

Implementing Tableau Calculi Using BDDs: BDDTab System Description

Rajeev Goré, Kerry Olesen, Jimmy Thomson

Research School of Computer Science, Australian National University

Abstract. We present a modification of the DPLL-based approach to decide modal satisfiability where we substitute DPLL by BDDs. We demonstrate our method by implementing the standard tableau calculi for automated reasoning in propositional modal logics K and S4, along with extensions to the multiple modalities of $\mathcal{ALC}$. We evaluate our implementation of such a reasoner using several K and S4 benchmark sets, as well as some $\mathcal{ALC}$ ontologies. We show, with comparison to FaCT++, InKreSAT and *SAT, that it can compete with other state of the art methods of reasoning in propositional modal logic. We also discuss how this technique extends to tableau for other propositional logics.

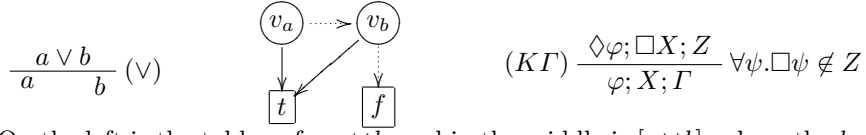
Introduction. Many approaches have been proposed to decide satisfiability and validity in various non-classical logics. In the eager approach the entire problem is encoded into SAT and solved with a SAT-solver, either in one phase [10] or incrementally [7] (e.g. InKreSAT). In the DPLL-based approach [3], DPLL is used to provide propositionally satisfying assignments, which are recursively checked for modal consistency in a master-slave arrangement (e.g. *SAT). Our approach, BDDTab, is a modification on the DPLL-based approach, where we use BDDs instead of DPLL. BDDs have also been used to implement the finite property method directly [4], which is unrelated to the approach we present here.

BDDTab is written in C++, using the BuDDy [2] BDD library. Source and all benchmarks can be found at <http://users.cecs.anu.edu.au/~rpg/BDDTab>

We use a common mapping $[\cdot]$ from propositional modal formulae to BDDs:

$$\begin{aligned} [p] &= \langle v_p, \text{true-node}, \text{false-node} \rangle & [\varphi \vee \psi] &= [\varphi] \vee [\psi] \\ [\neg\varphi] &= \neg[\varphi] & [\Diamond\varphi] &= \neg[\Box\neg\varphi] \\ [\varphi \wedge \psi] &= [\varphi] \wedge [\psi] & [\Box\varphi] &= \langle v_{\Box\varphi}, \text{true-node}, \text{false-node} \rangle \end{aligned}$$

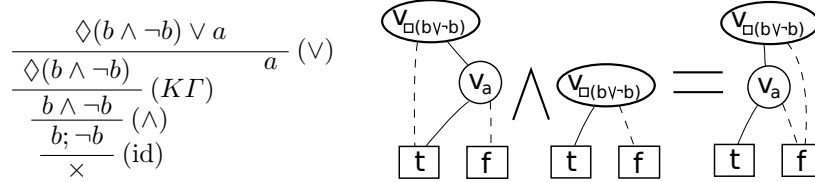
This mapping is both purely propositional and modally shallow: each modal formula $\Box\varphi$ is represented by a BDD variable $v_{\Box\varphi}$, and the mapping “stops” when it reaches a modal formula. We treat modal formulae as if they were atomic propositions by allocating them a BDD variable $v_{\Box\varphi}$. As constructed, the set of all valuations on which $[\varphi]$ is true represents all the open branches of a saturated classical propositional logic (CPL) tableau for φ . However, the correspondence between valuations and tableau leaves is not quite one-to-one, as we demonstrate next.



On the left is the tableau for $a \vee b$, and in the middle is $[a \vee b]$, where the *low* branch is a dashed line, and the *high* branch is a solid line. Each tableau branch has only one of the atomic propositions, but $[a \vee b]$, with variable order $v_a < v_b$, has the valuation $a = f$, $b = t$, which would be equivalent to the tableau node $\neg a; b$. Thus using BDDs can involve a kind of semantic branching. This is not limited to the local branching demonstrated above; every satisfying valuation in a BDD is mutually disjoint with every other satisfying valuation in that BDD, so the branches in its representation of a saturated tableau are mutually disjoint.

Method. Our method is basically a standard tableau procedure, but we use BDDs to perform the saturation phases, as described above. Thus we compute a CPL saturation phase using BDDs, get a branch from the saturation, then compute modal jumps via $(K\Gamma)$ (shown above) as necessary, where Γ is a finite set of global assumptions (TBox). Branches are explored recursively, depth-first.

An important part of our method is when a modal jump closes, and a new OR-branch needs to be explored. Instead of traversing the BDD for the next satisfying valuation, we refine the BDD to remove the closed leaf, and then ask for a new one. The refinement is of the form $BDD_{final} = BDD_{initial} \wedge \neg(v_0 \wedge v_1 \wedge \dots)$, where v_i are some of the variables of the branch that closed. This process is analogous to clause learning in SAT procedures.



Above left is the tableau for $\{\Diamond(b \wedge \neg b) \vee a\}$. The first BDD on the right is $[\Diamond(b \wedge \neg b) \vee a]$. In the tableau, exploring the modal jump over $\Diamond(b \wedge \neg b)$ will close, so we would refine $[\Diamond(b \wedge \neg b) \vee a]$ with the second BDD, giving the third BDD, from which we would get a new satisfying valuation, and continue the procedure.

This process has the potential to eliminate many branches at a time. For instance, if a in the previous example were some more complicated formula, we would have also eliminated every other branch that included $\Diamond(b \wedge \neg b)$.

The flip-side of this is that the refinement process may introduce new variables to existing branches. In the above example, every other branch will now explicitly include $\neg\Diamond(b \wedge \neg b)$, even if it didn't mention it previously. If these new variables are "irrelevant" to the branch, they may cause a net increase in the cost of exploring the branch. We refer to this problem as "**irrelevant variables**."

Optimisations. Responsible Variables: Given the choice of refining with $\neg(v_0)$ or $\neg(v_0 \wedge v_1)$, we prefer $\neg(v_0)$ because refining with $\neg(v_0)$ would eliminate at least as many branches as $\neg(v_0 \wedge v_1)$. Thus a significant optimisation of this algorithm

is to find small subsets of modal variables over which to refine. We recursively track sets of variables that we deem “responsible” for any *unsatisfiable* result.

This method for minimising the set v_1, v_2 is like modal back-jumping [6], and is the BDD counterpart of the learning technique from standard lazy SMT [1], but we use only an approximation of the dependency graph, as explained next.

The base case is when a modal jump closes immediately because the subsequent saturation returned the false-bdd. In this case we can (relatively cheaply) incrementally reconstruct the saturation phase from the modal variables, and explicitly determine a small subset that still yields the false-bdd.

In general, we will only have the responsible variables from beyond the jump. From this, we can only approximate which modal variables were responsible for those variables, as the details are hidden in the BDDs. Our approximation is to consider all the subformulae of a modal variable. If one of its subformulae is a responsible variable, then we consider it responsible, but we must also consider other modal variables with common subformulae, and so on for those variables.

$$\frac{\frac{\frac{\vdots}{\Diamond(a \wedge \neg a); \neg b} \text{ (jump 2)}}{\vdots} \text{ (id)}}{\Diamond(b \vee \Diamond(a \wedge \neg a)); \Box(\neg b) \text{ (jump 1)}}$$

Consider the above example in K. Jump 2 closes immediately due to $a \wedge \neg a$. This is our base case, and we would find that $v_{\Diamond(a \wedge \neg a)}$ is the only responsible variable. As there is only one branch, $v_{\Diamond(a \wedge \neg a)}$ would be returned to jump 1. We would then identify $v_{\Diamond(b \vee \Diamond(a \wedge \neg a))}$ as responsible, because its subformula $\Diamond(a \wedge \neg a)$ was returned as a responsible variable from beyond jump 1. We would also identify $v_{\Box(\neg b)}$ as responsible, as it shares the subformula b with $v_{\Diamond(b \vee \Diamond(a \wedge \neg a))}$.

Each BDD corresponds to a saturation phase. If the saturation phase contains an L-satisfiable branch, then we say that the corresponding BDD is L-satisfiable.

Sat Cache: We use a global cache of L-satisfiable BDDs. A new saturation (BDD) is immediately flagged as L-satisfiable if the cache already contains it (as a unique integer). The cache size is limited by a first-in-first-out removal policy.

Unsat Cache: We use a global cache of L-unsatisfiable modal jumps (the refinement BDD $\neg(v_0 \wedge v_1 \wedge \dots)$ is cached). When creating a saturation, we iterate through this cache, and if the variables in a cached refinement are a subset of the variables of the saturation, we apply that refinement to the saturation. The subset check aids in avoiding introducing irrelevant variables. This cache is also size-limited, with a first-in-first-out removal policy.

Consider again the example shown under **Method**. We made a refinement of $[\neg\Diamond(b \wedge \neg b)]$, which we would cache. So if a future modal jump includes $\Diamond(b \wedge \neg b)$, we would immediately apply this refinement. From this point in time onward, no branches would ever be explored that contain $\Diamond(b \wedge \neg b)$.

BDD Unsat Cache: If we were not concerned about irrelevant variables we could skip the (potentially expensive) subset checks used in the unsatisfiable

Table 1. Results on the LWB benchmarks for K (left) and S4 (right)

subclass	BDDTab	BUC	SUC	NUC	Norm	Reorder	R to L	FaCT++	InKreSAT	*SAT
branch_n	18	18	18	18	18	19	17	10	13	12
branch_p	21	1	21	21	21	21	21	10	16	18
lin_n	21	21	21	21	21	21	21	21	21	14
path_n	21	2	21	21	21	21	21	21	10	21
path_p	21	3	21	21	21	21	21	21	10	21
ph_n	10	10	10	10	10	10	10	13	21	12
ph_p	9	9	9	9	10	9	9	7	9	9
poly_p	21	9	21	21	21	21	21	21	21	21
t4p-p	21	0	21	21	21	21	21	21	21	21

subclass	BDDTab	BUC	SUC	NUC	Norm	Reorder	R to L	FaCT++	InKreSAT
branch_n	19	19	19	19	19	19	18	6	12
grz_n	21	21	21	21	21	21	14	21	21
ipc_n	12	12	12	12	12	12	21	11	10
ipc-p	21	21	21	21	21	21	21	10	9
md_n	15	15	15	15	15	15	15	10	8
md-p	21	21	21	21	21	21	21	4	3
path_n	3	3	3	3	3	3	7	21	4
path_p	4	4	4	4	4	4	8	21	5
ph_n	5	5	5	5	5	5	4	8	14
ph-p	9	9	9	9	9	9	9	6	9
s5_n	21	21	21	21	21	21	21	21	16

cache above. Instead, we could maintain a single BDD of all the refinements and apply that to every modal jump.

Saturation Unsatisfiable Cache: Instead of leaves, we could cache saturated tableaux like we do for the satisfiable cache, which would also skip the subset checks.

Normalisation: The mapping from a formula φ to its BDD $[\varphi]$ is syntactic, so two semantically equivalent but syntactically different formulae can exist as two separate BDD variables. To avoid such duplications, we add an extra normalisation step where all modal formulae are constructed as BDDs. Since BDDs are canonical, and equality checking is just integer comparison, duplicate variables can then be easily identified and merged into one variable before the tableau process begins. This is similar to the syntactic normalisations (sorting and associativity) in other approaches [9], but is stronger, as it also catches distributivity.

Reordering: The order of the BDD variables can have a significant effect on the size of the constructed BDDs and thus on how long they take to construct. The BDD package we use, Buddy [2], provides automatic reordering, which periodically attempts to reorder the variables to reduce the number of BDD nodes.

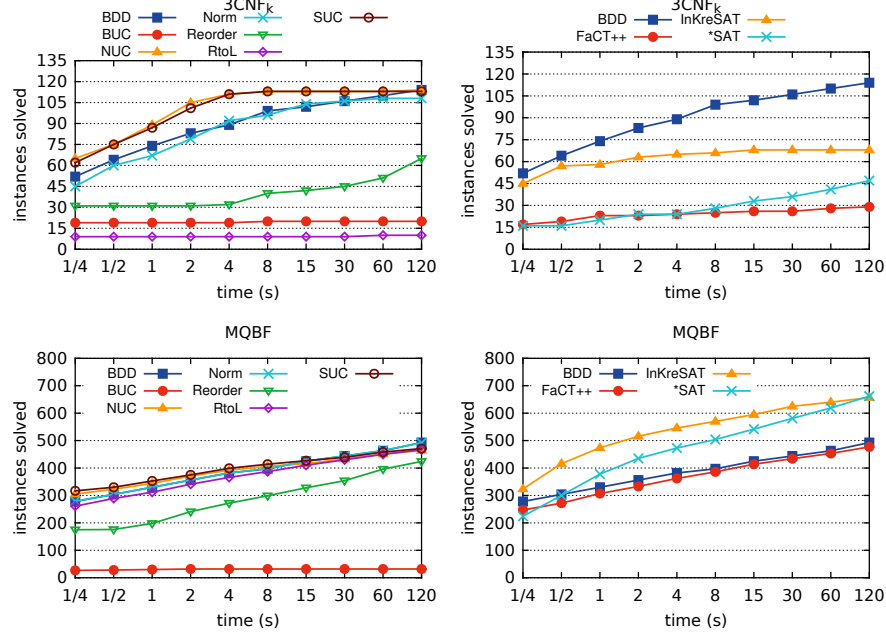
Results. We evaluated BDDTab, against InKreSAT [7], FaCT++ [12] and *SAT [11], all with default settings, as representatives of the state-of-the-art.

We also evaluated several versions of our prover: **BDDTab** uses the optimisations RV, SC and UC, to which the other versions introduce one modification; **BUC** uses a single BDD for the unsatisfiable cache; **SUC** caches unsatisfiable saturations instead of leaves; **NUC** does not do any unsatisfiable caching at all; **Norm** does extra formula normalisation; **Reorder** uses automatic BDD variable reordering; and **R to L** explores saturated BDDs from right to left.

We have followed Kaminski and Tebbi [7] in their choice of benchmarks. Our tests were performed on an Intel 3.4GHz CPU with 8GB of memory.

Table 1 contains results for the Logic Work Bench (LWB) benchmarks for K and S4 [5]. Each subclass consists of a formula shape which takes an argument n

Fig. 1. $3CNF_K$ and MQBF benchmarks



and generates a formula. As n increases, the formulae become larger. The entries in Table 1 mark the highest problem instance solved by each program, given a 30 second time limit per formula. For each subclass the best result is marked in bold. Classes where all provers solved all problems are omitted.

On the LWB benchmarks, BDDTab has arguably the best results, completing every subclass that two or more other provers complete, and gaining a higher score than all of them for several subclasses, including `branch_n` and `_p` for K, and `branch_n`, `ipc_p`, `md_n` and `md_p` for S4. BDDTab performs quite poorly on only a few subclasses, and at least one other prover also performs badly on these.

For the `ph_n` subclasses, this poor result is expected since the formulae are primarily propositional, and tableau-based approaches can halt once they find a single open branch. Our approach always computes the entirety of the saturation phase, and so wastes time finding every open branch, when only one is needed.

BDDTab's memory use for the `ph` subclasses grows exponentially with n , as expected, but is not always a significant factor in its performance on these subclasses. The first unsolved instance of S4 `ph_p` uses 2.7GB before timing out. On the first unsolved instances of the other `ph` subclasses it uses at most 50MB.

Figure 1 shows results for a set of randomly generated $3CNF_K$ [3] formulae of modal depth 2, 4 and 6, as well as for a subset of the TANCs-2000 [8] Unbounded Modal QBF (MQBF) benchmarks for K. On the $3CNF_K$, BDDTab shows the best performance by a large margin. On the MQBF benchmarks BDDTab performs just as well as FaCT++ and worse than InKreSAT and *SAT.

BUC is catastrophically worse over all the benchmarks than BDDTab, except for LWB S4. It is clear that this is due to irrelevant variables being introduced.

NUC performs similarly to BDDTab over the benchmarks, apart from the $3CNF_K$, where it performs much better; it solves the same instances, but is significantly faster on almost every single one, showing that the overhead of the subset checking can easily outweigh the benefits of using the cache, especially when the cache is large. On the MQBF each version is able to solve instances that the other could not, showing that the caching can be worthwhile.

SUC performs almost identically to NUC, which suggests that even cheap unsatisfiability caching does not provide a net benefit over these benchmarks.

Norm is almost identical to BDDTab. For LWB and MQBF the normalisation does not find any duplicate formulae, so the only difference is the time taken to perform the normalisation. These show that this is not necessarily an expensive procedure. In the $3CNF_K$ the normalisation typically reduces the number of variables by around 30%, but this doesn't translate into any net benefit.

Reorder is much worse than BDDTab on $3CNF_K$ and MQBF. Thus naive automatic reordering can be expensive without providing significant benefit.

R to L is perhaps the most striking result, since it just reverses an arbitrary ordering decision. On the LWB it performs significantly better than BDDTab on a number of classes and worse on some others. On the $3CNF_K$ it performs atrociously. This demonstrates the importance of these ordering decisions.

Description logics We tested BDDTab on classifying 15 $\mathcal{ALC}$ ontologies from public online collections. This required a trivial extension to multiple modalities, and we present only general results. Each ontology was translated into a single multi-modal K formula, and then BDDTab was asked to read this formula and perform an exhaustive n^2 check of all possible subsumptions of named classes. FaCT++ was asked to classify the original ontologies through OWL API.

Often, the majority of BDDTab's time is taken up by constructing the BDD representation of the TBox (the global assumptions), after which the actual classification is relatively quick. Reordering is very beneficial here, which is primarily because keeping the size of this BDD small makes it much faster to construct.

BDDTab's worst performance with reordering was two seconds, except for ABA-Adult-Mouse-Brain where, even with reordering, BDDTab failed to construct the global assumptions within 10 minutes. FaCT++ took at most 4 seconds to load and classify each. These results are not conclusive, but suggest it is possible for our method to handle large sets of global assumptions.

Conclusion and Further Work. Determining better ordering schemes and heuristics is needed. These include BDD variable ordering and reordering, modal jump exploration order, saturation exploration order and cache removal policies.

The approach could be extended to include features of other logics such as graded modalities or nominals. Each syntactic element would require its own BDD variable, and saturation phases would always be performed in one hit, but otherwise any high level tableau algorithm could use BDDs in this way.

The refinement process could be modified to only remove paths from the BDD, and avoid introducing new variables to existing paths.

Finally, our notion of “irrelevant variables” can be refined. DPLL-based modal reasoners (and lately SMT solvers) implement a technique called “pure-literal filtering”: if a non-Boolean atom (e.g. a box or a diamond formula) occurs only positively [resp. negatively] in the *original* formula (*i.e.* before adding the $\neg(v_0 \wedge v_1 \dots)$) and it is assigned negatively [resp. positively] in a branch, then the negated [resp. positive] atom can be ignored when recursively checking the branch. In our example, we can simply consider v_a in the final OBDD ignoring the “ $\neg \diamond(\dots)$ ” since $\diamond(\dots)$ occurs only positively in the original formulae.

Our work shows that BDDs can be an effective base data structure for computing tableaux. The results explicitly show this for K, S4 and $\mathcal{ALC}$ with respect to the state-of-the-art as represented by FaCT++, InKreSAT and *SAT.

Acknowledgments. We thank an anonymous reviewer for many suggestions.

References

- [1] Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
- [2] Buddy: (2013), <http://sourceforge.net/projects/buddy/>
- [3] Giunchiglia, F., Sebastiani, R.: Building decision procedures for modal logics from propositional decision procedures - the case study of modal K(m). Information and Computation 162(1/2) (October/November 2000)
- [4] Goré, R., Thomson, J.: BDD-based automated reasoning for propositional bi-intuitionistic tense logics. In: IJCAR. pp. 301–315 (2012)
- [5] Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4 (1996)
- [6] Horrocks, I., Patel-Schneider, P.F.: Optimizing Description Logic Subsumption. Journal of Logic and Computation 9(3), 267–293 (1999)
- [7] Kaminski, M., Tebbi, T.: Inkresat: Modal reasoning via incremental reduction to SAT. In: CADE. pp. 436–442 (2013)
- [8] Massacci, F., Donini, F.M.: Design and results of TANCS-2000 non-classical (modal) systems comparison. In: TABLEAUX. pp. 52–56. LNCS 1847, Springer (2000)
- [9] Sebastiani, R., Tacchella, A.: SAT techniques for modal and description logics. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 781–824. IOS Press (2009)
- [10] Sebastiani, R., Vescovi, M.: Automated Reasoning in Modal and Description Logics via SAT Encoding: the Case Study of K(m)/ALC-Satisfiability. Journal of Artificial Intelligence Research (JAIR) 35, 343–389 (2009)
- [11] Tacchella, A.: *sat system description. In: Description Logics (1999)
- [12] Tsarkov, D., Horrocks, I.: Fact++ description logic reasoner: System description. In: IJCAR 2006. LNAI, vol. 4130, pp. 292–297. Springer (2006)