Journal of Automated Reasoning (2020) 64:461–484
https://doi.org/10.1007/s10817-018-09503-x

KSP A Resolution-Based Theorem Prover for K_n: Architecture, Refinements, Strategies and Experiments

Cláudia Nalon[1] · **Ullrich Hustadt**[2] · **Clare Dixon**[2]

Received: 31 October 2018 / Accepted: 28 November 2018 / Published online: 17 December 2018

Abstract
In this paper we describe the implementation of KSP, a resolution-based prover for the basic multimodal logic K_n. The prover implements a resolution-based calculus for both local and global reasoning. The user can choose different normal forms, refinements of the basic resolution calculus, and strategies. We describe these options in detail and discuss their implications. We provide experiments comparing some of these options and comparing the prover with other provers for this logic.

Keywords Modal logics · Theorem proving · Resolution method

1 Introduction

Modal logics have long been used in Computer Science for describing and reasoning about complex systems, including programming languages [42], knowledge representation and reasoning [8,21,43], verification of distributed systems [18–20] and terminological reasoning [46]. The most basic of such logics is the multimodal K_n, which extends the classical language with new operators, $\Box_a$ and $\Diamond_a$, with $a \in A = \{1, \ldots, n\}$, a fixed finite set of indexes. A formula φ is interpreted with respect to a *Kripke Structure*, which comprises a set of worlds, a set of relations over the worlds, and an evaluation function which assigns an interpretation to every atomic formula at every world. This interpretation can then be lifted from atomic

C. Dixon was partially supported by the EPSRC funded RAI Hubs FAIR-SPACE (EP/R026092/1) and RAIN (EP/R026084/1), and the EPSRC funded programme Grant S4 (EP/N007565/1).

✉ Clare Dixon
CLDixon@liverpool.ac.uk

Cláudia Nalon
nalon@unb.br

Ullrich Hustadt
U.Hustadt@liverpool.ac.uk

[1] Department of Computer Science, University of Brasília, Brasília, DF, Brazil

[2] Department of Computer Science, University of Liverpool, Liverpool, UK

formulae to arbitrary formulae. Three related reasoning tasks have been extensively discussed in the literature:

(i) given a formula φ, the *local satisfiability problem* consists of showing that there is a model and a world in it that satisfies φ;
(ii) given a formula φ, the *global satisfiability problem* consists of showing that there is a model such that all worlds in this model satisfy φ;
(iii) given a set of formulae Γ and a formula φ, the *local satisfiability of φ under the global constraints (or assumptions) Γ* consists of showing that there is a model that globally satisfies all formulae in Γ and that there is a world in this model that satisfies φ.

Those reasoning tasks are far from trivial. The local satisfiability problem for the multi-modal propositional case is PSPACE-complete [21]. The global satisfiability and the local satisfiability under global constraint problems for K_n are EXPTIME-complete [50].

Several proof methods and tools for reasoning in K_n exist, either in the form of methods applied directly to the modal language [17,27] or obtained by translation into a more expressive target language (First-Order Logic [24] or Hybrid Logic [2], for instance). Translation-based methods benefit not only from the existence of available theorem provers, therefore not requiring additional effort for implementation, but also the strategies available for the target language can be almost immediately applied to the translated problem [23]. This is not the case for direct methods, where strategies need to be adapted to deal with the underlying normal forms and inference rules. However, the translation into a more expressive logic combined with a standard proof method for that logic may involve a computational overhead and may not necessarily result in a decision procedure for the set of translated formulae. Additionally, standard proof methods for the target logic may not normally include all optimisations and strategies that can be included in a direct method. For example, here we employ 'hyper-resolution-like' inferences that avoid the generation of intermediate resolvents and thereby reduce the search space.

We will focus on the resolution-based methods for K_n which are presented in [33,34]. Both calculi are clausal: a formula to be tested for satisfiability is first translated into a normal form, to which a set of inference rules are applied. The inference rules applied to propositional clauses (i.e. those where modal operators do not occur) are basically variants of the binary resolution rule [45]. For dealing with modal clauses, a set of hyper-resolution rules [44] are applied to modal and propositional clauses. The main difference between those proof methods resides in their normal form: in [33], for completeness, all clauses are considered for application of the resolution rules; whilst in [34], because clauses are labelled, the resolution rules only need to be applied when the labels of clauses can be unified. Differently from other calculi which use labels for guiding the application of inference rules [3,5,10,11,54,55], the labels in the clausal form given in [34] do not refer to worlds, but to the modal layer (i.e. the distance from the root of a model) where a subformula holds.

Both proof methods have been implemented in our prover, KSP [36], a resolution-based prover for the multi-modal logic K. The structure of the normal form restricts application of the resolution inference rules reducing the search space whilst remaining complete. The prover also uses the set of support strategy [56], a strategy that requires that the set of clauses is partitioned in two sets and, then, restricts that clauses used as premises for resolution inferences are from different sets in this partition. For the modal case, the use of labels relating to modal layer of subformulae allows us to restrict clause selection even further. The prover also incorporates a range of simplification techniques and refinements intended to improve efficiency of the prover. Here, we concentrate on the implementation aspects of those techniques and refinements, further discussing the architecture of the prover, a variety

of choices available to the user, and their impact on the efficiency of the prover. The resulting prover outperforms other modal provers for formulae that have a high degree of nesting of modal formulae. This paper extends the work in [35] providing full details of the prover, its main control loop, pre-processing options, notions of redundancy, refinements and strategies. The prover is based on the calculus presented in [34] and its sources are available at [36]. We also provide experimental results comparing KSP with other provers and analysing some of the combinations of refinements and strategies. The results update and extend those in [35] by using more recent versions of the provers involved, presenting additional experimental results and considering the performance of portfolios of provers.

The paper is organised as follows. We introduce the syntax and semantics of K_n in Sect. 2. In Sects. 3 and 4 we briefly describe the normal form and the calculus. Section 5 describes the available strategies and their implementations. The evaluation of strategies and of the performance of the prover compared to existing tools are given in Sect. 6. We summarise our results and provide conclusions in Sect. 7.

2 Language

Let $A = \{1, \ldots, n\}$, $n \in \mathbb{N}$, be a finite fixed set of indexes and $P = \{p, q, s, t, p', q', \ldots\}$ be a denumerable set of propositional symbols. The *set of well-formed formulae*, WFF_K, is the least set such that every $p \in P$ is in WFF_K; if φ and ψ are in WFF_K, then so are $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $\boxed{a}\varphi$, and $\diamondsuit\!\!\!\!\!\!\diamondsuit\varphi$ for each $a \in A$. The formulae **false**, **true**, $(\varphi \Rightarrow \psi)$, and $(\varphi \Leftrightarrow \psi)$ are introduced as the usual abbreviations for $(\varphi \wedge \neg\varphi)$,¬**false**, and $(\neg\varphi \vee \psi)$, and $((\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi))$, respectively (where $\varphi, \psi \in \mathsf{WFF}_\mathsf{K}$).

A *literal* is either a propositional symbol or its negation; the set of literals is denoted by L. We denote by $\neg l$ the *complement* of the literal $l \in L$, that is, $\neg l$ denotes $\neg p$ if l is the propositional symbol p, and $\neg l$ denotes p if l is the literal $\neg p$. A *modal literal* is either $\boxed{a}l$ or $\diamondsuit\!\!\!\!\!\!\diamondsuit l$, where $l \in L$ and $a \in A$. The modal depth of a formula is recursively defined as follows:

Definition 1 Let $\varphi, \psi \in \mathsf{WFF}_\mathsf{K}$ be well-formed formulae. The modal depth of a formula is given by the function $\mathsf{mdepth} : \mathsf{WFF}_\mathsf{K} \longrightarrow \mathbb{N}$, where:

- $\mathsf{mdepth}(p) = 0$, for $p \in P$;
- $\mathsf{mdepth}(\neg\varphi) = \mathsf{mdepth}(\varphi)$;
- $\mathsf{mdepth}(\varphi \wedge \psi) = \mathsf{max}(\mathsf{mdepth}(\varphi), \mathsf{mdepth}(\psi))$;
- $\mathsf{mdepth}(\varphi \vee \psi) = \mathsf{max}(\mathsf{mdepth}(\varphi), \mathsf{mdepth}(\psi))$;
- $\mathsf{mdepth}(\boxed{a}\varphi) = 1 + \mathsf{mdepth}(\varphi)$;
- $\mathsf{mdepth}(\diamondsuit\!\!\!\!\!\!\diamondsuit\varphi) = 1 + \mathsf{mdepth}(\varphi)$.

The modal level of a subformula is given relative to its position in the syntactic tree of its superformula.

Definition 2 Let φ, φ' be well-formed formulae. Let Σ be the alphabet $\{1, 2, .\}$ and Σ^* the set of all finite sequences over Σ. The empty sequence in Σ^* is denoted by ε. Let $\tau : \mathsf{WFF}_\mathsf{K} \times \Sigma^* \times \{+, -\} \times \mathbb{N} \longrightarrow 2^{\mathsf{WFF}_\mathsf{K} \times \Sigma^* \times \{+,-\} \times \mathbb{N}}$ be the partial function inductively defined as follows (where $\lambda \in \Sigma^*$, $pol \in \{+, -\}$, $ml \in \mathbb{N}$, and the complement of a symbol in $\{+, -\}$ is given as $\mathsf{comp}(+) = -$, $\mathsf{comp}(-) = +$):

- $\tau(p, \lambda, pol, ml) = \{(p, \lambda, pol, ml)\}$, for $p \in P$;

- $\tau(\neg\varphi, \lambda, pol, ml) = \{(\neg\varphi, \lambda, pol, ml)\} \cup \tau(\varphi, \lambda.1, \mathsf{comp}(pol), ml)$;
- $\tau(\boxed{a}\varphi, \lambda, pol, ml) = \{(\boxed{a}\varphi, \lambda, pol, ml)\} \cup \tau(\varphi, \lambda.1, pol, ml + 1)$;
- $\tau(\diamondsuit\varphi, \lambda, pol, ml) = \{(\diamondsuit\varphi, \lambda, pol, ml)\} \cup \tau(\varphi, \lambda.1, pol, ml + 1)$;
- $\tau(\varphi \wedge \varphi', \lambda, pol, ml) = \{(\varphi \wedge \varphi', \lambda, pol, ml)\} \cup \tau(\varphi, \lambda.1, pol, ml) \cup \tau(\varphi', \lambda.2, pol, ml)$;
- $\tau(\varphi \vee \varphi', \lambda, pol, ml) = \{(\varphi \vee \varphi', \lambda, pol, ml)\} \cup \tau(\varphi, \lambda.1, pol, ml) \cup \tau(\varphi', \lambda.2, pol, ml)$.

The function τ applied to $(\varphi, \varepsilon, +, 0)$ returns the *annotated syntactic tree for* φ, where each node is uniquely identified by a subformula, its path order (or its position) in the tree, its polarity, and its modal level.

Definition 3 Let φ be a formula and let $\tau(\varphi, \varepsilon, +, 0)$ be its annotated syntactic tree. If $(\varphi', \lambda, pol, ml) \in \tau(\varphi, \varepsilon, +, 0)$, then the modal level of φ' in φ is given by $\mathsf{mlevel}(\varphi, \varphi', \lambda) = ml$ and the polarity of φ' in φ is given by $\mathsf{pol}(\varphi, \varphi', \lambda) = pol$.

If $\mathsf{mlevel}(\varphi, \varphi', \lambda) = ml$ we say that φ' at position λ of φ occurs at the modal level ml. For instance, p occurs three times in the formula $\boxed{a}\,\boxed{a}(p \wedge (\neg\boxed{a}\,p \vee \diamondsuit p))$, at position 1.1.1 at modal level 2; and at positions 1.1.2.1.1.1 and 1.1.2.2.1 at modal level 3. Let φ' be a subformula at position λ of a formula φ. If $\mathsf{pol}(\varphi, \varphi', \lambda) = +$, we say that φ' has positive polarity at λ. Similarly, if $\mathsf{pol}(\varphi, \varphi', \lambda) = -$, we say that φ' has negative polarity at λ. If for all positions λ at the modal level ml, we have that either $\mathsf{pol}(\varphi, \varphi', \lambda) = +$ or $\mathsf{pol}(\varphi, \varphi', \lambda) = -$, then φ' is said to be *pure at the modal level ml*. Finally, if φ' is pure at all modal levels, then φ' is said to be a *pure*. For example, taking φ to be the formula $\boxed{a}\,\boxed{a}(p \wedge (\neg\boxed{a}\,p \vee \diamondsuit p))$ above, then p occurs only with positive polarity at modal level 2 and with both negative and positive polarity at the modal level 3. Thus, p is pure at the modal level 2, but it is not pure at the modal level 3 (hence, it is not pure when considering the whole formula). A literal l is *pure at the modal level ml* if is either of the form p or $\neg p$ and p is pure at the modal level ml. If a literal l is pure at all modal levels, then we say l is a *pure literal*.

Modal formulae are interpreted over (rooted) Kripke models:

Definition 4 A *Kripke model M for n agents over P* is given by a tuple

$$(W, w_0, R_1, \ldots, R_n, \pi),$$

where W is a set of possible *worlds* with a distinguished world w_0, each *accessibility relation* R_a is a binary relation on W such that their union is a tree with *root* w_0, and $\pi : W \to (P \to \{true, false\})$ is a function which associates with each world $w \in W$ an interpretation to propositional symbols.

Definition 5 Satisfaction of a formula at a world w of a model M is defined inductively, as follows:

- $(M, w) \models p$ if, and only if, $\pi(w)(p) = true$, where $p \in P$;
- $(M, w) \models \neg\varphi$ if, and only if, $(M, w) \not\models \varphi$;
- $(M, w) \models (\varphi \wedge \psi)$ if, and only if, $(M, w) \models \varphi$ and $(M, w) \models \psi$;
- $(M, w) \models (\varphi \vee \psi)$ if, and only if, $(M, w) \models \varphi$ or $(M, w) \models \psi$;
- $(M, w) \models \boxed{a}\varphi$ if, and only if, for all w', wR_aw' implies $(M, w') \models \varphi$;
- $(M, w) \models \diamondsuit\varphi$ if, and only if, there is w' such that wR_aw' and $(M, w') \models \varphi$.

Let $M = (W, w_0, R_1, \ldots, R_n, \pi)$ be a model. A formula φ is *locally satisfied in* M, denoted by $M \models_L \varphi$, if $(M, w_0) \models \varphi$. The formula φ is *locally satisfiable* if there is a model M such that $(M, w_0) \models \varphi$. A formula φ is *globally satisfied in* M, denoted by $M \models_G \varphi$, if for all $w \in W$, $(M, w) \models \varphi$. A formula φ is said to be *globally satisfiable* if there is a model M

such that M globally satisfies φ. Satisfiability of a set of formulae is defined as usual. Given a set of formulae Γ, a formula φ is *locally satisfiable under global assumptions* Γ, if there is a model M such that $M \models_G \Gamma$ and $M \models_L \varphi$.

A model $M = (W, w_0, R_1, \ldots, R_n, \pi)$ is *tree-like* if $\bigcup_{a=1}^{n} R_a$ is a tree, i.e. a directed acyclic graph (with root w_0). As a formula is locally satisfiable if, and only if, it is locally satisfiable in a tree-like model [21], from now on we will only consider such a class of models. We denote by $\mathsf{depth}(w)$ the length of the unique path from w_0 to w through the union of the accessibility relations in M. We call a *modal layer* the equivalence class of worlds at the same depth in a model.

We note that checking the local satisfiability of a formula φ can be reduced to the problem of checking the local satisfiability of its subformulae at the modal layer of a model which corresponds to the modal level where those subformulae occur (see [1]). Due to this close correspondence of modal layer and modal level we use the terms interchangeably.

Also, checking the global satisfiability of φ can be reduced to checking the local satisfiability of φ at all modal layers (up to an exponential distance from the root) of a model [14,50]. The following definitions and results are needed later. Let K_n^* be the extension of K_n with an additional operator $\boxed{*}$, the universal operator. Let $M = (W, w_0, R_1, \ldots, R_n, \pi)$ be a tree-like model for K_n. The model M^* is the tuple $(W, w_0, R_1, \ldots, R_n, R_*, \pi)$, where $R_* = W \times W$. A formula $\boxed{*}\varphi$ is locally satisfied at the world w in the model M^*, written $(M^*, w) \models_L \boxed{*}\varphi$, if, and only if, for all $w' \in W$, we have that $(M^*, w') \models \varphi$. Given these definitions, for φ in WFF_K, deciding $M \models_G \varphi$ is equivalent to deciding $M^* \models_L \boxed{*}\varphi$. Also, deciding if a formula φ is satisfiable under the global assumptions $\Gamma = \{\gamma_1, \ldots, \gamma_m\}, m \in \mathbb{N}$, is equivalent to deciding $M^* \models_L \varphi \wedge \boxed{*}(\gamma_1 \wedge \ldots \wedge \gamma_m)$.

Thus, a uniform approach based on modal levels can be used to deal with all satisfiability problems.

3 Layered Normal Form

The calculi presented in [33,34] are both clausal. We present the normal form described in [34], as this normal form can be also used to simulate the one given in [33]. A formula to be tested for local or global satisfiability is first translated into a normal form called *Separated Normal Form with Modal Levels*, SNF_{ml}. A formula in SNF_{ml} is a conjunction of clauses labelled by the modal level at which they occur. We write $ml : \varphi$ to denote that φ holds at the modal level $ml \in \mathbb{N} \cup \{*\}$. By $* : \varphi$ we mean that φ holds at all modal levels. Formally, let $\mathsf{WFF}_\mathsf{K}^{ml}$ be the set of formulae $ml : \varphi$ such that $ml \in \mathbb{N} \cup \{*\}$ and $\varphi \in \mathsf{WFF}_\mathsf{K}$. Let $M^* = (W, w_0, R_1, \ldots, R_n, R_*, \pi)$ be a model and $\varphi \in \mathsf{WFF}_\mathsf{K}$. Satisfiability of labelled formulae is given as follows:

- $M^* \models ml : \varphi$ if, and only if, for all worlds $w \in W$ such that $\mathsf{depth}(w) = ml$, we have $\langle M^*, w\rangle \models \varphi$;
- $M^* \models * : \varphi$ if, and only if, $M^* \models \boxed{*}\varphi$.

Note that labels in a formula work as a kind of *weak* universal operator, allowing us to talk about formulae that are satisfied at a given modal layer.

Clauses in SNF_{ml} are in one of the following forms:

- Literal clause $\quad ml : \bigvee_{b=1}^{r} l_b$
- Positive a-clause $\quad ml : l' \Rightarrow \boxed{a} l$
- Negative a-clause $\quad ml : l' \Rightarrow \diamondsuit l$

Table 1 The translation function ρ

Let φ and φ' be well-formed formulae, t' be a new propositional symbol, and let $* + 1 = *$. The translation function $\rho : \mathsf{WFF}^{ml}_{\mathsf{K}} \longrightarrow \mathsf{WFF}^{ml}_{\mathsf{K}}$ is defined as:

$$
\begin{aligned}
\rho(ml : t \Rightarrow \varphi \wedge \varphi') &= \rho(ml : t \Rightarrow \varphi) \wedge \rho(ml : t \Rightarrow \varphi') \\
\rho(ml : t \Rightarrow \boxed{a}\,\varphi) &= (ml : t \Rightarrow \boxed{a}\,\varphi), \text{ if } \varphi \text{ is a literal} \\
&= (ml : t \Rightarrow \boxed{a}\,t') \wedge \rho(ml + 1 : t' \Rightarrow \varphi), \text{ otherwise} \\
\rho(ml : t \Rightarrow \diamondsuit_a\,\varphi) &= (ml : t \Rightarrow \diamondsuit_a\,\varphi), \text{ if } \varphi \text{ is a literal} \\
&= (ml : t \Rightarrow \diamondsuit_a\,t') \wedge \rho(ml + 1 : t' \Rightarrow \varphi), \text{ otherwise} \\
\rho(ml : t \Rightarrow \varphi \vee \varphi') &= (ml : \neg t \vee \varphi \vee \varphi'), \text{ if } \varphi, \varphi' \text{ are disjunctions of literals} \\
&= \rho(ml : t \Rightarrow \varphi \vee t') \wedge \rho(ml : t' \Rightarrow \varphi'), \text{ otherwise}
\end{aligned}
$$

where $ml \in \mathbb{N} \cup \{*\}$ and $l, l', l_b \in L$. Clauses are kept in simplified form, that is, no duplicate literals are allowed and a clause such as $ml : C \vee l \vee \neg l$ simplifies to $ml : \mathbf{true}$. As the disjunction operator is commutative, associative, and idempotent, simplification takes place regardless of the order of literals in a clause. Positive and negative a-clauses are together known as *modal a-clauses*; the index a may be omitted if it is clear from the context. A literal clause $ml : C$ is said to be positive (resp. negative) if all literals l occurring in C are of the form p (resp. $\neg p$), for $p \in P$.

Let φ be a formula in the language of K_n. In the following, we assume φ is in Negation Normal Form (NNF), that is, a formula where the operators are restricted to $\wedge$, $\vee$, $\boxed{a}$, $\diamondsuit_a$ and $\neg$; also, only propositions are allowed in the scope of negations. The transformation of a formula φ into SNF_{ml} is achieved by recursively applying rewriting and renaming [41]. Let φ be a formula and t a propositional symbol not occurring in φ. For local satisfiability, the translation of φ is given by $0 : t \wedge \rho(0 : t \Rightarrow \varphi)$, where t is a new propositional symbol and the transformation function $\rho : \mathsf{WFF}^{ml}_{\mathsf{K}} \longrightarrow \mathsf{WFF}^{ml}_{\mathsf{K}}$ is defined in Table 1. We refer to clauses of the form $0 : D$, for a disjunction of literals D, as *initial clauses*. For global satisfiability, the translation of φ is given by $* : t \wedge \rho(* : t \Rightarrow \varphi)$ where t is a new propositional symbol. For testing the satisfiability of φ under global assumptions $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$, $m \in \mathbb{N}$, the translation is given by $* : t \wedge \rho(0 : t \Rightarrow \varphi) \wedge \rho(* : t \Rightarrow \gamma_1 \wedge \ldots \wedge \gamma_m)$.

As the conjunction operator is commutative, associative, and idempotent, in the following we often refer to a formula in SNF_{ml} as a set of clauses. The next lemma shows that the transformation into SNF_{ml} is satisfiability preserving.

Lemma 1 [34] *Let $\varphi \in \mathsf{WFF}_{\mathsf{K}}$ be a formula and let t be a propositional symbol not occurring in φ. Then: (1) φ is locally satisfiable if, and only if, $0 : t \wedge \rho(0 : t \Rightarrow \varphi)$ is satisfiable; (2) φ is globally satisfiable if, and only if, $* : t \wedge \rho(* : t \Rightarrow \varphi)$ is satisfiable.*

The proof is standard. For the only if part, if φ is satisfiable, then there is a model $M = (W, w_0, R_1, \ldots, R_n, \pi)$ that satisfies φ. We build a model $M' = (W, w_0, R_1, \ldots, R_n, \pi')$ where the valuation of π' of the new symbols introduced by renaming are set to *true* exactly at the worlds where the formulae they are replacing are also evaluated to *true*. For the if part, if there is a model M that satisfies the translation of φ, by ignoring the labels and the valuation of the propositional symbols not occurring in φ, we show that M also satisfies φ.

The fact that the transformation for formulae under global assumptions is also satisfiability preserving follows easily from Lemma 1.

Some of the refinements implemented in KSP require further transformation of the set of clauses. For instance, for completeness of negative resolution [44], we require that literals occurring in the scope of modal operators are positive. Given a set of clauses in SNF_{ml}, in addition to the rules given in Table 1, we exhaustively apply the following rewriting rules (where $ml \in \mathbb{N} \cup \{*\}$, t, $p \in P$, and t' is a new propositional symbol):

$$\rho(ml : t \Rightarrow \boxed{a}\neg p) = (ml : t \Rightarrow \boxed{a}t') \wedge \rho(ml + 1 : t' \Rightarrow \neg p)$$
$$\rho(ml : t \Rightarrow \diamondsuit\neg p) = (ml : t \Rightarrow \diamondsuit t') \wedge \rho(ml + 1 : t' \Rightarrow \neg p)$$

We call the resulting normal form SNF^{+}_{ml}. It can be shown that the resulting set of clauses is satisfiable if, and only if, the original set of clauses is satisfiable.

Completeness of ordered resolution [22] requires that literals in the scope of modal operators are "small enough" with respect to a given ordering on literals. Also for completeness, those literals need to be available in the set of literal clauses so that the relevant clauses used in the hyper-resolution rules are derived. We can ensure these conditions are met by further processing of the set of SNF_{ml} clauses. Let Φ be a set of clauses and P_Φ be the set of propositional symbols occurring in Φ. Let $\succ$ be a well-founded and total ordering on P_Φ. This ordering can be extended to literals L_Φ over P_Φ by setting $\neg p \succ p$ and $p \succ \neg q$ whenever $p \succ q$, for all $p, q \in P_\Phi$. A literal l is said to be *maximal* with respect to a clause $ml : C \vee l$ if, and only if, there is no l' occurring in C such that $l' \succ l$. Given a set of clauses Φ in SNF_{ml} and an ordering on the literals occurring in Φ, in addition to the rules given in Table 1, we exhaustively apply the following rewriting rules (where $ml \in \mathbb{N} \cup \{*\}$, $t \in P$, $l \in L$ and t' is a new propositional symbol):

$$\rho(ml : t \Rightarrow \boxed{a}l) = (ml : t \Rightarrow \boxed{a}t') \wedge \rho(ml + 1 : t' \Rightarrow l)$$
$$\rho(ml : t \Rightarrow \diamondsuit l) = (ml : t \Rightarrow \diamondsuit t') \wedge \rho(ml + 1 : t' \Rightarrow l)$$

where $p \succ t'$, for all p occurring in Φ. We call the resulting normal form SNF^{++}_{ml}. Again, it is easy to show that Φ is satisfiable if, and only if, the resulting set of clauses in SNF^{++}_{ml} is satisfiable.

4 Inference Rules

The calculus comprises a set of inference rules for dealing with propositional and modal reasoning. In the following, we denote by σ the result of unifying the labels in the premises for each rule. Formally, unification is given by the function $\sigma : 2^{\mathbb{N}\cup\{*\}} \longrightarrow \mathbb{N} \cup \{*\}$, where $\sigma(\{ml, *\}) = ml$; and $\sigma(\{ml\}) = ml$; otherwise, σ is undefined. The inference rules given in Table 2 can only be applied if the unification of their labels is defined (where $* - 1 = *$). Note that for GEN1 and GEN3, if the modal clauses in the premises occur at the modal level ml, then the literal clause in the premises occurs at the next modal level, $ml + 1$.

Definition 6 Let Φ be a set of clauses in SNF_{ml}. A *derivation from* Φ is a sequence of sets $\Phi_0, \Phi_1, \ldots$ where $\Phi_0 = \Phi$ and, for each $i > 0$, $\Phi_{i+1} = \Phi_i \cup \{D\}$, where $D \notin \Phi_i$ is the resolvent obtained from Φ_i by an application of either LRES, MRES, GEN1, GEN2, or GEN3. We also require that D is in simplified form and that D is not a tautology. A set of clauses Φ is *saturated* if every clause that is a resolvent obtained from Φ by an application of either LRES, MRES, GEN1, GEN2, or GEN3 is either a tautology or it is already contained in Φ. A *local refutation for* Φ is a derivation $\Phi_0, \ldots, \Phi_k$, $k \in \mathbb{N}$, where $0 : \mathbf{false} \in \Phi_k$. A *global refutation for* Φ is a derivation $\Phi_0, \ldots, \Phi_k$, $k \in \mathbb{N}$, where $* : \mathbf{false} \in \Phi_k$. A derivation

Table 2 Inference rules, where $ml = \sigma(\{ml_1, \ldots, ml_{m+1}, ml_{m+2} - 1\})$ in GEN1, GEN3, where $m \geq 0$; $ml = \sigma(\{ml_1, ml_2\})$ in LRES, MRES; and $ml = \sigma(\{ml_1, ml_2, ml_3\})$ in GEN2

$$[\mathsf{LRES}]\quad \frac{ml_1 : D \vee l \qquad ml_2 : D' \vee \neg l}{ml : D \vee D'}$$

$$[\mathsf{MRES}]\quad \frac{ml_1 : l_1 \Rightarrow \Box_a l \qquad ml_2 : l_2 \Rightarrow \Diamond_a \neg l}{ml : \neg l_1 \vee \neg l_2}$$

$$[\mathsf{GEN2}]\quad \frac{ml_1 : l'_1 \Rightarrow \Box_a l_1 \qquad ml_2 : l'_2 \Rightarrow \Box_a \neg l_1 \qquad ml_3 : l'_3 \Rightarrow \Diamond_a l_2}{ml : \neg l'_1 \vee \neg l'_2 \vee \neg l'_3}$$

$$[\mathsf{GEN1}]\quad \frac{\begin{array}{l} ml_1 : l'_1 \Rightarrow \Box_a \neg l_1 \\ \vdots \\ ml_m : l'_m \Rightarrow \Box_a \neg l_m \\ ml_{m+1} : l' \Rightarrow \Diamond_a \neg l \\ ml_{m+2} : l_1 \vee \ldots \vee l_m \vee l \end{array}}{ml : \neg l'_1 \vee \ldots \vee \neg l'_m \vee \neg l'}$$

$$[\mathsf{GEN3}]\quad \frac{\begin{array}{l} ml_1 : l'_1 \Rightarrow \Box_a \neg l_1 \\ \vdots \\ ml_m : l'_m \Rightarrow \Box_a \neg l_m \\ ml_{m+1} : l' \Rightarrow \Diamond_a l \\ ml_{m+2} : l_1 \vee \ldots \vee l_m \end{array}}{ml : \neg l'_1 \vee \ldots \vee \neg l'_m \vee \neg l'}$$

$\Phi_0, \ldots, \Phi_i$ from Φ is *terminating* if it is either a local (resp. global) refutation for Φ or if there is a Φ_i, $i \in \mathbb{N}$, such that Φ_i is saturated.

For the satisfiability problem under global assumptions, a refutation is either a local or a global refutation. The following theorems, taken from [34] where full proofs are given, ensure the calculus is sound, complete and terminating.

Theorem 1 (Soundness [34]) *Let Φ be a set of clauses in SNF_{ml} and $\Phi_0, \ldots, \Phi_k$, $k \in \mathbb{N}$, be a derivation for Φ. If Φ is satisfiable, then every Φ_i, $0 \leq i \leq k$, is satisfiable.*

Soundness of the calculus is proved by showing that, for each inference rule, if the premises are satisfiable, so it is the resolvent.

Theorem 2 (Completeness [34]) *Let Φ be an unsatisfiable set of clauses in SNF_{ml}. Then there is a refutation for Φ by applying the resolution rules given in Table 2.*

Completeness is proved by showing that if a set Φ of clauses in SNF_{ml} is unsatisfiable, there is a refutation produced by the method presented here. The proof is by induction on the number of nodes of a graph, known as *behaviour graph* [9], built from Φ. Intuitively, nodes in the graph correspond to worlds and the set of edges correspond to the accessibility relations in a model. The graph construction is similar to the construction of a canonical model, followed by filtrations based on the set of clauses, often used to prove completeness for proof methods in modal logics [6]. We show that deletions of nodes in the graph correspond to application of the inference rules given in Table 2. If the reduced graph is empty, then the set of clauses is unsatisfiable and there is refutation from Φ. If the reduced graph is not empty, then a model witnessing the satisfiability of Φ can be built from it.

Theorem 3 (Termination [34]) *Let Φ be a set of clauses in SNF_{ml}. Then, any derivation from Φ terminates.*

This result follows from the fact that none of the inference rules generates new literals, new modal literals, or new labels. Hence, there are a finite number of clauses that can be built from the literals and modal literals occurring in Φ.

We note that, for a given formula φ, the normal form given in [33], called SNF, is equivalent to $(* : \mathbf{start} \Rightarrow t) \wedge \rho(* : t \Rightarrow \varphi)$, where t is a new propositional symbol and **start** is a

Table 3 Inference rules for initial clauses

$$\text{[IRES1]}\quad \frac{*:\ \mathbf{start} \Rightarrow D \vee l \qquad *:\ \mathbf{start} \Rightarrow D' \vee \neg l}{*:\ \mathbf{start} \Rightarrow D \vee D'} \qquad \text{[IRES2]}\quad \frac{*:\ \mathbf{start} \Rightarrow D \vee l \qquad *:\ D' \vee \neg l}{*:\ \mathbf{start} \Rightarrow D \vee D'}$$

constant denoting the root of the tree-like model. Two additional inference rules, given in Table 3 are required for completeness. In this case, a *refutation for a set of clauses* Φ is either a global refutation or a derivation $\Phi_0, \ldots, \Phi_k, k \in \mathbb{N}$, where $* : \mathbf{start} \Rightarrow \mathbf{false} \in \Phi_k$. Also, by taking $(0 : t) \wedge \rho(* : t \Rightarrow \varphi)$ as the normal form of φ, the calculus presented in [33] can be simulated by the one given in [34], without the rules given in Table 3. Instead of having clauses of the form $* : \mathbf{start} \Rightarrow \varphi$, we have $0 : \varphi$. Then, LRES simulates IRES1 with $ml_1 = ml_2 = 0$; and IRES2 with $ml_1 = 0$ and $ml_2 = *$.

5 Implementation

KSP is an implementation, written in C, of the calculus described in Sect. 4. The prover was designed to support experimentation with different combinations of refinements of its basic calculus. Refinements and options for (pre)processing the input are coded as independently as possible in order to allow for the easy addition and testing of new features. This might not lead to optimal performance (e.g. some techniques need to be applied consecutively, whereas most tools would apply them concurrently), but it helps to evaluate how the different options independently contribute to achieve efficiency. In its current version, KSP implements local and global reasoning. The implementation of a proof search procedure for local satisfiability under global assumptions is ongoing work.

First we discuss the main processing cycle and in then we give an overview of the available options and their implementations. For a comprehensive list of options, see [36], where the sources and instructions on how to install and use KSP can be found.

5.1 Main Processing Cycle

The proof search procedure for local satisfiability implemented in KSP is shown in Fig. 1. The preprocessing steps (Lines 2–4) are explained in Sects. 5.2, 5.3, and 5.4. The main loop (Lines 6–16) is based on the given-clause algorithm implemented in Otter [31], a variation of the set of support strategy [56], a refinement which restricts the set of choices of clauses participating in a derivation step. For the classical case, a set of clauses Δ is partitioned into two sets Γ and $\Lambda = \Delta \backslash \Gamma$, where Λ must be satisfiable for completeness. The set Γ is the *set of support* (the sos, aka *passive* or *unprocessed* set); and Λ is called the *usable* (aka *active* or *processed* set). The *given clause* is chosen from Γ, resolved with clauses in Λ, and moved from Γ to Λ. Resolvents are added to Γ. For the modal calculus, the set of clauses is further partitioned according to the modal level at which clauses occur. That is, for each modal level ml there are three sets: Γ^{lit}_{ml}, Λ^{lit}_{ml} and Λ^{mod}_{ml}, where the first two sets contain literal clauses while the latter contains modal clauses. As the calculus does not generate new modal clauses and because the set of modal clauses by itself is satisfiable (as they are implications

```
1  Algorithm: KSP-Proof-Search
2  input_processing;
3  snf_transformation;
4  clause_preprocessing;
5  Γ^lit ← ⋃Γ^lit_ml;
6  while (Γ^lit ≠ ∅) do
7      for (all modal levels ml) do
8          clause ← given(ml);
9          if (not redundant(clause)) then
10             GEN1(clause, ml, ml − 1);
11             GEN3(clause, ml, ml − 1);
12             LRES(clause, ml, ml);
13             Λ^lit_ml ← Λ^lit_ml ∪ {clause};
14         Γ^lit_ml ← Γ^lit_ml \ {clause};
15         if (0 : false ∈ Γ^lit_0) then return unsatisfiable;
16     Γ^lit ← ⋃Γ^lit_ml;
17 return satisfiable;
```

Fig. 1 Main loop

and the left-hand sides are a single non-negated proposition), there is no need for a set for unprocessed modal clauses. Attempts to apply an inference rule are guided by the choice, for each modal layer ml, of a literal clause in Γ^{lit}_{ml}, which can be resolved with either a set of modal clauses in Λ^{mod}_{ml-1} or with a literal clause in Λ^{lit}_{ml}.

In more detail, let Γ^{lit} be the union of all sets of literal clauses (Line 5). A *cycle* corresponds to one iteration of the outer loop (Lines 6–16). This loop is executed while the set of unprocessed clauses is not empty. In the inner loop (Lines 7–15), for every modal level ml, a literal clause is returned by the function given. The options for selecting the modal level and the literal clause at this level are described in Sects. 5.5 and 5.6. Once a literal clause is selected, it is tested for redundancy (Line 9), i.e. clauses that can be deleted without affecting the satisfiability of the clause-set. The choices for redundancy elimination are presented in Sect. 5.9. If the given clause is not redundant, then it is processed against all usable modal clauses in the previous modal level (Lines 10 and 11) and against all usable literal clauses at the same modal level (Line 12). Note that as no modal clauses are generated during the proof search, the inference rules MRES and GEN2 are applied before the prover enters the main loop. This is discussed further in Sect. 5.4. The refinements that can be used to apply LRES are given in Sect. 5.7. Once the inference rules are applied the chosen clause is moved to the set of processed clauses (Line 13) and removed from the set of unprocessed clauses (Line 14). If the empty clause (0: **false**) is generated at the modal level 0, then the procedure returns that the set of clauses is unsatisfiable (Line 15). Note that in Fig. 1, only the condition for local reasoning is given. If the prover is set for global satisfiability, then that condition changes to $* : \mathbf{false} \in \Gamma^{lit}$. If the empty clause is not found, the procedure returns that the set of clauses is satisfiable (Line 17).

5.2 Input Processing

The input is read from either a file or from the command line. A configuration file can also be given. In this case, the options given at the command line override those in the configuration

file. A input file is a set of declarations (the options for the prover) followed by sets of modal formulae or clauses. The user can also specify the sets of formulae and clauses as either processed (usable) or unprocessed (sos).

The tokeniser and the parser were built with Flex [12] and Bison [13], which are both free, open source software and easily available. The input language of KSP is LR(2) which can be handled by Bison with options for generating a generic parser. Relying on generators for the lexer and the parser might not lead to the most efficient implementation of the automata which recognise the given language. However, changes in the grammar require very little effort to be implemented, making this part of the code easier to maintain.

The outputs of the parser are a double-linked annotated abstract tree and a symbol table. In the annotated tree, a number is assigned to every node in order to avoid unnecessary transversal of the tree while performing simplification. The assigned number may not be unique[1], but two formulae are checked for repetition, for instance, only if they are assigned the same number. Conjunctions are treated as n-ary operators and nested conjunctions are flattened. The operands of conjunctions are ordered: first come Boolean constants, second propositional symbols, third compound propositional formulae, and fourth modal formulae. The same applies to disjunctions. The symbol table contains information about the propositional symbols, constants, and modal operators occurring in the formula: their type, id, number of occurrences, number of positive and negative occurrences (both globally and by modal level). A double level hash table contains the locations of the positions of propositional symbols and constants in the tree: the first level corresponds to the modal level at which they occur and the second level to the addresses themselves, so that book-keeping the deletions in the tree can be done fast (typically, in constant time).

As the parsing is bottom-up, as usual for LR grammars, linearisation (i.e. the removal of double-implications) and the calculation of polarity requires at least another pass in the tree. This extra transversal of the tree is only done in case there is any double-implication occurring in the input formula or if the option for (modal level) pure literal elimination is set. Modal level pure literal elimination consists of replacing every propositional symbol p which is pure at a modal level ml by a constant. If p occurs only with positive polarity at ml, then p is replaced by **true**; if it occurs only with negative polarity, then p is replaced by **false**.

If the input is a set of formulae, depending on the options given by the user, the formulae are first transformed into their Negation Normal Form (NNF) or into Box Normal Form (BNF) [39]. The translation into BNF also removes the $\Diamond$ operator. Thus, the $\Box$ operator is also allowed in the scope of negations. More precisely, the translation into BNF differs from the NNF just in one case. When transforming a formula as $\neg\Box_a\varphi$ into NNF, the result is $\Diamond\mathsf{NNF}(\neg\varphi)$; the transformation of the same formula into BNF results in $\neg\Box_a\mathsf{BNF}(\neg\varphi)$. For example, the formula $\Box_a(p\wedge q)\wedge\Diamond\neg(p\wedge q)$ is transformed into $\Box_a(p\wedge q)\wedge\neg\Box_a(p\wedge q)$, which is easier to check for simplification than checking the resulting NNF which is $\Box_a(p\wedge q)\wedge\Diamond(\neg p\vee\neg q)$.

Then transformation into prenex (option *prenex*) or antiprenex normal form (option *antiprenex*) or one after the other can be applied. The definitions of those normal forms are given in [32]. Basically, the prenex normal form corresponds to pushing the modal operators occurring in a formula φ as far as possible outwards the formula in order to obtain φ' which is equivalent to φ. For instance, the prenex normal form of $(\Box_a p\wedge\Box_a q)$ is $\Box_a(p\wedge q)$. Similarly, the antiprenex normal form corresponds to pushing the modal operators occurring

[1] We would need arbitrary precision arithmetics for doing so.

Table 4 Simplification rules

$(\psi \vee \psi) \mapsto \psi$	$(\psi \wedge \psi) \mapsto \psi$	$(\psi \vee \neg\psi) \mapsto \mathbf{true}$
$(\psi \wedge \neg\psi) \mapsto \mathbf{false}$	$(\psi \vee \mathbf{true}) \mapsto \mathbf{true}$	$(\psi \wedge \mathbf{false}) \mapsto \mathbf{false}$
$(\psi \vee \mathbf{false}) \mapsto \psi$	$(\psi \wedge \mathbf{true}) \mapsto \psi$	$\neg\mathbf{false} \mapsto \mathbf{true}$
$\neg\mathbf{true} \mapsto \mathbf{false}$		
$[a]\,\mathbf{true} \mapsto \mathbf{true}$	$\langle a \rangle\,\mathbf{false} \mapsto \mathbf{false}$	$([a]\varphi \wedge \langle a \rangle \neg\varphi) \mapsto \mathbf{false}$
$([a]\varphi \wedge [a]\neg\varphi) \mapsto [a]\,\mathbf{false}$	$([a]\,\mathbf{false} \wedge \langle a \rangle \varphi) \mapsto \mathbf{false}$	$([a]\,\mathbf{false} \wedge [a]\varphi) \mapsto [a]\,\mathbf{false}$
$(\langle a \rangle \varphi \vee \langle a \rangle \neg\varphi) \mapsto \langle a \rangle\,\mathbf{true}$	$(\langle a \rangle\,\mathbf{true} \vee [a]\varphi) \mapsto \mathbf{true}$	$(\langle a \rangle\,\mathbf{true} \vee \langle a \rangle \varphi) \mapsto \langle a \rangle\,\mathbf{true}$

in φ as far as possible inwards the formula in order to produce φ' equivalent to φ. For instance, the antiprenex normal form of $\langle a \rangle (p \vee q)$ is $(\langle a \rangle p \vee \langle a \rangle q)$.

With options *nnfsimp* (resp. *bnfsimp*), simplification is applied to formulae in NNF (resp. BNF); with options *early_ple* and *early_mlple*, pure literal elimination is applied globally or at every modal level, respectively. The simplification rules are given in Table 4. Most of those simplification rules can be found in the literature (e.g. [49]). The only rules we are not aware that were reported before are $(\langle a \rangle \mathbf{true} \vee [a]\varphi) \mapsto \mathbf{true}$ and $([a]\mathbf{false} \wedge \langle a \rangle \varphi) \mapsto \mathbf{false}$. We show that the first of those rules is correct. The formula $(\langle a \rangle \mathbf{true} \vee [a]\varphi)$ is semantically equivalent to $(\langle a \rangle \varphi \vee \langle a \rangle \neg\varphi \vee [a]\varphi)$, which is semantically equivalent to $(\langle a \rangle \varphi \vee \neg [a]\varphi \vee [a]\varphi)$. As $\neg [a]\varphi \vee [a]\varphi$ is a tautology, the whole formula simplifies to **true**. The proof that the transformation of $([a]\mathbf{false} \wedge \langle a \rangle \varphi) \mapsto \mathbf{false}$ is correct is similar.

5.3 Transformation to Normal Form

By default, formulae are transformed into SNF_{ml}. There are four different options that determine the normal form. Two of those options are used for transforming a set of clauses into SNF^{+}_{ml} and into SNF^{++}_{ml}, as described in Sect. 3. The two other options are used for transforming a set of clauses into SNF^{-}_{ml} and SNF^{--}_{ml}, which are defined analogously to SNF^{+}_{ml} and SNF^{++}_{ml}, but where literals in the scope of modal operators are renamed by new negative literals.

The transformation into any of those normal forms requires renaming [41]. All renaming is performed bottom-up and it uses an auxiliary hash table based on the number assigned to the formulae in the tree structure and the modal level at which the formula occurs. A bottom-up renaming might not be optimal with regard to the number of generated clauses for linear formulae (those where bi-implications do not occur and subformulae are not repeated) [7]. An alternative would be top-down renaming. However, for formulae which are not linear, top-down renaming cannot ensure that the number of generated clauses is smaller than that obtained with bottom-up renaming. A future version of KSP will implement small normal forms [37], but at the moment only four different forms of renaming are available. With the option *normal_renaming*, every subformula is renamed by a new propositional symbol. With option *limited_reuse_renaming*, the same new propositional symbol is used for all occurrences of the same subformula being renamed at a particular modal level. Yet another option, *extensive_reuse_renaming*, also uses the same propositional symbol for all occurrences of the same subformula being renamed; in addition, if a formula φ was renamed by a new propositional symbol t, then the NNF of $\neg\varphi$ is renamed by $\neg t$. With the option

conjunct_renaming, modal subformulae that occur in conjunctions are renamed, instead of applying the usual rewriting rule. When applied together, *conjunct_renaming* and either *limited_reuse_renaming* or *extensive_reuse_renaming* might lead to a smaller set of clauses. For instance, $ml : t \Rightarrow \boxed{a}\, p \wedge (p \vee \boxed{a}\, p)$ is transformed into $\{ml : t \Rightarrow t_1, ml : t \Rightarrow t_2, ml : t_1 \Rightarrow \boxed{a}\, p, ml : t_2 \Rightarrow p \vee t_1\}$.

The set of clauses resulting from the transformation into the normal form is stored in a trie-like structure (implemented as multi-level hash tables), according to the set they belong to (usable, sos), their type (initial, literal, modal positive, modal negative), their modal level, the index of the modal operator (in the case of modal clauses), their maximal literal, and their size (in the case of initial and literal clauses). Clauses are implemented as simplified, ordered lists of literals. Also, for every propositional symbol, a list of clauses by modal level is kept in the symbol table. The trie structure helps to make the implementation of clause and literal selection efficient, to reduce the number of clauses being checked during redundancy tests (repetition and subsumption), and also to reduce the number of misses during the construction of the set of candidate clauses to which a particular resolution rule is applied. The list of clauses kept in the symbol table allows for efficiently finding the clauses to be processed during the application of unit resolution (options *unit* and *lhs_unit*) and pure literal elimination (option *ple*, which is applied if the literal is pure in the whole set of clauses; or option *mlple*, which is applied if the literal is pure at a modal level).

5.4 Preprocessing of Clauses

The preprocessing of clauses comprises several tasks which may be set by the user. By default, we prevent duplicate clauses to be stored in the trie-like structure that we use to maintain the set of clauses, but only duplicates in the set they are stored are checked. For instance, if a clause is to be stored in Γ^{lit}_{ml}, then we only check this particular set for a duplicate. With option *check_full_repeated*, repetition of clauses is checked against all sets of clauses at the same modal level.

Propagation of a literal in the scope of the operator $\diamondsuit$ is applied at this stage, with option *propdia*. The propagation rule is given by

$$\frac{ml : l' \Rightarrow \diamondsuit l}{ml + 1 : l}$$

for literals l and l', modal level ml, and index a, with the side condition that there is only one negative modal clause in Λ^{mod}_{ml} (otherwise, the rule is not sound). Propagation of a literal is not needed for completeness, but as the inference rule generates a unit clause at the propositional set of formulae, unit propagation and subsumption can be applied, reducing the number of literals at the modal level $ml + 1$.

If the set of modal clauses at the modal level ml does not contain a negative modal clause, than all positive modal clauses at that modal level and all clauses (modal and literal) with modal level greater than ml can be deleted. This is justified by the fact that a world w satisfies $\boxed{a}l$, for a literal l and index a, if there is no world w' such that $(w, w') \in R_a$. Thus, we can take the empty relation for all worlds at the level ml, which satisfy all positive modal clauses. As the worlds at greater modal levels are no longer accessible, the sets of clauses corresponding to those levels can also be deleted. The option for this simplification is *mle*.

As no modal clauses are generated during the proof search, the inference rules MRES and GEN2 are also exhaustively applied at this step, that is, before the prover enters the main loop. We note that the transformation into SNF^{+}_{ml}, SNF^{++}_{ml}, SNF^{-}_{ml}, or SNF^{--}_{ml} is performed after the

preprocessing of clauses. If any of those four options related to the transformation into the normal form are set by the user, then all literals in the scope of modal operators will have the same polarity. Therefore, the inference rules MRES and GEN2 are not applicable and will be blocked. However, as those inference rules produce very short resolvents, which can be particularly useful to reduce the number of clauses if subsumption is also set, the user can force those inference rules to be applied even when they are not needed for completeness, by setting the options *mres* and *gen2*.

If forward and/or backward subsumption [29] are set, then self-subsumption is also applied at this point. Forward and backward subsumption are performed in lazy mode and only against the usable (see Sect. 5.9). However, for self-subsumption, clauses are tested against all sets, irrespective of where they are stored.

By default, the usable sets Λ_{ml}^{lit} of literal clauses are empty (unless those sets are given as input). However, the user has the choice of automatically populating those usable sets with literal clauses. There are six options: *populate_non_negative* moves literal clauses which are not negative from Γ_{ml}^{lit} to Λ_{ml}^{lit}; *populate_non_positive* moves clauses which are not positive from Γ_{ml}^{lit} to Λ_{ml}^{lit}; *populate_negative* moves negative literal clauses from Γ_{ml}^{lit} to Λ_{ml}^{lit}; *populate_positive* moves positive literal clauses from Γ_{ml}^{lit} to Λ_{ml}^{lit}; *populate_max_lit_negative* moves literal clauses whose maximal literal is negative from Γ_{ml}^{lit} to Λ_{ml}^{lit}; and the option *populate_max_lit_positive* moves literal clauses whose maximal literal is positive from Γ_{ml}^{lit} to Λ_{ml}^{lit}.

We note that populating the usable sets must be done with some care, as some combinations of the set of support and other refinements are not complete. For instance, using ordered resolution as a refinement, consider the set $\Delta = \{0 : p \vee q, 0 : p \vee \neg q, 0 : \neg p \vee q, 0 : \neg p \vee \neg q\}$, which is unsatisfiable, and let $p \succ q$ be the ordering over the propositional symbols. Using the option *populate_non_negative*, we obtain $\Lambda_0^{lit} = \{0 : p \vee q, 0 : p \vee \neg q, 0 : \neg p \vee q\}$, which is satisfiable, and $\Gamma_0^{lit} = \{0 : \neg p \vee \neg q\}$. In the first cycle, there is only one clause to choose, $\neg p$ is the maximal literal, and the only non-redundant generated resolvent is $0 : \neg q$, which is added to Γ_0^{lit}. Now, as $0 : \neg p \vee \neg q$ is moved to the set of usable clauses, $0 : \neg q$ is the only clause that can be chosen in the sos. However, there is no clause in Λ_0^{lit} where q is the maximal literal. Thus, no inference steps can be applied, and the procedure would output the set as satisfiable. In contrast, using the options *populate_non_negative* with negative resolution or *populate_max_lit_positive* with either negative or ordered resolution are safe choices.

5.5 Controlling the Inner Loop

The inner loop executed during the proof search (Lines 7–15) iterates over the modal levels in the set of literal clauses. As mentioned before, each set Γ_{ml}^{lit} is implemented as multi-level hash tables, where the modal level is one of the keys. By default, these sets are scanned by following the order of the entries in the hash table. However, the user can set different orderings. With option *ordlevel_ascend*, the **for** loop iterates in ascending order of modal levels, that is, from $ml = 0$ to the maximal modal level; and with option *ordlevel_descend*, the modal levels are scanned from the maximal modal level down to $ml = 0$. With option *ordlevel_shuffle*, the list of modal levels is partitioned in half and the two lists are merged, just before entering the inner loop; the modal levels are then scanned in the resulting order. Preliminary evaluation of these features, checked over the LWB benchmarks [25], shows that the default performs better. However, we have not performed an extensive evaluation yet.

5.6 Clause Selection

Besides the set of support strategy, which restricts clause selection to those in the sos, there are five different heuristics for choosing a literal clause as the given clause at a modal level within each cycle. With option *shortest*, the clause with the smallest size at that particular modal level is chosen. With option *newest*, clause selection simulates a stack (last in, first out). With option *oldest*, clause selection simulates a queue (first in, first out). With option *smallest* (resp. *greatest*), the given clause is the shortest clause with the smallest (resp. greatest) maximal literal in the set.

5.7 Refinements

Besides the implemented restrictions on clause selection, the user can further restrict LRES by choosing options *ordered* (clauses can only be resolved on their maximal literals with respect to an ordering chosen by the prover in such a way to preserve completeness), *negative* (one of the premises is a negative clause, i.e. a clause where all literals are of the form $\neg p$ for some $p \in P$), *positive* (one of the premises is a positive clause), or *negord*, where both negative and ordered resolution inferences are performed.

The completeness of some of these refinements depends on the particular normal form chosen. For instance, negative resolution is incomplete without SNF^{+}_{ml} or SNF^{++}_{ml}. For example, the set $\{0 : p, 0 : p \Rightarrow \boxed{a}\neg q, 0 : p \Rightarrow \diamondsuit s, 1 : \neg s \vee q\}$ is locally unsatisfiable, but as there is no negative literal clause in the set, no refutation can be found with negative resolution. By renaming $\neg q$ with t in the scope of $\boxed{a}$, we obtain the set $\{0 : p, 0 : p \Rightarrow \boxed{a}t, 0 : p \Rightarrow \diamondsuit s, 1 : \neg s \vee q, 1 : \neg t \vee \neg q\}$ in SNF^{+}_{ml}, from which a refutation using negative resolution can be found. Similarly, ordered resolution requires SNF^{++}_{ml} for completeness, while positive resolution requires SNF^{-}_{ml} or SNF^{--}_{ml}.

5.8 Inference Rules

Besides the inference rules given in Table 2, three more inference rules are also implemented. With option *unit*, unit clauses are propagated through all literal clauses and the right-hand side of modal clauses, that is, the following inference rules are applied:

$$\text{[UNIT]}\ \frac{ml_1 : l_1 \vee \ldots \vee l_m \vee l \qquad ml_2 : \neg l}{\sigma(\{ml_1, ml_2\}) : l_1 \vee \ldots \vee l_m} \qquad \text{[UNIT-GEN1]}\ \frac{ml_1 : l_1 \Rightarrow \diamondsuit l \qquad ml_2 : \neg l}{\sigma(\{ml_1, ml_2 - 1\}) : \neg l_1} \qquad \text{[UNIT-GEN3]}\ \frac{ml_1 : l_1 \Rightarrow \boxed{a} l \qquad ml_2 : l_2 \Rightarrow \diamondsuit l_3 \qquad ml_3 : \neg l}{\sigma(\{ml_1, ml_2, ml_3 - 1\}) : \neg l_1 \vee \neg l_2}$$

Clearly, UNIT, UNIT-GEN1, UNIT-GEN3 are special cases of the inference rules LRES, GEN1, and GEN3, respectively. Their implementation, however, is different, as the set of candidates to resolve with the unit clause $ml : \neg l$ is built from the list of clauses stored in the symbol table instead of using the trie-like structure for clauses. Subsumption, if set, is also immediately applied. By default, redundancy is only checked when a clause is chosen, but for those unit resolution rules the premises $ml_1 : l_1 \vee \ldots \vee l_m \vee l$ in UNIT and $ml_1 : l_1 \Rightarrow \diamondsuit l$ in UNIT-GEN1 are deleted from the set of clauses as soon as subsuming resolvents are generated.

The option *lhs_unit* propagates unit clauses through the left-hand side of modal clauses, that is, the following inference rules are applied:

$$\text{[LHS-UNIT-1]}\quad \frac{ml_1 : \ l \qquad ml_2 : \ l \ \Rightarrow \ \Diamond l'}{\sigma(\{ml_1, ml_2\}) : \mathbf{true} \Rightarrow \Diamond l'} \qquad \text{[LHS-UNIT-2]}\quad \frac{ml_1 : \ l \qquad ml_2 : \ l \ \Rightarrow \ \Box l'}{\sigma(\{ml_1, ml_2\}) : \mathbf{true} \Rightarrow \Box l'}$$

Again, if the modal clauses in the premises are subsumed by the resolvents in those inference rules, they are immediately deleted from the clause set. The rules LHS-UNIT-1 and LHS-UNIT-2 are not needed for completeness. However, exhaustive application of the two rules together with subsumption and the usual unit resolution removes all occurrences of the literal $\neg l$ at the modal level ml; thus, if the options for pure literal elimination are set, more clauses can be removed from the clause set.

The inference rules shown in Table 3 are set with the option *ires*, which together with the *global* option, implements initial resolution and, therefore, the calculus given in [33]. The inference rules IRES1 and IRES2 are, by default, applied after the main loop described in Fig. 1. For an unsatisfiable set of clauses, if the literal clauses are not by themselves unsatisfiable, this means that a proof can only be found after the set of literal clauses is saturated, which might be very time consuming. With option *interires*, initial and literal resolution are interleaved, that is, IRES1 and IRES2 are applied within the main loop given in Fig. 1, which may shorten the time to finding a proof.

5.9 Redundancy Elimination

Pure literal elimination can be applied globally (option *ple*) or by modal level (option *mlple*). For modal level pure literal elimination, if a literal l is pure at a modal level ml, then the literal can be set to **true** at that level. This means that any literal clause at the modal level ml in which l occurs can be deleted. If l occurs in the scope of $\Box$ on the right-hand side of a positive modal clause, then the positive modal clause can also be deleted (because $\Box\mathbf{true}$ is a tautology). If l occurs in the scope of $\Diamond$ on the right-hand side of a positive modal clause $ml - 1 : l' \Rightarrow \Diamond l$, then the clause $ml - 1 : l' \Rightarrow \Diamond l$ is deleted and the clause $ml - 1 : l' \Rightarrow \Diamond\mathbf{true}$ is generated. Because $\Diamond\mathbf{true}$ is not a tautology, the newly generated clause is kept in the set of clauses. As the number of literal occurrences is stored in the symbol table, at the implementation level, the procedure for modal level pure literal elimination consists of scanning the information related to all propositional symbols p and deleting the list of clauses at a particular modal level ml if the number of either positive or negative occurrences of p at the modal level ml is zero. For pure literal elimination, the procedure is similar.

Both forward (option *fsub*) and backward subsumption (option *bsub*) are implemented. A literal clause $ml : C$ is subsumed by a literal clause $ml' : D$ if, and only if, $ml' : D$ implies $ml : C$. For forward subsumption, a literal clause $ml : C$ is deleted if it is subsumed by any older literal clause. For backward subsumption, a literal clause $ml : C$ is deleted if it is subsumed by any newer literal clause. In both cases, subsumption is applied in lazy mode: a clause is tested for subsumption only when it is selected from Γ^{lit}_{ml} and only against clauses in Λ^{lit}_{ml}. As pointed out in [47], lazy subsumption avoids expensive checks for clauses that might never be selected during the search of a proof. Also, the trie-like structure for clauses is used to improve the selection of candidates for subsumption. A clause $ml : C$ is subsumed by a clause $ml' : D$ if, and only if, the following holds:

1. $\sigma(\{ml, ml'\})$ is defined;
2. the size of $ml : C$ is greater or equal the size of $ml' : D$;
3. the maximal literal in $ml : C$ is less or equal the maximal literal in $ml' : D$;
4. the minimal literal in $ml : C$ is greater or equal the minimal literal in $ml' : D$.

The first three conditions are keys in the trie-like structure, thus the first approximation for a set of candidates can be obtained in linear time on the size of the clause set for a particular modal level. The last condition requires testing all clauses which satisfy the first three conditions, but it can also be easily checked: as clauses are implemented as ordered lists, it only requires to test the head of those lists.

The user can force subsumption checking in the whole set of clauses by setting the option *sos_sub*.

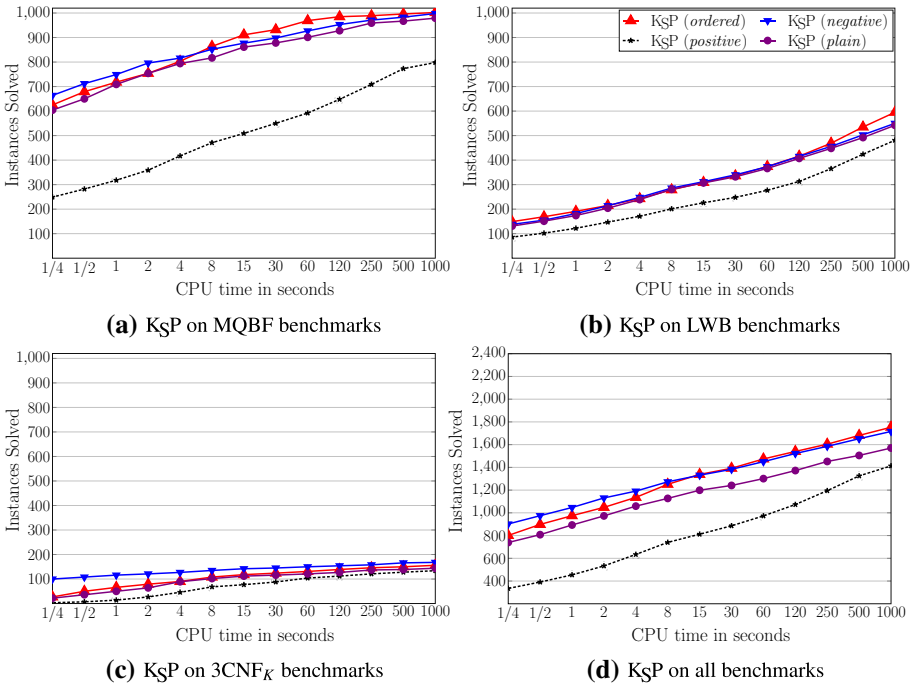
6 Evaluation

We have compared KSP 0.1.2 with the provers BDDTab [15,38], FaCT++ 1.6.3 [51,52], InKreSAT 1.0 [26,27], Spartacus 1.1.3 [16,17], and a combination of the optimised functional translation [23] with Vampire 4.2.2 [28,53][2]. In this context, FaCT++ represents the previous generation of reasoners while the remaining systems have all been developed in recent years. Unless stated otherwise, the reasoners were used with their default options.

Our benchmarks [36] consist of three collections of modal formulae:

1. The complete set of TANCS-2000 modalised random QBF (MQBF) formulae [30] complemented by the additional MQBF formulae provided by Kaminski and Tebbi [27]. This collection consists of five classes, called qbf, qbfL, qbfS, qbfML, and qbfMS in the following, with a total of 1016 formulae, of which 617 are known to be satisfiable and 399 are known to be unsatisfiable (due to at least one of the provers being able to solve the formula). The minimum modal depth of formulae in this collection is 19, the maximum 225, average 69.2 with a standard deviation of 47.5.
2. LWB basic modal logic benchmark formulae [4], with 56 formulae chosen from each of the 18 parameterised classes. In most previous uses of these benchmark classes, only parameter values 1 to 21 were used for each class, resulting in 378 benchmark formulae with a median size of a benchmark formula of 1072.5 and a maximum size of 24,972. For such low parameter values, most benchmark formulae were easily solvable by state-of-the art provers. To overcome this problem we have instead chosen the 56 parameter values so that only the best current provers, if any at all, will be able to solve all the formulae within a time limit of 1000 CPU second. The median value of the maximal parameter value used for the 18 classes is 1880, far beyond what has ever been tested before. The median size of benchmark formulae is 342,077.5 and the maximum size is 288,072,146. Of the 1008 formulae, half are satisfiable and half are unsatisfiable by construction of the benchmark classes. The minimum modal depth of formulae in this collection is 1, the maximum 30,004, average 1065.7 with a standard deviation of 2670.1.
3. Randomly generated 3CNF_K formulae [40] over 3 to 10 propositional symbols with modal depth 1 or 2. We have chosen formulae from each of the 11 parameter settings given in the table on page 372 of [40]. For the number of conjuncts we have focused on a range around the critical region where about half of the generated formulae are satisfiable and half are unsatisfiable. The resulting collection contains 1000 formulae, of which 457 are known to be satisfiable and 464 are known to be unsatisfiable. Note that this collection is quite distinct to the one used in [27] which consisted of 135 3CNF_K formulae over 3 propositional symbols with modal depth 2, 4 or 6, all of which were

[2] We have excluded *SAT from the comparison as it produced incorrect results on a number of benchmark formulae.



(a) KSP on MQBF benchmarks

(b) KSP on LWB benchmarks

(c) KSP on 3CNF$_K$ benchmarks

(d) KSP on all benchmarks

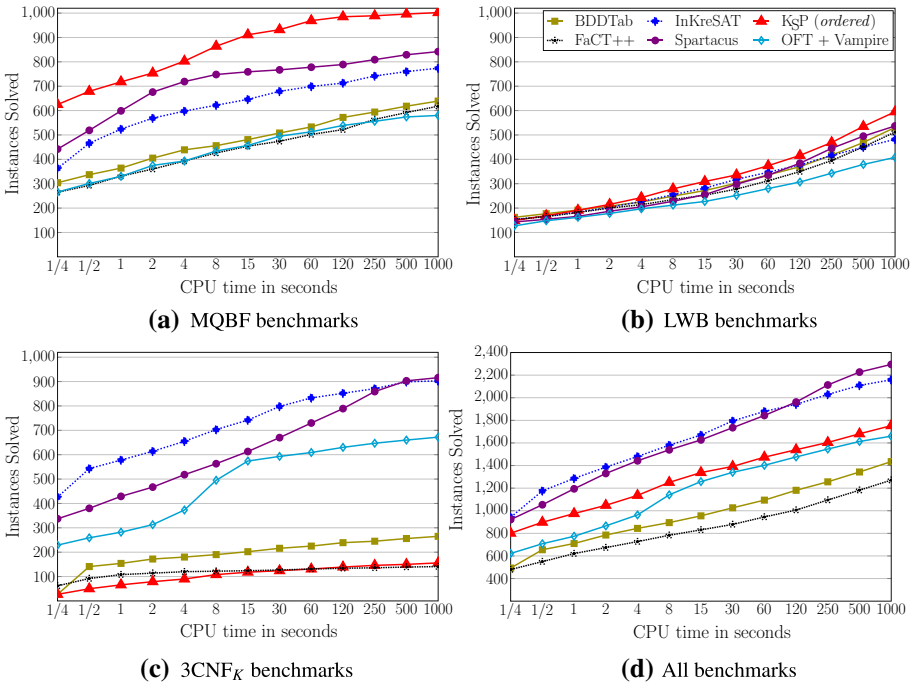
Fig. 2 Benchmarking results for KSP

satisfiable. The minimum modal depth of formulae in this collection is 1, the maximum 2, average 1.8 with a standard deviation of 0.4.

In [35] we have used the same benchmark formulae and the same versions of BDDTab, FaCT++ and InKreSAT, but used KSP 0.1.1, Spartacus 1.0, and Vampire 3.0 instead of the more recent versions employed here. We also applied a different method of computing the optimised functional translation; the method used now is faster, but typically results in a larger formula as fewer simplifications are performed during the computation. As a consequence the results reported for these three provers are different to those in [35].

Benchmarking was performed on PCs with an Intel i7-2600 CPU @3.40 GHz and 16 GB main memory. For each formula and each prover we have determined the median run time over five runs with a time limit of 1000 CPU seconds for each run.

Figure 2 shows the impact of different refinements on the performance of KSP on the MQBF, LWB, and 3CNF$_K$ collections and all benchmark formulae together. KSP (*plain*) uses the rules shown in Table 2, without additional refinement, on a set of SNF_{ml} clauses. KSP (*ordered*) applies ordered resolution on a set of SNF^{++}_{ml} clauses. KSP (*negative*) uses negative resolution on a set of SNF^{+}_{ml} clauses, while KSP (*positive*) applies positive resolution on a set of SNF^{-}_{ml} clauses. Irrespective of the refinement, the shortest clause is selected to perform inferences; both forward and backward subsumption are used; the unit and lhs-unit resolution rules are applied; prenex and *early_mlple* are set; and no simplification steps are applied. KSP (*ordered*) offers the best performance on the MQBF and LWB collections, while on the 3CNF$_K$ collection KSP (*negative*) performs best. KSP (*plain*) performs slightly worse than these two refinements on all three collections and KSP (*positive*) is significantly worse than the other three refinements. Overall, as shown in Fig. 2d, KSP (*ordered*) performs best. Ordered resolution restricts the applicability of the rules further than the other refinements. Not only



(a) MQBF benchmarks **(b)** LWB benchmarks

(c) 3CNF$_K$ benchmarks **(d)** All benchmarks

Fig. 3 Benchmarking results for all provers

is this an advantage on satisfiable formulae in that a saturation can be found more quickly, but also unsatisfiable formulae where with this refinement KSP typically finds refutations more quickly than with any of the other refinements. That being said, the difference between KSP (*plain*), KSP (*negative*) and KSP (*ordered*) is smaller than one might expect. This is due to the fact that for a modal formula, the corresponding set of SNF^{++}_{ml} clauses is larger than the set of SNF^{+}_{ml} clauses which in turn is typically larger than the set of SNF_{ml} clauses. This counterbalances the advantages gained from the more constrained proof search of the refinements.

Figure 3 compares the performance of all the provers on the MQBF, LWB, and 3CNF$_K$ collections and all benchmark formulae together. For the MQBF collection we see that KSP (*ordered*) performs better than any of the other provers. The graphs in Fig. 4 offer some insight into why KSP performs well on these formulae. Each of the four graphs shows for one formula from each class how many atomic subformulae occur at each modal level, the formulae originate from MQBF formulae with the same number of propositional symbols, conjuncts and QBF quantifier depth. Formulae in the class qbfS are the easiest, the total number of atomic subformulae is low and spread over a wide range of modal levels, thereby reducing the possibility of inference steps between the clauses in the layered normal form of these formulae. In contrast, in qbfMS formulae almost all atomic subformulae occur at just one modal level. Here the layered normal form can offer little advantage over a simpler normal form. But the number of atomic subformulae is still low and KSP seems to derive an advantage from the fact that the normal form 'flattens' the formula: KSP is at least two orders of magnitude faster than any other prover on this class. The classes qbf and qbfL are more challenging. While the atomic subformulae are more spread out over the modal levels than for qbfMS, at a lot of these modal levels there are more atomic subformulae than in a

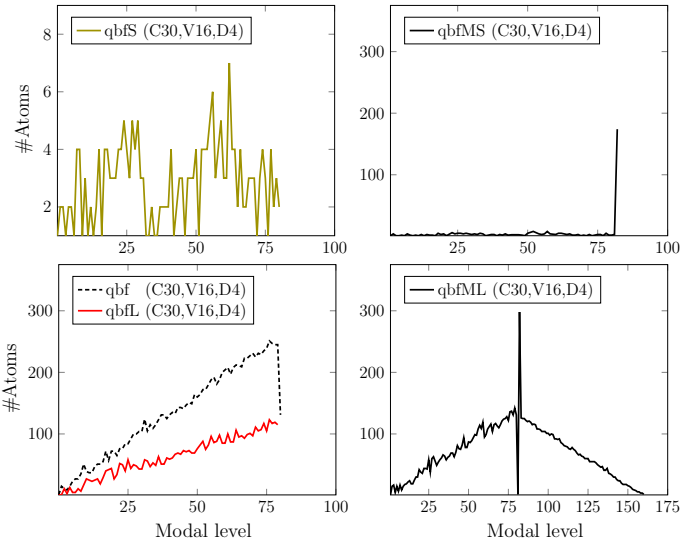


Fig. 4 Modal structure of MQBF formulae

qbfMS formula in total. The layered modal translation is effective at reducing the number of inferences for these classes, but more inference possibilities remain than for qbfMS. Finally, qbfML combines the worst aspects of qbfL and qbfMS, the number of atomic subformulae is higher than for any other class and there is a 'peak' at one particular modal level. This is the only MQBF class containing formulae that KSP cannot solve.

On the LWB collection KSP performs slightly better than the other provers with Spartacus being the second best system and the combination of the optimised functional translation with Vampire (OFT + Vampire) performing worst. Table 5 provides more detailed results. For each prover it shows in the left column how many of the 56 formulae in a class have been solved and in the right column the parameter value of the most difficult formula solved. For InKreSAT we are not reporting this parameter value for three classes on which the prover's runtime does not increase monotonically with the parameter value but fluctuates instead. As indicated in bold in the table, BDDTab is the best performing prover on one class, InKreSAT on three, OFT + Vampire on three, Spartacus on four, and KSP on five classes; KSP and Spartacus are joint best on a further two classes. A characteristic of the classes on which KSP performs best is again that atomic subformulae are evenly spread over a wide range of modal levels.

It is worth pointing out that simplification alone is sufficient to detect that formulae in lin_p are unsatisfiable. For grz_p, pure literal elimination can be used to reduce all formulae in this class to the same simple formula; the same is true for grz_n and lin_n. Thus, these classes are tests of how effectively and efficiently, if at all, a prover uses these techniques.

On the 3CNF_K collection, InKreSAT is the best performing prover and KSP the worst performing one. This should now not come as a surprise. For 3CNF_K we specifically restricted ourselves to formulae with low modal depth which in turn means that the layered normal form has little positive effect.

It is evident from these results that no prover is best on all the collections and on every benchmark formula. It therefore makes sense to employ a portfolio of provers when trying to determine the satisfiability of a modal formula. Following Schuppan and Darmawan [48], we can consider an 'Oracle Procedure' based on an oracle that for each benchmark formula picks

Table 5 Detailed evaluation results on the LWB benchmark

	BDDTab		FaCT++		InKreSAT		KSP (*ordered*)		Spartacus		OFT +Vampire	
branch_n	22	*22*	12	*12*	15	*15*	18	*18*	12	*12*	**36**	*42*
branch_p	22	*22*	12	*12*	22	*22*	24	*24*	14	*14*	**35**	*40*
d4_n	20	*440*	6	*40*	34		**53**	*1760*	31	*880*	14	*200*
d4_p	26	*640*	24	*600*	18	*360*	**56**	*1880*	35	*1040*	21	*960*
dum_n	39	*2400*	42	*2640*	23	*1120*	**52**	*3440*	49	*3200*	16	*560*
dum_p	42	*2640*	38	*2320*	28	*1520*	52	*3440*	52	*3440*	19	*800*
grz_n	35	*2600*	27	*1800*	50	*4500*	5	*50*	**51**	*5000*	20	*1100*
grz_p	35	*2600*	27	*1800*	**51**	*5000*	29	*2000*	49	*4000*	26	*1700*
lin_n	46	*4000*	43	*3400*	33	*2500*	1	*10*	34	*2500*	**40**	*3100*
lin_p	14	*500*	28	1×10^5	**56**	5×10^5	24	*6000*	55	4×10^5	28	1×10^5
path_n	37	*290*	48	*400*	7	*14*	**56**	*1200*	48	*400*	42	*340*
path_p	35	*270*	48	*400*	5	*12*	**56**	*1200*	48	*400*	42	*340*
ph_n	10	*10*	8	*16*	**24**	*90*	3	*6*	22	*80*	13	*35*
ph_p	**11**	*11*	9	*8*	10	*10*	5	*5*	9	*9*	10	*10*
poly_n	39	*600*	34	*500*	30		33	*480*	**47**	*760*	18	*180*
poly_p	38	*580*	34	*500*	28	*400*	33	*480*	**47**	*760*	17	*160*
t4p_n	40	*3500*	24	*1500*	17	*800*	41	*4000*	**46**	*6500*	10	*100*
t4p_p	48	*7500*	49	*8000*	28		54	*13000*	54	*13000*	12	*300*

For each prover, the left column shows how many of the 56 formulae have been solved and the right column shows the parameter of the most difficult formula solved. The best performance is shown in bold. Parameters are typeset in italics while indices are typeset in roman

Table 6 Use of each prover by an Oracle Solver

BDDTab	FaCT++	InKreSAT	KSP	Spartacus	OFT + Vampire	Unsolved
674	111	912	849	748	57	227

the best performing prover among the six that we have evaluated, or, alternatively, executes all six provers in parallel and then only accounts for the one with the shortest runtime. Table 6 shows for how many of benchmark formulae the Oracle Procedure would pick a particular prover to get the shortest runtime. As sometimes two or more provers can solve a benchmark formula in the same amount of time, some double counting is involved. Also, there are 227 benchmark formulae that none of the provers can solve within the time limit of 1000 CPU seconds.

A more realistic approach takes advantage of the observation that KSP performs best for formulae of high modal depth while other provers perform well on modal formulae of low modal depth. We consider a procedure that uses KSP (*ordered*) to solve all formulae with a modal depth greater than 3 while for all other formulae one of the other provers is used. The threshold of 3 gave us the best result in terms of the total number of problems solved by this procedure. Figure 5 shows how well this approach performs. It shows benchmarking results for combinations of KSP (*ordered*) with BDDTab, InKreSAT, and Spartacus. It also shows the performance of the Oracle Procedure as well as the performance of a restricted version of the Oracle Procedure that is only allowed to choose between InKreSAT and Spartacus. As we can see, the combination of KSP (*ordered*) with InKreSAT performs best, slightly better than the combination with Spartacus, and significantly better than the combination with BDDTab.

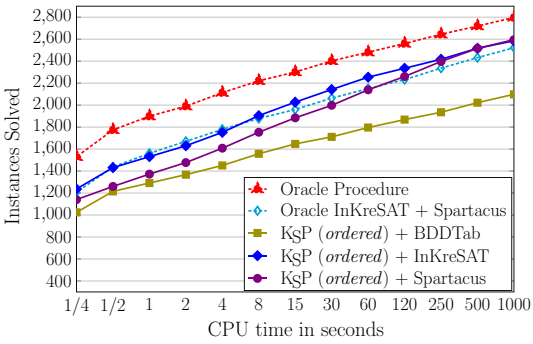


Fig. 5 Benchmarking results for portfolios of provers

The combination of KSP (*ordered*) with InKreSAT also performs better than the restricted Oracle Procedure. This indicates that such a combination not only offers a practical approach to obtaining a procedure that performs better than any single prover, but also that the use of KSP by such a procedure offers performance advantages over other combinations of provers.

7 Conclusions

We have presented the clausal resolution prover KSP for both local and global reasoning for the multi-modal propositional modal logic, K_n. This is based on a complete calculus where resolution inferences are restricted to clauses at the same modal level. This paper focuses on the implemented prover providing full details of the input processing, normal forms, clause preprocessing, the main control loop including proof search strategies and clause selection, refinements via variants of ordered resolution, inference rules and dealing with redundant clauses. We carried out an experimental evaluation with the prover comparing KSP with other provers and also analysing some of these combinations. The evaluation shows that KSP works well on problems with high modal depth where the separation of modal layers can be exploited to improve the efficiency of reasoning.

As with all provers that provide a variety of strategies and optimisations, to get the best performance for a particular formula or class of formulae it is important to choose the right strategy and optimisations. KSP currently leaves that choice to the user. The development of an *auto mode* in which the prover makes a choice of its own, based on an analysis of the given formula, is future work.

The same applies to the transformation to the layered normal form. Again, KSP offers a number of ways in which this can be done as well as a number of simplifications that can be applied during the process. It is clear that this affects the performance of the prover, but we have yet to investigate the effects on the benchmark collections introduced in this paper.

References

1. Areces, C., Gennari, R., Heguiabehere, J., de Rijke, M.: Tree-based heuristics in modal theorem proving. In: W. Horn (ed.) ECAI 2000, pp. 199–203. IOS Press, Amsterdam (2000)
2. Areces, C., Heguiabehere, J.: Hylores 1.0: Direct resolution for hybrid logics. In: A. Voronkov (ed.) CADE 2002, LNCS, vol. 2392, pp. 156–160. Springer, Berlin (2002)
3. Balbiani, P., Demri, S.: Prefixed tableaux systems for modal logics with enriched languages. In: M.E. Pollack (ed.) IJCAI 1997, pp. 190–195. Morgan Kaufmann, Los Altos (1997)
4. Balsiger, P., Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4. J. Automat. Reason. **24**(3), 297–317 (2000)
5. Basin, D., Matthews, S., Vigano, L.: Labelled propositional modal logics: theory and practice. J. Logic Comput. **7**(6), 685–717 (1997)
6. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
7. de la Tour, T.B.: An optimality result for clause form translation. J. Symb. Comput. **14**(4), 283–301 (1992)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995)
9. Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Trans. Comput. Logic **2**(1), 12–56 (2001)
10. Fitting, M.: Prefixed tableaus and nested sequents. Ann. Pure Appl. Logic **163**(3), 291–313 (2012)
11. Fitting, M., Mendelsohn, R.L.: First-Order Modal Logic. Kluwer, Dordrecht (1998)
12. Flex: The fast lexical analyzer generator. http://github.com/westes/flex (2017). Accessed 6 Dec 2018
13. GNU Bison: The yacc-compatible parser generator. http://www.gnu.org/software/bison/ (2017). Accessed 6 Dec 2018
14. Goranko, V., Passy, S.: Using the universal modality: gains and questions. J. Logic Comput. **2**(1), 5–30 (1992)
15. Goré, R., Olesen, K., Thomson, J.: Implementing tableau calculi using BDDs: BDDTab system description. In: S. Demri, D. Kapur, C. Weidenbach (eds.) IJCAR 2014, LNCS, vol. 8562, pp. 337–343. Springer, Berlin(2014)
16. Götzmann, D., Kaminski, M.: Spartacus: sources and benchmarks. Saarland University, Saarbrücken, Germany. http://www.ps.uni-saarland.de/spartacus/. Accessed 6 Dec 2018
17. Götzmann, D., Kaminski, M., Smolka, G.: Spartacus: a tableau prover for hybrid logic. Electron. Notes Theor. Comput. Sci. **262**, 127–139 (2010)
18. Hailpern, B.T.: Verifying Concurrent Processes Using Temporal Logic. LNCS, vol. 129. Springer, Berlin (1982)
19. Halpern, J.Y.: Using reasoning about knowledge to analyze distributed systems. Ann. Rev. Comput. Sci. **2**, 37–68 (1987)
20. Halpern, J.Y., Manna, Z., Moszkowski, B.: A hardware semantics based on temporal intervals. In: J. Díaz (ed.) ICALP 1983, LNCS, vol. 154, pp. 278–291. Springer, Berlin (1983)
21. Halpern, J.Y., Moses, Y.: A guide to completeness and complexity for modal logics of knowledge and belief. Artif. Intell. **54**(3), 319–379 (1992)
22. Hayes, P.J., Kowalski, R.A.: Semantic trees in automatic theorem proving. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 87–101. Elsevier, Amsterdam (1969)
23. Horrocks, I.R., Hustadt, U., Sattler, U., Schmidt, R.: Computational modal logic. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, pp. 181–245. Elsevier, Amsterdam (2006)
24. Hustadt, U., Schmidt, R.A.: MSPASS: modal reasoning by translation and first-order resolution. In: R. Dyckhoff (ed.) TABLEAUX 2000, LNCS, vol. 1847, pp. 67–71. Springer, Berlin (2000)
25. Jaeger, G., Balsiger, P., Heuerding, A., Schwendimann, S., Bianchi, M., Guggisberg, K., Janssen, G., Heinle, W., Achermann, F., Boroumand, A.D., Brambilla, P., Bucher, I., Zimmermann, H.: LWB: the logics workbench 1.1. University of Berne, Switzerland. http://home.inf.unibe.ch/~lwb/benchmarks/benchmarks.html. Accessed 6 Dec 2018
26. Kaminski, M., Tebbi, T.: InKreSAT: sources and benchmarks. Saarland University, Germany. http://www.ps.uni-saarland.de/~kaminski/inkresat/. Accessed 6 Dec 2018
27. Kaminski, M., Tebbi, T.: InKreSAT: modal reasoning via incremental reduction to SAT. In: CADE 2013, LNCS, vol. 7898, pp. 436–442. Springer, Berlin (2013)
28. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: N. Sharygina, H. Veith (eds.) CAV 2013, LNCS, vol. 8044, pp. 1–35. Springer, Berlin (2013)
29. Lee, R.C.T.: A completeness theorem and computer program for finding theorems derivable from given axioms. Ph.D. thesis, University of California, Berkeley, USA (1967)
30. Massacci, F., Donini, F.M.: Design and results of TANCS-2000 non-classical (modal) systems comparison. In: R. Dyckhoff (ed.) TABLEAUX 2000, LNCS, vol. 1847, pp. 52–56. Springer, Berlin (2000)

31. McCune, W.W.: OTTER 3.0 reference manual and guide. Technical report ANL-94/6, Argonne National Lab, Lemont, IL, USA (1994). http://www.osti.gov/servlets/purl/10129052-6WVVjK/native/. Accessed 6 Dec 2018
32. Nalon, C., Dixon, C.: Anti-prenexing and prenexing for modal logics. In: M. Fisher, W. van der Hoek, B. Konev, A. Lisitsa (eds.) JELIA 2006, LNCS, vol. 4160, pp. 333–345. Springer, Berlin (2006)
33. Nalon, C., Dixon, C.: Clausal resolution for normal modal logics. J. Algorithms **62**, 117–134 (2007)
34. Nalon, C., Hustadt, U., Dixon, C.: A modal-layered resolution calculus for K. In: H. de Nivelle (ed.) TABLEAUX 2015, LNCS, vol. 9323, pp. 185–200. Springer, Berlin (2015)
35. Nalon, C., Hustadt, U., Dixon, C.: KSP: A resolution-based prover for multimodal K. In: N. Olivetti, A. Tiwari (eds.) IJCAR 2016, LNCS, vol. 9706, pp. 406–415. Springer, Berlin (2016)
36. Nalon, C., Hustadt, U., Dixon, C.: KSP: sources and benchmarks. University of Brasília, Brazil (2018). http://www.cic.unb.br/~nalon/#software. Accessed 6 Dec 2018
37. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 335–367. Elsevier, Amsterdam (2001)
38. Olesen, K.: BDDTab: sources and benchmarks. Australian National University, Canberra, Australia. http://users.cecs.anu.edu.au/~rpg/BDDTab/. Accessed 6 Dec 2018
39. Pan, G., Sattler, U., Vardi, M.Y.: BDD-based decision procedures for the modal logic K. J. Appl. Non Class. Logics **16**(1–2), 169–208 (2006)
40. Patel-Schneider, P.F., Sebastiani, R.: A new general method to generate random modal formulae for testing decision procedures. J. Artif. Intell. Res. (JAIR) **18**, 351–389 (2003)
41. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
42. Pratt, V.R.: Application of modal logic to programming. Stud. Log. **39**(2/3), 257–274 (1980)
43. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: R. Fikes, E. Sandewall (eds.) KR 1991, pp. 473–484. Morgan Kaufmann, Los Altos (1991)
44. Robinson, J.A.: Automatic deduction with hyper-resolution. Int. J. Comput. Math. **1**, 227–234 (1965)
45. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
46. Schild, K.: A correspondence theory for terminological logics. In: J. Mylopoulos, R. Reiter (eds.) IJCAI 1991, pp. 466–471. Morgan Kaufmann, Los Altos (1991)
47. Schulz, S.: Simple and efficient clause subsumption with feature vector indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics: Essays in Honour of William W. McCune. LNCS, vol. 7788, pp. 45–67. Springer, Berlin (2013)
48. Schuppan, V., Darmawan, L.: Evaluating LTL satisfiability solvers. In: T. Bultan, P.A. Hsiung (eds.) ATVA 2011, LNCS, vol. 6996, pp. 397–413. Springer, Berlin (2011)
49. Sebastiani, R., Vescovi, M.: Automated reasoning in modal and description logics via SAT encoding: the case study of K(m)/ALC-satisfiability. J. Artif. Intell. Res. **35**(1), 343–389 (2009)
50. Spaan, E.: Complexity of modal logics. Ph.D. thesis, University of Amsterdam, The Netherlands (1993)
51. Tsarkov, D.: FaCT++: sources. https://github.com/ethz-asl/libfactplusplus. Accessed 6 Dec 2018
52. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: U. Furbach, N. Shankar (eds.) IJCAR 2006, LNCS, vol. 4130, pp. 292–297. Springer, Berlin (2006)
53. Voronkov, A., Kovács, L., Reger, G., Suda, M., Kotelnikov, E., Robillard, S., Gleiss, B., Rawson, M., Bhayat, A., Riener, M.: Vampire. https://vprover.github.io/. Accessed 6 Dec 2018
54. Waaler, A.: Connections in nonclassical logics. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1487–1578. Elsevier, Amsterdam (2001)
55. Wallen, L.A.: Automated Deduction in Non-classical Logics. MIT Press, Cambridge (1990)
56. Wos, L., Robinson, G.A., Carson, D.F.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM **12**, 536–541 (1965)