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

Technical Report - TR 04-11-03 (v0.8)

sKizzo

a QBF decision procedure based on Propositional Skolemization and Symbolic Reasoning

Marco Benedetti[§]
benedetti@itc.it

November 11, 2004

[§] This work was supported by PAT (*Provincia Autonoma di Trento*, Italy), under grant n. 3248/2003.

Abstract

We introduce a novel algorithm for the evaluation of *quantified boolean formulas* (QBFs), which we call sKizzo. Our work is firstly framed in the broad field of algorithms for automated deduction. Then, we enlighten the strong application-related interest in decision procedures for QBFs.

The algorithm itself is thoroughly discussed. It integrates within a uniform framework several different ideas and techniques: classical resolution-based QBF reasoning, algorithms for structure reconstruction, propositional skolemization, BDD-based representations, symbolic reasoning, search-based decision procedures, compilation to SAT techniques, and more. A detailed account of each module in the solver is presented, together with the overall architecture explaining how they interact with one another.

We also report of our first implementation for the algorithm. It was used to experimentally evaluate our approach, yielding very interesting results. The related literature is carefully reviewed, aiming to point out the many distinguishing features of sKizzo. Finally, a large section is devoted to the presentation of our ongoing efforts and future work on the topic.

Table of Contents

1 Introduction

Many application problems (Planning, Scheduling, Formal Verification, and more) can be successfully tackled by stating them in some formal language featuring an inference apparatus (i.e., a *logic*). Thereafter, they are solved by means of an automated-reasoning tool able to manage statements in the chosen language.

This language should be expressive enough to capture the scenario of interest. For example, if one needs to predicate about time-dependent properties, he or she would better use a logic containing time-related operators. But there is another crucial issue to take into account while choosing the target logic: As far as applications are concerned, the *efficacy* of the known decision procedures is of primary importance.

Should a deduction engine for a certain logic come out to be incredibly effective with respect to the average case, it would probably become attractive for many applications. Even if the underlying logic is not expressive enough for the problem at hand, it is still possible to re-write a somehow restricted version of the problem, or to retain the whole meaning of the problem at the expense of a (possibly huge) enlargement in the size of each instance (also suffering from a remarkable obfuscation).

From a theoretical point of view, properties of most commonly used logics are well known. For example, first-order logic (FOL) is known to be semidecidable, while propositional logic ($PROP$) is a simpler decidable logic in which the satisfiability problem is NP-complete; quantified propositional logic (QBF) and linear propositional temporal logics ($PLTL$) are PSPACE, and so on. However, what does really matter to applications is the average case, i.e. the capability of a reasoning engine to effectively solve those problems that arise in practice. As we move from a more expressive to a less expressive formalism, we may improve the worst-case complexity, but we also loose the expressive power of certain syntactic operators that not only provide a more natural way to state relevant facts or rules, but could also be effectively exploited during the solving process, at least in principle. In general, the actual balance between these pros and cons is unclear.

As a matter of fact, the most effective solving tools for a large class of industrial-scale problems [21] (such as computer-aided design of integrated circuits [39, 42], Planning [38], Model Checking for dynamic systems [10], Scheduling [23], Operations Research, and Cryptography [50], to name a few) are *SAT solvers*, which are reasoning engines designed to decide the existence of models for $PROP$ instances.

One step ahead of propositional logic, we encounter the more expressive *quantified propositional logic*, which adds the valuable possibility to *quantify* over the truth value of variables. Most of the industrial-made problems reported above have a more natural QBF formulation, which is—in addition—possibly exponentially more succinct than the propositional one. Compelling questions arise: *Are QBF solving tools worthy of this inheritance*? *Do they add any value to the reasoning capabilities of purely propositional solvers*?

The answer is: *no*. Or, at least, *not yet*. QBF logic is a promising formalism still in need for substantial improvements as to satisfiability procedures. A lot of research efforts

are currently focusing on designing new solving paradigms to capture the added value of quantified reasoning. In this work, we contribute to such research with several new ideas, and with a novel decision procedure, called sKizzo[1].
From an historical perspective, the approach we propose acts like a glue that joins together techniques developed over decades in the framework of automated reasoning. The first and most important component of our construction traces back to the Twenties (the Skolem theorem [71, 15]). Following the timeline, we capitalize on some seminal contributions to automated theorem proving from the early Sixties (DPLL algorithms [24, 25]). Then, a compact formalism from the Eighties to reason about boolean functions [12, 76] is employed. The Nineties gave us key contributions towards effective quantified reasoning [16, 40]. In the same years, successful techniques to compile real-world problems into SAT instances were proposed [38, 42, 10]. We adapt such techniques to our case. Finally, symbolic representations for propositional problems gained attention in the last few years [17, 53, 57], and are largely exercised here.

Our approach exploits all these techniques within a coherent framework, by *symbolically reasoning* on the *compact representation* of the *propositional expansion* of the *skolemized problem*, resorting to ground, *SAT-based propositional reasoning* whenever it pays back. Over and above building on top of existing contributions—and purposely to leverage all of them at once—our work essentially introduces a new way of looking at quantified boolean reasoning (see Section 5.2).

The rest of the paper is organized as follows. Section 2 introduces QBFs, presents the foundamental decision strategies employed in QBF solvers, and gives a first overview of our algorithm. A more detailed description of the solver is presented in Section 3, together with exemplifications over small formulas. Section 4 is concerned with implementation and experimentation. The current version of the solver is presented, benchmark suites used for the evaluation are discussed, statistical results over real-world instances and a preliminary performance evaluation are reported. The related literature is discussed in Section 5, where sKizzo is also compared with other existing solvers. Future research directions are presented. Section 6 closes the paper with a few concluding remarks.

2 Overview

2.1 Quantified Boolean Formulas

We consider quantified boolean formulas in *prenex conjunctive normal form*, such us, for example, this one:

$$\exists a \forall b \exists c \, (a \vee b \vee \neg c) \wedge (b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \qquad (1)$$

which is comprised of the *prefix* "$\exists a \forall b \exists c$" followed by the *matrix* "$(a \vee b \vee \neg c) \wedge (b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$" which is a *conjunctive normal form* (CNF)

[1] Arbitrarily shortened form of *skolemizzo*, which is an Italian word meaning "I apply skolemization".

formula, i.e. a propositional formula made up by conjuncting clauses, each clause being a disjunction of literals (a variable or a negated variable). More in general, we consider formulas in the form

$$\mathcal{Q}_1 V_1 \mathcal{Q}_2 V_2 \ldots \mathcal{Q}_n V_n \; \mathcal{F}$$

where the matrix $\mathcal{F}$ is a CNF propositional formula on variables $var(\mathcal{F})$, and the prefix $\mathcal{Q}_1 V_1 \mathcal{Q}_2 V_2 \ldots \mathcal{Q}_n V_n$ is such that $\mathcal{Q}_i \in \{\forall, \exists\}, i = 1, \ldots, n$ and $\mathcal{Q}_i \neq \mathcal{Q}_{i+1}, i = 1, \ldots, n-1$, while $\{V_i\}$ is a partition of $var(\mathcal{F})$ (i.e.: $\cup_{i=1}^{n} V_i = var(\mathcal{F})$ and $V_i \cap V_j = \emptyset$ for $i \neq j$). We suppose that each V_i is non-empty.

Each V_i is called *scope*. A scope V_i is *existential* (*universal*) if $\mathcal{Q}_i = \exists$ ($\mathcal{Q}_i = \forall$). The scope $\sigma(v)$ of a variable is the index i such that $v \in V_i$. The scope $\sigma(\Gamma)$ of a clause Γ is the maximal scope of its variables. Variables $v \in V_i$ are said to be existentially (universally) quantified if $\mathcal{Q}_i = \exists$ ($\mathcal{Q}_i = \forall$). The set of existentially (universally) quantified variables in f is denoted by $var_\exists(f)$ ($var_\forall(f)$, respectively).

2.2 Solving QBFs

To *solve* (equivalently: to *evaluate*, or to *decide*) a QBF amounts to determine its truth value, according to a semantics which we give intuitively for the sample formula (1). That formula is solved by answering *true* or *false* to the question: "Does a truth value for a exist such that for both possible truth values for b a truth value for c exists such that the matrix $(a \vee b \vee \neg c)(b \vee c)(a \vee \neg b \vee c)(\neg a \vee \neg b \vee \neg c)$ evaluates to *true*?" (the notion of evaluation for the matrix is the standard one for propositional logic).

A formula is said to be *satisfiable* if the answer to the above question is "yes", *unsatisfiable* otherwise. Each satisfiable formula has at least one *model*, i.e. a way of deciding the truth value of every existential variable as a function of all the universal variables with lower scope, in such a way that the matrix evaluates to *true* whichever the values of those universal variables. Thus, a model is a collection of boolean functions, one for each existential variable, meant to represent the way existential variables have to depend on the preceding universal variables (w.r.t. the order in the prefix) to guarantee an always satisfied matrix. Such a set of functions has a natural, tree-shaped representation, which is represented in the left side of Figure 1 for the sample case (1). Nodes labeled by universal variables have two child subtrees, one for each truth value, and a assignment satisfying the matrix is encountered along each path to a leaf.

QBF solvers are algorithms designed to tell formulas having at least one model from those having none. For example, a QBF solver can be engaged to decide that (1) is indeed satisfiable, whereas the statement

$$\exists a \forall b \exists c \; (a \vee b \vee c) \wedge (b \vee \neg c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b) \tag{2}$$

has no model (it is *unsatisfiable*).

Unfortunately, QBFs arising from the applications mentioned in the introduction are not as small as shown in these examples. They may indeed contain tens of quantifier alternations, thousands of variables, and millions of clauses.

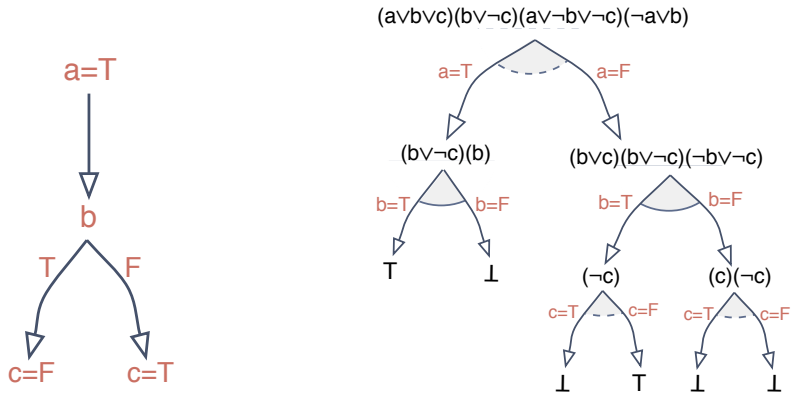

Fig. 1. On the left: A model for the satisfiable instance (1); On the right: The semantic evaluation tree of the unsatisfiable instance (2)

2.3 Classical Decision Strategies

Two classical strategies exist to decide QBF formulas.

Search. The prefix is dealt with in a left-to-right way. The order of the variables in the prefix is respected along each branch of the *semantic evaluation tree* for the formula, which is the and/or tree depicted in the right side of Figure 1 for the sample case of (2). The root is labeled by the original matrix, while the formula attached at one node is obtained from the formula labeling its father node by assigning one propositional variable. Leaves are labeled by either the empty formula $\top$ (meaning that the assignment from the root to the current leaf satisfy the original matrix) or the empty clause $\bot$ (the assignment contradicts the formula). According to the semantics of quantifiers, an existential variable generates an *or* node that disjunctively split each branch, while universal quantifiers are associated to *and* nodes that split branches conjunctively. A model, if one exists, is a subtree with all the leaves labeled by $\top$, extracted by choosing only one child for each existential node, and both children for conjunctive nodes. A search-based solver visits the evaluation tree to determine whether such a subtree does exist. As no model can be extracted out of the tree in the right side of Figure 1, the formula (2) is unsatisfiable.

Solve. Rather than search for a model, it is possible to *solve* the formula by applying a *refutationally complete procedure*. Such strategy aims to derive necessary consequences from the given formula, ending up with the empty clause if and only if the original formula is unsatisfiable. These methods build upon generalizations of the resolution approach for standard satisfiability, such as *q-resolution* [40, 13]. There are several possible complete strategies for applying resolution. For example, we can focus on *eliminating quantifiers* in a right-to-left order (w.r.t. the order in the prefix). We get rid of existential quantifiers by q-resolution, and *expand* universal

quantifiers to the two cases they represent. In the sample case of the unsatisfiable instance (2), we would start by resolving each clause containing c against each clause containing $\neg c$, thus obtaining $\exists a \forall b (a \vee b) \wedge (\neg a \vee b)$ (where c vanished). The universal quantifier b can be eliminated by constructing the conjunction of a copy $(a' \vee b') \wedge (\neg a' \vee b')$ of the matrix where b' has to be assigned to *true* with a different copy $(a'' \vee b'') \wedge (\neg a'' \vee b'')$ where b'' is assigned to *false*. We obtain $\exists a''(a'') \wedge (\neg a'')$, which by resolution finally yields the empty clause.

QBF solvers employ one of these two strategies. However, the underlying strategy represents only one out of several ingredients in competitive solvers, possibly not the prominent one. Some of the many enhancements that have to be introduced to construct an effective solver are discussed in Section 5.

2.4 sKizzo at a glance

sKizzo does not comfortably fit into either of the classical strategies we have just reviewed (even if both *search*-related and *solving*-related elements are present in our solver, as discussed in Section 5.2).

The key idea is to *restate* the original instance as a purely *existential* problem (SAT) by moving it to a different boolean space, where variables represent higher-level concepts. This new space contains decision problems over the existence of a consistent set of boolean functions representing models for the originating QBF instance (according to what we have seen in Section 2.2).

For example, to solve the QBF instance (1) we restate the problem as follows: "*Do a constant boolean function s^a and a unary boolean function $s^c(b)$ exist that compute the truth values to be assigned to a and c, respectively, in such a way that the matrix always evaluates to* true *whichever the truth value chosen for b?*".

Such translation from one problem space into another is almost identical to a well-known technique, called *skolemization*. It is indeed fair to say that skolemization is at the very heart of sKizzo. Unfortunately, it is not enough to designate skolemization as a core technique to profitably deal with QBFs. A lot of striking and inescapable issues do arise in practice. Most of the algorithms sKizzo employes have been designed to get rid of these complications. For example: (a) the prefix of a QBF formula does not always closely reflect the dependencies between existential and universal variables; (b) the result of a classical skolemization is neither a purely propositional formula nor even a QBF formula; (c) if we cast a skolemized instance into a purely propositional shape, we obtain such a huge instance that not only a direct solution is impractical, but an explicit representation is unfeasible; (d) if we adopt a compact or symbolic representation, we have to re-design all the classical inference rules to work on symbolic objects, as we cannot afford raw groundization nor even for intermediate results; (e) complete decision procedures may be inefficient in the symbolic framework, so we may be forced—sooner or later—to resort to a search-based complete algorithm; (f) the combinatorial core coming out of all the preceding steps is best dealt with by a state-of-the-art SAT solver, which we have to properly involve in the solving process.

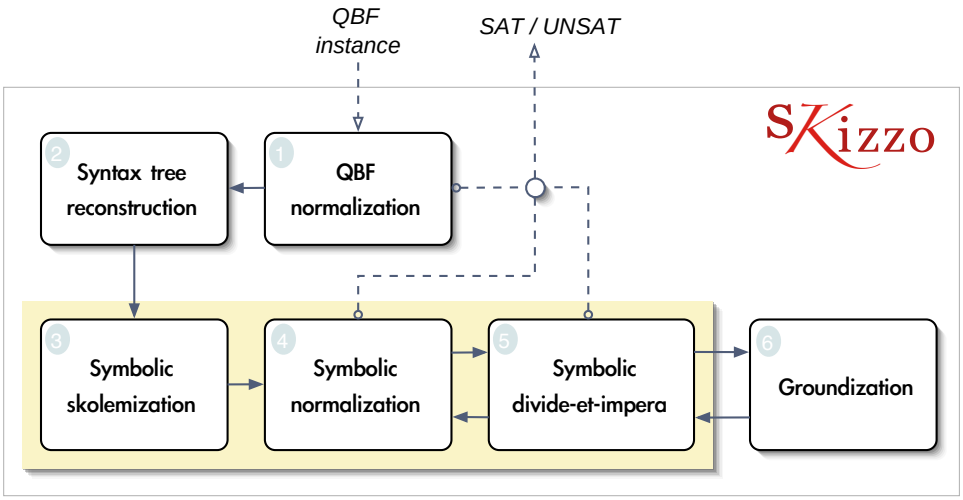

Fig. 2. High level view of sKizzo's internals

sKizzo has been designed to solve all the above problems. Figure 2 represents a simplified waterfall model for the algorithm, which is made up of six steps. The data and control flow is almost one-way, as each module takes its input form the previous step and produces an output used by subsequent steps. The major exception is Step 5, which may resort back to Step 4. The architectural model we adopt effectively decomposes relevant atomic functionalities and clearly isolates all the key ideas. It is adopted to simplify the exposition, thought it does not closely reflect the implementation presented in Section 4. The algorithm works as follows.

Step 1. Start by *normalizing the input formula*. This is just a pre-processing step which is performed working on the original QBF representation. It consists in applying simple inference rules (such us unit clause propagation and pure literal elimination) up to the fixpoint. It aims to (a) anticipate the simple part of the work—that could anyway be performed later on as the inference power of Step 1 is subsumed by the following steps, and (2) put the formula in a normal form in which each clause has an existential scope as required by subsequent steps. See Section 3.1.

Step 2. *Extract a tree-shaped syntactic structure* out of the flat conjunctive normal form coming from Step 1. The prefix of our QBF formula specifies a linearly shaped syntactic tree. This tree is responsible for conveying a relevant part of the semantics of a formula, namely, it says which existential variables are in the scope of which universal variables. Semantically equivalent trees may exists that are no longer linear, thus more closely reflecting the intrinsic dependencies in the matrix. See Section 3.2.

Step 3. Produce a *symbolic representation* of the *propositional version* of the *skolem translation* of the structured formula coming out of Step 2. Things seem complicated, but they are not. In essence, we apply the Skolem theorem to obtain a purely

universal but satisfiability equivalent formula. By doing so, we introduce *function symbols* not belonging to the toolkit of QBF logic. To eliminate such functions, we expand their propositional meaning over each point of their definition domains. As a side effect, our purely universal instance becomes a purely existential one, i.e. a SAT instance. Unfortunately, this instance is possibly exponentially larger than the original QBF formula. In practice, it is intractable, unless some kind of compact representation is employed. That is exactly what we do, by means of a two-level symbolic representation. See Section 3.3.

Step 4. Try to solve (or, at least, to strongly simplify) the symbolically-represented, purely propositional instance coming out of Step 3. To perform this work, apply up to the fixpoint a set of inference rules that manage symbolic objects to produce symbolic inferences, thus never expanding the formula to its ground meaning. See Section 3.4.

Step 5. If Step 4 is unable to solve the problem (as it enrolls a refutationally incomplete inference apparatus), start dividing the problem into smaller and simpler pieces, until a sufficiently small sub-problem is obtained that either (a) can be directly solved by employing the symbolic rules given in Step 4, or (2) is affordable by a SAT solver once extended to its full, *ground meaning*. See Section 3.5

Step 6. Should Step 5 decide to face the ground version of some sub-problem, we would have to produce the necessary flat propositional space, generate all the necessary clauses, give the resulting instance to the SAT solver, wait for its answer, and give the result back to Step 5. We do this in Step 6. See Section 3.6.

Some steps (namely, Step 1, 4 and 5) are able to decide specific classes of formulas. Others just perform a kind of "pre-processing" aiming at modifying the representation of the problem or reducing its size. They never decides anything.

2.5 Notation

We denote clauses by uppercase greek letters, and represent them either as explicit conjunctions $\Gamma = l_1 \vee l_2 \vee \cdots \vee l_n$, or as sets of literals $\Gamma = \{l_1, \ldots, l_n\}$. Given a clause $\Gamma = \{l_1, \ldots, l_n\}$, we denote by $\Gamma * l$ the result of *applying the assignment* l to that clause (l being a positive or negative literal). The result is that (1) the clause stays the same if $var(l)$ doesn't appear in $var(\Gamma)$, (2) the clause *disappears* (is subsumed) if $l \in \Gamma$, and (3) the clause *resolves* to $\Gamma \setminus \{\neg l\}$ if $\neg l \in \Gamma$. This notion is readily extended to sets of clauses and sets of literals, in so as $f * \Delta$ is the formula resulting after applying the (partial) assignment Δ to each clause in f. The total ordering $V_1 < V_2 < \cdots < V_n$ among scopes induces a partial ordering $\prec$ among variables, in which each V_i is an unordered subset of variables, and $v \prec w$ whenever $v \in V_i$ and $w \in V_j$ for some $i < j$. Given a subset S of the variables and the partial ordering induced by the prefix, we define $Sup(S) = \{v \in S | \nexists v' \in S . v \prec v'\}$. The Sup function is extended to clauses, in so as $Sup(\Gamma) = \{v \in var(\Gamma) | \nexists v' \in var(\Gamma) . v \prec v'\}$.

We say that a variable v *dominates* another variable w iff $v \prec w$ (we also say that w is *deeper* than v). The universal depth $\delta(v)$ of an existential variable $v \in V_i$ is the number of dominating universal variables for v: $\delta(v) = |\{v | v \in V_j, j < i, \mathcal{Q}_j = \forall\}|$.

We denote the *exclusive or* operator by "$\otimes$". We use such operator to construct literals out of variables, when the polarity of the literal depends on some binary parameter. For example, $a \otimes v$ means v when $a = 0$, and $\neg v$ when $a = 1$.

3 The algorithm

In this section we describe each step of our decision procedure in turn, according to Figure 2.

3.1 Step 1: QBF Normalization

The aim of this step is to normalize and simplify the input formula.

Step 1 takes as input the original instance, and produces either a (possibly) simplified version of that formula, or—for "simple" instances—a SAT/UNSAT decision.

A set of (easy-to-implement but incomplete) inference rules for QBF is utilized. Each rule is repeatedly applyed, until its deductive closure is computed. Deductive closure is expanded for each rule in a round-robin way, until fixpoint. The formula is then said to be *normalized* with respect to the set of rules employed. For classes of formulas that are decided during this step, a SAT/UNSAT outcome is obtained and the algorithm terminates.

A candidate set of rules is the following.

PLE (Pure Literal Elimination) selects literals that only appears positively (negatively); if a pure literal has existential scope, it is safe to assign it to true (the standard propositional case: some clauses are satisfied, none is resolved, and we are free to select either truth value). Conversely, this choice is too optimistic for universal pure literals. The clauses in which an universal pure literal appears should be satisfiable even if the literal is false. So, PLE chooses the worst case.

UCP (Unit Clause Propagation) is a powerful rule that only considers clauses $\{\gamma\}$ made up of one single literal. If the scope of the clause is universal, the formula is immediately unsatisfiable (γ has to be true, and, at the same time, the formula should hold for both truth values). If the scope is existential, the literal must be assigned (and propagated) to avoid an immediate contradiction, as in the standard propositional case.

FAR (ForAll Reduction) is an equivalence preserving rule specific to QBF. It allows to remove the deepest literal from each clause Γ with universal scope ($\mathcal{Q}_{\delta(\Gamma)} = \forall$). If the resulting clause still has universal scope, the rule is applied again. If the empty clause is derived, the formula is unsatisfiable. Otherwise, the resulting clause has existential scope. The intuition behind FAR is that for clauses in which the deepest literal is universal the "worst-case" assignment is always the one that resolves against that literal, so we cannot confide in that literal to satisfy the clause. We simply remove it.

Only the FAR rule is *essential* to compute the normal form required by subsequent steps. When all the rules reach their fixpoint (and neither a contradiction is detected, nor the instance is satisfied), the formula is passed to the subsequent steps. In particular, each formula is supposed—from now on—to be exclusively made up of clauses with existential scope.

3.2 Step 2: Syntax-tree Reconstruction

The aim of this step is to partly reconstruct the lost/hidden syntactic structure of the formula, thus producing information that greatly helps subsequent steps.

Step 2 takes as input the prenex QBF formula f produced by Step 1, and constructs a *quantifier tree* for f. A quantifier tree $tree(f)$ for f is a tree-shaped structure with the following properties:

1. The root node is labelled with an "and" connective, and may have any number of children.
2. The internal nodes have free degree and are labeled with a (universally or existentially) quantified variable in f. Each variable in f appears somewhere in the internal nodes of $tree(f)$. Existentially quantified variables appears in $tree(f)$ *exactely once*. Conversely, any two internal nodes n and n' can be labelled with the *same universal variable*, provided they do not lay on the same branch.
3. Each leaf node n is labelled with a non-empty list of clauses; the set of variables in such clauses is always a subset of the variables encountered along the path from the root to n; every clause in f appers somewhere among the leaves, but no clause is reported twice or more.
4. If a variable v appears in the prefix of f before a variable v', and v and v' have different quantifiers, then along every branch of $tree(f)$ that contains both v and v', v' is a successor of v (i.e. quantifier alternation is preserved).

In general, different quantifier trees exist for the same f, the simplest one being a linear tree made up of one single branch linearly replicating the sequence of variables in the prefix of f. However, more structured trees also exist in general. To our purpose, the smaller is the average universal depth for existential variables, the better the reconstructed tree. The best trees are those minimizing the universal depth of *all* the existential variables[2]. We employ Algorithm 1 to construct our quantifier tree.

It is straightforward to interpret a quantifier tree t as the syntactic tree of a non-prenex quantified boolean formula $qbf(t)$ inductively defined as:

$$qbf(n) = \begin{cases} \Gamma_1 \wedge ... \wedge \Gamma_n, & \text{for leaf nodes labeled by } \{\Gamma_1, ..., \Gamma_n\} \\ qbf(c_1) \wedge ... \wedge qbf(c_n), & \text{for the root} \\ Qv.\ (qbf(c_1) \wedge ... \wedge qbf(c_n)), & \text{for nodes with } label(n) = v, \mathcal{Q}_{\sigma(v)} = Q \end{cases}$$

[2] We call such trees *minimal quantifier trees*. Their existence, uniqueness, and the complexity of their construction will be investigated elsewhere.

input : A prenex QBF formula f
output: A quantifier tree for f

// First, we create the root;
$r \leftarrow$ the root node for the tree;
$label(r) \leftarrow$ " $\wedge$ ";

// Then, we create the leaves together with their lists of attached clauses;
$activeNodes \leftarrow \emptyset$;
foreach $v \in var_{\exists}(f)$ **do**
  $n \leftarrow$ new node;
  $label(n) \leftarrow v$;
  $clauses(n) \leftarrow \{\Gamma \in f | v \in Sup(\Gamma)\}$;
  $depends(n) \leftarrow \emptyset$;
  foreach $\Gamma \in clauses(n)$ **do**
    foreach $\gamma \in \Gamma$ **do**
      $depends(n) \leftarrow depends(n) \cup var(\gamma)$;
    end
  end
  $depends(n) \leftarrow depends(n) \setminus \{v\}$;
  $f \leftarrow f \setminus clauses(n)$;
  $activeNodes \leftarrow activeNodes \cup \{n\}$;
end

// Finally, the rest of the tree in a bottom-up way;
while $activeNodes \neq \emptyset$ **do**
  $n \leftarrow$ pick one variable from $Sup(activeNodes)$;
  if $depends(n) = \emptyset$ **then**
    $father \leftarrow r$;
  else
    $v \leftarrow$ pick one from $Sup(depends(n))$;
    if $isUniversal(v)$ **then**
      $father \leftarrow$ new node;
      $label(father) \leftarrow v$;
      $activeNodes \leftarrow activeNodes \cup \{father\}$;
      $depends(father) \leftarrow depends(n) \setminus \{label(n)\}$;
    else
      $father \leftarrow$ the node n with $label(n) = v$;
      $depends(father) \leftarrow depends(father) \cup depends(n) \setminus \{label(n)\}$;
    end
  end
  $father(n) \leftarrow father$;
  $activeNodes \leftarrow activeNodes \setminus \{n\}$;
end

Algorithm 1: An algorithm to construct a quantifier tree for a QBF formula

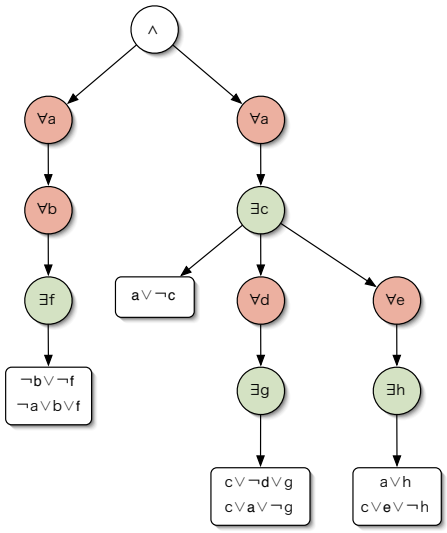

Fig. 3. A minimal quantifier tree for (3)

where $c_1, \ldots, c_n$ are the children of n.

It is possible to prove that for each quantifier tree, the key property $f \equiv qbf(tree(f))$ holds. The rest of the algorithm thus safely works on $t = \mathrm{tree}(f)$ rather than on f, experiencing two classes of benefits (see subsequent paragraphs for details):

- The reduced universal depth of the existential variables will allow to produce simpler instances during Step 3, faster computations during Step 4, and to directly address a larger class of problems in Step 6.
- The duplication of universal connectives will allow Step 5 to effectively split the main problem into unrelated subproblems.

As an example, let us consider the following quantified boolean formula and its minimal quantifier tree depicted in Figure 3.

$$\begin{aligned} \forall a \forall b \exists c \forall d \forall e \exists f \exists g \exists h.\ & (a \vee \neg c) \wedge (\neg a \vee b \vee f) \wedge (\neg b \vee \neg f) \wedge (a \vee h) \wedge \\ & \wedge (c \vee e \vee \neg h) \wedge (c \vee \neg d \vee g) \wedge (a \vee c \vee \neg g) \end{aligned} \tag{3}$$

It is interesting to notice that:

- The formula $qbf(t)$ represented by this quantifier tree is logically equivalent to the original formula. Namely:

$$\begin{aligned} qbf(t) = \forall a \forall b \exists f.\ & ((\neg a \vee b \vee f) \wedge (\neg b \vee \neg f)) \wedge \\ \forall a \exists c.\ & (a \vee \neg c) \wedge \\ & (\forall d \exists g.\ (c \vee \neg d \vee g) \wedge (a \vee c \vee \neg g)) \wedge \\ & (\forall e \exists h.\ (a \vee h) \wedge (c \vee e \vee \neg h)) \end{aligned}$$

– The universal depth of the existential variables in the prenex form is 2 for c and 4 for f, g and h. In the quantifier tree, universal depth is reduced to 1 for c, and to 2 for f, g, h;
– There is one replicated universal variable immediately below the root (a). Quantifier trees exploit the distributive property of universal quantifiers over conjunctions: $\forall a.\ (f(a,\ldots) \wedge g(a,\ldots)) \equiv (\forall a.\ f(a,\ldots)) \wedge (\forall a.\ g(a,\ldots))$. In this simple example, the transformation allows to consider the two child sub-problems of the root as completely independent instances (thought they do actually share some variables).

The notion of ordering among quantifiers, dominating quantifiers and universal depth is adapted to the tree-shaped prefix. By construction, the partial ordering induced by the prefix of a formula f is always a *restriction* of the ordering $\prec_T$ defined by its quantifier tree $tree(f)$. In particular, the partial ordering relation between variables (quantifiers) is defined in such a way that $v \prec_T w$ when the node labeled by w lays in the subtree rooted at v. The relation between "dominated" and "dominating" changes accordingly. The dominating quantifiers for v (and for a clause Γ) are the ones encountered along the path from the root to n, i.e. the set $d_{ing}(v) = \{d | d \prec_T v\}$ (and $d_{ing}(\Gamma) = \{d | \exists v \in var(\Gamma) | d \prec_T v\}$). The dominated quantifiers are those in the subtree rooted at v, i.e. the set $d_{ed}(v) = \{d | v \prec_T d\}$. The universal depth $\delta(e)$ of an existential variable e is the number $|d_{ing}(e) \cap var_\forall(f)|$ of dominating universal quantifiers.

3.3 Step 3: Symbolic Skolemization

The aim of this step is to translate the problem from QBF to a symbolic representation of a $PROP$ instance.

The Skolem theorem is employed to translate the tree-shaped representation $t = tree(f)$ produced by Step 2 into a compact, *symbolic* representation of a purely existential instance which is equivalent to f as to satisfiability.

There are three main ingredients here: (1) the *Skolem theorem* and its use in the present context, (2) the way symbolic representations for the problems are introduced and managed, and (3) the role of the stree-shaped structure produced during Step 2.

3.3.1 Propositional Skolemization. In the framework of *First Order Logic* (and other logics as well), the Skolem theorem is employed to resort to a purely existential (purely universal) formula while retaining satisfiability equivalence. This transformation is especially useful to automate deduction, and this is indeed the reason why we utilize it now.

The Skolem theorem—as applyed to FOL formulas—introduces *Skolem functions* and *Skolem constants* that have no direct representation in $PROP$. Even if no syntactic tool exists to directly represent such functions, they are not beyond the expressive power of $PROP$, at the expense of an exponential blowup in the size of the instance. We adopt a *propositional skolemization* in three steps:

1. translation of the QBF instance f into an equivalent FOL instance $FOL(f)$.

2. application of the Skolem theorem to $FOL(f)$ to obtain a (satisfiability preserving) FOL instance $Sk(FOL(f))$ with no universal quantifier.
3. compilation of $Sk(FOL(f))$ into an equivalent SAT instance $Prop(Sk(FOL(f)))$.

The first step (translation to FOL) is just syntactic sugar, but it allows to plainly capture the intuition of a purely propositional skolemization. Skolem funtions leverage the existence of two semantics levels in FOL, namely the level of *predicates* and the level of *terms*. Skolem functions are terms that are substituted for other terms (the existential variables) as arguments of predicates. QBF and $PROP$ lack both the syntactic tools and the interpretation mechanism necessary to cope with those two levels. They just feature the predicate level, though this is slightly obfuscated by their variable-oriented syntax.

To uncover such level, we introduce a FOL unary predicate $b/1$ defined over the boolean space $\{0, 1\}$, and interpreted as $b(0) = FALSE$, $b(1) = TRUE$, and restrict the domain of interpretation of every variable to be the boolean space as well. This immediately allows us to rewrite a QBF formula as a syntactically correct and logically equivalent FOL formula. For example, we rewrite the QBF formula

$$\forall x \exists y \forall z \exists k.\ (x \vee y) \wedge (\neg x \vee \neg y \vee z) \wedge (y \vee \neg z \vee k) \wedge (\neg k \vee z)$$

as a FOL formula

$$\forall x \exists y \forall z \exists k.\ (b(x) \vee b(y)) \wedge (\neg b(x) \vee \neg b(y) \vee b(z)) \wedge (b(y) \vee \neg b(z) \vee b(k)) \wedge (\neg b(k) \vee b(z))$$

In the second step (skolemization), we eliminate existential variables by substituting to each existential variable v a different Skolem function s^v, depending on the proper subset of dominating universal quantifiers. We obtain a satisfiability-equivalent (not logically equivalent) purely universal formula.

$$\begin{aligned} \forall x \forall z.\ & (b(x) \vee b(s^y(x))) \ \wedge\ (\neg b(x) \vee \neg b(s^y(x)) \vee b(z)) \wedge \\ & \wedge\, (b(s^y(x)) \vee \neg b(z) \vee b(s^k(x, z))) \ \wedge\ (\neg b(s^k(x, z)) \vee b(z)) \end{aligned}$$

It is interesting to notice that from a FOL point of view, existential quantifiers are simply disappeared, and that the dute we pay for this simplification is the loss of logical equivalence. From a higher-level point of view, we can predicate over the interpretation of terms and explicitly state what the Skolem theorem implicitly says when it reduces the satisfiability of $FOL(f)$ to the satisfiability of $Sk(FOL(f))$, i.e. that each *inner* existential FOL quantification over v has been substituted by an *outer* higher-order existential quantification over s^v (over the existence of a proper interpretation for the Skolem terms we have introduced). Informally:

$$\forall x \exists y \forall z \exists k.\ f(x, y, z, k) \quad \Longleftrightarrow \quad [\exists s^y \exists s^k] \forall x \forall z.\ f(x, s^y(x), z, s^k(x, z))$$

In the third step (translation to $PROP$), the actual work is done. It amounts to *flatten* the two semantics levels above onto one single propositional level. This transformation is made easy by the constructive property that for every formula $Sk(FOL(f))$ (where $f \in QBF$) both the predicate-level interpretation and the term-level interpretation map

boolean spaces onto boolean values. We may join their definition spaces and interpretation functions, and give an inductive translation procedure from $Sk(FOL(f))$ to $PROP$.

The only non-trivial piece of work consists of building a CNF propositional representation for every Skolem function. As a constructive consequence of steps 1-2, every Skolem function $s(a_1, a_2, \ldots, a_n)$ we manage is a relations over $\{0,1\}^{n+1}$ that maps $\{0,1\}^n$ onto $\{0,1\}$. Each one is completely specified by 2^n boolean parameters giving the truth value of the function on each point of its domain, so 2^{2^n} different Skolem n-ary functions exist. Let us denote by s_A the boolean parameter that represents the truth value of a boolean n-ary function s evaluated in A, where each single point A in the definition domain of s is conveniently represented by a string of n bits $\{A_1, A_2, \ldots, A_n\}$. We directly obtain a CNF propositional encode for s as follows:

$$Prop(s(a_1, a_2, \ldots, a_n)) = \bigwedge_{A \in \{0,1\}^n} s_A \vee \neg A_1 \otimes a_1 \vee \neg A_2 \otimes a_2 \vee \cdots \vee \neg A_n \otimes a_n$$

Let us consider as an example the binary Skolem function $s(a, b)$. It is

$$Prop(s(a,b)) = (s_{00} \vee \neg a \vee \neg b) \wedge (s_{01} \vee \neg a \vee b) \wedge (s_{10} \vee a \vee \neg b) \wedge (s_{11} \vee a \vee b)$$

The propositional formula $Prop(s(a,b))$ may be seen as a function mapping a point $\langle a, b\rangle \in \{0,1\}^2$ in the domain of s onto the proper truth value s_{ab}.

The next step is to extend the propositional encoding from the level of terms to the level of predicates. We limit our attention to the encoding of a FOL clause in $Sk(FOL(f))$ into a satisfiability equivalent set of propositional clauses (the encoding of the union of a set of FOL clauses being just the union of the encodings of each clause).

Let us first consider the simple case of a clause containing only one existentially quantified variable e:

$$\forall u_1 \forall u_2 \cdots \forall u_n \exists e.\ p_1 \otimes u_{i_1} \vee p_2 \otimes u_{i_2} \vee \cdots \vee p_r \otimes u_{i_r} \vee e$$

where $\{u_i, i = 1, \ldots, n\}$ are all the universal variables dominating e, while $\{u_{i_j}, j = 1, \ldots, r,\ r \leq n\}$ is the subset of such variables that appear in the clause with polarities $p_1, p_2, \ldots, p_r$ respectively. The existential literal e is assumed to be positive for simplicity.

By *substituting* the propositional version $Prop(s(u_1, \ldots, u_n))$ of the Skolem function $s : \{0,1\}^n \to \{0,1\}$, defined by the 2^n boolean parameters $\{s_{0\ldots00}, s_{0\ldots01}, \cdots, s_{1\ldots11}\}$ for e, we obtain:

$$\begin{array}{l} \exists s_{0\ldots00} \exists s_{0\ldots01} \cdots \exists s_{1\ldots11} \\ \quad \forall u_1 \forall u_2 \cdots \forall u_n \\ \quad\quad p_1 \otimes u_{i_1} \vee p_2 \otimes u_{i_2} \vee \cdots \vee p_r \otimes u_{i_r} \vee \\ \quad\quad\quad \vee \left(\bigwedge_{A \in \{0,1\}^n} s_A \vee \neg A_1 \otimes u_1 \vee \cdots \vee \neg A_n \otimes u_n \right) \end{array}$$

As a consequence of the semantics flattening we have performed, the "meta" existential quantifier over an n-ary Skolem function has been transformed into a set of 2^n outer

existential quantifiers. In the worst case, we have to distribute the conjunction over all the clauses in the last term, thus obtaining 2^n clauses. Fortunately, some (many) of those clauses are trivially satisfied by complementary literals. In particular, whenever $A_{i_j} \otimes p_j = 1$ for at least one $j \in \{1, \ldots, r\}$, the clause is satisfied, so that we get only $2^{\delta(e)-r}$ clauses. Moreover, skolemized clauses no longer contain existential variables dominated by universal variables, hence all the universal literals are *forall reducible*. As a result of these two properties, we obtain the set of unit clauses:

$$\exists s_{0\ldots00} \exists s_{0\ldots01} \cdots \exists s_{1\ldots11}. \quad \bigwedge_{\substack{A \in \{0,1\}^n \\ \forall j. A_{i_j} \otimes p_j = 0}} s_A$$

In the general case we have clauses containing m existential variables $\{e_1, e_2, \ldots, e_m\}$ with $\delta(e_1) \leq \delta(e_2) \leq \ldots \leq \delta(e_m)$ and polarities $q_1, \ldots, q_m$, where each e_i is dominated by a set $\cup_{j=0}^{i} U_j$ of universal variables. Each clause also contains a possibly empty subset of universal variables $\{u_k, k = i_{j-1}+1, \ldots, i_j\} \subseteq U_j$ for each $j = 1, \ldots, m$, with $i_0 = 0$ and polarities p_k. The general shape for the clause is

$$\begin{aligned} \forall U_1 \exists e_1 \cdots \forall U_m \exists e_m. \quad & p_1 \otimes u_{i_1} \wedge \cdots \wedge p_{j_1} \otimes u_{i_{j_1}} \wedge \ q_1 \otimes e_1 \wedge \\ & p_{j_1+1} \otimes u_{i_{j_1+1}} \wedge \cdots \wedge p_{j_2} \otimes u_{i_{j_2}} \wedge \ q_2 \otimes e_2 \wedge \\ & \qquad \vdots \\ & p_{j_{m-1}+1} \otimes u_{i_{j_{m-1}+1}} \wedge \cdots \wedge p_{j_m} \otimes u_{i_{j_m}} \wedge \ q_m \otimes e_m \end{aligned} \tag{4}$$

By (a) propositionally skolemizing all the existential variables in such clause (the order does not matter), and (b) applying forall reduction to all the variables in $\cup_{j=0}^{m} U_j$, we obtain:

$$\exists S_1 \cdots \exists S_m. \quad \bigwedge_{\substack{A \in \{0,1\}^{\delta(e_m)} \\ \forall j. A_{i_j} \otimes p_j = 0}} q_1 \otimes s^1_{A|_{\delta(e_1)}} \wedge q_2 \otimes s^2_{A|_{\delta(e_2)}} \wedge \cdots \wedge q_m \otimes s^m_{A|_{\delta(e_m)}} \tag{5}$$

where s^i_A is a boolean parameter representing the truth value over $A \in \{0,1\}^{\delta(e_i)}$ of the Skolem function s^i introduced for e_i, and $S_i = \{s^i_A. A \in \{0,1\}^{\delta(e_i)}\}$, while $A|_k$ denotes the k-bit long prefix of the binary vector A. We denote by $PropSk(\cdot)$ the translation function

$$PropSk : QBF \longrightarrow PROP$$

that applied to a generic QBF clause represented by Expression (4) yields the result of our three-step translation, i.e. the set of clauses represented by Expression (5). The cardinality of this clause set is $2^{\delta(e_m)-j_m}$.

To skolemize an entire formula, we observe that Skolem functions are introduced once per variable, not once per clause. So, the propositionally skolemized version of any formula is obtained by joining together the skolem clauses obtained out of each QBF clause, always re-using the same skolem function parameters for the same existential variable. The overall skolemization procedure may thus be seen as a mapping between

the original QBF space and a purely propositional space defined over the variables s^i_A.

As an example, suppose we want to propositionally skolemize[3] the formula

$$\forall x \exists y \forall z \exists k.\ (x \vee y \vee \neg k) \wedge (\neg x \vee z \vee k) \wedge (\neg y \vee \neg k) \tag{6}$$

We obtain the following propositional instance:

$$\begin{array}{l} \exists s^y_1 \exists s^k_{01} \exists s^k_{10} \exists s^k_{11}.\ (s^y_1 \vee \neg s^k_{10}) \wedge (s^y_1 \vee \neg s^k_{11}) \wedge (s^k_{01}) \wedge (\neg s^y_0 \vee \neg s^k_{00}) \wedge \\ \qquad \wedge (\neg s^y_0 \vee \neg s^k_{01}) \wedge (\neg s^y_1 \vee \neg s^k_{10}) \wedge (\neg s^y_1 \vee \neg s^k_{11}) \end{array} \tag{7}$$

Now, suppose we find a model for (7). We would then be entitled to conclude that the skolemized version of (6):

$$\forall x \forall z.\ (x \vee s^y(x) \vee \neg s^k(x,z)) \wedge (\neg x \vee z \vee s^k(x,z)) \wedge (\neg s^y(x) \vee \neg s^k(x,z)) \tag{8}$$

is satisfiable, hence that (6) is satisfiable. Not only we are ensured that a proper interpretation for the Skolem functions $s^y/1$ and $s^k/2$ do exist to satisfy the formula, but we have *explicitily computed* such an interpretation. The model for (7) indeed gives us the desired truth value of each skolem function over each point of their domains (for models that are partial assignments, unassigned variables corresponds to don't-care values in the truth table of the corresponding skolem function). As we will see in Section 5.3.3, this information will be used by sKizzo to construct a model for (6).

3.3.2 Symbolic Representation. The ground CNF translation of a QBF problem may be exponentially larger than the originating instance. As a consequence, not only it may be unfeasible to solve the resulting SAT instance, but it might not even fit into the memory of any real machine (space explosion). Without some powerful tool for compactely representing and managing propositional skolemizations, the resulting ground instances are definitely out of reach.

The term "symbolic representation" has a broad AI-related sense, but it is used with a much more specific meaning in the realm of model checking (MC). According to MC's usage of the word, a symbolic representation is one that allows to shift from *explicit* MC techniques—where each state of a system to be checked is individually represented and manipulated—to *symbolic* MC approaches—where data structures are employed that allow to compactly and implicitly represent (possibly huge) sets of states, and also to reason about them as a whole. We adopt MC's viewpoint here.

We are interested in symbolically representing and manipulating sets of clauses. Related approaches do exist in the literature (see Section 5.1), but we have to manage a very special case here. In particular, we are only interested in representing *sets of clauses arising from the propositional skolemization of a QBF formula*, with a representation that is *closed* under the symbolic operations applied in Step 4 and 5 (see

[3] We don't mix propositional skolemization and tree-shaped prefixes until Section 3.3.3, so this formula is to be considered as a prenex CNF.

Section 3.4 and 3.5). Our representation employes one single *symbolic clause* to compactly represent the whole clause set described by the expression (5). So, we have a one-to-one correspondence between symbolic clauses and original QBF clauses.

To represent the set of clauses described by expression (5) we need to memorize three pieces of information:

1. The *ordered* list $\Gamma = [q_1 \otimes e_1, \ldots, q_m \otimes e_m]$ of existential literals in the originating QBF clause;
2. The *set of indexes* $\mathcal{I} = \{A \in \{0,1\}^{\delta(e_m)} \mid \forall j. A_{i_j} \otimes p_j = 0\}$;
3. The list $[\delta(e_1), \ldots, \delta(e_m)]$ of universal depths together with the sets $\cup_{j=0}^{i} U_j$ of universal variables dominating each existential variable.

The information in Item 3 is not related to a single clause. Rather, it is an attribute of the formula as a whole that only depends on the set of dominating variables for the literals at hand. The prefix of a prenex formula (or the quantifier tree for a structured formula), suffices to extract this information for every clause.

By contrast, the information in Items 1 and 2 actually define a *symbolic clause* $symb(C)$ obtained by the QBF clause C, which we compactly denote by writing $\Gamma_{\mathcal{I}}$. So, the symbolic transformation

$$symb : QBF \longrightarrow QBF_{SYMB}$$

maps QBF instances onto symbolic instances belonging to the space of *symbolic QBF instances* which we denote by QBF_{SYMB}. The inverse function reconstructs a QBF clause $symb^{-1}(\Gamma_{\mathcal{I}})$ out of a symbolic clause. The propositional skolemization is extended to symbolic clauses as $PropSk(\Gamma_{\mathcal{I}}) \doteq PropSk(symb^{-1}(\Gamma_{\mathcal{I}}))$.

For example, the matrix of the propositionally skolemized formula (7) is compactly represented as:

$$[y, \neg k]_{\{00,01\}} \wedge [k]_{\{01\}} \wedge [\neg y, \neg k]_{\{00,01,10,11\}} \tag{9}$$

Each symbolic clause is made up of *symbolic literals*[4], that we represent as symbolic unit clauses, possibly omitting the square braces. For example, the symbolic clause $[y, \neg k]_{\{00,01\}}$ is made up by the symbolic literals $y_{\{0\}}$ and $\neg k_{\{00,01\}}$. We say that a symbolic literal $\gamma_{\mathcal{I}}$ belongs to a symbolic clause $\Gamma_{\mathcal{J}}$, written $\gamma_{\mathcal{I}} \in \Gamma_{\mathcal{J}}$, when $\gamma \in \Gamma$ and $\mathcal{I} \subseteq \mathcal{J}$.

A symbolic formula has both a *symbolic size* and a *ground size*. The symbolic size is the number of symbolic clauses (symbolic literals) in the formula. The ground size is the number of clauses and literals in the plain propositional instance the symbolic formula stands for. So, the symbolic size (number of clauses) for a symbolic formula f is

[4] We could have introduced concepts the other way around, i.e. by defining symbolic clauses in terms of symbolic literals. However, all the symbolic literals in a symbolic clause $\Gamma_{\mathcal{I}}$ share the same set of indexes $\mathcal{I}$. It is simpler to break a clause into literals that inherit indexes than defining composition rules for obtaining correct symbolic clauses. Moreover, each bit in $i \in \mathcal{I}$ "suddenly" refers to some universal variable, and a linear shape suffices to identify which one just because clauses are attached to the proper point in the quantifier tree. As a result, arbitrary compositions of symbolic literals may not represent meaningful symbolic clauses.

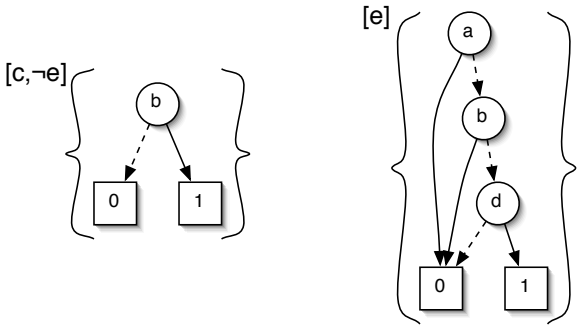

Fig. 4. Symbolic representations for the skolemized version of two clauses clauses $b \vee c \vee \neg e$ (on the left) and $\neg a \vee \neg b \vee d \vee e$ (on the right) under the relevant prefix $\forall a \forall b \exists c \forall d \exists e$.

$|f|_{symb} = \sum_{\Gamma_{\mathcal{I}} \in f} |\Gamma|$, while its ground size is $|f|_{ground} = \sum_{\Gamma_{\mathcal{I}} \in f} |\mathcal{I}|$. For example, the formula (9) has symbolic size equal to 3 and ground size equal to 7. The ground size is always greater than the symbolic size, as each symbolic clause represents at least one ground clause.

As a second layer of symbolic representation, we compactly represent index sets by means of *binary decision diagrams* (BDDs) defined over the set $Var_{\forall}(f)$. According to the semantics of BDDs, an entire set $\mathcal{I} = \{A \in \{0,1\}^{\delta(e_m)} \mid \forall j. A_{i_j} \otimes p_j = 0\}$ is represented by a single linear-sized BDD (in m) requiring one internal node for each universal variable in the originating QBF clause. Hence, the whole symbolic clause has a linear size w.r.t. the number of literals in the originating QBF clause. Figure 4 depicts our symbolic representation for two sample clauses.

Given that the size of each clause is linear in the size of the originating QBF clause, and that we produce only one symbolic clause for each QBF clause, *the symbolic size of* $symb(f)$ is linear in $|f|$. However, this only holds for the *initial* symbolic representation. The symbolic size may increase as a consequence of the manipulations described in Section 3.4.

3.3.3 Quantifier Tree. The main role of the quantifier tree constructed in Step 3 is to reduce the dimensionality of the Skolem terms. Let us consider again the formula

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

and its syntactic tree depicted in Figure 3.

According to the linear prefix $\forall a \forall b \exists c \forall d \forall e \exists f \exists g \exists h$, we should introduce—among the others—the skolem functions $s^c(a,b)$ and $s^h(a,b,e)$. But, certain dependencies are artificially forced by the linear shape of the prefix. According to the quantifier tree,

it suffices to introduce $s^c(a)$ and $s^h(a, e)$. So, when introducing Skolem functions, sKizzo always looks at the quantifier tree to select as few arguments as possible.

The path from the root to the node where a clause is attached defines the *relevant* prefix for that clause. Along the relevant prefix for a clause we encounter all the existential variables $e_1, e_2, \ldots, e_n$ in that clause; *before* each existential variable e_i, we also encounter all the dominating universals for e_i. These sets of dominating universals, together with the sequence of universal depths $\delta(e_1) \leq \delta(e_2) \leq \cdots \leq \delta(e_n)$, are the missing information to interpret a symbolic clause and reconstruct its ground meaning. In fact, by expression (5), each index set $\mathcal{I}$ for a literal $\gamma_{\mathcal{I}}$ is a subset of $\{0,1\}^{\delta(\gamma)}$, where the j-th bit i_j in every $i \in \mathcal{I}$ corresponds to the j-th dominating universal quantifier for γ. By adopting such convention, we obtain two related effects:

1. we simplify the notation for symbolic clauses, and avoid representing redundant information (i.e. to duplicate the information on the relevant sets of dominating universals in each symbolic clause);
2. symbolic clauses are not self-contained objects, as we need to refer to the quantifier tree to extract their ground meaning.

The second point would be of no importance for a linear, prenex formula, because the meaning of the j-th bit in any index would be same for every literal (and every clause). When we move to a tree-like syntactic structure, this property fails to be true, as the meaning of each bit also depends on the branch of the tree the literal lays on.

Most manipulations we perform over symbolic clauses and literals only involve symbolic objects on the same branch, so the above notation is unambiguous. But a few of these operations (namely, those described in Section 3.4.3 and Section 3.4.4) may involve clauses attached to different branches.

To make operations among generic index sets unambiguous, we redefine the notion of projection and selection in terms of an underlying interpretation for index sets that refers to the whole set of universal variables $Var_\forall(f) = \{u_1, u_2, \ldots, u_n\}$. We interpret each index set $\mathcal{I} \subseteq \{0,1\}^m$ relative to the universal variables $u_{j_1}, u_{j_2}, \ldots u_{j_m}$, as a subset $U_{\mathcal{I}} \subseteq \{0,1\}^{|var_\forall(f)|}$, where $\langle p_1, p_2, \ldots, p_n\rangle \in U_{\mathcal{I}}$ iff an index $i = \langle i_0 i_1 \ldots i_m\rangle \in \mathcal{I}$ exists such that $i_k \otimes p_{j_k} = 0$ for $i = 1, \ldots, m$. The complement set $\overline{U_{\mathcal{I}}}$ of $U_{\mathcal{I}}$ is $\{0,1\}^{|var_\forall(f)|} \setminus U_{\mathcal{I}}$. The projection $U_{\mathcal{I}}|_e$ of $U_{\mathcal{I}}$ onto the existential variable (or literal) e with dominating universals $u_{j_1}, u_{j_2}, \ldots u_{j_m}$ is the set $\{\langle i_1 i_2 \ldots i_{\delta(e)}\rangle \mid \exists \langle p_1, p_2, \ldots, p_n\rangle \in U_{\mathcal{I}}$ with $i_k = p_{j_k}, k = 1, \ldots, \delta(e)\} \subseteq \{0,1\}^{\delta(e)}$. Finally, by *selecting* $U_{\mathcal{I}}$ with an universal literal $l = q \otimes u_k$ (on the variable v_k with polarity $q \in \{0,1\}$) as a condition, we obtain the set $U_{\mathcal{I}} * l = \{\langle p_1, p_2, \ldots, p_n\rangle \in U_{\mathcal{I}} |\ p_k = q\}$.

For objects laying on the same branch of the tree, we maintain a light notation. For example, given any two symbolic literals $\alpha_{\mathcal{I}}$ and $\beta_{\mathcal{J}}$ on the same branch, we write $\alpha_{\mathcal{I} \cap \overline{\mathcal{J}}}$ to mean $\alpha_{(U_{\mathcal{I}} \cap \overline{U_{\mathcal{J}}})|_\alpha}$.

3.4 Step 4: Symbolic Normalization

The aim of this step is to (attempt to) decide the symbolic instance produced during the previous step (hence, to decide the satisfiability of the original QBF problem).

It works by computing the deductive closure of a set of *symbolic inference rules*. When the set of rules adopted is not refutationally complete, instances exist that stay undecided at the end of the current step. In these cases, however, a satisfiability-equivalent, symbolic output formula is generated that is guaranteed to show a (much) *smaller ground size* than the input formula. The aim of this step may thus be seen as an attempt to reduce the complexity of the problem that Steps 5 and 6 will have to manage.

Thought symbolically represented, the formula we face is a purely existential CNF propositional instance attainable via the $PropSk$ function. The inference rules we adopt need to add nothing to the well-known inference systems described in the literature to simplify/decide such "ground" formulas. Rather, the emphasis is on designing symbolic versions of the standard rules that work without expanding symbolic clauses to ground clauses, symbolic literals to ground literals, and so on. In essence, it is a matter of defining how the basic steps (subsumption, resolution, assignments substitution, etc.) and their compositions can be performed at a purely symbolic level (i.e.: on sets insted of on single set's elements).

We may figure out what symbolic reasoning does by referring to the following commutative diagram (where $Norm_{\mathcal{R}}(\cdot)$ denotes the subset of normalized formulas w.r.t a set of inference rules $\mathcal{R}$).

$$\begin{array}{ccc} QBF_{SYMB} & \xrightarrow{PropSk} & PROP \\ \text{symbolic inferences}\downarrow & & \downarrow\text{standard inferences} \\ Norm_{\mathcal{R}}(QBF_{SYMB}) & \xrightarrow[PropSk]{} & Norm_{\mathcal{R}}(PROP) \end{array}$$

Symbolic reasoning consists in walking the diagram top-down first, then left-to-right.

The first step towards symbolic reasoning amounts to extend the star operator introduced in Section 2.5 to the case of symbolic clauses and symbolic literals. This is done as follows.

$$\Gamma_{\mathcal{I}} * l_{\mathcal{J}} = \begin{cases} \Gamma_{\mathcal{I}\cap\overline{\mathcal{J}}} & \text{when } l \in \Gamma \\ \Gamma_{\mathcal{I}\cap\overline{\mathcal{J}}} \wedge \Phi_{(\mathcal{I}\cap\mathcal{J})|_{\delta(\Phi)}} & \text{with } \Phi = \Gamma \setminus \{\neg l\}, \text{when } \neg l \in \Gamma \\ \Gamma_{\mathcal{I}} & otherwise \end{cases} \tag{10}$$

The following inference rules build on top of the symbolic star operator.

3.4.1 SUCP: Symbolic Unit Clause Propagation. The SUCP rule is the simplest one. It builds on top of the observation that each symbolic unit clause $[\gamma]_{\mathcal{I}}$ in the formula represents a set $\{\gamma_i | i \in \mathcal{I}\}$ of ground unit literals. All of them need to be assigned to avoid contradictions. These assignments can be performed all-at-once by simply exploiting the symbolic star operator. See Algorithm 2.

input : A symbolic formula f

output: A symbolic formula $f' \overset{SAT}{\equiv} f$ with no unit clause

while $[\gamma]_{\mathcal{I}} \in f$ *and* $\bot \notin f$ **do**
  $f \leftarrow f * \gamma_{\mathcal{I}}$;
end

Algorithm 2: A basic version of the symbolic unit clause propagation rule

input : A symbolic formula f

output: A symbolic formula $f' \overset{SAT}{\equiv} f$ with no pure literal

$\mathsf{V} \leftarrow var_{\exists}(f)$;
while $\mathsf{V} \neq \emptyset$ *and* $f \neq \emptyset$ **do**
  pick one $v \in \mathsf{V}$;
  $\mathsf{P} \leftarrow \emptyset$;
  foreach $\Gamma_{\mathcal{I}} \in f$ *such that* $v \in \Gamma$ **do**
    $\mathsf{P} \leftarrow \mathsf{P} \cup \mathcal{I}|_{\delta(v)}$;
  end
  $\mathsf{N} \leftarrow \emptyset$;
  foreach $\Gamma_{\mathcal{I}} \in f$ *such that* $\neg v \in \Gamma$ **do**
    $\mathsf{N} \leftarrow \mathsf{N} \cup \mathcal{I}|_{\delta(v)}$;
  end
  $\mathcal{I}^{+} \leftarrow \mathsf{P} \cap \overline{\mathsf{N}}$;
  $\mathcal{I}^{-} \leftarrow \mathsf{N} \cap \overline{\mathsf{P}}$;
  foreach $\Gamma_{\mathcal{I}} \in f$ *such that* $v_{\mathcal{I}^+} \in \Gamma_{\mathcal{I}}$ *or* $\neg v_{\mathcal{I}^-} \in \Gamma_{\mathcal{I}}$ **do**
    foreach $\gamma \in \Gamma$ **do**
      $\mathsf{V} \leftarrow \mathsf{V} \cup var(\gamma)$;
    end
  end
  $\mathsf{V} \leftarrow \mathsf{V} \setminus \{v\}$;
  $f \leftarrow f * v_{\mathcal{I}^+}$;
  $f \leftarrow f * \neg v_{\mathcal{I}^-}$;
end

Algorithm 3: A basic version of the symbolic pure literal elimination rule

3.4.2 SPLE: Symbolic Pure Literal Elimination. The SPLE rule does what we would expect from the standard rule, but performs its job in a purely symbolic manner. It (a) constructs a complete symbolic representation of the set of every *pure ground literal*, and (b) applies this literal to the formula. The simplest way[5] to perform symbolic PLE is reported in Algorithm 3.

[5] On the implementation side, the simplest way might fail to be the best way. Thought quite intuitive, Algorithm 3 sometimes gets into troubles because big BDDs are generated as intermediate results. So, we also designed a step-by-step version that (a) computes pure literals out of each clause (rather than for each variable), and (b) always manages several (still unfinished) computations at once, with a greedy, cost-minimizing scheduler to control the job.

input : A symbolic formula f
output: A symbolic formula $f' \overset{SAT}{\equiv} f$ normalized w.r.t SHBR

continue $\leftarrow$ TRUE;
while *continue and* $f \neq \emptyset$ **do**
  continue $\leftarrow$ FALSE;
  G $\leftarrow$ the symbolic implication graph over f;
  foreach *source node* $s \in G$ **do**
    foreach *path* $s = a_0 \xrightarrow{\mathcal{I}_1} a_1 \cdots \xrightarrow{\mathcal{I}_{n-1}} a_{n-1} \xrightarrow{\mathcal{I}_n} a_n$ *in* G *with* $a_n = \neg a_k$ **do**
      $U_{\mathcal{I}} \leftarrow \cap_{j=k+1}^{n} U_{\mathcal{I}_j}$;
      if $U_{\mathcal{I}} \neq \emptyset$ **then**
        $\mathcal{I} \leftarrow U_{\mathcal{I}}|_a$;
        $f \leftarrow f * \neg a_{\mathcal{I}}$;
        if *new binary clause created* **then**
          continue $\leftarrow$ TRUE;
        end
        if $\bot \in f$ **then**
          return UNSAT;
        end
      end
    end
  end
end
return f;

Algorithm 4: The symbolic hyper binary resolution algorithm

3.4.3 SHBR: Symbolic Hyper Binary Resolution. The SHBR rule enumerates all the resolution chains of symbolic binary clauses, looking for *failed* symbolic literals, i.e. for literals $\neg a_{\mathcal{I}}$ such that each $\neg a_i \in \neg a_{\mathcal{I}}$ can be derived (via a finite number of resolution steps only involving binary clauses) as a consequence of the hypothesis a_i. Each ground literal in $a_{\mathcal{I}}$ generates a contradiction ($f * a_i$ is UNSAT for every $i \in \mathcal{I}$), so we force the opposite symbolic assignment, thus shifting our attention onto $f * \neg a_{\mathcal{I}}$.

To compute all the failed literals we employ an approach similar to the standard one for propositional logic (see Section 5.1). We build a *symbolic implication graph*, which has a node for each positive and negative existentially quantified variable in the original formula, and a couple of arcs $a \xrightarrow{\mathcal{I}} \neg b$ and $b \xrightarrow{\mathcal{I}} \neg a$ for each binary symbolic clause $[a, b]_{\mathcal{I}}$. So, unlike standard implication graphs, symbolic graphs feature *labeled arcs*. The arc originating from $[a, b]_{\mathcal{I}}$ is labeled by $\mathcal{I}$. Should an arc be originated from more than one clause, it would be labeled by the *union* of the sets of indexes of each clause. Each symbolic arc $a \xrightarrow{\mathcal{I}} b$ represents a set of *ground arcs* $\{a_{i|_{\delta(a)}} \longrightarrow b_{i|_{\delta(b)}}, i \in \mathcal{I}\}$.

At this point, following the two-level symbolic representation of clauses, we employ a two-step algorithm for extracting symbolic failed literals (see Algorithm 3.4.3):

1. We discover all the *potential* failed literals by discarding the labels on the arcs. A depth-first, non-redundant visit starting from each *source* of the graph is employed. A potential failed literal a is one for which we have encountered the following (portion of a) resolution path: $a \xrightarrow{\mathcal{I}_1} a_1 \xrightarrow{\mathcal{I}_2} \cdots \xrightarrow{\mathcal{I}_n} \neg a$;

2. As each symbolic arc represents a set of ground arcs, a symbolic path from a to $\neg a$ in the symbolic implication graph represents a (*possibly empty*) set of ground paths. We are interested in symbolically extracting all such ground paths, by (a) intersecting the indexes encountered along the path, and (b) projecting the resulting set onto the index domain relevant to a.

It is interesting to note that a failed n-step path over a symbolic implication graph immediately maps onto a sequence of $n + 1$ nodes in the quantifier tree of the formula (where the first node is equal to the last one) which we call a *loop*. Every two subsequent nodes in this loop always lay on the same branch (because an arc is originated from a binary clause which by construction is attached to its lowest existential variable in a branch where the other one is *already* appeared). So, the loop is made up of top-down and bottom-up steps, but no *lateral* step is allowed. Bottom-up steps may only follow the path towards the root, as the underlying structure is a tree, while top-down steps may "choose" a branch whenever more than one is given. As a consequence, loops made up of at least 4 steps may cover a subtree of the quantifier tree, not just a linear branch. This means that the definition domains of the symbolic literals involved in a chain of derivations are not necessarily sub-domains of one another. The extraction of an *actual* failed literal from a potential failed literal takes into account this property by exploiting the notions introduced in the last part of Section 3.3.3.

The rules described so far only rely on symbolic assignments. The next rule in addition requires *symbolic equivalency*, which we now introduce informally. A symbolic literal $a_{\mathcal{I}|_{\delta(a)}}$ is equivalent to a symbolic literal $b_{\mathcal{I}|_{\delta(b)}}$ when (1) a and b lay on the same branch of the quantifier tree, and (2) for each $i \in \mathcal{I}$, $a_{i|_{\delta(a)}} \leftrightarrow b_{i|_{\delta(b)}}$ is a consequence of the formula. When this happens, we can simplify the formula by substituting each occurence of $a_{\mathcal{I}|_{\delta(a)}}$ with $b_{\mathcal{I}|_{\delta(b)}}$ (or vice-versa).

Clauses may fall into one of three classes when a standard propositional substitution is applied: (1) those which remain untouched, (2) those which only contain the substituted literal and thus exchange one literal for another, and (3) those which contains both variables involved in the substitution, and may thus be either (3a) satisfied (when the substitution generate a couple of opposite literals) or (3b) shortened (when the substitution generates two copies of the same literal).

This three-fold consequence of substitution stays the same in the symbolic case, with the caveat that symbolic clauses actually represent sets of ground clauses and literals, and symbolic substitutions represent sets of equivalencies over ground literals. In general, it is not the case that the whole set of literals (if any) in a symbolic clause is covered by the substitution. This means that in the case (2) and (3b) above we may obtain *two* symbolic clauses out of each originating clause after substitution is applied.

3.4.4 SER: Symbolic Equivalency Reasoning. The SER rule works on the very same symbolic implication graph used during SHBR. It aims at identifying sets of symbolic literals that are equivalent to one another. Thereafter, symbolic equivalency is applied to simplify the formula. Following the two-level symbolic representation of objects in sKizzo we employ a two-step algorithm to perform symbolic binary equivalence reasoning:

```
input  : A symbolic formula f
output: A symbolic formula f' ≡^SAT f normalized w.r.t. SER
continue ← TRUE;
while continue do
    continue ← FALSE;
    G ← the binary implication graph over f;
    SCC ← the set of strongly connected components in G [Kosaraju, Sharir];
    foreach S ∈ SCC do
        lits ← the set of literals in S;
        while lits ≠ ∅ do
            m ← one literal in lits with the maximal universal depth;
            foreach loop a_0 -I_1-> a_1 ··· -I_{n-1}-> a_{n-1} -I_n-> a_0 in S with m = a_0 do
                U_I ← ∩_{j=0}^{n} U_{I_j};
                if U_I ≠ ∅ then
                    if ∃i∃j.a_i = ¬a_j then
                        return UNSAT;
                    else
                        a ← one a_i with the minimal universal depth;
                        I_a ← U_I|_a;
                        foreach b = a_j, b ≠ a do
                            I_b ← U_I|_b;
                            Apply equivalency a_{I_a} ↔ b_{I_b} to f;
                            if new binary clause created by step (3b) then
                                continue ← TRUE;
                            end
                        end
                    end
                end
                lits ← lits \ {m};
            end
        end
    end
end
return f;
```

Algorithm 5: The symbolic equivalency reasoning algorithm

1. First, we extract all the *strongly connected components* (SCCs) from the implication graph, discarding labels on arcs. In the standard propositional case, each SCC identifies an equivalence class over the set of literals, but this would be a too strong conclusion were it directly applied to symbolic literals: in the propositional case, any two node in a SCC are part of a non-intersecting loop entirely belonging to the SCC, and this is the reason why they are equivalent. In the symbolic case, any two ground literals in a SCC belong to some symbolic loop; this is a *necessary* condition for equivalence, but their actual equivalence has to be tested;
2. To test the equivalence of literals in each SCC, we cannot consider the component as a whole (we would reach a too weak conclusion by assuming that the equivalency of literals stem by traversing *all* the symbolic arcs); rather, we have to *enumerate*

the loops belonging to the SCC, and for each symbolic loop we have to compute the actual set of ground loops it stands for (using the same technique as in HBR).

A significant difference w.r.t. the standard propositional case comes into play at this point. What ER does in that case is to extract one representative literal out of each equivalency class and then apply substitution. The selection is done arbitrarily, as all literals are equivalent: the formula resulting after substitution is the same whichever literal is chosen (apart from the name of the variable representing the class of equivalence). This is not true for the symbolic case, because two equivalent symbolic literals may be defined over Skolem domains of different dimensionality (thus generating a different set of ground literals and clauses). So, after we have tested that the symbolic loop is non-empty, we still need to select a representative literal from that loop.

In addition to this, when we move from the SCC-as-a-whole technique to the extraction of a sequence of loops in the SCC, we implicitly generate an *ordering* among substitutions whose effect is worth considering.

Our answer to these degrees of freedom aims at reducing the ground size of the resulting formula, and is given in Algorithm 5. Note that the correctness of this algorithm relies on a hidden property, i.e. that the minimal-depth literal in every loop over the quantifier tree always dominates all the other literals in the same loop.

3.4.5 Notes on symbolic inferences. Step 4 might be forced to *enlarge* the size of its own representation of the problem in order to reduce the size of the instance that Step 5 and 6 will have to manage. It is possible indeed for a smaller ground instantiation of the problem to correspond to a more complicated symbolic representation for the problem itself. There are two major sources of enlargement, reflecting the two-level symbolic representation of the clauses:

- Set representation is done via BDDs. It is well known that the size of a binary decision diagram is not directly related to the size of the set it represents [76]. For the initial symbolic skolemization, this size is linear (see Section 3.3.2), but when the star operator and the other rules are applied, BDDs start representing "non-convex" sets of indexes. Though smaller as to ground size, these sets may require more space at the symbolic level (see experimental results at Section 4.3).
- Symbolic assignments may split each single symbolic clause they touch into couples of clauses, in such a way that even if the overall ground cardinality is never increased, the number of symbolic clauses may grow, and their memory representation is enlarged as well.

We also notice that there is a class of formulas that can be decided without requiring more than what can be done during Step 4. In general, this class is implicitly defined by the inference power of the combination of the symbolic rules we adopt. In particular, at least all the instances that contain no more than two existentially quantified variables per clause are decided during this step. This is a consequence of two properties of the standard rules for binary reasoning inherited from their symbolic counterparts. Namely:

- A PROP formula only containing 2-clauses is satisfiable iff no contradiction is detected during hyper binary resolution;

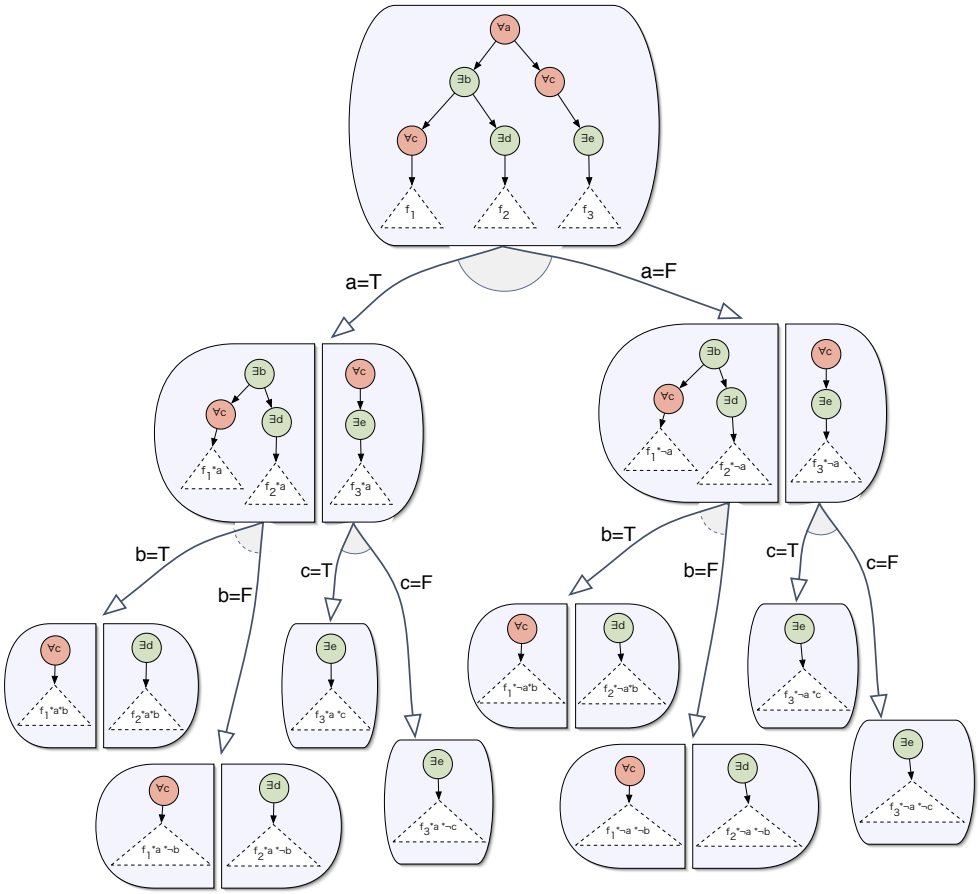

Fig. 5. The top-most part of an AND/OR, divide-et-impera search tree for the formula $\forall a((\exists b(\forall c\, f_1(a,b,c)) \wedge \forall d\, f_2(a,b,d)) \wedge \forall c \forall e\, f_3(a,c,e))$

– A PROP formula only containing 2-clauses is satisfiable iff no SCC contains a variable in both polarities[6].

3.5 Step 5: Symbolic Divide-et-Impera

The aim of this step is to apply a systematic procedure to decide all the problems that no previous step has been able to decide.

The procedure we apply employs a top-down strategy that is best described in an inductive manner (see Algorithm 6):

[6] For the symbolic case this is just a necessary condition. As we saw, it is also necessary that both literals lay on a ground cycle contained in that SCC.

Algorithm 6: symbDecide

input : A tree-shaped symbolic formula t
output: A SAT/UNSAT answer

```
begin
    // normalization performed by Step 4
    t_norm ← normalize(t);
    if t_norm = ∅ then
        outcome ← SAT
    else if ⊥∈ t_norm then
        outcome ← UNSAT;
    else
        if groundVersionIsAffordable (t_norm) then
            groundCnf ← PropSk(t_norm);
            outcome ← SatSolver.solve (groundCnf);
        else
            l ← labelattherootoft_norm;
            if (l = " ∧ ") then
                outcome ← SAT;
                foreach child subtree t' of t_norm do
                    outcome ← outcome and symbDecide (t');
                end
            else
                leftOutcome ← SAT;
                foreach child subtree t' of t_norm do
                    leftOutcome ← leftOutcome and symbDecide (t' ∗ l);
                end
                rightOutcome ← SAT;
                foreach child subtree t' of t_norm do
                    rightOutcome ← rightOutcome and symbDecide (t' ∗ ¬l);
                end
                if isUniversal(l) then
                    outcome ← leftOutcome and rightOutcome;
                else
                    outcome ← leftOutcome or rightOutcome;
                end
            end
        end
    end
    return outcome ;
end
```

base case 1 When symbolic normalization suffices to solve the instance, report the symbolic solution.

base case 2 When the instance can be addressed in a *ground way*, just do it.

inductive case When no base case applies, divide the instance into smaller sub-instances according to the quantifier tree, and report that the whole instance is SAT iff each sub-instance is SAT.

The inductive case deals with two conceptually different trees: the syntax-related quantifier tree of the symbolic formula, and the semantics-related AND/OR search tree that is visited to decide the instance. The former is explicitly manipulated as a parameter, the latter is implicitly explored via the recursive structure of the decision procedure. The two trees are related in the sense that each node of the quantifier tree is to be decided by checking both truth values for the labeling variable (or just one, should lazy evaluation suffice), while each truth value generates a set of quantifier sub-trees to be recursively decided. The resulting situation is depicted for a sample case in Figure 5.

To have the whole procedure working we need to extend the meaning of the star operator when it is applied with a ground universal literal v as a second argument:

$$\Gamma_{\mathcal{I}} * v = \begin{cases} \Gamma_{\mathcal{I}*v} & \text{when } v \in d_{ing}(\Gamma) \\ \Gamma_{\mathcal{I}} & otherwise \end{cases} \tag{11}$$

where the $\mathcal{I} * v$ operation denotes the existential abstraction defined in Section 3.3.3.

3.6 Step 6: Groundization

The aim of this step is to compute an actual compilation to SAT of a symbolic CNF representation, whenever Step 5 decides to encode sub-problems into SAT instances to be passed to a SAT solver.

In essence, this step computes the $PropSk$ function by applying expression (5). Altought theorically straightforward, this operation deserves a lot of attention on the practical side (see Section 4.1.3). Groundization is made up of two steps: (1) generation of the ground space and (2) generation of the ground clauses.

3.6.1 Ground space generation. The key operation to be performed is to construct a mapping between the *structured namespace* of symbolic literals and a *flat namespace* for ground literals, which is more SAT-solver friendly. This amounts to uniquely associate a positive integer k (representing a propositional variable p_k) to each ground literal v_i that belongs to at least one symbolic clause in the current symbolic formula. The association should also work the other way around (necessary to model reconstruction), so we need a bijective function.

In addition to this functional property, we also desire two additional *non-functional* requirements:

Compactness. The set of integers generated for the formula as a whole should be composed of all and only the integers in the interval $[1, n]$, for some sufficiently large n (no unused variable code: this is to avoid that the SAT solver allocates unnecessarily large data structures);

Invertibility. It should be possible to compute both the direct and the inverse mapping function efficiently (ideally, near to $O(1)$).

The signature of the resulting mapping function is

$$V_{map} : D_\exists \times D_\forall \rightarrow [1, n]$$

where $D_\exists = var_\exists(f)$ and $D_\forall = \{0,1\}^{|var_\forall(f)|}$. The function is partial, as it is defined only over ground literals actually belonging to f.

3.6.2 Ground clauses generation. Once V_{map} is constructed, a clause $\Gamma_\mathcal{I}$, with $\Gamma = \{p_1 \otimes e_1, p_2 \otimes e_2, \ldots, p_m \otimes e_m\}$, is expanded to its ground meaning by producing for each $i \in \mathcal{I}$ the ground clause

$$sign(p_1) \cdot V_{map}(e_1, i|_{e_1}) \wedge sign(p_2) \cdot V_{map}(e_2, i|_{e_2}) \wedge \ldots \wedge sign(p_m) \cdot V_{map}(e_m, i|_{e_m})$$

where $sign(p) = +1$ for $p = 1$ and $sign(p) = -1$ for $p = 0$.

4 Implementation and experimentation

In this section we present a first implementation of our decision procedure and a preliminary experimental evaluation. The interested reader may find further details and a wider experimentation at [4]. The section is organized as follows. Section 4.1 discusses our implementation. Section 4.2 introduces other solvers and describes the benchmarks used for evaluation. Functional results for sKizzo are reported in Section 4.3. Finally, Section 4.4 focuses on relative solvers' performance.

4.1 Implementation

sKizzo features a preliminary implementation (current version: sKizzo$^{v0.1}$), which is a 60k-line piece of code written in C using an object-oriented programming style. It has been developed from scratch on a 14" iBook running MacOS X 10.3, using Xcode 1.(2-5) as a programming environment, gcc 3.3 as a compiler and Shark 4.0 as a profiler. Such platform has been used to extract the results presented in Section 4.3. The performance-related experimental results given in Section 4.4 have been obtained on a different platform. Namely, the whole system has been ported to Linux and tested on a 2.6GHz P4 processor with 1GB main memory, running RedHat 9.3. Compiler version for the Linux platform is 3.4.
sKizzo relys on two libraries (a C and a C++ library) to perform its work. Figure 6 depicts the interactions among sKizzo's steps and the following two libraries:

A BDD package. We employ the CUDD package [72], version 2.4.0, by Fabio Somenzi (Department of Electrical and Computer Engineering, University of Colorado at Boulder) which is meant for manipulation of Binary Decision Diagrams (BDDs), Algebraic Decision Diagrams (ADDs) and Zero-suppressed Binary Decision Diagrams (ZDDs).

A SAT solver. We exploit zChaff [52], version 2004.5.13, a state-of-the-art, search-based SAT solver from the SAT Research Group at the Princeton University.

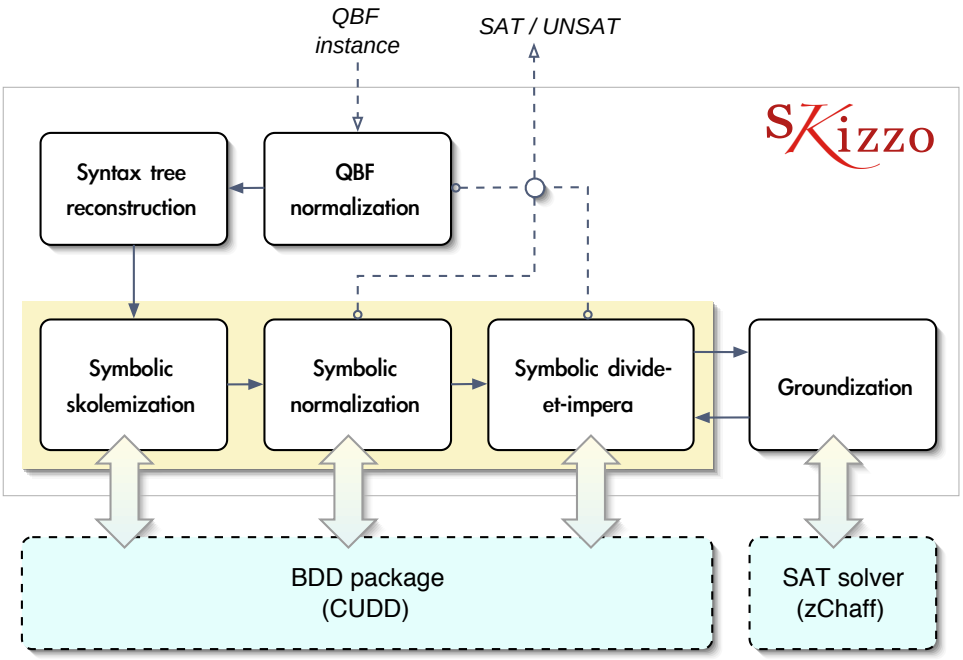

Fig. 6. Interaction between sKizzo and the libraries it exploits

Our implementation produces statistical information for families of instances, and detailed reports of the solution process of single instances, also including the time and memory requirements per phase. Some steps (such as tree reconstruction and preliminary QBF simplification) may be optionally disabled. On demand, both the reconstructed syntactic trees and the sets of ground instances produced may be dumped on secondary memory in a textual format for later analysis.

In the rest of this section, we devote our attention to a few implementation-related issues that may have a strong impact over the run-time performance of sKizzo.

4.1.1 Efficiency monitoring. Our implementations of the symbolic inference rules used during Step 4 show quite instance-depending performances (deductions per time unit). For example, the pure literal elimination rule is quite slow on certain instances, while the unit clause propagation quickly attacks the same problem, or vice-versa. All the rules but PLE may also occasionally lead to an explosion of the number of symbolic clauses, but this explosion appears to strongly depend on which other rules have already reached the fixpoint.

Being still absent a theorical framework explaining these and other effects, we decided to temporarily classify this topic as an implementation-related issue. We indeed resort to a heuristic, on-the-fly scheduling policy for inference rules.

In particular, we implemented a fix-priority policy with on-the-fly resource consumption monitoring and preemptive interruption capabilities. A fixed priority is assigned off-line to each rule, according to some statistical evidences. The scheduler keeps on picking the highest-priority inference rule that has not yet reached the fixpoint, and applies it until either the fixpoint is reached, or the rule begins to *trash*. Trashing is defined in terms of resource consumption, in particular memory consumption (see below) and time efficiency. The latter measure is computed by monitoring the number of inferences per second the rule is performing, and the consequences of these inferences both on the symbolic size and on the ground size of the problem. By analysing the typical behaviour of symbolic rules, we defined for each rule some trashing condition (for example: "PLE is trashing whenever it shows for more than N contiguous inference cycles a *shrinking rate* for the *projected ground size* of the problem which is both monotonically decreasing and constantly below treshold T"). When the monitor decides that a rule is trashing, the rule itself is preemptively interrupted and marked as a "trashing rule". Then, the next highest-priority, non-trashing rule (if any) is applied.

The scheduler not only monitors what the currently rule is doing, but it also tracks what is happening to the rules that are currently inactive (either at the fixpoint or trashing). Inactive rules may indeed loose both their staus of "fixpointed rule" and their status of "trashing rule" as a consequence of other rules' behaviour, with consequences on the rest of the inference trace.

4.1.2 Memory management. The virtual memory facility provided by all modern operating systems is largely unuseful (if not dangerous) for resolution-based QBF solvers. These solvers tend indeed to be memory-eager. When physical memory is over, the OS—hoping to help—suddenly moves on to the next level of the memory hierarchy (virtual memory is expanded to the disk). Memory access becomes orders of magnitude slower and raw performance heavily falls down. The point is that sKizzo (together with some other QBF solvers) needs to "contineously" refer to all (or to a great part of) the information it has stored in memory, so no portion of such information is safely moved to secondary memory (as opposed to what happens in many other common situations). The only way of avoiding memory trashing, is to avoid consuming the whole physical memory.

sKizzo performs a contineous monitoring of memory consumption, with the aim of avoiding that swap-to-disk even begins. Steps that may consume great amount of memory are Symbolic Normalization (the number of symbolic clauses increase, BDDs get larger), Symbolic Divide-et-impera (many symbolic instances at the same time have to be maintained when deep decision levels are reached), and Groundization (the potential compilation-to-SAT blow-up gets real). The first two situations are rather uncommon. They are dealt with as non-recoverable problems. sKizzo surrenders and communicates to the user that memory limitations prevented him from solving the instance. The last situation——fairly more common—is managed by estimating the memory requirements for every SAT instance, before the instance itself is generated. Should those projected requirements overcome available physical memory, the instance would not generated and the divide-et-impera procedure would be requested to split the problem at hand into (more but) smaller sub-problems. This dynamic adjustment creates an interesting

(and automatic) time/memory tradeoff, whose simplest effect is to produce different execution traces (and execution times) on the very same machine by just adding/removing physical memory (*without* adjusting any parameter).

4.1.3 Mapping to CNF. The mapping function realized by Step 6 is a time-critical one. It is in general responsible for producing several ground problems per session, each problem having up to million clauses (each clause in turn made up of several literals to be translated). It is quite common for the mapping function to be called hundreds million times during one session's lifetime. Also, generation time overcomes solution time for a large class of instances. For these reasons, a careful engineering of data structure is necessary. In particular, hash tables and logical properties of the underlying ordered decision diagrams are heavily exploited. The inverse function is also to be computed efficiently, as model reconstruction—thought not yet implemented—will be a key feature of sKizzo.

Another interesting computation in need for efficiency concerns the number of ground clauses the current symbolic instance would produce, should it be made ground immediately. A lazy, purely symbolic estimation of the ground size (based on properties of the BDDs) is performed. This capability is exploited several times, though performance is a concern only during *efficiency monitoring*, due to the high number of estimations required (see Section 4.1.1).

4.2 Benchmarks and solvers

To evaluate sKizzo we refer to the QBFLIB's archive [33] maintained by the STAR-lab group at the University of Genova. This growing set of benchmarks is currently comprised of more than 4000 instances and have been used in the "QBF Solver Evaluation" sessions during SAT03 and SAT04.

In this preliminary evaluation, we focus on a subset of the non-random families of instances collected in the QBFLIB. In particular, we consider:

Rintanen's benchmarks [59], the first and best-known collection of QBF problems, made up of 47 instances divided into 5 families, obtained by encoding planning problems into QBF. These instances are currently within the solving capabilities of most state-of-the-art solvers, so they can be exploited to compare the time/memory requirements of different solvers.

Ayari's benchmarks [1], made up of 72 instances divided into 5 families, obtained from real-world verification problems on circuits and protocol descriptions. These instances are still quite challenging for modern solvers, and some of them have never been solved.

Biere's benchmarks [9], made up of 64 instances divided into 4 families; the n-th instance in each family translates a model checking (MC) problem stating an invalid safety property over an n-bit counter (the optional *reset* and *enable* inputs yield 4 combinations clustered into 4 families). The properties state that the counters never reach the *all-one* state starting from the *all-zero* state (each one thus fails to be true after $2^n - 1$ steps). Such problems are easy for BDD-based symbolic MC.

Conversely, they are rather difficult for SAT-based bounded MC techniques, as they capture the worst-case scenario in which the number of steps necessary to falsify the property equals the diameter of the system. These instances *could be* simple but effective witnesses of the fact that QBF-reasoning really adds something to SAT-based methods.

In Section 4.3 we also refer to other families, such us those obtained by encoding modal logic instances.

In Section 4.4 we compare with the SOTA solver (from State-Of-The-Art) and with a few among the best real solvers. The SOTA solver is an ideal solver built by starting in parallel all the existing real solvers. It conquers an instance if (and as soon as) one of the real solver does. Thus, in no benchmark the SOTA solver performs worse than any real solver, as it *dominates* all of them. The time taken to solve a set of instances with the SOTA solver is the sum of the best time on each instance (the calculation may thus involve more than one real solver per family). In practice, to construct the SOTA performance profile we have to limit our attention to a specific set of real solvers, and target a limited set of benchmarks with all these solvers. Here we refer to the SOTA solver made up by all the solvers participating in the QBF04 evaluation, as it results from [43].

To directly compare with a few real solvers, we will restrict our attention to four state-of-the-art solvers, among which we find the three top-rated solvers according to most of the results presented in [43] (see also Section 5.1). Namely:

QuBE-LRN [33], version 1.3, a search-based solver featuring lazy data structures for unit clause and pure literal propagation, plus conflict and solution learning.

Quantor [9], version 2004.01.25, a solution-based solver employing q-resolution and expansion to eliminate quantifiers, plus a number of other features to improve efficiency.

SEMPROP [45], version 24.02.02[7], a search-based solver featuring directed backtracking and lemma/model caching.

yQuaffle [78], version 09.30.04, a search-based solver featuring multiple conflict-driven learning, inversion of quantifiers and solution-based backtracking.

Two more interesting solvers for QBF are ZQSAT [32] and QMRES [57]. They have been developed quite recently, and apply symbolic techniques to QBF. We plan to directly compare with these solvers when public releases will be available. In the meanwhile, indirect comparisons can be deduced from the data presented in [57, 32].

4.3 Functional results

Here we briefly address—from an experimental point of view—three aspects:

1. The actual role of the QBF inference rules employed in Step 1.

[7] A more recent version does exist, but we have experienced some problems in making it work on our test platform.

Instance	*Variables*	*Clauses*	*Alt.*	*Prefix shape*
flipflop-3-c	551	203	2	E[9]A[15]E[140]
cf-2-2x3-w	94,206	1,375	6	E[14]A[2]E[164]A[2]E[164]A[2]E[164]
cf-2-4x8-d	99,432	43,333	32	E[455]A[4]E[818]· · ·E[818]A[4]E[407]
cf-2-9x5-w	745,140	95,180	46	E[67]A[9]E[1357]· · ·E[1357]A[9]E[674]
ripple-carry-10-c	292,399	423,084	2	E[29]A[220]E[289368]
ripple-carry-11-c	414,410	601,952	2	E[32]A[264]E[410918]
ripple-carry-12-c	571,099	832,132	2	E[35]A[312]E[567114]
ripple-carry-13-c	768,478	1,122,585	2	E[38]A[364]E[763968]
ripple-carry-14-c	1,013,039	1,482,992	2	E[41]A[420]E[1007972]

Table 1. Some non-trivial instances decided by preliminary QBF reasoning

2. The effectiveness of the tree-reconstruction algorithm (Step 2) on real-world instances.
3. The deductive power of symbolic-only reasoning (Step 4).

Though the main goal of Step 1 is to reduce the formula to an existential-scope-only normal form, its simplification effects are sometimes surprisingly strong. For example, several non-trivial formulas (w.r.t. their size) exist in the test benchmarks that are not beyond the deductive power of the incomplete set of rules adopted. Table 1 shows a few instances from the QBF library that are completely solved during Step 1. Some sparing "monster" instances having more than one million variables (such as the biggest ones in the *ripple-carry* series) were already noticed to be addressable despite their huge size [QBF03]. According to sKizzo's experimental evidences, some families of instances lay in the class of tractable QBF problems, as the inference engine in Step 1 has polynomial complexity.

As far as sKizzo is concerned, the final objective of Step 2 is to reduce the *arity* of the skolem functions that are being introduced. When tree reconstruction is not performed, the prefix we manage is a linearly shaped structure, with a length equal to the number of variables in the instance (i.e. a tree with one single branch). In the worst case, such structure stays untouched after tree reconstruction. But in general, we may expect that the resulting tree is a non-collapsed structure, with more than one branch, and a maximal depth which is lower than the number of variables in the instance. Consequently, we also expect that both the average and the maximal universal depth of existential variables decrease. How relevant are these results over real-world instances?

Table 2 gives a first answer. It compares the depth, average universal depth, and maximal universal depth computed over the linear prefix (the first three columns), against the same values computed on the reconstructed syntactic tree. On these instances, the impact of our reconstruction algorithm is clearly strong, and in some cases even surprisingly strong (see for example the instance-independent shape of the *tree* family).

	Before reconstruction			*After reconstruction*		
Instance	*Var*	*Max ∀-depth*	*Avg ∀-depth*	*Depth*	*Max ∀-depth*	*Avg ∀-depth*
adder-12-sat	2,665	942	804.8	227	80	43.1
adder-12-unsat	2,687	486	277.9	2189	354	242.8
adder-14-sat	3,641	1,281	1,093.5	267	94	50.2
adder-14-unsat	3,667	665	381.1	2,988	483	331.8
adder-16-sat	4,769	1,672	1,426.4	307	108	57.4
adder-16-unsat	4,799	872	500.6	3,911	632	434.6
Adder2-10-c	7,970	445	417.8	670	300	287.8
Adder2-10-s	7,970	545	524.8	98	56	29.5
Adder2-12-c	11,580	642	603.0	957	432	414.5
Adder2-12-s	11,580	786	756.9	116	68	35.3
Adder2-14-c	15,862	875	822.0	1,296	588	564.2
Adder2-14-s	15,862	1,071	1,031.5	134	80	41.2
flipflop-6-c	6864	30	29.9	560	18	17.6
flipflop-7-c	15,213	35	35.0	1,330	21	20.7
flipflop-8-c	30,427	40	40.0	2,824	24	23.8
flipflop-9-c	56,175	45	45.0	5,466	27	26.9
flipflop-10-c	97,272	50	50.0	9,820	30	29.9
flipflop-11-c	159,837	55	55.0	16,610	33	32.9
k-branch-n-20	13822	127	97.9	5568	127	64.3
k-branch-p-19	12544	121	93.2	5063	121	61.3
k-d4-n-16	1438	69	51.7	755	69	35.3
k-d4-p-19	1176	62	45.9	638	62	31.6
k-dum-n-18	885	44	32.2	495	44	22.4
k-dum-p-20	854	41	30.5	469	41	21.4
k-grz-n-18	792	24	17.4	393	24	11.2
k-grz-p-19	767	24	17.7	379	24	11.5
k-lin-n-19	4103	18	11.8	2248	18	8.2
k-lin-p-18	932	12	9.9	430	12	8.4
k-t4p-n-19	2725	123	90.9	1446	122	61.4
k-t4p-p-19	1470	69	50.6	782	68	34.5
k-poly-n-18	1465	110	84.0	926	110	69.1
k-poly-p-17	1384	104	79.4	875	104	65.4
k-path-n-13	937	43	32.1	450	43	22.9
k-path-p-20	1358	61	45.3	645	61	32.0
k-ph-n-21	11131	12	9.7	5347	12	6.5
k-ph-p-20	10444	12	9.7	5067	12	6.4
toilet-a-06-01.11	227	6	3.9	84	6	1.8
toilet-a-06-01.12	247	6	3.9	92	6	1.8
toilet-c-10-05.10	805	4	1.2	498	4	0.5
toilet-c-10-05.12	965	4	1.2	610	4	0.5
toilet-g-15-01.2	80	4	3.2	7	4	1.3
toilet-g-20-01.2	106	5	4.0	8	5	1.7
TOILET7.1.iv.13	400	3	2.2	216	3	1.5
TOILET7.1.iv.14	431	3	2.2	234	3	1.5
TOILET10.1.iv.20	855	4	3.0	457	4	2.0
TOILET16.1.iv.32	2,133	4	3.0	1,117	4	2.0
tree-exa10-10	21	10	10.0	4	2	2.0
tree-exa10-15	31	15	15.0	4	2	2.0
tree-exa10-20	41	20	20.0	4	2	2.0
tree-exa10-25	51	25	25.0	4	2	2.0
tree-exa10-30	61	30	30.0	4	2	2.0

Table 2. The effect of tree-reconstruction over the structure of the syntactic tree

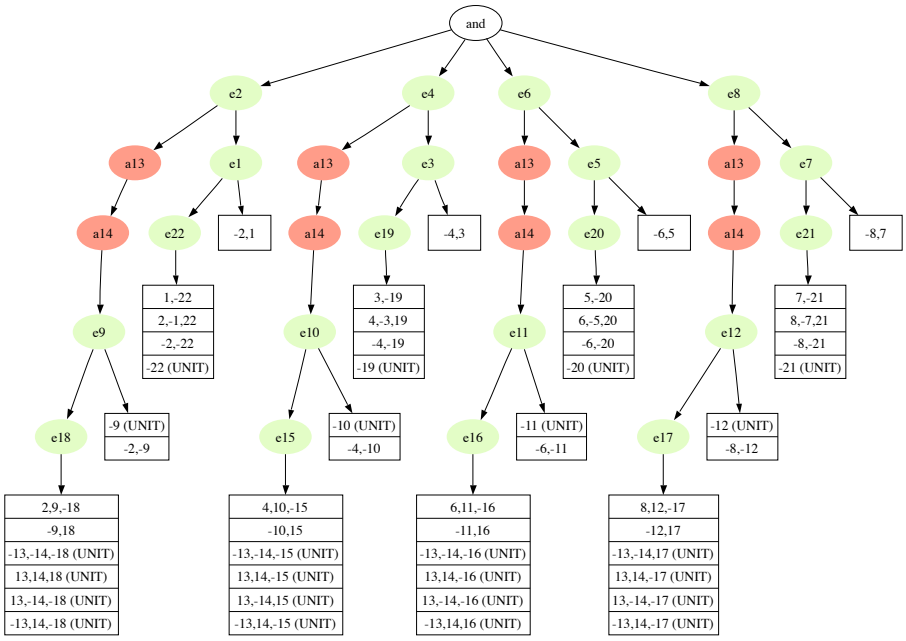

Fig. 7. The reconstructed tree for the instance "*toilet-g-04-01*"

An intuitive way to get the feeling of these effects is to take a look at some tree reconstructed from real-world problems. As we said in Section 4.1, sKizzo is indeed able to produce a textual representation for such trees that can be later on graphically rendered by suited programs (such us *graphviz*). Unfortunately, the trees of all non-trivial instances are too big to be fully represented while keeping readable fonts for clauses and variables (sometimes, they are even too big to be rendered at all). However, the smallest instances fit within our space limitations and retain some interesting features. For example, Figure 7 depicts the reconstructed three of the small "toilet-g-04-01" instance. On the other hand, we can give up the requirement of a complete and readable repre-

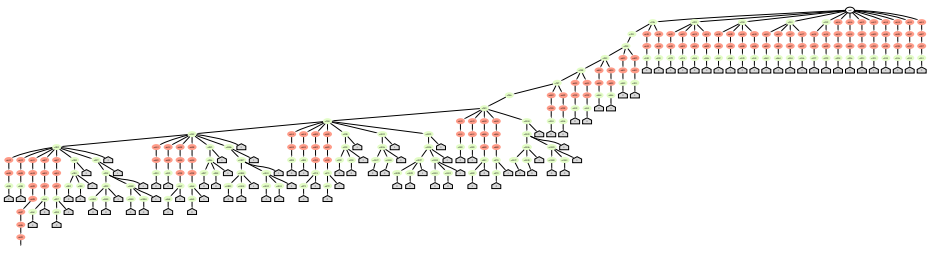

Fig. 8. The compressed, top-most part of the reconstructed tree for the instance "*flipflop-5-c*"

sentation, and just get the overall picture, such us in Figure 8, where only the top-most part of a much bigger tree is compactly represented.

To be fair, we have to say that for other classes of instances the results of our reconstruction are much less impressive than they appear in Table 2. Further investigations on this topic will be conducted.

The last point we consider here is the relative importance of symbolic reasoning with respect to all the other machinery within sKizzo. Table 3 gives a few results that we now discuss to get the feeling of what happens within Step 4. A few instance for each family we are interested in are reported, one instance per row. The first three columns give the symbolic size of the instance (see Section 3.3.2) before and after Step 4 is executed for the first time, together with the relative variation occurred as an effect of such execution. The subsequent three columns report similar information for the ground size of the instances. The last column gives the relative amount of time spent in (the first application of) symbolic reasoning. When this percentage is equal to 100%, the instance is completely solved during Step 4, and subsequent steps are not activated (when the number of remaining clauses is zero, the instance is satisfiable, otherwise it is unsatisfiable). We use a dash when completion time is not known (it is beyond 1000 seconds). When this percentage is less than 100% it measures the cumulative Step 3 + Step 4 time against the total running time for the instance. An important point is that Step 4 is possibly executed several times, according to Algorithm 6. However, we are only measuring the effects of the very first execution against all the rest.

Consistently with Section 3.4.5, we notice that the ground size of instances is always reduced, whilst the symbolic size of some of them is increased as an effect of symbolic reasoning. The reduction ratio for the ground size is quite family-dependent, though not sensibly instance-depending. Most of the simpler families are completely solved by symbolic reasoning. Conversely, for more complex instances symbolic reasoning does not suffices, and in certain cases shows a little effect. For some families (such as "mutex"), the shift from symbolically solvable and unsolvable instances happens *within* the family, moving from the smallest instances to the medium ones. The symbolic/ground size rate strongly depends on the structure of the instances, as we expect from the discussion presented in Section 3.3.1. Within the same family—and for families that grows according to a clearly parametric instance complexity ("adder","counter","mutex",...)—the logaritm of the number of ground clauses grows roughly linearly with the symbolic size. Quite often, the number of ground clauses before symbolic reasoning is intractable (state-of-the-art solvers can afford millions clauses, not billions). Some of them stays unaffordable even after Step 4, but many undergo a strong reduction of the ground size (see the "adder" family, for example). Several problems exist that—thought not strongly reduced during the first execution of Step 4—are hardly simplified during subsequent ones (not shown in the table).

The overall effect of symbolic reasoning is quite incisive. Further experiments will investigate how the solving effort is divided among the various modules of the solver, also taking into account the role of the SAT solver, the time spent in BDD reordering, and the behavior of the divide-et-impera procedure.

	Symbolic clauses			Ground clauses			
Instance	*Before*	*After*	*Diff.*	*Before*	*After*	*Diff.*	Symb. time
Adder2-2-c	234	193	-18%	$1.0 \cdot 10^6$	$5.4 \cdot 10^5$	-46.0%	100%
Adder2-6-s	3,315	2,236	-33%	$1.8 \cdot 10^{12}$	$1.0 \cdot 10^6$	-99.9%	23%
Adder2-8-s	6,060	4,070	-33%	$1.0 \cdot 10^{16}$	$2.2 \cdot 10^7$	-99.9%	14%
Adder2-10-s	9,625	6,447	-33%	$5.3 \cdot 10^{19}$	$4.6 \cdot 10^8$	-99.9%	-
adder-10-sat	3,641	3,580	-2%	$9.9 \cdot 10^{21}$	$1.9 \cdot 10^8$	-99.9%	-
BLOCKS3i.5.4	2,640	2,814	+7%	$4.0 \cdot 10^4$	$3.0 \cdot 10^4$	-25.0%	100%
BLOCKS3ii.5.2	1,886	2,095	+11%	$2.9 \cdot 10^4$	$2.1 \cdot 10^4$	-28.0%	100%
BLOCKS3iii.5	1,226	1,614	+32%	$1.9 \cdot 10^4$	$1.3 \cdot 10^4$	-32.0%	100%
BLOCKS4i.6.4	10,710	12,355	+15%	$1.3 \cdot 10^6$	$1.0 \cdot 10^6$	-23.0%	1%
CHAIN12v.13	486	0	-100%	$1.8 \cdot 10^6$	0	-100.0%	100%
CHAIN17v.18	861	0	-100%	$1.1 \cdot 10^8$	0	-100.0%	100%
CHAIN23v.24	1,443	0	-100%	$1.2 \cdot 10^{10}$	0	-100.0%	100%
cnt04	321	0	-100%	$1.7 \cdot 10^3$	0	-100.0%	100%
cnt04re	397	300	-24%	$2.1 \cdot 10^3$	$3.2 \cdot 10^2$	-85.0%	25%
cnt08	1,237	0	-100%	$6.1 \cdot 10^4$	0	-100.0%	100%
cnt08re	1,309	1,240	-5%	$6.5 \cdot 10^4$	$1.1 \cdot 10^4$	-83.0%	$<$1%
cnt12	2,505	0	-100%	$1.3 \cdot 10^6$	0	-100.0%	100%
cnt12re	2,733	2,820	+3%	$1.5 \cdot 10^6$	$2.6 \cdot 10^5$	-83.0%	$<$1%
flipflop-9-c	74,066	71,691	-3%	$9.4 \cdot 10^{12}$	$9.2 \cdot 10^{12}$	-2.0%	100%
flipflop-10-c	128,245	124,844	-3%	$1.3 \cdot 10^{14}$	$1.3 \cdot 10^{14}$	-1.0%	100%
flipflop-11-c	210,674	205,995	-2%	$1.7 \cdot 10^{15}$	$1.7 \cdot 10^{15}$	-1.0%	100%
impl04	32	0	-100%	$1.4 \cdot 10^2$	0	-100%	100%
impl12	96	0	-100%	$3.7 \cdot 10^4$	0	-100%	100%
impl20	160	0	-100%	$9.4 \cdot 10^6$	0	-100%	100%
k-branch-n-9	12,923	20,608	+59%	$2.1 \cdot 10^{18}$	$1.6 \cdot 10^{18}$	-23.8%	3%
k-branch-p-13	28,676	78,006	+172%	$3.7 \cdot 10^{24}$	$2.9 \cdot 10^{24}$	-21.6%	100%
k-d4-n-16	5,133	5,535	+8%	$4.2 \cdot 10^{22}$	$2.4 \cdot 10^{22}$	-42.9%	2%
k-d4-p-16	2,959	5,044	+70%	$4.7 \cdot 10^{17}$	$3.1 \cdot 10^{17}$	-34.0%	100%
mutex-4-s	362	0	-100%	$1.9 \cdot 10^7$	0	-100%	100%
mutex-8-s	834	367	-56%	$2.9 \cdot 10^{12}$	$3.5 \cdot 10^4$	-99.9%	70%
mutex-16-s	1,778	947	-47%	$2.6 \cdot 10^{22}$	$2.4 \cdot 10^9$	-99.9%	-
TOILET10.1.iv.20	3,466	3,326	-4%	$2.1 \cdot 10^4$	$7.4 \cdot 10^3$	-64.8%	55%
TOILET16.1.iv.32	10,495	8,175	-22%	$5.6 \cdot 10^4$	$8.6 \cdot 10^3$	-84.6%	72%
toilet-a-08-01.11	3,109	1,069	-66%	$6.0 \cdot 10^4$	$2.7 \cdot 10^4$	-55.0%	3%
toilet-c-10-01.14	1,974	1,874	-5%	$7.5 \cdot 10^3$	$4.0 \cdot 10^3$	-46.6%	1%
toilet-g-20-01.2	460	0	-100%	$1.1 \cdot 10^3$	0	-100.0%	100%
tree-exa2-40	51	1	-100%	$5.6 \cdot 10^{14}$	1	-100%	100%
tree-exa10-30	58	0	-100%	58	0	-100%	100%

Table 3. The effect of symbolic reasoning over the size of instances, measured as the number of ground clauses and symbolic clauses before and after Step 4 is executed.

4.4 Performance

As a preliminary performance evaluation, we target some non-random benchmarks from the QBFLIB, and compare both with the SOTA solver (see Section 4.2) and with a few among the best real solvers. In the former case, we address a subset of the test

Test set		SOTA'04		sKizzo	
From family	*instances*	*Solved*	*Time (sec.)*	*Solved*	*Time (sec.)*
adder	8	6	(132.54)	2	(20.18)
blocks	8	8	215.02	7	70.28
chain	8	8	9.25	8	0.45
counter	8	4	(0.05)	6	(1232.29)
flipflop	8	8	3.16	8	81.67
impl	8	8	0.06	8	127.75
jmc-quant	16	6	30.13	6	196.61
k-branch	16	7	(274.93)	9	(994.84)
mutex	7	7	(7.30)	3	(2.31)
szymanski	8	8	(211.20)	2	(1.75)
toilet	8	8	7.75	8	1.23
tree	8	8	4.82	8	0.18
vonN	8	8	16.40	8	16.91

Table 4. Comparison between sKizzo and the SAT'04 SOTA solver

cases employed in [43], in the latter case we target the families described in Section 4.2 (a wider experimentation can be found at [4]).

Table 4 gives the result of our comparison, and is to be interpreted as follows. Each row compares sKizzo with the SOTA solver on a specific test-set. Thirteen test-sets have been considered. Each one is a subset of a family of instances, randomly extracted from that family. The first two columns give the name of the originating family, and the number of instances extracted to build the test-set. To perform a fair comparison, we used the same instances extracted during the QBF evaluation. The values in the third and fourth columns have been reported from [43]. They represent the number of instances solved by the SOTA solver, and the cumulative time taken to solve them. The last two columns give analogous results for sKizzo[8]. Solving times are parenthesized when they are not directly comparable (as the two solvers didn't solve the same number of instances).

Results are quite encouraging. The SOTA solver solves more instances per family than sKizzo in only 4 out of 13 cases. Interestingly, the converse also happens: in 2 cases our algorithm solved more instances than the SOTA solver. In the remaining 7 test-cases, the number of instances solved is the same, so we compare running times. In 4 out of these 7 cases, sKizzo is faster (up to an order of magnitude) than the SOTA. Conversely, in most of the remaining cases the SOTA solver outperforms our procedure by an order of magnitude (with the noticeable exception of the "Impl" family, where the SOTA's advantage is several orders of magnitude wide).

[8] The machines used for experimenting with the two solvers have the same operating systems, type of processor and amount of pshysical memory, but also have a minor difference. Indeed, sKizzo has been run on a 2.6Ghz processor, while the SOTA solver has been tested on a 3.2GHz processor. Running times in Table 4 are not normalized, so we would expect a slightly better performance of sKizzo should it be tested on the very same machine as the SOTA.

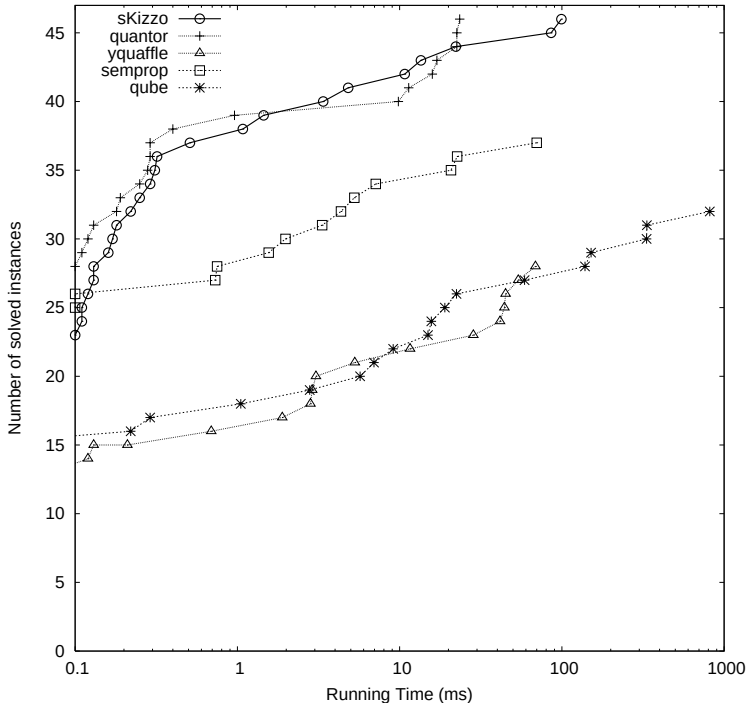

Fig. 9. Comparison over the “rintanen” group of families

Though very preliminary, this experimentation suggests that difficult instances lay in different places for the two solvers. This confirms that the reasoning engine of sKizzo really diparts from those used in other solvers. The test-cases in which sKizzo is dramatically outperformed indirectly point out how it needs to be improved. In this respect, a minor exception is the “adder” family, that we thoroughly re-consider later in this section with surprising results. Conclusively, the comparison with the SOTA solver on the test-cases we considered is fairly good, especially if we take into account that the version of sKizzo we employed is a first, unadjusted implementation of a completely new algorithm.

To directly compare with real solvers, we restrict our attention to the solvers described in Section 4.2. The representation style we adopt for our experimental results is taken from [34, 57, 74], where the number of solved instances is plotted against the (non-cumulative) time taken to solve those instances. So, the y-value of a point in the plot gives the number of runs that didn’t time-out, each one being timed out after an amount of time represented by the x-value of the same point.

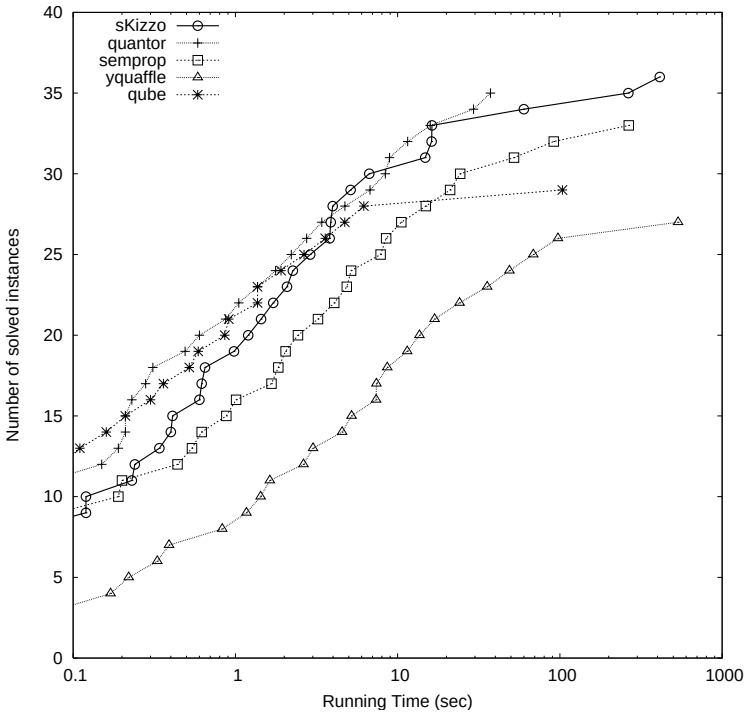

Fig. 10. Comparison over the "ayari" group of families

Figure 9 concerns Rintanen's benchmarks. sKizzo and Quantor were the only two solvers to complete the task (apart from one instance that no solver has been able to decide) within the alloted 1000-second timeout. The performance of sKizzo is aligned to that of Quantor—by far the best solver on this group of families. Both solvers sensibly dominates the rest of the competitors. Quantor finishes its task requiring less time than our algorithm. Interestingly, the instance that contributes the more to this difference with a surprisingly high solving time (almost one hundred seconds on *impl20*) is one that all the other solvers find to be absolutely trivial. The reason for the poor performance of sKizzo over the $impl$ family has been analyzed, and it comes out to be an idiosyncrasy that can be easily worked-around (it will in future releases).

The results on the Ayari's benchmarks are reported in Figure 10. This benchmark is much more difficult than the previous one. Indeed, no solver has been able to solve even half the instances in the group, except for sKizzo which exactly matches this result by solving 36 instances out of 72, immediately followed by Quantor with 35 solved instances. The performance of sKizzo is quite satisfactory on this benchmark, especially if we take into account that some of the instances we didn't solve are easy for other solvers. This give important hints on how to address them more efficiently in fu-

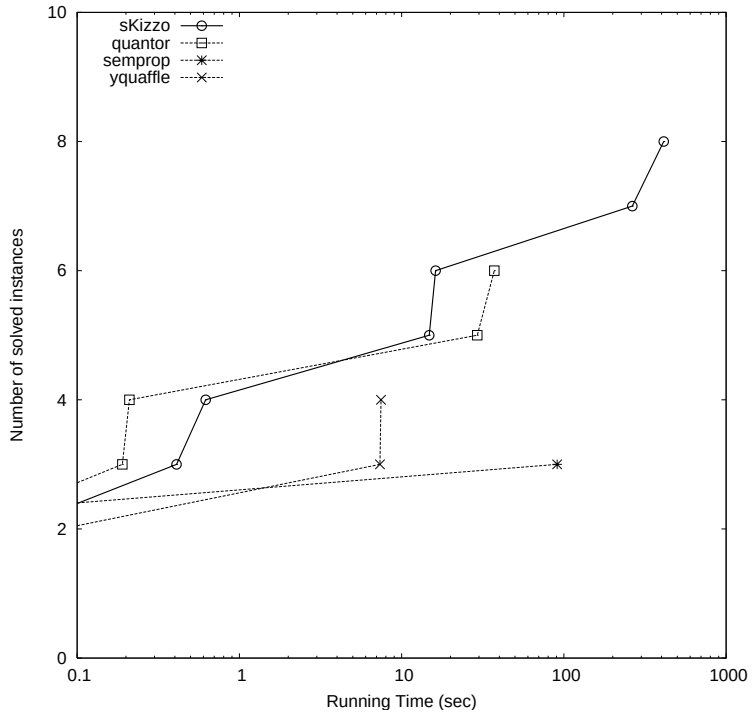

Fig. 11. Comparison over all the satisfiable "adder" instances

ture releases.

The good performance obtained in the Ayari benchmarks led us to reconsider some of the results presented in the comparison with the SOTA solver (Table 4). How is it possible for sKizzo to perform quite well in the whole benchmark while being dramatically outperformed in some of the Ayari families in Table 4?

A first simple answer is that many more solvers than the four we consider here may have contributed to the performance of the SOTA solver. Even if those solvers didn't show an outstanding overall performance in isolation, they might have been able to (very) efficiently solve at least some instances, thus contributing to the remarkable SOTA performance. A second intriguing answer concerns the role of the random sampling over families of instances. In particular, we are interested in the statistical significance of the subset extracted.

To shed a few light over this question, we considered in more detail the "adder" test-set used in Table 4, which is inherited from the QBF04 solver evaluation. At a closer look, it results that most instances in that test set are *unsatisfiable*, even if in the original family 50% of the instances are SAT. Moreover, one of the two SAT instances chosen for the test-set is very complex and no solver has been able to conquer it (nor sKizzo

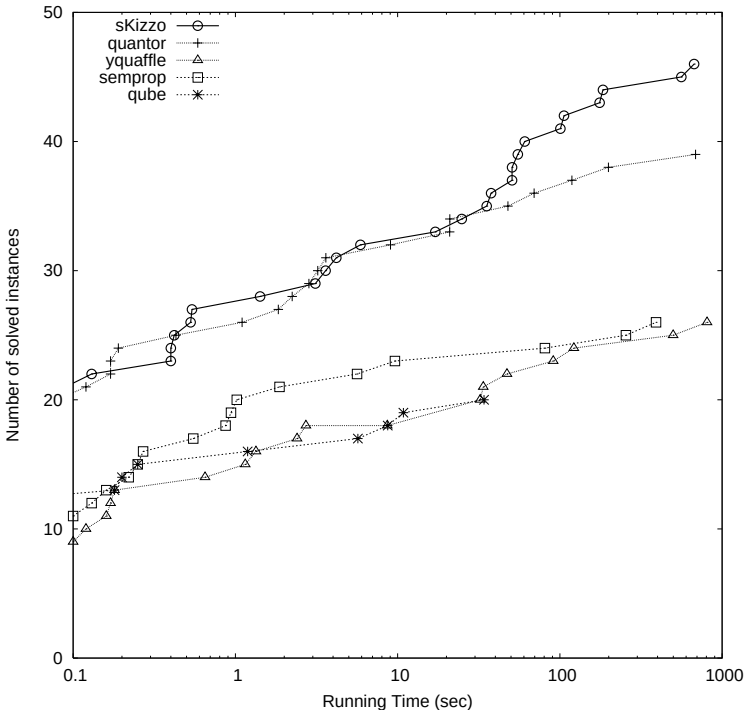

Fig. 12. Comparison over all the "counter" instances

did). If we now consider that—empirically—sKizzo finds satisfiable "adder" instances easier to solve than equal-sized unsatisfiable ones, we have a good explanation for the phenomenon.

Figure 11 compares[9] five solvers on the whole satisfiable subset of the "adder" family[10]. Results are coherent with our analysis.

To conclude this preliminary experimentation, we target the "counters" family of instances. We introduced this family and motivated its relevance in Section 4.2. From our experimentation, and according to what Biere already noticed in [9], it comes out that QBF-based model checking *doesn't* outperform plain SAT-based model checking, at least on the "counter" instances. Within 1 minute, BMC is able to decide 7-bit-counters (with both reset and enabled signals modeled), while the QBF solver *decide* can handle 3-bit-counters, *qube* and *semprop* up to 4 bits, and only *quantor* matches the 7 bits result. sKizzo is just slightly better than the state of the art, with 8-bit solved within 60

[9] All the solvers have been run on the same 2.6GHz, 1GB machine, with a 1000s. timeout.

[10] These instances are by no means *easy*. The five more complex instances resolved by sKizzo in Figure 11 are 2003-hard (no solver was able to solve them in the QBF03 evaluation). Three of them have been conquered later on by Quantor.

seconds. However, it is also able to solve the all 16-element family of counters (with no enable and no reset signal) in less than 5 minutes, and its relative performance gets more interesting as higher running times are considered. Figure 12 compares our algorithm against four state-of-the art solvers, including quantor, which is the best solver on this specific family.

5 Related and future work

5.1 Related work

Most work on QBF algorithms aims at extending the successful solving techniques used in propositional satisfiability to the case of quantified boolean formulas. So, let us briefly recall those techniques at first.

Successful SAT solvers are refined versions of a *search algorithm* that looks for models in the space of *partial truth assignments*. They all build on top of the seminal work [24, 25] by Davis, Putnam, Logemann, and Loveland, where a depth-first, recursive visit of the semantic evaluation tree of the formula [62, 41] is described. This algorithm, often referred to as DPLL procedure, has been improved in several ways over the decades (see [22, 36, 29, 67, 77, 3, 35, 48, 52, 8] for specific examples and [5] for a detailed survey). Many enhancements—such us conflict analysis, failure-driven assertion, non-chronological backtracking, and learning—are *lookback* techniques (information is gathered form past search and then conveniently reused). Others - such us forward checking, forward reasoning and heuristics choices—are lookahead techniques (information concerning the remaining search space is exploited by ad-hoc exploring mechanisms to help the main search procedure doing its work efficiently).

The unquestionable result of all these improvements is that SAT solvers are now regarded as effective tools for solving industrial-scale problems [21], and have been successfully applied to several domains, such us computer-aided design of integrated circuits [39, 42], Planning [38], Model Checking for dynamic systems [10], Scheduling [23], Operations Research, and Cryptography [50], just to name a few.

The QBF language is more expressive than PROP, but it is also more complex to decide [73] (PSPACE instead of NP). Problems arising from Temporal Reasoning [70], Planning [59], Formal Verification [65, 1], Reasoning about Knowledge [55], and two-player games [31] find a more natural (and possibly exponentially more succinct) representation in QBF than in PROP. Hence, the question arises on whether QBF solvers can inherit and perhaps overcome the popularity (and efficiency) of SAT-based methods for such applications. The answer mainly depends on the effectiveness of the tools developed for deciding QBF formulas. Such tools should prove on the field that a way exists to leverage the augmented expressive power of QBF without suffering too much of the worstened worst-case complexity.

Currently, most QBF solvers leverage revised versions of techniques that have been originally introduced for the SAT framework. These techniques range from the extension of resolution-based reasoning [40] to the employment of lookback enhancements [46], encountering along the way a key contribution by Cadoli, Giovanardi and

Schaerf [16] in which the original extension of DPLL to QBFs is presented. Up to a certain point, the extension of SAT techniques to QBF solvers has been successful. In the QBF solver evaluation reported in [44], all the competitive QBF solvers (such us QSAT [60], QSOLVE [27], QUAFFLE [78], QuBE [33], SEMPROP [45]) were search-based.

Yet, the important question we raised in the introduction remains unanswered. Namely: *can QBF-based reasoning really beat SAT-based reasoning*?

Some results [65, 1, 55, 9, 57, 43, 44] suggest that the shift to QBF does not pay off yet. In one such contribution [9], for example, the way unbounded model checking is performed via SAT is compared to the way it could be done via QBF. The results of this preliminary evaluation confirms that current QBF-based model checking may at most match the results obtained by plain BMC. Moreover, as the examples analyzed in [9] are very easy for classical BDD-based model checking, some kind of remarkable improvement may be reasonably expected from alternative QBF solver architectures.

A few alternative solving algorithms for QBF are indeed emerging [43]. Some of them reverse the order in which quantifiers are considered [9] (bottom-up instead of top-down), others [57, 32] employ some kind of compact representation for the problem (usually, OBDD or ZBDD based symbolic representations). Many of them restate the very goal of the solver: it is no longer a matter of *searching for a solution*, rather an attempt to directly *solve the instance*.

Interestingly, the distinction between *searching* and *solving* is quite old, and—in the framework of algorithms for existential propositional reasoning (SAT)—traces back to the two early contributions [24, 25]. Recently, it has been reconsidered from different perspectives [61, 56, 5]. Well known resolution-based solving techniques (see [13] for a survey) have received renewed attention, especially when used in conjunction with compressed representation for clauses [17, 28, 53]. When compared to search-based algorithms for SAT, these so-called *symbolic* approaches show a certain strength on specific classes of instances, but seem to be not competitive in general [56].

Things change in the QBF scenario. Both the idea of *compressed/symbolic* representations, and the shift from *searching* to *solving* seem to be promising [57, 9, 32] as far as QBF is concerned. Possible reasons for this asymmetry discussed in the literature are (1) that symbolic representations manage existential and universal quantifiers in a symmetric way[11], whereas search-based procedures have a hard way with the latter ones, and (2) that QBF problems are more structured and less *combinatorial* in nature than their propositional counterparts.

[11] This is true for the symbolic representations employed so far in the literature. Within sKizzo, the technique used to get rid of universal variables strongly differs from the way existential quantifiers are dealt with. As sKizzo is a quite competitive solver, the prominence of "symmetric hypothesis" is weakened.

Let us now take a closer look at some of these new algorithms for QBF satisfiability, starting with the really noticeable case of Quantor [9]. It is a "young" and simple[12] algorithm that after one year of improvements[13] has over-performed (on non-random benchmarks) all the well-known search-based algorithms [43]. It does not use compressed representation, and adopt a *solving* paradigm. The emphasis is on eliminating quantifiers from the innermost to the outermost, using q-resolution [40] for existential quantifiers and expansion for universal ones. A lot of other details (equivalence reasoning, subsumption control, estimation of expansion and resolution costs, scheduling heuristics) contribute to the impressive overall performance of this algorithm.

ZQSAT is an algorithm that uses the *searching* paradigm, but employs a compressed representation for clauses based upon ZDDs, in the spirit of [17]. It is able to efficiently solve classes of instances that are known to be hard for non-symbolic DPLL-based QBF solvers, and is also able to process QBF formulas in NNF (negated normal forms), which can bring in some cases an exponential benefit (provided suited input formulas are available). QMRES [57] is another algorithm that exploits ZDDs to symbolically represent clauses. Rather than performing a DPLL-like search, this algorithm adopts multi-resolution [17] to *solve* the instance. This yields a quite competitive solver that has been able to tackle previously unsolved instances during the last QBF solver evaluation [43], resulting particularly efficient over high-alternation, structured instances. In the same paper [57], a BDD-based solver—called QBDD—is also introduced, which employ a symbolic quantifier elimination technique and uses BDDs to encode satisfying assignments. Though both algorithms appear to be more scalable than search-based ones, QBDD is in general dominated by QMRES.

To conclude, we give a few pointers to contributions that report about the tools and techniques used by sKizzo. *Structure exploitation* is a somewhat elusive concept. On the one hand, the structure of an instance is always *implicitly* dealt with (just think of the way search-based algorithms visit their prefix, or heuristics compute their preferences). On the other hand, an *explicit* exploitation of the QBF structure is rarely if ever attempted in other approaches. Noticeable exceptions are (1) the QBDD and QMRES algorithms presented in [57] that exploit the *Gaifman graph* of the matrix to decide a good variable ordering for the ordered DD used to perform quantifier elimination, (2) the work presented in [26] to pre-process QBF prefixes and obtain semantically equivalent alternatives with an "optimal" number of quantifier inversions, (3) the generalization from CNF to NNF of the normal form used as input language for QBF formulas, presented in [32], and (4) the notes on tree-like prefixes reported in [9].

The foundational work of Skolem [71], together with the related works of Herbrand and Löwenheim (see [15] for an overview of these works), are at the very basis of au-

[12] As usual, even if the abstract description of an algorithm is "simple", implementation details may be absolutely non-trivial. Practical experience with SAT and QBF solvers suggests that not only implementation is important, but it can be the main responsible for the performance edges. For example, most of the reasons for Chaff efficiency are hidden in the code, not in its abstract description.

[13] A preliminary version of Quantor did already participate in the SAT'03 evaluation with limited success.

tomated deduction for expressive logics, and have had the widest possible application. We here just cite a recent work by Jackson [37] that employ a notion of skolemization similar to ours.

Forms of reasoning about binary sub-formulas are widely known, and regarded as an effective pre-processing step in the propositional framework [11, 30, 8]. Our main source of inspiration for the binary hyper reasoning techniques presented in Section 3.4.3 and 3.4.4 is [2], while the algorithm we use for detecting strongly connected components is due to Kosaraju (unpublished) and Sharir [66] and dates back to 1978.

The interest in binary decision diagrams as a tool for manipulating boolean functions traces back to the seminal work by Bryant [12], and is nowadays so wide that entire monographs on the topic exist (see [76] for a comprehensive account on the field). Their usage in SAT/QBF satisfiability algorithms have been explored at least in [75, 51, 17, 63, 54, 53, 28, 56, 57, 32]. The CUDD package we have used [72] is one of the most widely known, thought many alternatives do exist (see `www.bdd-portal.org` for further details).

5.2 Discussion

We discuss the differences and similarities between our technique and the other approaches to QBF satisfiability reported in Section 5.1.

5.2.1 Solving vs. search. We cited in Section 5.1 some empirical results suggesting a tradeoff between what can be done efficiently by *solving*, and what you'd better perform via *search*, both in the propositional case and in the QBF framework. Some classes of formulas have been recognized to clearly fall into one of these two classes. At the same time, solvers are partitioned between search-based ones (QuBE, SEMPROP, QZSAT, QSOLVE, etc.) and solving-based ones (Quantor, QMRES, QBDD).

sKizzo behaves differently. It tries to obtain the best of both worlds: first, simplify (or decide, if you can) the instance via state-of-the-art symbolic reasoning tools (CUDD); then, face the remaining combinatorial core by means of state-of-the-art, search-based solvers (zCHAFF). The employment of a refutationally incomplete set of rules in Step 4 is not necessarily a drawback for the algorithm. A positive side-effect of this limitation is that those inferences that are easy and effective for the symbolic machinery are left to Step 4. When Step 4 reaches the limits of its deductive power, it is likely to have extracted a *core* of the instance which is more *combinatorial* in nature. Search techniques have proved to be more effective on such instances, and Step 6 indeed address them by means of search-based methods.

These complementary searching and solving behaviors are deeply *interconnected* within sKizzo: Step 1 applies a non-symbolic, solution-oriented, pre-processing reasoning toolset. Step 4 implements a fully symbolic, refutationally incomplete inference procedure. Step 6 applies a non-symbolic, ground, search-based, complete decision method. Finally, Step 5 is itself a search procedure that moves from one node to the other of its search space through symbolic steps, and then resorts to a ground, search-based approach (Step 6) whenever possible.

5.2.2 The symbolic approach. We have reported about several symbolic/compressed representations for CNF formulas and/or propositional models. Very few of them concern QBFs. Namely, the search-based solver ZQSAT, and the solving procedures QMRES and QBDD.

The representation employed within sKizzo differs from all of them. Step 3 indeed exploits the special structure of the propositionally skolemized QBF clauses, i.e. the fact that clauses in the propositionally skolemized version of a QBF instance are not randomly scattered. Rather, they are grouped into "clusters", one for each originating QBF clause. Consequently, our symbolic representation for clauses is not just a decision diagram that represent sets of clauses (or sets of assignments), but a much more articulated data structure. It involves different representation levels for existential and universal variables, and needs to refer to the syntactic tree of the formula to fully expand its ground meaning. As a minor note, we observe that our algorithm is currently the only competitive BDD-based QBF solver, as all the others build on top of a ZBDD-based representation. As opposite to other symbolic approaches, we also notice that our representation doesn't prevent from easily detecting both pure literals and unit clauses.

5.2.3 Structure exploitation. Few solvers attempt to directly leverage the structure of instances. By contrast, sKizzo confers to the word *structure* a central, many-sided role: (1) the solver architecture is articulated in several steps and this allows for investigations on how to classify formulas w.r.t. the stage they are solved in[14], (2) the recovery of some *hidden syntactic structure* for the formula is the main concern of Step 2, (3) the data structure used in Steps 4 are themselves organized according to the prefix of the formula, and (4) the syntactic tree of the formula is used to guide the search during Step 5.

While all of these topics deserve further attention (and an in-depth comparison with the techniques presented in [57, 26, 32]), we here limit our attention to the structure recovery performed during Step 2. A fascinating consequence of Step 2 is that we loose the simple notion of *direction* for the prefix (innermost $\rightarrow$ outermost or outermost $\rightarrow$ innermost), because quantifier alternations no longer show a linear shape. Though in linearly represented prefixes there is only a partial ordering among quantifiers (as variables in the same scope are not ordered) the sequence of scopes is still totally ordered. This total order identifies the two possible directions over the prefix used by most algorithms. When a tree-shaped syntactic structure is used, the set of scopes is only partially ordered, and no monodimensional notion of direction applies. Even more radically, the notion of "order among quantifiers" (which is inescapable in all the other solvers) is simply absent in the reasoning techniques we adopt in Steps 3-4. They indeed abstracts over the presence of multiple, partially ordered quantifiers, as the only place in which quantifier alternations do matter is in the symbolically represented (and atomically ma-

[14] A more general issue exists about constructing solvers that behave efficiently on QBF formulas known to belong to restricted sub-classes, "simpler" than general QBFs. See [14].

nipulated) sets of ground clauses[15]. The partial order among scopes re-gain part of its importance during Step 5.

5.2.4 Divide-et-impera. The decision procedure operated in Step 5 closely resembles the one used in DPLL-like solvers for QBF. However, what we are managing is not a QBF formula, but a (symbolically-represented) tree-shaped propositional formula. As far as splitting over existential variables is concerned, this makes the whole procedure more similar to SAT solvers than to QBF decision procedures. By contrast, when the split is performed over *universal* variables, something conceptually different happens: the instance is partitioned into two completely disjoint sub-formulas, according to expression (11). The splitting operation is performed symbolically, both on universal variables and on existential variables, according to expressions (10) and (11). This is reminiscent of other symbolic approaches to QBF described in Section 5.1.

Unlike most other search-based solvers, the base-case of the procedure is never a direct decision over trivial sub-formulas; well in advance, either symbolic normalization or compilation-to-SAT decide every sub-problem. These techniques may thus be seen as powerful look-ahead tools.

Finally, internal nodes of the quantifier tree generating more than one child induce sets of *independent* subproblems. The whole procedure is named "divide-et-impera" after this feature, which is absent in standard DPLL procedures. Step 2 gives a fundamental contribution towards partitioning subproblems. Should indeed Step 5 work on a linear prefix, it would never be able to disjoin sub-instances.

5.2.5 Memory consumption. In our experimental evaluations, Quantor often timed-out because of memory problems. It is in the very nature of that algorithm to be memory eager: it indeed uses an explicit representation for intermediate formulas derived via q-resolution and expansion. Unfortunately, for large instances, such intermediate representations get so large than the process starts allocating more memory than is physically available[16]. As a consequence, performance falls down. By contrast, search-based solvers usually employ only a fraction of the physical memory, and their time-outs are actual *time*-outs.

sKizzo lays somewhere in the middle: Like quantor, it heavily employs all the available physical memory, and could proportionally benefit from larger physical memories (its performances are expected to scale with memory, an already noticed feature of symbolic procedures [57]). Like DPLL solvers, it never causes the virtual memory system to start managing secondary memory. Obviously, every process can keep on monitoring

[15] At the implementation level, a notion of order among variables may still exist—for example if one employs ordered decision diagrams—though this is a hidden kind of order with absolutely no effect (other than, possibly, performance) on the underlying reasoning procedure.

[16] The large size of such intermediate results strongly resembles what happens with BDD-based computations. Not by chance, the strong memory requirements of sKizzo also originate form the BDD-based representation of the intermediate clause sets managed during symbolic reasoning. These intermediate sets can indeed be interpreted as the symbolic, all-at-once counterpart to the step-by-step, ground approach used by Quantor. See Section 3.4.5.

its own memory consumption and just stop working when it is getting too large. However, sKizzo is able to achieve this result without being forced to give up the solving process. It will just take more time.

5.3 Future work

Our future work on sKizzo mainly aims at improving the algorithm and the implementation, both of which show a lot of room for improvements. We now briefly point out where such space lays, distinguishing among (1) some implementation-related improvements, (2) the introduction of already known techniques not yet exploited within our solver, (3) the addition of new features, and (4) some applicative perspectives.

5.3.1 Implementation. sKizzo$^{v0.1}$ is a first implementation of a completely new algorithm. As such, it lacks a lot of optimizations and a careful implementation-level engineering. Actual bottelnecks in the whole process are still to be discovered and possibly removed. Some aspects of sKizzo$^{v0.1}$'s internals suffer from the underlying theory being developed at the same time of the implementation[17]. With few exceptions, simple-minded data structures are used. Many of them should be redesigned to pursue efficiency. For example, techniques to perform fast binary constraint propagation are very effective in the purely propositional framework. They could be easily lifted to our case.

There are several parameters to tune in the two linked libraries. Nothing has yet been attempted in this respect. The decision-diagram package, for example, needs an initial variable order (known to possibly have a dramatic impact on the overall performance), the choice of an algorithm for dynamic reordering[18], the control of the size of computed tables and allocated memory, and so on. Relevant effects on the overall performance may be expected, since a large fraction of the whole running time on complex instances is spent on either BDD-related operations or variable reordering. Moreover, all the competitive DD-based QBF solvers reported so far employ ZDDs rather than BDDs. sKizzo encapsulate the interface towards DDs in a dedicated package, so that the replacement of one package with another is relatively easy.

In addition to this, there are alternative (and quite different) versions of such libraries that worth the case to be considered. For example, a lot of other efficient SAT solvers (BerkMin, siege, Jerusat, ...) could be plugged into the modular architecture of sKizzo (where a module exists to abstract over the specific solver employed). Deci-

[17] The first implementation core of sKizzo was written six months ago, when very few ideas of those reported in Section 3 and Section 4.1 were already clearly stated.

[18] Preliminary experimentations show—as expected—a complex tradeoff between the time spent by sKizzo performing variable reordering and the efficiency of the operations on the resulting representation, also depending on the reordering algorithm employed.

sion diagram packages other than the CUDD should also be tested[19] (see the web page `www.bdd-portal.org` for a comprehensive list of possibilities).

Beyond the effective setup of libraries' parameters, a general tuning activity is also necessary for the solver itself. For example, the strategies to control resource consumption presented in Section 4.1 are powerful tools that still deserve attention to reach their potential.

5.3.2 Exploiting known techniques. At present, sKizzo employs almost none of the enhancements that most other solvers heavily exploit. For example:

- No *heuristics* is employed, even though there are a lot of heuristics decisions to be taken. Heuristics may greatly help our solver in reaching earlier conclusions. The place currently missing heuristics the most is Step 5 (divide-et-impera). The problem of choosing the next variable to branch on (and possibly the truth value to be firstly assigned to that variable) is a sensible problem for DPLL-like algorithms. A tradeoff exists between the complexity of deciding which literal is to be selected and the pruning ability of the resulting choice w.r.t. the size of the search tree actually explored. At one extreme, one could randomly choose the next variable, thus consuming no time in the selection decision. At the other extreme, one could solve the problem of finding the variable ordering and truth value assignment which minimize the number of nodes subsequently explored by the search procedure. However, this problem is even harder [49] than the SAT problem itself, so one resorts to approximate decisions. Comprehensive studies of the effect of different heuristics on the performance of purely existential solvers can be found in [69, 34]. Well known examples of heuristics are the *Maximum Occurrences Minimal Size* (MOMS) [29, 58, 22], the combined rule presented in [77], the *Unit Propagation Lookahead* (UPL) [47, 8], and the *Variable State Independent Decaying Sum* (VSIDS) [52]. Adaptations of such heuristics to the quantified case have to deal with the partial order among variables induced by the prefix. Most search-based QBF solvers leverage adapted SAT heuristics (see [43]).

- No form of *learning* is employed, other than the one possibly performed by the SAT solver as a black box. However, at least four kinds of learning may be introduced: (a) learning of failures/successes of the divide-et-impera procedure, resulting in the insertion of new symbolic clauses; (b) learning of the optimal instance size to switch from the symbolic reasoning to the search-based behaviour; (c) learning of the optimal tradeoff between splitting the instance at hand into more pieces and giving it as a whole to the SAT solver; as the absolute ground size of an instance gives no clear indication on its hardness, the divide-et-impera procedure should learn from past SAT instances generated for the same QBF instance; a preliminary

[19] While the overall performance of different DD packages is usually comparable, significant discrepancies may emerge when only a small subset of the operations over BDDs is of interest (this is the case for sKizzo). In such cases, design choices such as the employment of a pointer-based vs. an index-based representation, or the constant-time negation via pointers labeling could make a strong difference.

version of this mechanism is already implemented in sKizzo$^{v0.1}$; we are also studying a method based upon the structural analysis of the matrix; (d) the clauses learned by the SAT solver during one run may be helpful in subsequent runs over the same QBF instances, provided the whole generate-and-solve procedure is made *incremental*. We are working on this improvement (see also Section 5.3.3).

– A major disadvantage of the divide-et-impera procedure presented in Section 3.5 is that it implicitly relies on chronological backtracking, that is: truth values for variables are assigned/unassigned following a *last-in, first-out* policy. So, whichever the reason the algorithm has to backtrack, it can only backtrack on the (chronologically) last assignment. *Conflict-directed backjumping* (sometimes called intelligent backtracking) is used in DPLL-like solvers to overcome such limitation. It has been originally introduced for the SAT case [68, 3], and then extended to QBF. When a contradiction is detected, the backjumping engine computes the strictly necessary subset of the current partial assignment which is responsible for the contradiction to arise. The value of the most recently assigned variable appearing in the contradiction has to be changed (alternative schemes from CSP feature hypotheses' reordering). In the QBF case, additional complications have to be dealt with, as working hypotheses may be either universally or existentially quantified, thus playing quite different roles in backjumping. However, Step 5 deals with a purely existential scenario. In general, a contradiction is detected in Step 6 by the SAT solver, and then forwarded to Step 5, where it becomes necessary to infer which symbolic assignments are responsible for that ground contradiction to arise. We are working on one such scheme, which—as in the usual propositional case—can also be extended to *learn* symbolic clauses that will further reduce redundancy.

– *Trivial truth* and *trivial falsity* checks are not performed. These techniques have been proved to be quite effective since early contributions on QBF [16]. As far as sKizzo is concerned, the check for trivial truth amounts to remove all the universal nodes from the quantifier tree, perform a complete for-all reduction, and then test for satisfiability the resulting existential instance. Should it come out to be satisfiable, the original QBF would be guaranteed to be satisfiable as well (the Skolem functions do not actually depend on their arguments: they are *constant*). Conversely, trivial falsity would follow from the unsatisfiablity of the subset of clauses only made up by existential literals.

– *Subsumption control* is not performed. As soon as a new clause is produced by some inference step, a *backward subsumption* control can be performed. It amounts to remove all the (already present) clauses that are subsumed by the just added clause. The converse operation, i.e.: *forward subsumption* control, amounts to avoid adding a clause if it is subsumed by some already present clause. The benefits of these methods have to be carefully weighted against the time they consume. Anyway, they are much more relevant to solution-based methods than to search-based solvers. sKizzo might exploit one such method at three levels: in the original QBF framework, in the intermediate symbolic representation, and in the final ground

reasoning. The second one is by far the more attractive, also considering that many of the inference rules presented in Section 3.4 produce redundant clauses.

5.3.3 New features. One obvious way of strengthening our approach is to augment the inference power of Step 1 and/or Step 4 by adding new inference rules. There are a lot of interesting candidates, though none of them have been extensively tested yet (beyond the seven rules already employed). As an example, some limited form of q-resolution could be implemented during Step 1 to remove certain existentially quantified variables, a la Quantor[20].

The SAT instances handled in Step 6 are not unrelated to one another. We are working on a very natural way of producing an incremental encoding and an *incremental resolution* for such instances. It would allow to exploit the incremental solving capability of state-of-the-art propositional solver, like zChaff.

Another interesting challenge for QBF solvers is to produce and manage compact certificates for their SAT/UNSAT answers w.r.t. a specific instance. The simplest forms of certificates are: for a SAT answer, a model of the QBF instance; for an UNSAT answer, a somewhat solver-dependent representation of a (minimal) sequence of inference steps deriving the empty clause from a (minimally) unsatisfiable subset of the input formula. Once the representation for certificates is decided, a piece of software can be written (independently of any particular solver) to verify their validity. In this respect, sKizzo has the advantage that it compactly represents its potential certificates in a native way. We are working on the model verifier "ozziKs" that takes such a certificate as input and verify its validity.

5.3.4 Applications. On the applicative side, it would be interesting to *connect* our solver to a real-world model checker. In particular, we have some experience [7, 6] in modifying/extending the module of NuSMV[21] devoted to instance generation for SAT-based model checking.

Following the guidelines reported in [64, 73, 60], it would be possible to extend the purely propositional generation mechanism employed in the BMC module to produce quantified boolean formulas[22]. Thereafter, the whole collection of SMV modules

[20] Quantor employs q-resolution as a core inference rule to decide the instance, together with universal quantifier expansions. Conversely, q-resolution should be just a pre-processing step for sKizzo with no commitment to its usage when the enlargement of the formula is not worth its cost. For example, its application could be limited to remove those variables—if any—that do not increase the size of the formula, or to eliminate those variables that imply a significant reduction in the projected ground size of the instance. This happens whenever in both resolvents the two deepest existential literals have some universal quantifications in between them along the quantifier tree, and resolution is performed over the deepest variable.

[21] NuSMV [19, 20, 18] is a state-of-the-art symbolic model checker that integrates BDD-based and SAT-based model checking techniques on the whole input language. It has been used for the verification of industrial designs, as a core for custom verification tools, and as a testbed for formal verification techniques.

[22] On the practical side, the embedding of sKizzo into NuSMV would be quite natural as this model checker already integrates the CUDD package (for unbounded symbolic model check-

available for NuSMV and related model checkers would be automatically translated into QBF instances. Moreover, we would have a roboust, industry-standard platform to extensively compare SAT solvers and QBF solvers on model checking instances.

6 Conclusions

We have introduced a novel algorithm for evaluating quantified boolean formulas. The originality of our approach is twofold. On the one hand, a decisively new approach to quantified reasoning is introduced. It amounts to reassess quantified reasoning as a quantifier-free reasoning in a purposely designed symbolic representation. On the other hand, numerous contributions to automated reasoning have been rearranged within a coherent framework. This is both advantageous and instrumental in realizing the above mentioned symbolic approach.

Our work is motivated by the outstanding potential of quantified reasoning in applications. Advances in decision procedures for this formalism are ardently expected, as the only missing step toward making that language irresistibly attractive is to invent more efficient decision procedures.

The conviction is growing among researchers that the expressive power of quantification is not necessarily a shortcoming as far as decision procedures are concerned. In this respect, sKizzo firstly succeeds to show how to retain both the expressive power of quantification and the strength of all the known solving techniques for propositional reasoning. Our preliminary experimental evaluation indeed yields remarkable results. In addition, the more we realize how wide the room for improvement is, the more those results sound promising.

Ours and other recent developments witness the youthfulness of the field. Notwithstanding their early stage, these approaches show the seeds of success. Thus, the looming possibility of constructing quantified reasoners worhty of inheriting the amazing success of SAT solvers in applications largely promotes further investigations along sKizzo's guidelines.

Acknowledgements

I thank Gigina Aiello for so many reasons I cannot even mention here, and Paolo Traverso for faithfully supporting my research efforts. I'm grateful to Sara Bernardini for the many days she has spent on listening to my early ideas about sKizzo. I also thank Marco Cadoli for several helpful advices on technical issues and for being so kind in discussing this work with me. Luciano Serafini read a preliminary version of this work and provided a lot of useful comments. Finally, Amedeo Cesta deserves a special thought for his indefatigable and strong encouragement.

ing) and the zChaff solver (for bounded model checking). Moreover, we would benefit from its pre-processing capabilities, that include parsing of SMV models, flattening, boolean encoding, predicate encoding, and cone of influence reduction (see [20]).

References

1. A. Ayari and D. Basin. Bounded Model Construction for Monadic Second-order Logics. In *Proceedings of CAV'00*, 2000.
2. F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *Proceedings of SAT'03*, 2003.
3. R. Bayardo and R. Schrag. Using CSP look-back rechniques to solve real-world SAT instances. *Proceedings of AAAI97*, 1997.
4. M. Benedetti. Web site of skizzo, `sra.itc.it/people/benedetti/sKizzo`, 2004.
5. M. Benedetti. *Bridging Refutation and Search in Propositional Satisfiability*. PhD thesis, Dipartimento Informatica e Sistemistica, Università "La Sapienza", Roma, 2001.
6. M. Benedetti and S. Bernardini. Incremental Compilation-to-SAT Procedures. In *Proceedings of SAT 2004*, 2004.
7. M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In *Proceedings of TACAS 2003*, number 2619 in LNCS, pages 18–33, 2003.
8. D. Le Berre. Exploiting the real power of unit propagation lookahead. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier Science Publishers, 2001.
9. A. Biere. Resolve and Expand. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 238–246, 2004.
10. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of Design Automation Conference*, volume 1579, pages 193–207, 1999.
11. R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 515—522, 2001.
12. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computing*, C-35(8):677–691, 1986.
13. H. K. Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.
14. H. K. Büning and X. Zhao. On Models for Quantified Boolean Formulas. In *Proceedings of SAT'04*, 2004.
15. S. N. Burris. Clarification: Skolem, available on-line at `www.thoralf.uwaterloo.ca/htdocs/scav/skolem/skolem.html`.
16. Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 262–267. American Association for Artificial Intelligence, 1998.
17. P. Chatalic and L. Simon. Multi-Resolution on compressed sets of clauses. In *Proceedings of the Twelfth International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000.
18. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of Int.nl Conf. on Computer-Aided Verification (CAV 2002)*, 2002.
19. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS, pages 495–499, 1999.
20. A. Cimatti, E. Giunchiglia, M. Roveri, M. Pistore, R. Sebastiani, and A. Tacchella. Integrating BDD-based and SAT-based Symbolic Model Checking. In *Proceeding of 4th International Workshop on Frontiers of Combining Systems (FroCoS'2002)*, 2002.
21. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. vardi. Benefits of bounded model checking at an industrial setting. *LNCS*, pages 436–453.

22. J. Crawford and L. Auton. Experimental Results on the Cross Over point in Random 3-SAT. *Artificial Intelligence*, 81, 1996.
23. J. M. Crawford and A. D. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of 12th AAAI '94*, pages 1092–1097, 1994.
24. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5:394–397, 1962.
25. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7, 1960.
26. U. Egly, H. Tompits, and S. Woltran. On Quantifier Shifting for Quantified Boolean Formulas. In *Proceedings of the SAT-02 Workshop on Theory and Applications of Quantified Boolean Formulas (QBF-02)*, pages 48–61, 2002.
27. R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 285–290, 2000.
28. J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. Vanfleet. SBSAT: a state-based, BDD-based satisfiability solver. In *Proceedings of SAT'03*, 2003.
29. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, The University of Pennsylvania, 1995.
30. A. Van Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1996.
31. Ian Gent and Andrew Rowley. Encoding connect-4 using quantified boolean formulae. Technical Report APES-68-2003, APES Research Group, July 2003. Available from `(http://www.dcs.st-and.ac.uk/~apes/apesreports.html)`.
32. M. GhasemZadeh, V. Klotz, and C. Meinel. ZQSAT: A QSAT Solver based on Zero-suppressed Binary Decision Diagrams, available online at `www.informatik.uni-trier.de/TI/bdd-research/zqsat/zqsat.html`, 2004.
33. E. Giunchiglia, M. Narizzano, and A. Tacchella. QuBE: A system for deciding Quantified Boolean Formulas Satisfiability. In *Proc. of the International Joint Conference on Automated Reasoning (IJCAR'2001)*, 2001.
34. Enrico Giunchiglia, Massimo Maratea, Armando Tacchella, and Davide Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of the First International Joint Conference on Automated Reasoning*, pages 347–363. Springer-Verlag, 2001.
35. Jan Friso Groote and Joost P. Warners. The propositional formula checker heerhugo. *JAR*, 24:101–125, 1999.
36. M. Stickel H. Zhang. Implementing Davis-Putnam's method by tries. Technical report, The University of Iowa, 1994.
37. Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139. ACM Press, 2000.
38. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI 1992*, pages 359–363.
39. J. Kim, J. Whittemore, J. P. M. Silva, and K. A. Sakallah. On Applying Incremental Satisfiability to Delay Fault Problem. In *Proc. of DATE 2000*, pages 380–384, 2000.
40. H. Kleine-Buning, M. Karpinski, and A. Flogel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
41. R. Kowalski and P. Hayes. Semantic Trees in Automated Theorem Proving. *Machine Intelligence*, 4:87–101, 1969.
42. T. Larrabee. Test pattern generation using boolean satisfiability. In *IEEE Transaction on Computer-aided Design*, pages 4–15, 1992.

43. D. Le Berre, M. Narizzano, L. Simon, and A. Tacchella. Second QBF solvers evaluation, avaliable on-line at `www.qbflib.org`, 2004.
44. D. Le Berre, L. Simon, and A. Tacchella. Challenges in the QBF areana: the SAT'03 evaluation of QBF solvers, avaliable on-line at `www.qbflib.org`, 2003.
45. R. Letz. Advances in Decision Procedures for Quantified Boolean Formulas. In *Proceedings of the First International Workshop on Quantified Boolean Formulae (QBF'01)*, pages 55–64, 2001.
46. Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 160–175. Springer-Verlag, 2002.
47. C. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI-97*, pages 366–371, 1997.
48. Chu-Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *proceedings of AAAI-2000*, pages 291–296, 2000.
49. P. Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial Intelligence*, 1-2(116):315–326, 2000.
50. F. Massacci and L. Marraro. Logical Cryptanalysis as a SAT Problem. *Journal of Automated Reasoning*, 24, 2000.
51. S. Minato. *Binary Decision Diagrams and Applications to VLSI CAD*. Kluwer, 1996.
52. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *proceedings of the 38th Design Automation Conference*, 2001.
53. D. B. Motter and I. L. Markov. A compressed, breadth-first search for satisfiability. *LNCS*, 2409:29–42, 2002.
54. G. Pan, U. Sattler, and M.Y. Vardi. BDD-based decision procedures for K. *LNAI*, 2392:16–30, 2002.
55. G. Pan and M.Y. Vardi. Optimizing a symbolic modal solver. In *Proceedings of CADE 2003*, 2003.
56. G. Pan and M.Y. Vardi. Search vs. Symbolic Techniques in Satisfiability Solving. In *Proceedings of SAT 2004*, 2004.
57. G. Pan and M.Y. Vardi. Symbolic Decision Procedures for QBF. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP04)*, 2004.
58. Daniele Pretolani. *Satisfiability and hypergraphs*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.
59. J. Rintanen. Construction Conditional Plans by a Theorem-prover. *Journal of A. I. Research*, pages 323–352, 1999.
60. J. Rintanen. Partial implicit unfolding in the davis-putnam procedure for quantified boolean formulae. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01)*, 2001.
61. I. Rish and R. Dechter. Resolution versus search: Two strategies for sat. In I. Gent et al., editor, *SAT2000*, pages 215–259. IOS Press, 2000.
62. J. A. Robinson. The Generalized Resolution Principle. *Machine Intelligence*, 3:77–94, 1968.
63. A. San Miguel Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In *Proceedings of the 7th International Conference on Principle and Practice of Constraint Programming*, pages 121–136. Springer-Verlag, 2001.
64. W. J. Savitch. Relation between nondeterministic and deterministic tape complexity. *Journal of Computer and System Sciences*, 4, 1970.
65. C. Scholl and B. Becker. Checking equivalence for partial implementation. Technical report, Institute of Computer Science, Albert-Ludwigs University, 2000.
66. M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 1(7):67–72, 1981.

67. J. P. Silva and K. A. Sakallah. Grasp: a new search algorithm for satisfiability. *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 220–226, 1996.
68. J. P. Marques Silva. An overview of backtrack search satisfiability algorithms. In *Fifth International Symposium on Artificial Intelligence and Mathematics*, 1998.
69. J. P. Marques Silva. The impact of Branching Heuristic in Propositional Satisfiability Algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, 1999.
70. A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logic. *Journal of the ACM*, 32:733–749, 1985.
71. Th. Skolem. Über die mathematische Logik. *NMT*, 10:125–142, 1928.
72. Fabio Somenzi. CUDD: Colorado University Binary Decision Diagrams, available online at `vlsi.colorado.edu/~fabio/CUDD`, 1995.
73. L. J. Stockmeyer and A. R. Meyer. Word Problems Requiring Exponential Time. In *In 5th Annual ACM Symposium on the Theory of Computing*, 1973.
74. G. Sutcliffe and C. Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.
75. T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J. P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845, pages 34–49. Springer-Verlag Inc, 1994.
76. Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. Monographs on Discrete Mathematics and Applications. SIAM, 2000.
77. H. Zhang. Sato: An efficient propositional prover. *Proceedings of 14th International Conference on Automated Deduction*, pages 272–275, 1997.
78. L. Zhang and S. Malik. Towards Symmetric Treatment of Conflicts And Satisfaction in Quantified Boolean Satisfiability Solver. In *Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*, 2002.