Modal CEGAR-tableaux with RECAR and resolution-based SAT-shortcuts

Rajeev Goré
Faculty of Information Technology, Monash University, Australia
rajeev.gore@monash.edu

Cormac Kikkert
Cormac Kikkert Research

We investigate two approaches for extending CEGAR-tableaux with SAT-shortcuts using a previously known approach called RECAR but also a totally new approach using the modal resolution theorem prover KSP as an oracle. Our experiments using our C++ implementation CEGARBox++ of CEGAR-tableaux show that: (1) CEGARBox++ with RECAR SAT-shortcuts is not competitive (2) CEGARBox++ using KSP to provide SAT-shortcuts is superior to both CEGARBox++ and KSP, particularly on large satisfiable problems. As far as we know, this is the first effective integration of SAT, tableaux and resolution methods for modal satisfiability which performs better than its parts.

1 Introduction

Propositional modal, temporal and description logics are of fundamental importance in logic-based artificial intelligence, hardware and software verification [22] and knowledge representation and reasoning [1]. Thus efficient decision procedures for modal logic are an important area of research.

Such research has not matched the advances in SAT-solving, until recently, when various authors independently used SAT-solvers for modal satisfiability [19, 8, 13, 14, 9] using a technique originally due to Claessen and Rosén [5] for propositional intuitionistic logic, which was, itself, "inspired" by the work of Goré et al. on "modal clause learning" using binary decision diagrams [11].

We previously [9] presented a new type of tableaux calculus which uses a SAT-solver and the standard counter-example guided abstraction refinement (CEGAR) methodology [6] to search for a rooted Kripke model for a given formula in modal clausal normal form (mcnf). We showed how to modify the initial mcnf to "compile in" the axioms T and 4 via a preprocessing stage, adding a standard loop-check for termination when required. Our Haskell implementation CEGARBox was, overall, the best over the standard benchmarks for the modal logics K, KT and S4, sometimes by orders of magnitude. Indeed, the only benchmark where CEGARBox did not win were the K-MQBF benchmarks, where the modal resolution theorem prover KSP solved approximately 50 (out of 1000) more problems in 32 seconds each [9]: see Figure 3.

When using SAT-solvers as oracles, we usually first create a formula of classical propositional logic (cpl) which is an approximation of the given (modal) formula φ_0. The approximation is an under-approximation $\check{\varphi}_0$ if its cpl-UNSATisfiability implies the modal unsatisfiability of φ_0, giving us an $\lightning$UNSAT-shortcut. The approximation is an over-approximation $\hat{\varphi}_0$ if its cpl-SATisfiability implies the modal satisfiability of φ_0, giving us a $\lightning$SAT-shortcut.

Here we report on two ways to add $\lightning$SAT-shortcuts to CEGAR-tableaux: the RECAR approach and a totally new approach using the resolution-based solver KSP as an oracle. Using our new C++ implementation CEGARBox++, our experiments show that: (1) CEGARBox++ with RECAR $\lightning$SAT-shortcuts is not competitive; and (2) CEGARBox++ using KSP to provide $\lightning$SAT-shortcuts is superior to each, particularly on large satisfiable problems.

M. Bílková, M. Gattinger, I. van der Giessen, M. Girlando, Y. Wang (Eds.):
Advances in Modal Logic 2026 (AiML 2026)
EPTCS 447, 2026, pp. 427–444, doi:10.4204/EPTCS.447.24

© Rajeev Goré & Cormac Kikkert
This work is licensed under the
Creative Commons Attribution License.

$$M,w \Vdash \langle\alpha\rangle\varphi \quad \text{iff} \quad \exists v, R_\alpha(w,v)\ \&\ M,v \Vdash \varphi$$
$$M,w \Vdash [\alpha]\varphi \quad \text{iff} \quad \forall v, R_\alpha(w,v) \Rightarrow M,v \Vdash \varphi$$

Figure 1: Semantics of multi-modal logic $\mathrm{K_n}$ (aka ALC)

2 Preliminaries: syntax, semantics and modal clausal normal form

Let Ag be a non-empty finite set of agent names. Let Atm be a set of atomic formulae with $Atm \cap Ag = \emptyset$. Consider the language of formulae defined from atoms $p \in Atm$ and $\alpha \in Ag$ by the BNF grammar:

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [\alpha]\varphi \mid \langle\alpha\rangle\varphi$$

Define $(\varphi_1 \to \varphi_2) := (\neg\varphi_1 \vee \varphi_2)$ and $(\varphi_1 \leftrightarrow \varphi_2) := ((\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1))$. Let $[\alpha]^0\varphi := \varphi$ and $[\alpha]^{n+1}\varphi := [\alpha][\alpha]^n\varphi$. The modal-depth of a formula is the maximum number of nested modalities in it. For a finite set $\Gamma = \{\varphi_1, \cdots, \varphi_k\}$ of formulae, let $\widehat{\Gamma} = (\varphi_1 \wedge \cdots \wedge \varphi_k)$ and let $[\alpha]\Gamma = \{[\alpha]\varphi_1, \cdots, [\alpha]\varphi_k\}$.

The Kripke semantics for these multimodal (description) logics use Kripke models consisting of triples $M := \langle W, \{R_\alpha\}_{\alpha\in Ag}, \vartheta\rangle$ over some non-empty set W (of possible worlds), and binary relations R_α over W for every agent $\alpha \in Ag$, and a valuation $\vartheta(w,p) \subseteq W \times Atm$ telling us the truth value of each atomic formula p at each world $w \in W$. Truthhood at world w in model M, written as $M,w \Vdash \varphi$, extends the usual truth-tables for classical propositional logic (cpl): see Figure 1.

A formula φ is $\mathrm{K_n}$-satisfiable if there is a Kripke model M containing a world w such that $M,w \Vdash \varphi$. A formula φ is $\mathrm{K_n}$-valid if $\neg\varphi$ is not $\mathrm{K_n}$-satisfiable. Formulae φ and ψ are logically equivalent if $\varphi \leftrightarrow \psi$ is $\mathrm{K_n}$-valid, and are equi-satisfiable if φ is $\mathrm{K_n}$-satisfiable iff ψ is $\mathrm{K_n}$-satisfiable.

2.1 Separated modal clausal normal form for $\mathrm{K_n}$

A *positive literal* is an atomic formula p. A *negative literal* is a negated atomic formula $\neg p$. A *literal* is a positive literal p or else a negative literal $\neg p$. We use a to f and l and r for literals. We use A, B, C and D for a set of literals. Let $\bar{l} := \neg p$ if $l = p$ and $\bar{l} := p$ if $l = \neg p$ so that $\bar{\bar{l}} = l$. A formula is in *negation normal form* (NNF) if it is implication-free and negations appear only in front of atomic formulae. An NNF can be created with a linear descent of the formula.

Proposition 2.1. *A formula φ_0 of modal depth κ can be converted into a logically equivalent formula $nnf(\varphi_0)$ in NNF which is at most polynomially longer.*

A modal clause is any formula of one of the following forms [21, 15, 10]:

cpl-clause: a formula $(l_1 \wedge \cdots \wedge l_i) \to (r_1 \vee \cdots \vee r_j)$ of literals;

box-clause: a formula $a \to [\alpha]b$ with literals a and b and agent name $\alpha \in Ag$;

dia-clause: a formula $c \to \langle\alpha\rangle d$ with literals c and d and agent name $\alpha \in Ag$.

Using semicolon to indicate a set-union, an arbitrary set $\mathscr{C} = (\mathscr{C}^{cpl}\ ;\ \mathscr{C}^{box}\ ;\ \mathscr{C}^{dia})$, of modal-clauses can be partitioned into the cpl-clauses $\mathscr{C}^{cpl}$ and box-clauses $\mathscr{C}^{box}$ and dia-clauses $\mathscr{C}^{dia}$ as defined above.

Our modal clausal normal form requires notation to succinctly express finite sequences of modalities such as $[\alpha_1][\alpha_2]\cdots[\alpha_k]$ which naturally correspond to all finite paths $w_1 R_{\alpha_1} w_2 R_{\alpha_2} \cdots R_{\alpha_k} w_k$ in a Kripke model. We therefore abuse notation to extend the language of formulae using constructs from regular expressions even though they are not part of the official syntax.

We use the composition operator ";" from regular expressions with ε for the empty regular expression obeying $(\varepsilon;\alpha) = (\alpha;\varepsilon) = \alpha$. We let $\alpha^0 = \varepsilon$ and $\alpha^{k>0} = (\alpha;\alpha^{k-1})$. We write $[\alpha_1;\alpha_2;\cdots;\alpha_k]$ instead of

$[\alpha_1][\alpha_2]\cdots[\alpha_k]$, write $[\alpha^n;\beta^m]$ for $[\alpha]^n[\beta]^m$ and write $w_1R_{\alpha_1;\cdots;\alpha_k}w_k$ for $w_1R_{\alpha_1}w_2R_{\alpha_2}\cdots R_{\alpha_k}w_k$. Let Ag^* be the set of all finite regular expressions over Ag that use only ";" and let Ag^k be the set of all finite regular expressions of length k over Ag that use only ";".

Proposition 2.2. *A formula φ_0 of modal depth κ can be put into an equi-satisfiable modal clausal normal form $r \wedge \widehat{SNF_{mc}(\varphi_0)}$ by naming subformulae using new atomic propositions, using r as the name for φ_0, and where each $\mathscr{C}$ below is a finite set of modal clauses (essentially the SNF_{ml} of Nalon et al. [17] who prove this proposition in detail):*

$$SNF_{mc}(\varphi_0) \quad := \quad \mathscr{C}_\varepsilon \,;\, \bigcup_{\sigma\in Ag^1}[\sigma]\mathscr{C}_\sigma \,;\, \cdots \,;\, \bigcup_{\sigma\in Ag^\kappa}[\sigma]\mathscr{C}_\sigma$$

Example 2.3. Consider the negation $\varphi_0 := \neg([\alpha](p \to q) \to ([\alpha]p \to [\alpha]q))$ of the K axiom with modal depth $\kappa = 1$. Its NNF is $[\alpha](\neg p \vee q) \wedge [\alpha]p \wedge \langle\alpha\rangle\neg q$. Putting r as the name for φ_0 gives us: $\mathscr{C}_\varepsilon := r \to [\alpha]b \,;\, r \to [\alpha]p \,;\, r \to \langle\alpha\rangle\neg q$ and $\mathscr{C}_\alpha := b \to \neg p \vee q$ with $\text{SNF}_{mc}(\varphi_0) = \mathscr{C}_\varepsilon \,;\, [\alpha]\mathscr{C}_\alpha$.

Note, the set $\mathscr{C}_\varepsilon$ excludes r so we must add it explicitly to form $(r \,;\, \text{SNF}_{mc}(\varphi_0))$.

3 SAT-solvers with internal state as CPL-oracles

CEGAR-tableaux [9] assume we have access to a SAT-solver `s`, with internal state, to which we can add a cpl-clause φ via `addClause(s, φ)`, and which accepts a set $\mathscr{A}$ of literals, called unit assumptions, such as `MiniSAT` [7]. Intuitively, we want all literals in $\mathscr{A}$ to be assigned to true.

When pre-loaded with a set $\mathscr{C}^{cpl}$ of cpl-clauses and called with `solve(s, A)`, such a SAT-solver returns one of two results:

(sat, ϑ): if $(\mathscr{A} \,;\, \mathscr{C}^{cpl})$ is true under some cpl-valuation $\vartheta \supseteq \mathscr{A}$ or-else

$(unsat, UC)$: if $UC \subseteq \mathscr{A}$ is a, not necessarily unique, minimal "unsatisfiable core" of $\mathscr{A}$ such that $(UC \,;\, \mathscr{C}^{cpl})$ is cpl-unsatisfiable, and hence so is $(\mathscr{A} \,;\, \mathscr{C}^{cpl})$.

UC itself may be cpl-satisfiable but the term "unsatisfiable core" is standard.

Since SAT-solvers handle classical propositional logic, they should return a cpl-valuation $\vartheta \subseteq Atm$. That is, strictly speaking, a cpl-valuation ϑ is just a set of atomic formulae. But in concrete applications, it may be more useful for a user to assert "I know that atomic formula c is false", which can be achieved with $\neg c \in \mathscr{A}$. Thus, the valuations returned by a SAT-solver are actually a subset of the set of all literals that appear in $(\mathscr{A} \,;\, \mathscr{C}^{cpl})$.

We can then easily extend such a set of literals to a valuation over all atoms that appear in an mcnf by putting all missing atoms to false.

Example 3.1. From our previous example, φ_0 is equi-satisfiable with $(r \,;\, \mathscr{C}_\varepsilon \,;\, [\alpha]\mathscr{C}_\alpha)$. Create a SAT-solver `s0` for modal depth 0, pre-load `s0` with $\mathscr{C}^{cpl}_\varepsilon = \emptyset$, the cpl-part of $\mathscr{C}_\varepsilon$, put $\mathscr{A}_\varepsilon = \{r\}$ because r must be true at modal depth 0, and call `solve(s0, Aε)`: it must return $(sat, \vartheta_\varepsilon)$ where $\vartheta_\varepsilon = \{r\} \supseteq \mathscr{A}_\varepsilon$ since $(\mathscr{A}_\varepsilon \,;\, \mathscr{C}^{cpl}_\varepsilon) = (r \,;\, \emptyset) = \{r\}$ is true under $\vartheta_\varepsilon = \{r\}$.

Create a SAT-solver `s1` for modal depth 1, pre-load `s1` with $\mathscr{C}^{cpl}_\alpha = \{b \to \neg p \vee q\}$, the cpl-part of $\mathscr{C}_\alpha$, put $\mathscr{A}_\alpha = \{b, p, \neg q\}$, and call `solve(s1, Aα)`: it must return $(unsat, UC_\alpha)$ where $UC_\alpha = \{b, p, \neg q\} \subseteq \mathscr{A}_\alpha$ since $(UC_\alpha \,;\, \mathscr{C}^{cpl}_\sigma) = \{b, p, \neg q, b \to \neg p \vee q\}$ is cpl-unsatisfiable but every proper subset is cpl-satisfiable.

Finally, add $\neg r$ to `s0` via `addClause(s0, ¬r)`, and restart it via `solve(s0, Aε)`: it must return $(unsat, UC_\varepsilon)$ where $UC_\varepsilon = \{r\} \subseteq \mathscr{A}_\varepsilon$ because $(UC_\varepsilon \,;\, (\mathscr{C}^{cpl}_\varepsilon \,;\, \neg r)) = (r \,;\, \neg r)$ is cpl-unsatisfiable but no proper subset is so.

Algorithm 1 CEGARTAB(A, `Trie[`σ`]`)

1: {Inputs: A is a set of unit assumptions,
   `Trie[`σ`]` is a node in our Trie containing modal clauses and a SAT-solver.}
2: Let t_σ := solve(`Trie[`σ`].sat`, A) {*is* $\mathscr{C}_\sigma^{cpl}$ *cpl-satisfiable*}
3: **if** t_σ = (unsat, UC_σ) **then**
4: **return** Unsatisfiable(UC_σ) {*because* $\mathscr{C}_\sigma$ *is* $\mathrm{K_n}$*-unsatisfiable*}
5: **else if** t_σ = (sat, ϑ_σ) **then**
6: {*Check box and diamond clauses that fire under classical valuation* ϑ_σ}
7: **for** $(c \to \langle\alpha\rangle d) \in$ `Trie[`σ`].DiaCl` with $c \in \vartheta_\sigma$ **do**
8: Let $B = \{b \mid (a \to [\alpha]b) \in$ `Trie[`σ`].BoxCl` and $a \in \vartheta_\sigma\}$
9: {*evaluate the next* α*-successor using the jump rule*}
10: **if** CEGARTAB($(d;B)$, `Trie[`σ`].child(`α`)`) = Unsatisfiable($UC_{\sigma;\alpha}$) **then**
11: $CS := \{c\} \cup \{a \mid (a \to [\alpha]b) \in$ `Trie[`σ`].BoxCl` and $a \in \vartheta_\sigma$ and $b \in UC_{\sigma;\alpha}\}$
12: Let $\varphi := \bigvee_{l \in CS} \neg l$
13: addClause(`Trie[`σ`].sat`, φ) { *Learn new clause* $\varphi := \neg \bigwedge CS$}
14: **return** CEGARTAB(A, `Trie[`σ`]`) { *apply (restart)* }
15: **end if**
16: **end for**
17: **return** Satisfiable {*because every fired diamond is fulfilled*}
18: **end if**

Figure 2: Algorithm of CEGARTAB for multi-modal (description) logic $\mathrm{K_n}$ (ALC)

4 CEGAR-tableaux for multi-modal logic $\mathrm{K_n}$ (aka ALC)

We now describe the CEGAR-tableaux [9] procedure extended to multi-modal logic $\mathrm{K_n}$ (aka ALC) without local (ABox) and global (TBox) assumptions.

Conceptually, the extension from mono-modal logic K to multi-modal logic $\mathrm{K_n}$ is easy because there are no interactions between the different modalities, so we just have n "copies" of K, one for each modality $[\alpha]$ for every agent $\alpha \in Ag$.

We give the extension of the main CEGARTAB algorithm from Goré and Kikkert [9] as Algorithm 1 in Figure 2, and make the connection to the general form of $\mathrm{SNF}_{mc}(\varphi_0)$ but with a non-singleton Ag:

$$\mathrm{SNF}_{mc}(\varphi_0) \quad := \quad \mathscr{C}_\varepsilon \,;\, \bigcup_{\sigma \in Ag^1} [\sigma]\mathscr{C}_\sigma \,;\, \cdots \,;\, \bigcup_{\sigma \in Ag^\kappa} [\sigma]\mathscr{C}_\sigma$$

Because CEGARTAB handled only mono-modal logics, the mcnf of a given formula φ_0 of modal depth κ was as below for a singleton $Ag = \{\alpha\}$ (say) :

$$\mathrm{SNF}_{mc}(\varphi_0) := \mathscr{C}_\varepsilon \,;\, [\alpha]\mathscr{C}_\alpha \,;\, [\alpha^2]\mathscr{C}_{\alpha^2} \,;\, \cdots \,;\, [\alpha^\kappa]\mathscr{C}_{\alpha^\kappa}$$

Conceptually, these "modal contexts" were stored in a linear Trie (a.k.a. prefix tree) data structure with `Node[0]` initialised to contain $\mathscr{C}_\varepsilon$ and node `Node[i]`, for $i > 0$, initialised to contain $\mathscr{C}_{\alpha^i}$, eliding the box-prefix $[\alpha^i]$ without loss of generality, but saving quadratic space.

In the multi-modal case, we use a multi-dimensional trie and all trie-edges are labelled by an agent name and the trie-root is labelled with ε. For a regular expression σ, we write `Trie`$[\sigma]$ for the corresponding `TrieNode`. For example, from the trie-root node `Trie`$[\varepsilon] = \mathscr{C}_\varepsilon$ there is a trie-path labelled by an α-edge followed by a β-edge to the trie-node `Trie`$[\alpha;\beta] = \mathscr{C}_{\alpha;\beta}$.

In Algorithm 1 in Figure 2, and in the implementation, each $\mathscr{C}_\sigma$ is stored at trie-position $\texttt{Trie}[\sigma]$ in four fields by partitioning it into its three disjoint components $\mathscr{C}_\sigma = (\mathscr{C}_\sigma^{cpl}\ ;\ \mathscr{C}_\sigma^{dia}\ ;\ \mathscr{C}_\sigma^{box})$ and adding "next" pointers as follows:

1. $\texttt{Trie}[\sigma]\texttt{.sat}$ is a dedicated SAT-solver for this trie-node pre-loaded with $\mathscr{C}_\sigma^{cpl}$
2. $\texttt{Trie}[\sigma]\texttt{.DiaCl}$ contains $\mathscr{C}_\sigma^{dia} := \bigcup_{\alpha\in Ag}\{c \to \langle\alpha\rangle d \mid (c \to \langle\alpha\rangle d) \in \mathscr{C}_\sigma\}$
3. $\texttt{Trie}[\sigma]\texttt{.BoxCl}$ contains $\mathscr{C}_\sigma^{box} := \bigcup_{\alpha\in Ag}\{a \to [\alpha]b \mid (a \to [\alpha]b) \in \mathscr{C}_\sigma\}$
4. $\texttt{Trie}[\sigma]\texttt{.Child}(\alpha)$ is (a pointer to) the node $\texttt{Trie}[\sigma;\alpha]$, for each $\alpha \in Ag$.

The second and third fields are structured further so that we can select the dia-clauses and box-clauses for a particular agent $\alpha \in Ag$ via $\texttt{Trie}[\sigma]\texttt{.DiaCl}(\alpha)$ and $\texttt{Trie}[\sigma]\texttt{.BoxCl}(\alpha)$, respectively. We often write "formula in a TrieNode" or even "$\varphi \in \texttt{Trie}[\sigma]$" to refer to formulae that are stored in one of these fields.

Recall that r names the initial formula but it is not in the root $\texttt{Trie}[\varepsilon]$. Now we simply start by calling CegarTab($\{r\}$, $\texttt{Trie}[\varepsilon]$) as shown in Figure 2.

Example 4.1. Let us continue with our example. We know that φ_0 is equi-satisfiable with $r\ ;\ \mathscr{C}_\varepsilon\ ;\ [\alpha]\mathscr{C}_\alpha$, where $\mathscr{C}_\varepsilon^{cpl} = \emptyset$ and $\mathscr{C}_\varepsilon^{dia} = (r \to \langle\alpha\rangle\neg q)$ and $\mathscr{C}_\varepsilon^{box} = (r \to [\alpha]b\ ;\ r \to [\alpha]p)$, meaning $\texttt{Trie}[\varepsilon]\texttt{.sat}$ is $\texttt{s0}$. We also have $\mathscr{C}_\alpha^{cpl} = (b \to \neg p \vee q)$ and $\mathscr{C}_\alpha^{dia} = \emptyset$ and $\mathscr{C}_\alpha^{box} = \emptyset$, meaning $\texttt{Trie}[\alpha]\texttt{.sat}$ is $\texttt{s1}$.

Recursion 0. The initial call to CegarTab($\{r\}$, $\texttt{Trie}[\varepsilon]$) sets $\sigma := \varepsilon$ and $A := \{r\} = \mathscr{A}_\varepsilon$ so line 2 is $\texttt{solve(s0, }\mathscr{A}_\varepsilon\texttt{)}$: it returns $(sat, \vartheta_\varepsilon)$ where $\vartheta_\varepsilon = \{r\}$ so we enter the "then" part of line 5. At line 7, there is only one "fired" dia-clause $(r \to \langle\alpha\rangle\neg q) \in \texttt{Trie}[\varepsilon]\texttt{.DiaCl} = \mathscr{C}_\varepsilon^{dia}$ with $r \in \vartheta_\varepsilon$, so $c := r$ and $d := (\neg q)$ and we know that $\langle\alpha\rangle\neg q$ must be (modally) true at modal depth 0. Under $\vartheta_\varepsilon = \{r\}$, every box-clause in $\mathscr{C}_\varepsilon^{box}$ "fires", thus the set $\{[\alpha]b, [\alpha]p\}$ is (modally) true at modal depth 0, and hence the set $B = \{b, p\}$ formed at line 8 must be classically true at some R_α-successor for $\neg q$ at modal depth 1. To evaluate this R_α-successor, we compute $(d;B) = \{b, p, \neg q\}$ at line 10 and recurse via CegarTab($\{b, p, \neg q\}$, $\texttt{Trie}[\alpha]\texttt{.Child}(\alpha)$).

Recursion 1. So $\sigma := \alpha$ and $A := \{b, p, \neg q\} = \mathscr{A}_\alpha$ and the relevant SAT-solver is the "next" one so $\texttt{Trie}[\alpha]\texttt{.sat = s1}$. Line 2 calls this SAT-solver, so this is the call $\texttt{solve(s1, }\mathscr{A}_\alpha\texttt{)}$: it must return $(unsat, UC_\alpha)$ where $UC_\alpha = \{b, p, \neg q\}$ so we return Unsatisfiable($\{b, p, \neg q\}$) at line 4 and pop the recursion stack.

Recursion 0. We return to line 11 where $\sigma := \varepsilon$ and so $(\sigma;\alpha) = \alpha$, so $UC_{\sigma;\alpha} = UC_\alpha = \{b, p, \neg q\}$. We know the chosen ϑ_ε at modal depth 0 caused a clash at modal depth 1 involving $\{b, p, \neg q\}$, and hence that $\{[\alpha]b, [\alpha]p, \langle\alpha\rangle\neg q\}$ cannot be jointly true at modal depth 0. At line 11 we trace the relevant box- and dia-clauses to find the conflict set $CS = \{r\}$ at modal depth 0. To avoid this "mistake", we compute the "refinement" $\varphi = \neg r$ at line 12 of recursion level 0. Line 13 is then just $\texttt{addClause(s0, }\neg r\texttt{)}$ so $\mathscr{C}_\varepsilon^{cpl} = \{\neg r\}$. The refined $\text{SNF}_{mc}(\varphi_0)$ demands r is false at the root. At line 14, we call CegarTab($\{r\}$, $\texttt{Trie}[\varepsilon]$) but we stay at the current SAT-solver at modal depth 0 thereby restarting $\texttt{s0}$.

Recursion 1. Line 2 calls $\texttt{solve(s0, }\mathscr{A}_\varepsilon\texttt{)}$: it returns $(unsat, UC_\varepsilon)$ where $UC_\varepsilon = r$ which returns Unsatisfiable($\{r\}$) to line 14 of Recursion 0: our final answer.

Theorem 4.2. *If* $\texttt{Trie}$ *contains* $\text{SNF}_{mc}(\varphi_0)$ *and* r *names* φ_0 *then* CegarTab($\{r\}$, $\texttt{Trie}[\varepsilon]$) *terminates, and returns Satisfiable iff* φ_0 *is* K_n*-satisfiable.*

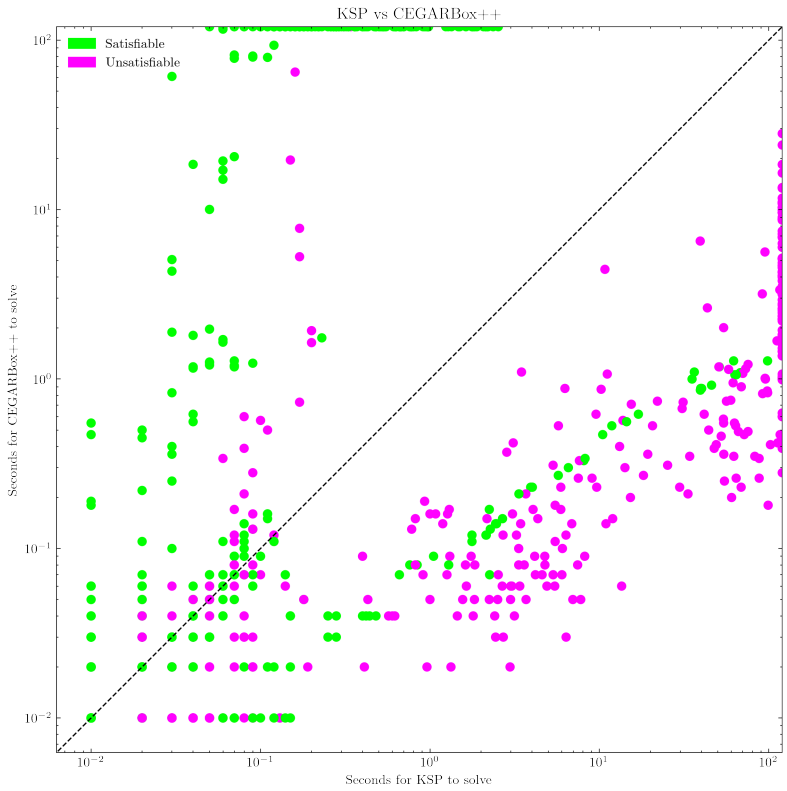

Figure 3: Performance of CEGARBox++ and KSP on the MQBF benchmarks. Problems on the top/right edge indicate a timeout for CEGARBox++/KSP respectively.

Proof. A simple modification of the proofs for mono-modal K from Goré and Kikkert [9] which proceed by induction on the number of restarts. □

5 CEGAR-tableaux with different (↯SAT) shortcuts

We previously showed [9] that CEGARBox outperformed all solvers we tested on the extended LWB K-benchmarks, and 3CNF K-benchmarks, mostly by orders of magnitude. However, on the MQBF K-benchmarks, CEGARBox loses to KSP by about 50 problems. We first determine why and then explore two solutions.

5.1 Why does CEGARBox++ not win on the MQBF benchmarks?

Figure 3 shows that CEGARBox++ solves all unsatisfiable benchmarks but struggles on the satisfiable ones, while KSP solves all satisfiable benchmarks but struggles on the unsatisfiable ones. KSP performs better overall as there are 617 satisfiable and 399 unsatisfiable problems in the MQBF benchmarks.

By inspecting the individual (satisfiable) benchmarks that CEGARBox++ struggles on, we saw that CEGARBox++ learns few clauses, and instead spends most of its time building a large model. That is, because CEGARBox++ explicitly builds a satisfying model, a huge model can cause it to timeout. This is particularly noticeable in benchmarks with lots of branching, as it can result in satisfying models having

an exponential number of worlds with respect to the size of the original benchmark. KSP on the other hand doesn't need to explicitly build a model to deem a formula as satisfiable, it simply applies rules until *saturation*.

This is a fundamental weakness in not just CEGARTAB, but the CEGAR procedure itself. For a formula φ, we first test whether an under-approximation $\breve{\varphi}$ consisting of the cpl-clauses of its mcnf is cpl-satisfiable. If $\breve{\varphi}$ is cpl-unsatisfiable, we can perform an ($\lightning$UNSAT) shortcut by immediately returning UNSAT as φ must also be modally unsatisfiable. However, there is no corresponding ($\lightning$SAT) shortcut.

Our observation is not new. Brummayer [3] devised an under-approximation for SMT that allows for early termination in the UNSAT case. Wang et al. [23] used induction to show the existence of large counter-models in model-checking. These frameworks are domain specific, so it is unlikely their insights would apply to CEGARTAB. However, for SAT-solving, Lagniez et al. [12] presented a *general* variant of CEGAR, which allows for both ($\lightning$SAT) and ($\lightning$UNSAT) shortcuts.

We first analyse the RECAR approach of Lagniez et al. [12]. We have previously shown [9] their implementation was unsound and their benchmarks were incorrect, but the theory of RECAR is sound so we incorporate their ideas [12] to create our own variant of RECAR-Tableaux.

We also create our own novel approach by detecting fixpoints in our under-approximation. Although it is specific to recursive CEGAR algorithms, we show that it works exceptionally well, allowing CEGARBox++ to solve all problems in the MQBF benchmark set bar one. Our approach combines KSP with CEGARBox++, giving a solver that elegantly combines SAT, tableaux and resolution methods, which previously were the three competing methods for K-satisfiability.

5.2 Background on modal resolution and RECAR

We first give some background on modal resolution and the RECAR framework.

5.2.1 Modal clausal resolution (KSP).

Recalling the definition of SNF_{mc}, we briefly outline the modal calculus rules of KSP [16] for multi-modal K_n, which is a resolution prover that operates on SNF_{ml}, a close variant of SNF_{mc}.

Proposition 5.1. *A formula φ_0 of modal depth κ can be put into the equi-satisfiable separated normal form with modal layers $SNF_{ml}(\varphi_0)$ of Nalon et al [17], who prove this proposition in detail, and where each $\mathscr{C}_i$ below is a set of modal clauses:*

$$SNF_{ml}(\varphi_0) \quad := \quad \mathscr{C}_0 \,;\, [\,]^1\mathscr{C}_1 \,;\, \cdots \,;\, [\,]^\kappa\mathscr{C}_\kappa$$

Informally, but not exactly because KSP requires extra clauses that are not required by CEGARBox++, we have $\mathscr{C}_l \approx \bigcup_{\sigma \in Ag^l} \mathscr{C}_\sigma$, where $\mathscr{C}_\sigma$ is from our $\text{SNF}_{mc}(\varphi_0)$. That is, each $\mathscr{C}_l$ is the "union" formed by taking a "horizontal slice" across layer $l = \mid \sigma \mid$ of our "vertical tree-like" $\text{SNF}_{mc}(\varphi_0)$, where $\mid \sigma \mid$ is the length of the Trie-path σ. But now, $\text{SNF}_{ml}(\varphi_0)$ can be implemented as a linear trie, rather than a branching trie as for $\text{SNF}_{mc}(\varphi_0)$. Let us write $n : \psi$ to indicate the clause ψ is stored at the n-th node in the (linear) trie, where the root is element 0. Using this notation, the resolution rules used by KSP are presented in Figure 4.

They differ from normal resolution as clauses are now labelled by their modal layer. Applying these rules will either derive an empty clause, meaning φ_0 is K_n-unsatisfiable, or terminate after saturation, meaning φ_0 is K_n-satisfiable. Observe the following close connection between modal resolution and CEGARTAB: all resolvents in KSP are classical clauses as are all learnt clauses in CEGARTAB. We will use this insight to combine both these approaches.

$$\text{[LRES]}\quad \frac{n : D \vee l \quad n : D' \vee \neg l}{n : D \vee D'} \qquad \text{[MRES]}\quad \frac{n : l_1 \Rightarrow [\alpha]\, l \quad n : l_2 \Rightarrow \langle\alpha\rangle \neg l}{n : \neg l_1 \vee \neg l_2}$$

$$\text{[GEN1]}\quad \frac{n : l'_1 \Rightarrow [\alpha]\neg l_1, \cdots, n : l'_m \Rightarrow [\alpha]\neg l_m, n : l' \Rightarrow \langle\alpha\rangle \neg t, n+1 : l_1 \vee \cdots \vee l_m \vee t}{n : \neg l'_1 \vee \cdots \vee \neg l'_m \vee \neg l'}$$

$$\text{[GEN2]}\quad \frac{n : l'_1 \Rightarrow [\alpha]\, l_1 \quad n : l'_2 \Rightarrow [\alpha]\neg l_1 \quad n : l'_3 \Rightarrow \langle\alpha\rangle\, l_2}{n : \neg l'_1 \vee \neg l'_2 \vee \neg l'_3}$$

$$\text{[GEN3]}\quad \frac{n : l'_1 \Rightarrow [\alpha]\neg l_1, \cdots, n : l'_m \Rightarrow [\alpha]\neg l_m, n : l' \Rightarrow \langle\alpha\rangle\, l, n+1 : l_1 \vee \cdots \vee l_m}{n : \neg l'_1 \vee \cdots \vee \neg l'_m \vee \neg l'}$$

Figure 4: The resolution rules used by KSP [16].

5.2.2 The RECAR framework (MOSAIC).

RECAR (Recursive Explore and Check Abstraction Refinement) is an extension of the CEGAR approach which introduces a recursive step to allow for an additional shortcut [12]. There are two vaiants, RECAR-under, where the main CEGAR loop contains an UNSAT shortcut (↯UNSAT), whilst the recursive step allows for a SAT shortcut (↯SAT), and RECAR-over where the main CEGAR loop contains a SAT shortcut (↯SAT), whilst the recursive step allows for an UNSAT shortcut.

We present just the RECAR-under algorithm here, as the main "CEGAR" loop of the RECAR-under framework has (↯UNSAT) shortcuts, like CEGAR-tableaux. This will help motivate our own application of RECAR later but note that the RECAR-over algorithm can be defined analogously (and is what is used in MOSAIC [12]). We require the following assumptions, see Figure 5, where *RC* is a Boolean function that determines whether or not a recursive call occurs:

Definition 5.2. Assumptions for RECAR-under [12]

1. Function 'check' is a sound, complete and terminating implementation that decides if an input formula is K-unsatisfiable.
2. Function refine(φ) is a function that constrains φ with more clauses so that:
   (a) If the under-approximation $\check{\varphi}$ is K-unsatisfiable, then so is refine$(\check{\varphi})$.
   (b) There exists an $n \in \mathbb{N}$ such that refine$^n(\check{\varphi})$ is K-satisfiable iff φ is so.
3. If the over-approximation $\hat{\varphi}$ is K-satisfiable, then so is φ.
4. Let over$(\varphi) = \hat{\varphi}$. There exists $n \in \mathbb{N}$ such that $RC(\text{over}^n(\varphi), \text{over}^{n+1}(\varphi))$ evaluates to false (guaranteeing termination).

Proposition 5.3. *RECAR-under is sound, complete and terminating [12].*

The RECAR framework was implemented in the solver MOSAIC [12]. Briefly, the over-approximation involves checking if some formula φ is K-satisfiable with $\leq n$ worlds via a naive SAT translation, with $\mathcal{O}(Atom(\varphi) \times n + n^2)$ variables, where $Atom(\varphi)$ denotes the number of atoms in φ. The translation has variables for each world, as well as extra variables indicating which worlds are accessible from each

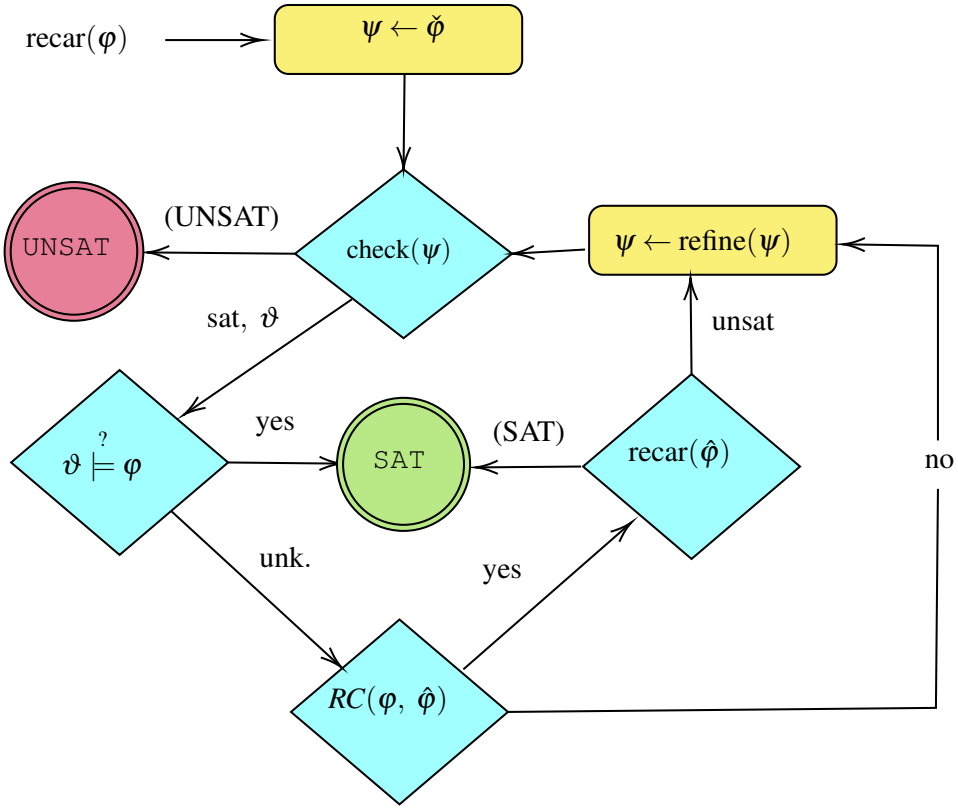

Figure 5: The RECAR-under framework [12].

other. Over the course of the algorithm, n will increase until it reaches the bound $n = Atom(\varphi)^{\text{depth}(\varphi)}$ where K-satisfiability can be determined as φ is K-satisfiable if and only if it has a model with n worlds [18]. The under-approximation involves "removing" conjuncts in φ. Thus both the under-approximation and over-approximation involve reasoning about the formula φ in its entirety via a SAT-solver. This is exactly what CEGAR-tableaux aim to avoid by using a SAT-solver to only determine worlds (and not models), thus requiring only $\mathcal{O}(n)$ variables in a SAT-solver.

5.3 RECARTAB: Extending CEGAR-tableaux with RECAR

We now extend CEGAR-tableaux with RECAR ($\lightning$SAT)-shortcuts.

First, we must take some liberties when applying RECAR to CEGAR-tableaux. Note our under-approximation $\breve{\varphi} := \mathscr{C}^{cpl}(\varphi)$ is a classical logic formula, whilst φ is a modal logic formula. As the algorithm progresses, $\breve{\varphi}$ will be refined with classical clauses only, and so whilst there exists an n where $\text{refine}^n(\breve{\varphi})$ is K-satisfiable iff φ is so, we are unable to detect when we have reached this n. This differs from MOSAIC, where the upper bound $n = UB(\varphi)$ can be pre-calculated by recursively counting the number of diamond sub-formulae in φ. Second, CEGAR-tableaux already have a recursive step because they are based on recursive CEGAR loops, unlike RECAR which is based on just one CEGAR loop.

These differences arise because CEGARTAB considers only one world of the Kripke model at a time, as opposed to naive SAT-translation approaches that translate the whole input modal formula. We believe that keeping worlds separate is one of the main factors of CEGARTAB's success, and so we keep the

under-approximation as is. Thus, instead of implementing a faithful RECAR algorithm, we shall instead perform RECAR on each world. Our under-approximation is unchanged, but our over-approximation is best motivated by an example.

Algorithm 3 Algorithm of RECARTAB for incorporating (↯SAT) shortcuts

1: {Inputs: A is a set of unit assumptions,
   `Trie[σ]` is a node in our Trie containing modal clauses and a SAT-solver.}
2: Let t_σ := solve(`Trie[σ].sat`, A) {*is* $\mathscr{C}_\sigma^{cpl}$ *cpl-satisfiable*}
3: **if** t_σ = (unsat, UC_σ) **then**
4: **return** Unsatisfiable(UC_σ) {*because* $\mathscr{C}_\sigma$ *is* $\mathrm{K_n}$*-unsatisfiable*}
5: **else if** t_σ = (sat, ϑ_σ) **then**
6: Let $L = \{l \mid (c \to_l \langle\alpha\rangle d) \in$ `Trie[σ].DiaCl` and $c \in \vartheta_\sigma\}$
7: {*Check box and diamond clauses that fire under classical valuation* ϑ_σ}
8: **for** $i \in L$ **do**
9: Let $D = \{d \mid (c \to_l \langle\alpha\rangle d) \in$ `Trie[σ].DiaCl` and $c \in \vartheta_\sigma$ and $l = i\}$
10: Let $B = \{b \mid (a \to [\alpha]b) \in$ `Trie[σ].BoxCl` and $a \in \vartheta_\sigma\}$
11: {*evaluate the next* α*-successor using the jump rule on all fired diamonds*}
12: **if** RECARTAB($D \cup B$, `Trie[σ].child(α)`) = Unsatisfiable($UC_{\sigma;\alpha}$) **then**
13: Let $D_{UC} := UC_{\sigma;\alpha} \cap D$ { *How many diamonds in conflict set?*}
14: **if** $|D_{UC}| = 1$ **then**
15: {*Normal clause learning*}
16: Find $c \to_l \langle\alpha\rangle d$ where $d \in D_{UC}$
17: $CS := \{c\} \cup \{a \mid (a \to [\alpha]b) \in$ `Trie[σ].BoxCl`, $a \in \vartheta_\sigma$, $b \in UC_{\sigma;\alpha}\}$
18: Let $\varphi := \bigvee_{l \in CS} \neg l$
19: addClause(`Trie[σ].sat`, φ) { *Learn new clause* $\varphi := \neg \bigwedge CS$}
20: **else if** $|D_{UC}| < 1$ **then**
21: {*No diamond in conflict — pick random* $d \in D$ *and simulate normal clause learning*}
22: Pick arbitrary $d \in D$
23: Find $c \to_l \langle\alpha\rangle d$
24: $CS := \{c\} \cup \{a \mid (a \to [\alpha]b) \in$ `Trie[σ].BoxCl`, $a \in \vartheta_\sigma$, $b \in UC_{\sigma;\alpha}\}$
25: Let $\varphi := \bigvee_{l \in CS} \neg l$
26: addClause(`Trie[σ].sat`, φ)
27: **else**
28: { *(↯SAT) shortcut failed: multiple conflicting diamonds*}
29: **for** $(c \to_l \langle\alpha\rangle d) \in$ `Trie[σ].DiaCl` with $c \in \vartheta_\sigma$, $l = i$, $d \in UC_{\sigma;\alpha}$ **do**
30: Replace l with a fresh label l' in $c \to_l \langle\alpha\rangle d$
31: **end for**
32: **end if**
33: **return** RECARTAB(A, `Trie[σ]`) { *apply (restart)* }
34: **end if**
35: **end for**
36: **return** Satisfiable {*because every fired diamond is fulfilled*}
37: **end if**

Suppose we are creating a model with depth n, and each world is 'firing' two diamond clauses. This results in creating $\mathcal{O}(2^n)$ worlds, and as CEGARTAB must iterate over each of these worlds, this will clearly timeout for large n. Instead, let us create an over-approximation that will aim to reduce the number of successors for each node. Instead of creating two worlds for $\langle\alpha\rangle p \wedge \langle\alpha\rangle q$, one for p and one for q, we will attempt to create one α-successor with $(p\ ;\ q)$. If this fails we will get a conflict A', and if $\{p,q\} \subseteq A'$, we know we must separate these diamond clauses. Otherwise, we have just managed to create a model for w_0 that has one successor, not two. If every world succeeds in grouping diamond clauses we can create a model with $\mathcal{O}(n)$ worlds instead of $\mathcal{O}(2^n)$ worlds, as formalised next.

5.4 RECARTAB: the RECAR-tableaux over-approximation algorithm

We begin by annotating each dia-clause with a 'label', initialised to 1: i.e. we replace $a \to \langle\alpha\rangle b$ with $a \to_1 \langle\alpha\rangle b$, for all dia-clauses. Now, we create a sole successor for each fired *label*, instead of each fired *dia-clause*. The number of labels will increase as the algorithm progresses so let the set of labels be L.

Theorem 5.4. *Algorithm 3 for* RECARTAB *is a sound and complete decision procedure for* $\mathrm{K_n}$.

Proof.

Termination: Separating diamond clauses can happen only finitely many times. Thus for RECAR-tableaux to not terminate, it must either jump or restart infinitely, but both are impossible by the proofs for CEGAR-tableaux [9].

Soundness: Any clause learned by RECARTAB is of the form $\neg c \vee \neg a_1 \vee \cdots \vee \neg a_m$ as in line 17 and 24 and is a logical consequence of the discovery that for some label l, the set $\{c, a_1, \cdots, a_m, c \to_l \langle\alpha\rangle d, a_1 \to_l [\alpha]b_1, \cdots, a_m \to_l [\alpha]b_m\}$ is K-unsatisfiable just as they are in CEGARTAB, so it is sound to learn such a clause. But when the conflict set contains multiple diamonds, as in line 28, we separate these conflicting diamonds using different labels, and restart, which will reduce the cardinality of L at line 6. This corresponds to finding that $(\langle\alpha\rangle D; [\alpha]B)$ is K-unsatisfiable and then attempting to K-satisfy $(\langle\alpha\rangle D'; [\alpha]B)$ for some $D' \subset D$ instead. The move is sound because we maintain the requirement for ($\lightning$SAT)-shortcuts that if $(\langle\alpha\rangle D'; [\alpha]B)$ is K-satisfiable then so is each individual set $(\langle\alpha\rangle d'; [\alpha]B)$ for all $d' \in D'$. In the worst case, this will lead to every dia-clause having its own label, meaning that RECARTAB will just simulate CEGARTAB, which we know to be sound.

Completeness Suppose RECARTAB returns Satisfiable and proceed by induction on the number of jumps on the maximal sequence of jumps. If there are no jumps the Kripke model is just a dead-end w_ε with ϑ_ε. Otherwise, consider the first jump from this root world on this sequence for any $i \in L$. The fired diamonds consist of some set $\langle\alpha\rangle D = \{\langle\alpha\rangle d_1, \cdots, \langle\alpha\rangle d_{n>0}\}$ while the fired boxes are $[\alpha]B = \{[\alpha]b_1, \cdots, [\alpha]b_{m\geq 0}\}$. We jump to a child containing the assumptions $(D;B) = \{d_1, \cdots, d_n, b_1, \cdots, b_m\}$ and the recursive call at line 12 must have returned Satisfiable so this child w^i_α must be K-satisfiable by the induction hypothesis. Thus $(\langle\alpha\rangle D; [\alpha]B)$ is K-satisfiable at w_ε in the K-model obtained by putting $w_\varepsilon R_\alpha w^i_\alpha$. But then so must each set $(\langle\alpha\rangle d; [\alpha]B)$ for every $d \in D$ since $\langle\alpha\rangle(p \wedge q) \to \langle\alpha\rangle p \wedge \langle\alpha\rangle q$ is K-valid. Thus each $i \in L$ generates such an R_α-child for its corresponding diamond jump and the for-loop over these $i \in L$ at line 8 succeeds only when all such children return Satisfiable, meaning K-satisfiable. Thus every $(\langle\alpha\rangle d; [\alpha]B)$ fired by ϑ_ε has a K-satisfiable witness, giving a K-model for $\mathrm{SNF}_{mc}(\varphi_0)$.

□

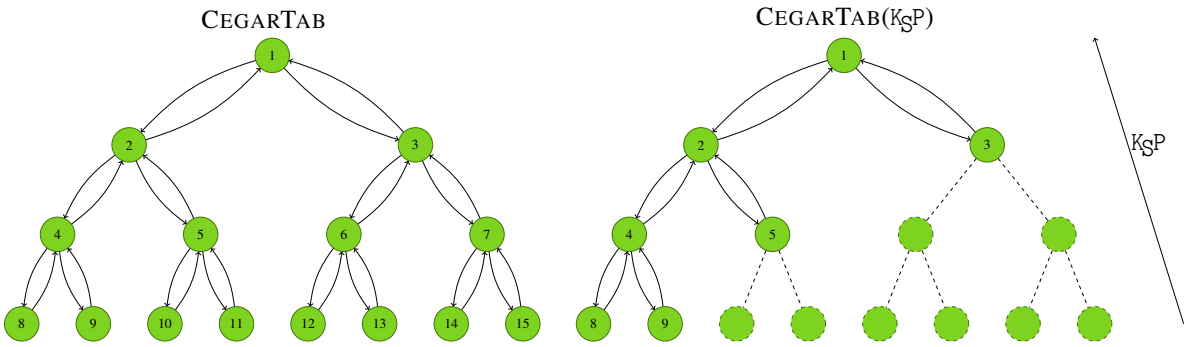

Figure 6: A comparison of the search process of CEGARTAB and CEGARTAB(KSP). CEGARTAB evaluates worlds with a pre-order traversal, and in the meantime, KSP saturates layers bottom-up. So CEGARTAB(KSP) generates no successors for "fixpoint" worlds 5 and 3 giving scenarios where CEGARTAB(KSP) evaluates a linear number of worlds, whilst CEGARTAB evaluates an exponential number.

We report on experiments on RECAR-Tableaux at the end of this section, but let us first propose our own approach to achieving (↯SAT) shortcuts.

5.5 (↯SAT) shortcuts via fixpoint detection using KSP

One of the main issues with RECARTAB is that it explicitly generates the model, meaning it suffers from the same shortcomings as regular CEGAR-tableaux. Sure, it uses heuristics to find smaller models, but there is no guarantee these smaller models exist, and this "optimism" might lead to wasted work.

We propose a different method of finding (↯SAT) shortcuts. Instead of explicitly creating a model, we simply check if one exists by searching for fixpoints in the CEGARTAB under-approximation where, intuitively, a fixpoint is a set of clauses which will never be refined with any new clauses by CEGARTAB.

Lemma 5.5 (Fixpoints are Satisfiable)**.** *If $\mathscr{A}_\sigma$ is a set of unit assumptions, and* CEGARTAB *will not refine $\mathscr{C}^{cpl}_\sigma$ with any clauses in the future, then $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$ is classically satisfiable iff $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$; $\mathscr{C}^{box}_\sigma$; $\mathscr{C}^{dia}_\sigma$ is* $\mathrm{K_n}$*-satisfiable.*

Proof. If $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$; $\mathscr{C}^{box}_\sigma$; $\mathscr{C}^{dia}_\sigma$ is $\mathrm{K_n}$-satisfiable then its subset $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$ is clearly classically satisfiable. If $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$ is classically satisfiable then CEGARTAB$(\mathscr{A}, \mathit{Trie}[\sigma])$ will find a classical valuation, then iterate over diamonds. For a contradiction, suppose a diamond jump returns Unsatisfiable. Then we learn a clause to add to $\mathscr{C}^{cpl}_\sigma$, contradicting that $\mathscr{C}^{cpl}_\sigma$ is a fixpoint. Thus all diamonds must be fulfilled, and so CEGARTAB$(\mathscr{A}, \mathit{Trie}[\sigma])$ returns Satisfiable. By the correctness of CEGARTAB, the set $\mathscr{A}_\sigma$; $\mathscr{C}^{cpl}_\sigma$; $\mathscr{C}^{box}_\sigma$; $\mathscr{C}^{dia}_\sigma$ is $\mathrm{K_n}$-satisfiable. □

Lemma 5.5 allows CEGARTAB to skip the potentially expensive process of generating a model if we know there are no clauses left to learn. This helps when a satisfying model is large, but "easy" to produce. But how to determine whether $\mathscr{C}^{cpl}_\sigma$ is a fixpoint? To this end we use KSP, as explained next.

5.6 CEGAR-tableaux with a resolution-based oracle

Recall that KSP uses SNF_{ml} rather than SNF_{mc} and that SNF_{ml} leads to a linear Trie data structure, as explained previously. Nevertheless, Lemma 5.5 still applies.

Algorithm 4 CEGARTAB(KSP)(A, Trie[l])

1: {Inputs: A is a set of unit assumptions,
Trie[l] contains the set $\mathscr{C}_l$ of modal clauses from $\text{SNF}_{ml}(\varphi_0)$ and a SAT-solver.}
2: {*Merge Step: Check if external KSP process has saturated layer l wrt layer $l+1$ (bottom-up)*}
3: **if** exists KSP-output-file for layer l **then**
4: Add new KSP clauses to Trie[l].sat
5: $Fixpoint$:= **true**
6: **end if**
7: Let t_l := solve(Trie[l].sat, A) {*is $\mathscr{C}_l^{cpl}$ cpl-satisfiable*}
8: **if** t_l = (unsat, UC_l) **then**
9: **return** Unsatisfiable(UC_l) {*because $\mathscr{C}_l$ is K_n-unsatisfiable*}
10: **else if** t_l = (sat, ϑ_l) **then**
11: **if** $Fixpoint$ = **true then**
12: **return** Satisfiable {*Fixpoint reached by KSP: skip child generation*}
13: **end if**
14: {*Check box and diamond clauses that fire under classical valuation ϑ_l*}
15: **for** every $(c \to \langle\alpha\rangle d) \in$ Trie[l].DiaCl with $c \in \vartheta_l$ **do**
16: Let $B = \{b \mid (a \to [\alpha]b) \in$ Trie[l].BoxCl and $a \in \vartheta_l\}$
17: {*evaluate the next layer using the jump rule on α*}
18: **if** CEGARTAB($(d;B)$, Trie[$l+1$])) = Unsatisfiable(UC_{l+1}) **then**
19: $CS := \{c\} \cup \{a \mid (a \to [\alpha]b) \in$ Trie[l].BoxCl and $a \in \vartheta_l$ and $b \in UC_{l+1}\}$
20: Let $\varphi := \bigvee_{l \in CS} \neg l$
21: addClause(Trie[l].sat, φ) { *Learn new clause $\varphi := \neg \bigwedge CS$*}
22: **return** CEGARTAB(A, Trie[l]) { *apply (restart)* }
23: **end if**
24: **end for**
25: **return** Satisfiable {*because every fired diamond is fulfilled*}
26: **end if**

Figure 7: Algorithm of CEGARTAB(KSP) on SNF_{ml} showing the integration of fixpoint detection.

Theorem 5.6 (KSP produces CEGARTAB fixpoints)**.** *Suppose that a **saturated** derivation from $\text{SNF}_{ml}(\varphi_0)$ by KSP results in a new set $\mathscr{C}_l'$ of modal clauses (at layer l). Then $\mathscr{C}_l'$ is a fixpoint of* CEGARTAB.

Proof. If $\mathscr{C}_l'$ is not a CEGARTAB fixpoint, the restart rule is applicable. Then the learnt clause corresponds exactly to one of the [GEN] resolution rule applications from Figure 4 by KSP, so the set of clauses is not saturated: contradiction. □

Our algorithm for "fixpoint detecting" CEGAR-Tableaux is CEGARTAB(KSP) but our implementation is CEGARBox++(KSP). For simplicity, CEGARBox++(KSP) is **multithreaded**, running CEGARBox++ and KSP independently, with naive communication via file writing/reading. We thank Cláudia Nalon for adjusting KSP to make this possible. The approach is as follows where both CEGARBox++ and KSP use the linear Trie for SNF_{ml}, not the tree-like Trie for SNF_{mc}, see Figure 6 and Figure 7:

1. Run KSP on an input formula φ_0, and have it print out $\text{SNF}_{ml}(\varphi_0)$;

2. KSP then performs (layered) resolution, from deepest (κ) to shallowest layers (0), and after fully saturating a layer, prints out all classical clauses it has learned;
3. CEGARTAB runs independently on $\mathrm{SNF}_{ml}(\varphi_0)$, from shallowest (0) to deepest (κ) layers, but when it receives clauses from KSP at layer l, it adds them to $\mathscr{C}_l^{cpl}$ (at modal layer l), marks the layer as a fixpoint and ignores the jump rule at that layer (sound by Lemma 5.5).

There are some subtle points in this integration which deserve noting. At any particular time, KSP is saturating some layer l with respect to layer $l+1$ using the resolution rules and adding cpl-clauses to layer l as required to avoid the jumps that would cause a clash at layer $l+1$. If CEGARBox++ arrives at layer l from layer $i-1$ *before* before KSP has saturated layer l wrt layer $l+1$, then the flag *Fixpoint* will be false and line 12 will be skipped, so CEGARBox++(KSP) mimics CEGARBox++ at layer l. Else, if KSP has already saturated layer l wrt layer $l+1$, and printed out new cpl-clauses, then *Fixpoint* is put to true at line 5. CEGARBox++ therefore imports the new cpl-clauses at line 4 and calls the SAT-solver at layer l at line 7 with the assumptions A. That is, CEGARBox++ checks whether the assumptions A demanded by the jump from layer $l-1$ to layer l are jointly cpl-satisfiable wrt the old cpl-clauses of layer l (inside the SAT-solver at layer l) plus these new cpl-clauses, But surely, KSP has already told us that layer l is satisfiable, so why do we need this check? Not quite: it is perfectly possible that the new cpl-clauses make layer l cpl-unsatisfiable under assumptions A as demanded by layer $l-1$, since KSP has not checked this possibility (yet). Thus the call to the SAT-solver at line 7 is essential in both cases.

At worst, this algorithm *should* be as strong as CEGARTAB. As we shall see, the combination works surprisingly well, despite the rather naive integration.

5.7 Experimental evaluation of the two new approaches

Our experiments used an Intel i7-11700F@2.50 GHz CPU with 16GB of RAM. We used the same K-benchmarks as in our original paper [9]. Our code can be found here: `https://github.com/cormackikkert/CEGARBoxCPP`. We used `MiniSAT` [7] as our SAT-solver, preliminary results showed that using other SAT-solvers leads to negligible performance differences.

Using CEGARBox++ as our control, we tested our two variants that implement (↯SAT) shortcuts: CEGARBox++(KSP) and RECARBox. Finally, as CEGARBox++(KSP) runs an instance of KSP "under the hood", we compare against KSP too. We use KSP(0.1.6), whereas our previous work [9] used version 0.1.3 (the latest at that time).

Since CEGARBox++(KSP) uses multi-threading, CPU-time is not an appropriate measure of performance, so all reported times are "wall time", not CPU-time.

Figure 8 shows that in all cases RECARBox is worse than CEGARBox++, likely meaning that the extra work required to look for smaller models is wasted. On the other hand we see that CEGARBox++(KSP) outperforms CEGARBox++ by a large margin on the MQBF benchmarks and is equal on the 3CNF benchmarks. However, CEGARBox++ outperforms CEGARBox++(KSP) on the LWB-K benchmarks. This results in both provers being about equal overall. RECARBox performs better than CEGARBox++(KSP) on the LWB-K benchmarks, which we believe is due to the presence of massive formulae in these benchmarks: see our conclusion.

Synergy vs. Parallelism Finally, as the integration involves multithreading, one might wonder whether the performance gains are due to the parallelism itself (effectively "cheating" by doubling resources). Specifically for the MQBF benchmarks, is CEGARBox++ *helping* to solve the satisfiable problems, or is it simply just KSP solving them all (since it is best on satisfiable formulae)?

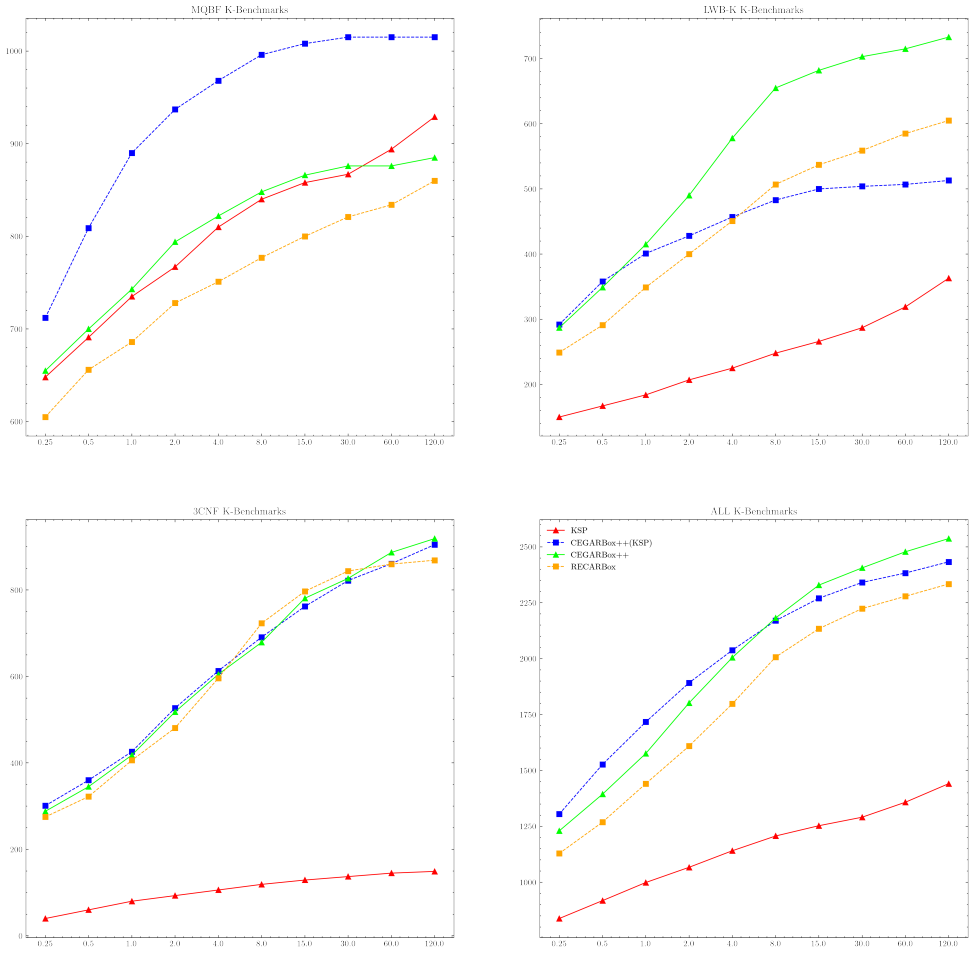

Figure 8: Performance of solvers on the standard K benchmarks

To this end, we compare CEGARBox++(KSP) with KSP in Figure 9. We can see that on hard satisfiable benchmarks, CEGARBox++(KSP) outperforms KSP, meaning that CEGARBox++ is contributing. We can see no satisfiable timeouts for CEGARBox(KSP), and even see a 100x improvement in some satisfiable problems, refuting the possibility that this improvement comes just from the doubling of resources due to parallelisation.

This highlights a distinction between a simple portfolio approach and our integration. While KSP and CEGARBox++ generally excel at different problem types (satisfiable vs. unsatisfiable), the integration appears to be better than the sum of their parts. Communicating fixpoints has allowed us to achieve results that a non-communicating portfolio could not.

6 Conclusions

We can see that RECAR-tableaux doesn't perform that well. We believe this is because it operates on a much stricter form of (↯SAT) shortcuts, where it tries to create smaller models, as opposed to CEGARTAB(KSP), which doesn't create models, but just shows they exist. Creating smallest models is intractable [4, 24] in general, and if they don't exist, RECAR-tableaux simply wastes its time.

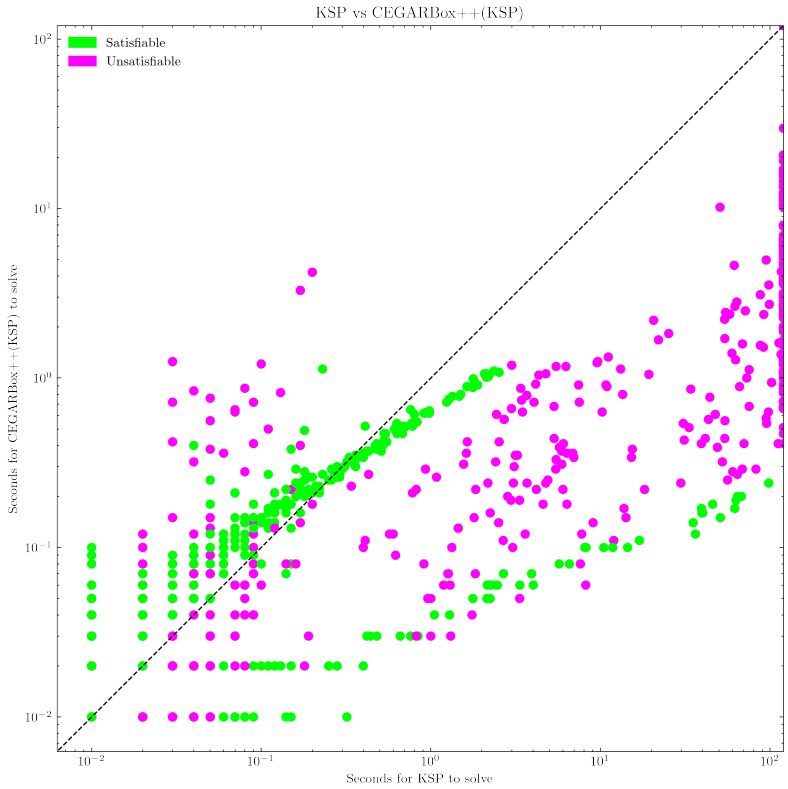

Figure 9: Performance of CEGARBox++(KSP) and KSP on the MQBF benchmarks. Problems on the top edge and the right edge indicate a timeout for CEGARBox++(KSP) and KSP respectively. Problems in the bottom right triangle/top left triangle are solved by CEGARBox++(KSP) / KSP quickest.

Our approach, CEGARTAB(KSP), for dealing with (↯SAT) shortcuts shows a lot of promise, outperforming all solvers on the MQBF benchmarks. We believe a better implementation would result in CEGARTAB(KSP) outperforming CEGARTAB across the board. In particular, the issue with the current approach is that the KSP normal forming procedure produces approximately two times as many clauses as CEGARTAB, resulting in slowdowns on **massive** formulae, which are present in the LWB-K benchmarks. These are why RECARBox performs better than CEGARBox++(KSP) on the LWB-K benchmarks. KSP can simplify this normal form (resulting in a smaller number of clauses than CEGARTAB), but on these massive formulae the simplification times out. A more polished integration would use CEGARTAB's normal forming procedure, and have better and more real-time communication protocols. We imagine this would be capable of outperforming CEGARTAB on the 3CNF and LWB-K benchmarks, whilst also full-solving the MQBF benchmarks. An implementation of such is easier said than done as KSP has certain requirements on its normal form and must produce extra clauses that are not needed for CEGARTAB. An alternative would be to instead optimise the simplification process of KSP, so it can produce reasonable normal forms quickly.

In general, this idea of searching for fixpoints is simply a method of detecting if $\check{\varphi}$ and φ are equisatisfiable, a normal idea in classical CEGAR, but more difficult in complicated CEGAR algorithms, where the under-approximation is a different type to the input (in our case a classical formula vs a modal formula). This fixpoint approach will be generally useful for other recursive CEGAR algorithms, such as

CAQE [20], or PDR [2], though of course it is most effective in applications such as CEGARTAB where branching is involved, meaning any (↯SAT) shortcut can possibly result in an exponential improvement in speed.

Whilst any approach could be used for fix-point detection, it is interesting to consider the implications of using resolution here. Previously, SAT-based, tableaux-based and resolution-based solvers were the state-of-the-art solvers used for reasoning in modal K. Now, CEGARTAB(KSP), is an elegant combination of all these approaches, that is greater than the sum of its parts. In particular, this highlights a key similarity of CEGARTAB and resolution methods. In CEGARTAB we learn clauses when we need to (i.e. *lazily*), whereas KSP learns clauses iteratively (i.e. *eagerly*).

We used KSP as an oracle for CEGARTAB. From discussion with Cláudia Nalon, Clare Dixon, Ullrich Hustadt, and Fabio Papacchini, creating an efficient implementation by reversing the roles is not obvious.

References

[1] Franz Baader, Ian Horrocks & Ulrike Sattler (2008): *Description logics*. *Foundations of Artificial Intelligence* 3, pp. 135–179.

[2] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In Ranjit Jhala & David A. Schmidt, editors: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, *Lecture Notes in Computer Science* 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7.

[3] Robert Brummayer & Armin Biere (2009): *Effective Bit-Width and Under-Approximation*. In Roberto Moreno-Díaz, Franz Pichler & Alexis Quesada-Arencibia, editors: *Computer Aided Systems Theory - EUROCAST 2009, 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers*, *Lecture Notes in Computer Science* 5717, Springer, pp. 304–311, doi:10.1007/978-3-642-04772-5_40.

[4] T. Y. Chen, Jean-Louis Lassez & Graeme S. Port (1986): *Maximal Unifiable Subsets and Minimal Nonunifiable Subsets*. *New Gener. Comput.* 4(2), pp. 133–152, doi:10.1007/BF03037439.

[5] Koen Claessen & Dan Rosén (2015): *SAT Modulo Intuitionistic Implications*. In Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, *Lecture Notes in Computer Science* 9450, Springer, pp. 622–637, doi:10.1007/978-3-662-48899-7_43.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. *J. ACM* 50(5), pp. 752–794, doi:10.1145/876638.876643.

[7] Niklas Eén (2006): *The Minisat page*. `http://minisat.se/`.

[8] Luca Geatti, Nicola Gigante & Angelo Montanari (2021): *BLACK: A Fast, Flexible and Reliable LTL Satisfiability Checker*. In Dario Della Monica, Gian Luca Pozzato & Enrico Scala, editors: *Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the Twelfth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2021), Padua, Italy, September 22, 2021*, *CEUR Workshop Proceedings* 2987, CEUR-WS.org, pp. 7–12. Available at `https://ceur-ws.org/Vol-2987/paper2.pdf`.

[9] Rajeev Goré & Cormac Kikkert (2021): *CEGAR-Tableaux: Improved Modal Satisfiability via Modal Clause-Learning and SAT*. In Anupam Das & Sara Negri, editors: *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, *Lecture Notes in Computer Science* 12842, Springer, pp. 74–91, doi:10.1007/978-3-030-86059-2_5.

[10] Rajeev Goré & Linh Anh Nguyen (2009): *Clausal Tableaux for Multimodal Logics of Belief*. *Fundam. Informaticae* 94(1), pp. 21–40, doi:10.3233/FI-2009-115.

[11] Rajeev Goré & Jimmy Thomson (2013): *An Improved BDD Method for Intuitionistic Propositional Logic: BDDIntKt System Description*. In Maria Paola Bonacina, editor: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, *Lecture Notes in Computer Science* 7898, Springer, pp. 275–281, doi:10.1007/978-3-642-38574-2_19.

[12] Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima & Valentin Montmirail (2017): *A Recursive Shortcut for CEGAR: Application To The Modal Logic K Satisfiability Problem*. In Carles Sierra, editor: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ijcai.org, pp. 674–680, doi:10.24963/ijcai.2017/94.

[13] Jianwen Li, Shufang Zhu, Geguang Pu & Moshe Y. Vardi (2015): *SAT-Based Explicit LTL Reasoning*. In Nir Piterman, editor: *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, *Lecture Notes in Computer Science* 9434, Springer, pp. 209–224, doi:10.1007/978-3-319-26287-1_13.

[14] Jianwen Li, Shufang Zhu, Geguang Pu, Lijun Zhang & Moshe Y. Vardi (2019): *SAT-based explicit LTL reasoning and its application to satisfiability checking*. *Formal Methods Syst. Des.* 54(2), pp. 164–190, doi:10.1007/S10703-018-00326-5.

[15] Grigori Mints (1988): *Gentzen-type systems and resolution rules part I propositional logic*. In: *International Conference on Computer Logic*, Springer, pp. 198–231, doi:10.1007/3-540-52335-9_55.

[16] Cláudia Nalon, Clare Dixon & Ullrich Hustadt (2019): *Modal Resolution: Proofs, Layers, and Refinements*. *ACM Trans. Comput. Log.* 20(4), pp. 23:1–23:38, doi:10.1145/3331448.

[17] Cláudia Nalon, Ullrich Hustadt & Clare Dixon (2017): *KSP: A Resolution-based Prover for Multimodal K, Abridged Report*. In Carles Sierra, editor: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ijcai.org, pp. 4919–4923, doi:10.24963/ijcai.2017/694.

[18] Linh Anh Nguyen (1999): *A New Space Bound for the Modal Logics K4, KD4 and S4*. In Miroslaw Kutylowski, Leszek Pacholski & Tomasz Wierzbicki, editors: *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, *Lecture Notes in Computer Science* 1672, Springer, pp. 321–331, doi:10.1007/3-540-48340-3_29.

[19] Thomas Pagram (2015): *Using decision diagrams for modal and intuitionisic theorem proving*. Australian National University.

[20] Markus N. Rabe & Leander Tentrup (2015): *CAQE: A Certifying QBF Solver*. In Roope Kaivola & Thomas Wahl, editors: *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, IEEE, pp. 136–143, doi:10.1109/FMCAD.2015.7542263.

[21] Grigori S Tseitin (1983): *On the complexity of derivation in propositional calculus*. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, doi:10.1007/978-3-642-81955-1_28.

[22] Moshe Y. Vardi (2009): *From Philosophical to Industrial Logics*. In Ramaswamy Ramanujam & Sundar Sarukkai, editors: *Logic and Its Applications, Third Indian Conference, ICLA 2009, Chennai, India, January 7-11, 2009. Proceedings*, *Lecture Notes in Computer Science* 5378, Springer, pp. 89–115, doi:10.1007/978-3-540-92701-3_7.

[23] Chao Wang, Aarti Gupta & Franjo Ivancic (2007): *Induction in CEGAR for Detecting Counterexamples*. In: *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, IEEE Computer Society, pp. 77–84, doi:10.1109/FAMCAD.2007.21.

[24] David A. Wolfram (1989): *Intractable Unifiability Problems and Backtracking*. *J. Autom. Reason.* 5(1), pp. 37–47, doi:10.1007/BF00245020.