation of multiple expressions is required in the same `-eval` argument (by employing the syntax described in Section 4.4), then all the roots describing the result of such expressions are represented in a single forest when `exp=bdd` or `exp=dot` are passed to `-dump`, while a different file dump for each expression is produced by using `exp=bdds` or `exp=dots`.

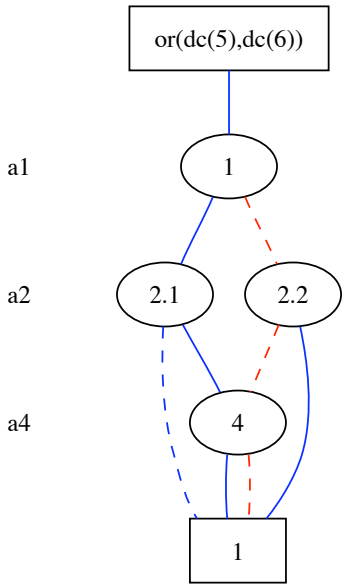


Figure 7: DOT representation for the scenarios in which at least one out of var. 5 and var. 6 are in a DONT-CARE condition in the (complete) sat-certificate "`example.qcnf.qbm`".

To obtain a CNF representation of the scenarios where $or(dc(4), dc(5))$ holds, we ask:

```
ozziKs -eval "or(dc 4, dc 5)" -dump exp=cnf example.qcnf.qbm
```

The content of the DIMACS file "`sat.qdimacs.qbm.exp.cnf`" we obtain as a result is as follows (some comments have been removed):

```
c sKizzo CNF expression dump file
c QBModel: sat.qdimacs.qbm
c Expression: or(dc(5),dc(6))

.
.
.


p cnf 4 3
1 4 0
1 -2 0
-1 -2 4 0
```

We recognize in this file a CNF on the (universal) variables $\{1, 2, 4\}$ whose models (in the SAT sense) are all and only the universal scenarios in which the required condition holds.

With respect to the evaluation of inverse expressions, ozziKs also does the following things:

- Allows us to request the evaluation of multiple expressions at once;
- Allows us to provide the input expression(s) in a textual file, like we did in Example 6;
- Provides a slightly different representation (similar to the one described in Example 4) for expressions that—unlike the case depicted in Figure 7—present DONT-CARE conditions.

For example, by issuing the command:

```
ozziKs -v 4 -eval inverse -dump exp example.qcnf.qbm
```

where the content of the textual file "`inverse`" is depicted in Figure 8, we request to dump to file in all the formats the results of the evaluation of all the expressions contained in such file.

```
*************************************************************
* This is a test file for ozziKs' expression evaluation engine *
*************************************************************

The formula whose certificate we query is:

\forall 1,2 \exists 3 \forall 4 \exists 5,6
(-1 2 -3) and (-1 -2 -4 5) and (-1 4 -5) and
(1 3 6) and (1 -4 -5) and (1 4 5) and
(-1 -3 -6) and (-2 5 6) and (1 -5 -6)

Which are the DONT-CARE condition of each existential variable?
<dc(3),dc(5),dc(6)>

What is the truth value of the xor of -5 and 6 in each scenario?
xor(3,6)
```

Figure 8: A text file containing inverse expressions to be evaluated over `example.qcnf.qbm`

We obtain the following on-screen feedback:

```
1.Processing "sat.qdimacs.qbm"...
  a.Certification
    1.Consistency check        [0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%] [OK, 0.00s]
    [OK, Verification skipped as requested, 0.01s]
  b.Evaluation (inverse)
    1. <dc(3),dc(5),dc(6)>
      a. dc(3) = never [OK, 0.00s]
      b. dc(5) = 12.5% [OK, 0.00s]
      c. dc(6) = 50% [OK, 0.00s]
      [OK, 0.00s]
    2.Dump to file
      a.BDD representation: "inverse.result.line14.exp.bdd" [OK, 0.00s]
      b.CNF representation: "inverse.result.line14.1.exp.cnf" [OK, 0.00s]
      c.CNF representation: "inverse.result.line14.2.exp.cnf" [OK, 0.00s]
      d.CNF representation: "inverse.result.line14.3.exp.cnf" [OK, 0.00s]
      e.DOT representation: "inverse.result.line14.exp.dot" [OK, 0.00s]
      [OK, 0.00s]
    3. xor(3,6)
      a. xor(3,6) = [37.5%:12.5%] [OK, 0.00s]
      [OK, 0.00s]
    4.Dump to file
      a.BDD representation: "inverse.result.line17.exp.bdd" [OK, 0.00s]
      b.CNF representation: "inverse.result.line17.true.exp.cnf" [OK, 0.00s]
      c.CNF representation: "inverse.result.line17.false.exp.cnf" [OK, 0.00s]
      d.DOT representation: "inverse.result.line17.exp.dot" [OK, 0.00s]
      [OK, 0.00s]
    [OK, 0.01s]
  c.Dump to file:
    1.Expressions [OK, Peformed on-the-fly during evaluation, 0.00s]
    [OK, 0.00s]
  [OK, VALID, 0.02s]
```

where:

- Lines 1.b.1.(a-c) tells us that 3 has no DONT-CARE condition, while 5 and 6 are DONT-CAREs in 12.5% and 50% of the scenarios respectively.
- Lines 1.b.2.(a-e) give feedback about the dumps to file. Files relative to different expressions in the same input source are distinguished by including in their name the line number at which the current expression was found. The content of `inverse.result.line14.exp.dot` is depicted in Figure 10. A CNF for each expression is also dumped.
- Line 1.b.3.a says `xor(3,6) = [37.5%:12.5%]`. This means that $xor(3,6)$ is true in 37.5% of the scenarios, is false in 12.5% of the scenarios, hence it is a DONT-CARE condition in the remaining 50% of cases, as confirmed by the `inverse.result.line17.exp.dot` version depicted in Figure 11. In this graphical representation the direct and the negated root of the expression are reported separately, due to the presence of DONT-CARE conditions, as opposed to what happens in Figure 7.
- Lines 1.b.4.(b-c) show that—as opposed to previous cases—two CNFs are dumped (instead of one) to completely capture the behavior of expressions with a non-empty set of DONT-CARE conditions.
- The input file `inverse` is translated into the output file `inverse.result` depicted in Figure 9.

```
*****************************************************************
* This is a test file for ozziKs' expression evaluation engine *
*****************************************************************

The formula whose certificate we query is:

\forall 1,2 \exists 3 \forall 4 \exists 5,6
(-1 2 -3) and (-1 -2 -4 5) and (-1 4 -5) and
(1 3 6) and (1 -4 -5) and (1 4 5) and
(-1 -3 -6) and (-2 5 6) and (1 -5 -6)

Which are the DONT-CARE condition of each existential variable?
<dc(3),dc(5),dc(6)> = <never,12.5%,50%>

What is the truth value of the xor of -5 and 6 in each scenario?
xor(3,6) = [37.5%:12.5%]
```

Figure 9: A text file containing inverse expressions and their evaluation outcome (over `example.qcnf.qbm`)

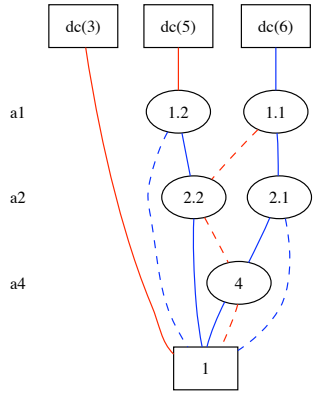


Figure 10: DOT representation for the DONT-CARE conditions of the existential variables in the sat-certificate "`example.qcnf.qbm`".

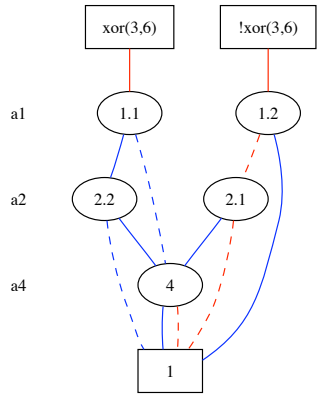


Figure 11: DOT representation for truth value of $xor(3,6)$ in the sat-certificate "`example.qcnf.qbm`".

Appendix A: QBF certificates

The term "certificate" has a fairly general meaning, originating in language recognition and complexity theory. Once *verified*, a certificate proves that the string it refers to actually belongs to a language of interest. Applied to logic, the term denotes any means of providing evidence of (un)satisfiability for a given statement, other than a refutationally-compete deductive approach. In essence, we verify that a given formula belongs to the language of (un)satisfiable statements.

The most natural *certificate of satisfiability* (sat-certificate) for a QBF formula is an explicit representation of any of its *models*. A formula is indeed satisfiable if and only if some model makes it evaluate to true. What we use in ozziKs as a QBF sat-certificate is indeed a compact BDD-based representation of one such model.

A certificate provides *solver-independent* evidence of satisfiability. In addition, it can be inspected to gather explicit information about the semantics of the formula.

For example, let us consider the formula

$$\forall a \exists b \forall c \exists d . M(a, b, c, d)$$

where a, b, c and d are propositional variables, and $M(a, b, c, d)$ is a matrix (a conjunction of clauses) mentioning the variables a, b, c, d. This QBF is true iff two functions exists:

1. A function $b = b(a)$ computing the truth value of b as a function of the truth value of a
2. A function $d = d(a, c)$ computing the value of d as a function of the truth value of a and c

... such that $M(a, b, c, d)$ is satisfied by the assignment $a = v_1, b = b(v_1), c = v_2, d = d(v_1, v_2)$ for every $v_1 \in \{\mathtt{T}, \mathtt{F}\}$ and every $v_2 \in \{\mathtt{T}, \mathtt{F}\}$. These two functions give us a *strategy* to satisfy the matrix, i.e. a way to make it evaluate to TRUE whichever the truth values for the universal variables. As a special case, notice that if the outermost scope is existential, variables laying therein have a *constant* value in any certificate.

What ozziKs does is to produce a compact, BDD-based representation of such a set of functions. Once we have the set of functions, we can (1) *certify* the satisfiability of the QBF problem by checking the matrix against the strategy (ozziKs does this for us), and (2) query the certificate to obtain information on the original encoded problem (for example, if we encoded a game in the QBF and asked about the existence of a winning strategy, the certificate gives us that strategy).

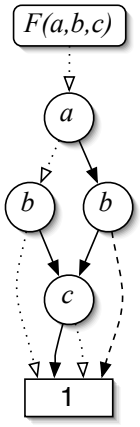
Let us consider these two aspects.

- The validity of a certificate can be verified by whoever is knowledgeable about the *evaluation apparatus* of the logic (deductive capabilities are unnecessary), independently of how it was obtained. The meaning of a successful verification is twofold: we are ensured (1) that the formula is sat, and (2) that the certificate encodes a model. Conversely, the verification fails when either the certificate is invalid or the formula is unsat (we cannot tell right away which circumstance occurred).
- A certificate exemplifies a definite scenario in which QBF-encoded problems reveal their satisfiability. For example, a sat-answer suffices to know that at least one winning strategy exists in a QBF-encoded two-player game, but it takes a certificate to exhibit an actual strategy. Let us consider the famous game "Connect-4". It is known that the player who moves first can always win. The rules of the game and the existence of a winning strategy can be encoded into a QBF instance f_4, expected to be sat. Which is the winning strategy? A certificate would disclose such information: the first player would prevail by just inspecting the certificate at each move, whatever the opponent does. Notice however that stand-alone certificates convey no self-contained semantics, as the meaning of each variable in the encoding is a piece of information held by the "encoders".

More formal definitions and properties are discussed in what follows (adpted from [3]), where the following conventions are adopted. Given a QBF F, we denote by $\widetilde{F}$ its matrix, by $var_\exists(F)$ ($var_\forall(F)$) the set of existentially (universally) quantified variables in F, and by $var_\forall(F, e) \subseteq var_\forall(F)$ the set of universal variables preceding (or *dominating*) $e \in var_\exists(F)$ in the prefix (we pose $\delta(e) \doteq |var_\forall(F, e)|$). Given a CNF matrix $\widetilde{F}$, the formula $\widetilde{F} * l$ is the CNF obtained by *assigning* the literal l, i.e. by removing from F each $\neg l$ literal and each clause containing l. This notation is readily extended to sets of literals. A matrix $\widetilde{F}$ is *satisfied* by a set of literals M (written $M \models \widetilde{F}$) when $\widetilde{F} * M$ is the empty formula.

A.1 Certificate representation

QBF models can be represented *explicitly* by employing data structures such as trees or truth tables. Or, we may pursue *compactness* at the expense of managing an *implicit* representation[14] requiring computation to yield values. An ideal certificate should be compact (easy to manage) *and* explicit (easy to verify and query). A successful tradeoff is obtained by employing *Binary Decision Diagrams* [5]. We consider their *reduced ordered* version (ROBDDs, or BDDs henceforth) with *complemented arcs*. A BDD $\mathcal{E}$ representing a total function $F(u_1, u_2, \ldots, u_n) : \mathfrak{B}^n \to \mathfrak{B}$ is a directed acyclic graph with one root (labeled by F) and one sink node (labeled "1").



Each internal node is labeled by one variable in $U = \{u_1, u_2, \ldots u_n\}$, and always has two children, one attached to the outcoming *then-arc*, the other to the *else-arc*. The *else-arc* may or may not be *complemented*. A unique path from the root to the sink is identified by assigning a value to each variable in U: The *then-arc* is chosen for variables assigned to 1, the *else-arc* is followed otherwise. The function F represented by $\mathcal{E}$ evaluates to 1 on $\langle \psi_1, \psi_2, \ldots, \psi_n \rangle \in \mathfrak{B}^n$ iff an even number of complemented arcs is encountered along the path defined by $\psi_1, \psi_2, \ldots, \psi_n$.

As an example, let us consider the BDD aside, where solid arrows denote then-arcs, while dashed (dotted) arcs are used for regular (complemented) else-arcs. It represents a binary function $F(a, b, c)$ of three binary variables a, b and c. It is, for example, $F(0, 1, 1) = 1$ and $F(1, 1, 1) = 0$. The represented function may be written as $F = b \wedge (a \vee c) \wedge (\neg a \vee \neg c)$. In a set-oriented interpretation, this BDD represents the *one-set* of F, i.e. the set having F as characteristic function. In our case, it stands for the set $\{\langle 0, 1, 1\rangle, \langle 1, 1, 0\rangle\}$ where F evaluates to 1.

The BDDs we utilize are *ordered* and *reduced*: The same variable ordering is followed along each path, and no two nodes representing the same set exist, so that each function has only one *canonic* representation. Furthermore, the version with complemented arcs is such that the set $\overline{S}$ is denoted by the same node as S (referred to with a complemented arc).

The BDD way of representing sets is regarded as *symbolic* in that it avoids the explicit enumeration of sets' elements in favor of a more abstract, diagram-based way of computing characteristic functions. Such representations may be exponentially more succinct than explicit ones (see [8]), and all the operations on the sets/functions they represent (union/disjunction, intersection/conjunction, etc.) can be performed by manipulating the involved BDDs [5]. With a small abuse of notation, we treat BDDs as if they were the sets they represent. For example, $x \in \mathcal{E}$ is an element in the subset of $\mathfrak{B}^n$ individuated by $\mathcal{E}$.

In collections of BDDs, canonicity spans over their set of nodes as a whole. This allows the sharing of structural information among diagrams. A BDD in such a set of interconnected diagrams—a *forest*—is identified by a (complemented) arc pointing to its root node.

[14] In [6] propositional formulas and QBFs with free variables are used. *Implicitness* is not an issue for the authors as they focus on characterizing classes of models/formulas.

Definition A.1 (QBF sat-certificate, validity) *A sat-certificate for a QBF F with $var_\exists(F)=\{e_1,\ldots,e_m\}$, $var_\forall(F)=\{u_1,\ldots,u_n\}$, and $\delta_i=\delta(e_i)$ is a forest of BDDs containing two roots $\langle\mathcal{E}_i^+,\mathcal{E}_i^-\rangle$ for each i in $[1,m]$. Both $\mathcal{E}_i^+$ and $\mathcal{E}_i^-$ are defined over $var_\forall(F,e_i)=\{u_1,\ldots,u_{\delta_i}\}$. The certificate*

$$\mathcal{C}(F)=[\langle\mathcal{E}_1^+,\mathcal{E}_1^-\rangle,\langle\mathcal{E}_2^+,\mathcal{E}_2^-\rangle,\ldots,\langle\mathcal{E}_m^+,\mathcal{E}_m^-\rangle]$$

is consistent *when $\forall i\in[1,m]$ it is $\mathcal{E}_i^+\cap\mathcal{E}_i^-=\emptyset$. It is* valid *for F when for any $\langle\psi_1,\ldots,\psi_n\rangle\in\mathfrak{B}^n$ the formula $\widetilde{F}_{[u_1=\psi_1,\ldots,u_n=\psi_n]}$ is satisfied by $\{e_i=s^{(i)}(\psi_1,\ldots,\psi_{\delta_i}), i\in[1,m]\}$, where the functions $s^{(i)}:\mathfrak{B}^{\delta_i}\rightarrow\mathfrak{B}$ are defined as*

$$s^{(i)}(\psi_1,\ldots,\psi_{\delta_i})=\begin{cases}1 & \textit{if } \langle\psi_1,\ldots,\psi_{\delta_i}\rangle\in\mathcal{E}_i^+\\ 0 & \textit{if } \langle\psi_1,\ldots,\psi_{\delta_i}\rangle\in\mathcal{E}_i^-\\ undef. & \textit{otherwise}\end{cases}$$

In essence, a sat-certificate is a compact but explicit representation of the dependencies that have to exist between existential (dependent) and universal (independent) variables in order to satisfy the matrix whichever the universal hypothesis.

Lemma A.1 *If $\mathcal{C}(F)$ is valid for a QBF F, then F is satisfiable. Every satisfiable QBF has at least one valid certificate.*

Proof sketch. A QBF is satisfiable iff it has at least one model, i.e. iff we find at least one tree-like structure (like the one introduced in Section 2), such that for every assignment $U=[u_1=\psi_1,\ldots,u_n=\psi_n]$ to the universal variables the set of existential literals collected along the branch individuated by U satisfies $\widetilde{F}*U$. Given a consistent certificate C for F, we insert the literal e_i into the label of the node reached following the $u_1=\psi_1,\ldots,u_{\delta_i}=\psi_{\delta_i}$ path iff $\langle\psi_1,\ldots,\psi_{\delta_i}\rangle\in\mathcal{E}_i^+$ (and, dually, $\neg e_i$ appears in the label iff $\langle\psi_1,\ldots,\psi_{\delta_i}\rangle\in\mathcal{E}_i^-$). By construction, if the certificate is valid according to the notion of validity given in Definition A.1, the tree-like structure obtained is a model. □

For example, a valid sat-certificate for the QBF we used as an example at Page 21:

$$\forall a\forall b\exists c\forall d\exists e\exists f.\ (\neg b\vee e\vee f)\wedge(a\vee c\vee f)\wedge(a\vee d\vee e)\wedge(\neg a\vee\neg b\vee\neg d\vee e)\wedge$$
$$(\neg a\vee b\vee\neg c)\wedge(\neg a\vee\neg c\vee\neg f)\wedge(a\vee\neg d\vee\neg e)\wedge(\neg a\vee d\vee\neg e)\wedge(a\vee\neg e\vee\neg f).$$

is depicted in Figure 12 (to be compared with the version automatically extracted by ozziKs, depicted in Figure 4, Page 21).

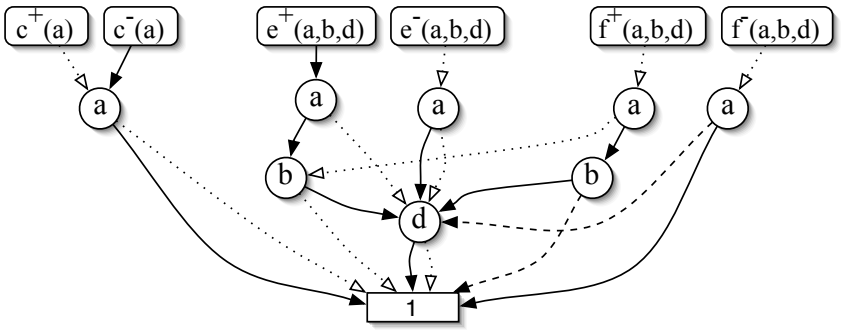


Figure 12: A BDD-based sat-certificate for the QBF at the end of Section A.1.

A.2 Certificate verification

The first thing we wish to do with a consistent certificate $\mathcal{C}$ for F is to verify its validity. We check that by choosing the truth values of the existential variables according to what the certificate suggests, we always satisfy the matrix.

An easy but impractical way of checking a certificate would be to check that $\mathcal{M}$ produces a satisfying assignment under all the possible universal hypotheses. Fortunately, the symbolic nature of the certificate helps us to perform a much more efficient, clause by clause, BDD-based verification, which is what the verification engine inside ozziKs actually does.

Let us use the exclusive or "$\otimes$" to construct literals out of variables ($\varphi \otimes v$ means v when $\varphi = 0$, and $\neg v$ when $\varphi = 1$).

Lemma A.2 *The algorithm* `checkValidity` *answers* TRUE *on* $\langle F, \mathcal{C} \rangle$ *if and only if* $\mathcal{C}$ *is a valid certificate for* F.

Let us consider, for example, the clause $\neg u_1 \vee e_1 \vee u_2 \vee \neg e_2 \vee e_3$ under the prefix $\forall u_1 \exists e_1 \forall u_2 \exists e_2 \exists e_3$. The only relevant universal hypotheses for this clause are those assigning both $u_1{=}1$ and $u_2{=}0$: All the others immediately satisfy the clause via one of its universal literals. So, it remains to verify that under the assignment $[u_1{=}1, u_2{=}0]$ at least one of the three remaining literals in the clause is true, i.e. that every universal hypothesis containing $u_1 = 1$, $u_2 = 0$ falls within the one-set of at least one out of $\mathcal{E}_1^+$ (for e_1), $\mathcal{E}_2^-$ (for $\neg e_2$), and $\mathcal{E}_3^+$ (for e_3). This is a two-step check: First, we collect the universal hypotheses $\mathcal{E} = \mathcal{E}_1^+ \cup \mathcal{E}_2^- \cup \mathcal{E}_3^+$ under which the clause is satisfied by some existential literal. Then, we check that all the hypotheses $\overline{\mathcal{E}}$ (in which no existential literal satisfies the clause) assign either $u_1 = 1$, or $u_2 = 0$, or both, so that the clause is satisfied by a universal literal.

Validity check is a coNP-complete problem [6].

Function `checkValidity`(*QBF* F*, certificate* $\mathcal{C}$)

Let $var_\exists(F)$ be $\{e_1, \ldots, e_m\}$;
Let $var_\forall(F)$ be $\{u_1, \ldots, u_n\}$;
Let $\mathcal{C}$ be $[\langle \mathcal{E}_1^+, \mathcal{E}_1^- \rangle, \langle \mathcal{E}_2^+, \mathcal{E}_2^- \rangle \ldots, \langle \mathcal{E}_m^+, \mathcal{E}_m^- \rangle]$;
forall *the clauses* $\Gamma \in \widetilde{F}$ **do**
  Let $u_{i_1}, \ldots, u_{i_h} \subseteq var_\forall(F)$ be the universal variables mentioned in Γ;
  Let $e_{j_1}, \ldots, e_{j_k} \subseteq var_\exists(F)$ be the existential variables mentioned in Γ;
  Let Γ be $\psi_1 \otimes u_{i_1} \vee \cdots \vee \psi_h \otimes u_{i_h} \vee \phi_1 \otimes e_{j_1} \vee \cdots \vee \phi_k \otimes e_{j_k}$;
  //The set of indexes/scenarios in which Γ is satisfied by some of its existential literals:
  $\mathcal{E} \leftarrow (\cup_{\phi_i=0} \mathcal{E}_{j_i}^+) \cup (\cup_{\phi_i=1} \mathcal{E}_{j_i}^-)$;
  //The set of indexes/scenarios in which Γ is not satisfied by any of its existential literals:
  $\mathcal{E}' \leftarrow \overline{\mathcal{E}}$;
  //The subset $\mathcal{E}'' \subseteq \mathcal{E}'$ of cases in which this clause exists (i.e. it is not satisfied by any universal literal)
  $\mathcal{E}'' \leftarrow \mathcal{E}'_{[u_{i_1}=\overline{\psi_1}, \ldots, u_{i_h}=\overline{\psi_h}]}$;
  //If any such case exists, the clause, hence the formula, is false.
  if $\mathcal{E}'' \neq \emptyset$ **then return** FALSE;
return TRUE;

A.3 Certificate construction

One may think it is a good idea to construct and maintain a certificate *during* the decision procedure. However:

1. Most of the information that is necessary to maintain to build a certificate is not required to continue the decision procedure;
2. The amount of information one would need to store may not fit in main memory;
3. On-the-fly certificate reconstruction may be wasting a lot of time (in constructing "sub-certificates" that later on turn out to be invalid);
4. For decision procedures not based on search (but e.g. on resolution, skolemization), an on-the-fly "forward" reconstruction procedure is completely unnatural.

These apparent drawbacks suddenly turn into advantages. They indeed suggest to *decouple* evaluation from model reconstruction, with almost no overhead for the former and a clear semantics for the latter. The two meshes of the chain are connected through an *inference log*, produced by the solver (sKizzo in our case), and subsequently read by a *model reconstructor* (ozziKs in our case). Once a sat-log (i.e. a log documenting all the inference steps taken to decide a true QBF instance) is known, the reconstructor comes into play. It trusts the solver about the log being a sat-log, and parses it *backward*, reasoning by induction on the number of entries:

Base case. At the end of the inference trace we find the empty formula $\mathcal{F}_t$, satisfied by an empty model $\mathcal{M}_t$.

Inductive case. Given a model $\mathcal{M}_i$ for $\mathcal{F}_i$, the reconstructor computes a model $\mathcal{M}_{i-1}{=}\mathcal{R}(\mathcal{M}_i, \mathbf{op}_i)$ for $\mathcal{F}_{i-1}$ by reasoning on how the instantiation $\mathbf{op}_i$ (of some underlying inference scheme) turned $\mathcal{F}_{i-1}$ into $\mathcal{F}_i$.

This leads inductively to a model $\mathcal{M}_0$ for $\mathcal{F}$, hence to a certificate for F, once the function $\mathcal{R}$ has been properly defined. The definition of this function depends on the kind of inference rules we may find in the log: Each one needs to be "inverted" in a peculiar way. ozziKs knows the inference strategies and rules employed by sKizzo, and is able to reconstruct models whatever sKizzo did to solve the instance. In particular, ozziKs is able to interpret:

- The (symbolic) assignment/substitution/resolution steps that take place while sKizzo reasons on the *symbolic skolemization* of the instance. The way to do this is described in [3].
- The (ground) assignment/resolution steps that take place on the original QBF representation.
- The compilation-to-SAT steps (followed by the invocation of some SAT solver) that sKizzo performs as soon as the ground expansion of the current subformula is affordable.
- The branching steps that take place in DPLL-like reasoning;
- Any sequential or nested combination of the above mentioned reasoning styles.

Appendix B: Content and format of input and output files

Several file types, with different contents and formats, play a role in the solution and certification of a QBF instance, according to the following table (notice that files related to the dump of expression evaluation results have already been discussed and are not reported here).

Filename extension	Content	Output of	Input for	Format
`.qdimacs`	A QBF instance F in prenex conjunctive normal form	Some manual/automatic compilation	The QBF solver (sKizzo)	QDIMACS
`.qdimacs.sKizzo.log`	An inference log recording the solving process for F	sKizzo when `-log` is specified	The inductive reconstruction engine of ozziKs	sKizzo log format v1.3
`.qdimacs.qbm`	A sat-certificate $C(F)$ (in a format amenable to be verified)	The inductive reconstruction engine of ozziKs	The verification engine of ozziKs	sKizzo/ozziKs QBM format v1.3
`.qdimacs.qbm.dot`	A sat-certificate $C(F)$ (in a format amenable to be visualized)	The inductive reconstruction engine of ozziKs	A DOT file visualizer like Graphviz	DOT
`.qdimacs.qdc`	A validity preserving assignment to the existential variables in the outermost scope (if any)	The inductive reconstruction engine of ozziKs	External procedure requiring valid outermost assignments	DIMACS 1.1

Three of these file types are specific to the sKizzo/ozziKs couple (i.e. inference logs, QBM certificates, DOT certificates). The next three subsections briefly describe them.

B.1 Inference logs

Inference log files are actually textual files whose content can be easily inspected. They are written in a language shared between the solver (sKizzo) and the reconstructor (ozziKs), of which we do not provide here a formal syntax and semantics.

Rather, we briefly comment on an example, which is a (possible) inference log for the decision process of our sample instance `s27_d2_s.qcnf`, an excerpt of which is depicted in Figure checkValidity. We observe that:

- In the preamble of each inference log some general information are provided, like
  - the name of the formula the certificate refers to,
  - the version of the inference-log language, and
  - the number and types of the variables in the instance.
- The bulk of the log is a numbered sequence of records—chronologically dumped—each one recording information about one inference step. In our example, there are 105 records. The last record for a TRUE instance is always and "END" record registering the final positive outcome of the evaluation.
- Records of different types exist. In our example:

  #1,#2 Two records reporting about a "ground" variable assignment (by UCP or PLE).

  #10 A ground variable elimination step via q-resolution (QRES).

  #19 A symbolic assignment, i.e. an assignment in the space of symbolic formulas (by SUCP, or SPLE, or SHBR). Notice how a BDD is dumped as well in this record (through the DDDMP library).

  #93 A symbolic equivalence substitution step (by SER), reporting involved variables, simplified clauses, and reference BDD.

  #103 A compilation to SAT of a sub-problem, specifying what exactly has been compiled, and which is the solution found by the SAT solver.

```
# sKizzo log file, 1.3
# created on Wed Nov  1 11:48:36 2006
# formula: "s27_d2_s.qcnf"
# dump format: text
# vars: (66,10)
# universals: 30 31 32 33 34 35 36 37 38 39
# int/ext mapping: 4 5 6 7 1 2 3 9 10 11 12
  13 14 15 16 17 32 33 34 35 37 38 39 40 41
  42 43 44 45 18 19 20 21 23 25 26 27 28 29
  60 61 62 63 64 65 66 67 68 69 70 71 72 73
  74 75 76 77 78 79 80 81 82 83 84 85

#1: G_ASSIGN [0]
-7

#2: G_ASSIGN [22]
52

                    . . .

#10: G_RES [141]
27 3 2
2 24 27
2 26 27
3 15 22 27
3 -24 -26 -27
2 -22 -27

                    . . .

#19: S_ASSIGN [180]
59
.ver DDDMP-2.0
.mode A
.varinfo 0
.nnodes 5
.nvars 11
.nsuppvars 4
.ids 6 7 9 10
.permids 1 10 9 8
.nroots 1
.rootids -5
.nodes
1 T 1 0 0
2 7 3 1 -1
3 9 2 2 -1
4 10 1 3 -1
5 6 0 4 1
.end

                    . . .

#93: S_EQUIV [197]
65 29 1
2
2 29 -65
2 -29 65
.ver DDDMP-2.0
.mode A
.varinfo 0
.nnodes 3
.nvars 11
.nsuppvars 2
.ids 3 8
.permids 1 3
.nroots 3
.rootids -3 -3 -3
.nodes
1 T 1 0 0
2 8 1 1 -1
3 3 0 1 2
.end
```

```
#103: SAT_ENCODING [180]
symbVar: 55(20) cnfVar: 546
0 0 0 1 1 0 0 0 1 0 0 1 1 1 0 1 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

                    . . .

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
var 1: (1,1)
0
 (1)
var 2: (2,1)
0
 (2)
var 3: (3,1)
0
 (3)
var 4: (4,1)
0
 (4)
var 9: (5,1)
0
 (5)
var 15: (6,1)
0
 (6)
var 16: (7,1)
0
 (7)
var 17: (8,1)

                    . . .

0
 (13)
var 24: (14,1)
0
 (14)
var 25: (15,1)
0
 (15)
var 26: (16,1)
0
 (16)
var 28: (17,1)
0
 (17)
var 29: (18,1)
0
 (18)
var 61: (19,8)
10 30 31 32 33 34 35 36 37 38 39
1 1 2 0 0 0 0 0 0 1  (19)
1 1 2 1 0 0 0 0 1 1  (21)
1 0 2 1 0 0 1 1 1 0  (23)
1 0 2 0 0 0 1 1 0 1  (25)
0 1 2 0 1 0 0 0 0 1  (27)
0 1 2 1 1 0 0 0 1 1  (29)
0 0 2 1 1 0 1 1 1 0  (31)
0 0 2 0 1 0 1 1 0 1  (33)
var 62: (35,1)
10 30 31 32 33 34 35 36 37 38 39
2 2 2 2 0 2 2 2 2 2  (35)

                    . . .

#105: END [549]
outcome: TRUE
time: 0.77
```

Figure 13: Excerpt of an inference log recording the steps taken to decide `s27_d2_s.qcnf`.

- This inference log is a tiny example:
  - It mentions only a part of the possible record entries. For example, symbolic resolution and branching DPLL-like steps are missing, as well as control records tracing rollback and checkpointing mechanisms in the solver.
  - Its size is approximately 27 KBytes. Logs for "difficult" instances from applications may contain millions of steps and may require gigabytes of disk space.

B.2 QBM certificates

QBM certificate files are textual files as well. The way information is organized within one such file will be discussed considering the content of the QBM certificate of our sample instance `s27_d2_s.qcnf`, as depicted in Figure 14:

- All certificates begin with a 6-line prefix having the following format:

  ```
  # QBM file, <VERSION>
  # solver: <SOLVER-NAME>
  # formula: <FILENAME>
  # complete: <IS-COMPLETE>
  # dump format: <FORMAT>
  # universals[<N_∀>]: <∀-LIST>
  # existentials[<N_∃>]: <∃-LIST>
  ```

  where:

  1. <VERSION> is the version of the certificate format, currently 1.3;
  2. <SOLVER-NAME> is the solver that produced the inference log from which the certificate has been obtained;
  3. <FILENAME> is the quote-delimited path of the qdimacs file containing the QBF instance this certificate refers to, relative to the position of the certificate itself;
  4. <IS-COMPLETE> is `yes` for full certificates (i.e containing the interpretation of every skolem function) or `no` for partial certificates (in which some skolem interpretations have been pruned away via the `-var` switch);
  5. <FORMAT> at present can only be `text`
  6. <$N_\forall$> is the number of universal variables in the formula and <∀-LIST> is the list of their space-separated names (i.e. numeric codes);
  7. <$N_\exists$> is the number of existential variables in the formula and <∃-LIST> is the list of their space-separated names (i.e. numeric codes).

- After the prefix, a textual representation for the forest of BDDs representing the certificate is dumped. Such dump is performed in the DDDMP-2.0 format[15]. The DDMP dump is itself made up of two parts:
  - A DDDMP prefix, characterized by lines of text starting with a dot. The last four lines of such prefix contain the most relevant piece of information. Their format is as follows:

[15] DDDMP—which stands for "Decision Diagrams DuMP"—is a library designed by Stefano Quer and Gianpiero Cabodi ("Politecnico di Torino", Italy) that works in tandem with the CUDD package (version 2.2.0 or higher). The DDDMP package defines formats for DD storage on secondary memory, and it contains a set of functions to dump DDs and DD forests on file, and to load them back. We use the latest version, 2.0.3, released August 01, 2005. For more information see `http://fmgroup.polito.it/quer/research/tool/tool.htm`

`.ids` <SUPPORT-LIST>
`.permids` <SUPPORT-LIST>
`.nroots` <N-ROOTS>
`.rootids:` <ID-LIST>

where:

1. <SUPPORT-LIST> is a space-separated list of codes for variables in the support set of the forest. For certificates in the 1.3 format, `.ids` and `.permids` are always followed by the same list, which is in fact the enumeration of integers from 1 to <$N_\forall$>.
2. <N-ROOTS> is the number of roots (i.e. the number of diagrams, i.e. the number of trees), inside the forest;
3. <ID-LIST> is a space-separated list of <N-ROOTS> (signed) root names, i.e. signed integer codes naming internal nodes in the forest (the terminal BDD node "1" always has code 1, the others are given an increasing code).

The following conventions hold:

- There are always exactly twice as much roots in the forest than existential variables in the formula, i.e. <N-ROOTS> = $2 \times$ <$N_\exists$> (cfr. in the example $110 = 55 \times 2$). This is because (see Appendix A) to each existential variable two BDDs are associated:
  - one *positive BDD* whose one-set specifies when the variable is true;
  - one *negative BDD* whose one-set specifies when the variable is false[16].
- The two lists <$\exists$-LIST> and <ID-LIST> are ordered consistently to one another. This means that the two roots of the BDDs related to the existential variable at position i in the list <$\exists$-LIST> are given in <ID-LIST> at position $2i$ (positive BDD) and $2i + 1$ (negative BDD). For example, variable 84 is related to the BDDs in the forest rooted at -8 and 8.
- The two lists <$\forall$-LIST> and <SUPPORT-LIST> always have the same size, and are consistently ordered. This means that the variable in the support set of the forest mentioned with code i (in the list of nodes decribed next) is related to the universal variable at position i in <$\forall$-LIST>.

– The list of nodes in the forest, enclosed between a heading `.nodes` line and a tailing `.end` line. Each line specifies a node, using the following format

<NODE-ID> <SUP-VAR-CODE> <VAR-PERM> <THEN-NODE> <ELSE-NODE>

where:

1. <NODE-ID> is an unique integer code for each node in the forest;
2. <SUP-VAR-CODE> is the code of the decision variable in the support set the current node refers to, i.e.—indirectly via <$\forall$-LIST>—is the universal variable related to the current node;
3. <VAR-PERM> is not relevant here
4. <THEN-NODE> is the (signed) integer code of the node pointed by the then-arc going out from the current node in the forest;
5. <ELSE-NODE> is the (signed) integer code of the node pointed by the else-arc going out from the current node in the forest.

[16]For no variable the intersection of such two BDDs can be non-empty. It is possible, however, that the union of the BDDs is not the full set. This happens for DONT-CARE conditions.

```
# QBM file, 1.3
# solver: sKizzo-v0.8.2
# formula: "./s27_d2_s.qcnf"
# complete: YES
# dump format: text
# universals[10]: 18 19 20 21 23 25 26 27 28 29
# existentials[55]: 4 5 6 7 1 2 3 9 10 11 12 13 14 15 16 17 32 33 34 35 37
                    38 39 40 41 42 43 44 45 60 61 62 63 64 65 66 67 68 69
                    70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
.ver DDDMP-2.0
.mode A
.varinfo 0
.nnodes 25
.nvars 11
.nsuppvars 10
.ids 1 2 3 4 5 6 7 8 9 10
.permids 1 2 3 4 5 6 7 8 9 10
.nroots 110
.rootids -1 1 -1 1 -1 1 1 -1 -1 1 -1 1 -1 1 1 -1 1 -1 -1 1 1 -1 1 -1 1 -1 -1
          1 -1 1 -1 1 -1 1 1 -1 -1 1 -1 1 1 -1 1 -1 1 -1 1 -1 -1 1 1 -1 -1 1
          -1 1 1 -1 3 -3 -4 4 1 -1 -5 5 1 -1 7 -7 -6 6 1 -1 9 -9 11 -11 13
          -13 -14 14 1 -1 15 -15 17 -17 18 -18 20 -20 21 -21 22 -22 -24 24 1
          -1 1 -1 1 -1 -25 25 -8 8 1 -1
.nodes
1 T 1 0 0
2 5 4 1 -1
3 1 0 1 2
4 1 0 2 -1
5 10 9 1 -1
6 6 5 1 -1
7 5 4 1 -6
8 8 7 1 -1
9 7 6 1 -8
10 7 6 1 -1
11 6 5 10 1
12 7 6 8 1
13 6 5 1 12
14 2 1 8 -1
15 2 1 1 8
16 9 8 1 -1
17 4 3 16 1
18 6 5 16 1
19 6 5 1 -16
20 4 3 1 19
21 9 8 1 5
22 7 6 1 5
23 9 8 5 -1
24 7 6 23 -1
25 3 2 1 -1
.end
```

Figure 14: The content of the certificate file `s27_d2_s.qcnf.qbm`.

B.3 DOT certificates

The DOT certificate contains the same information as the corresponding QBM certificate, but in a format which is amenable to be parsed and rendered by an automatic graph drawing tool like *graphviz*.

Here we do not describe the content of the DOT file, but the graphical appearance of the BDD forest (constituting the certificate) after it has been rendered. We refer to Figure 2 (page 19) and Figure 3 (page 20), where a total and a partial certificate for our sample formula `s27_d2_s.qcnf` are respectively depicted (the latter is obtained by specifying to ozziKs a "`-var 71-76`" switch).

Graphic conventions are as follows:

- At the top of the figure a row is depicted containing a list of all the existential (possibly signed) variable codes. Each variable is enclosed in one box, and each box may contain more than one variable: All the variable in the same box have the same skolem interpretation;
- In the leftmost part of the figure a column is depicted listing the names of all the universal variables in some order;
- All the internal nodes in the figure are choice points. The internal nodes at the vertical level of one universal variable in the leftmost column are decision nodes associated to that universal variable;
- Internal nodes always have two outgoing arcs: one then-arc (continuos line) and one else-arc (dashed line);
- Each arc has a positive (*blue* colored) or a negative (*red* colored) sign;
- Each box in the top row has one signed (same color convention as before) continuos outgoing edge heading for some internal node.

To know the truth value of an existential variable e as a function of its dominating universal variables, we proceed as follows

1. We start from the box in the top row containing the existential variable e;
2. We move downward to reach the sink node 1 through many subsequent choice points (no more than one per universal variable). If the universal variable u is set to FALSE, we choose the else-arc at the choice point related to u, if it is set to TRUE we follow the then-arc.
3. In the end, we obtain a path linking e to 1. We count how many negative arcs we encountered along the way. We say that the path as a whole is positive if it contains an even number of negative arcs, and is negative otherwise.
4. The variable is TRUE if the path is positive, and is FALSE if it is negative.

For cases in which DONT-CARE conditions occur (this does not happen in our sample formula), some existential variable e may appear two times in the top row: once as a positive literal "e", once as a negative literal "!e".

In this case, we compute the sign of both paths, and then apply this rule:

- The $e \rightarrow 1$ path is positive and the $!e \rightarrow 1$ path is negative: e is TRUE;
- The $e \rightarrow 1$ path is negative and the $!e \rightarrow 1$ path is positive: e is FALSE;
- Both paths are negative: e is a DONT-CARE condition;
- Both paths are positive: cannot happen in consistent certificates.

References

[1] M. Benedetti. sKizzo's web site, http://sKizzo.info, 2005.

[2] 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.

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

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

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computing*, C-35(8):677–691, 1986.

[6] H. K. Büning and X. Zhao. On Models for Quantified Boolean Formulas. In *Proceedings of SAT'04*, 2004.

[7] Fabio Somenzi. Colorado University Binary Decision Diagrams, vlsi.colorado.edu/∼fabio/CUDD, 1995.

[8] Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. Monographs on Discrete Mathematics and Applications. SIAM, 2000.

Copyright and License

Copyright

Definitions

- By "SOFTWARE" we mean the software "ozziKs" 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`