Abstract. In this paper, we describe an implementation of a hyper-resolution-based calculus for the propositional basic multimodal logic, $K_n$. The prover was designed to support experimentation with different combinations of refinements for its basic calculus: it is primarily based on the set of support strategy, which can then be combined with other refinements, simplification techniques and different choices for the underlying normal form and clause selection. The prover allows for both local and global reasoning. We show experimental results for different combinations of strategies and comparison with existing tools.

1 Introduction

In this paper, we present K$P$, a theorem prover for the basic multimodal logic $K_n$ which implements a variation of the set of support strategy [20] for the modal resolution-based procedure described in [13]. The prover also implements several other refinements and simplification techniques in order to reduce the search space for a proof. Besides the set of support strategy, all other refinements of the calculus are implemented as independent modules, allowing for a better evaluation of how effective they are.

The paper is organised as follows. We introduce the syntax and semantics of $K_n$ in Section 2. In Section 3, we briefly describe the normal form and the calculus presented in [13]. Section 4 describes the available strategies and their implementations. Evaluation of strategies and of the performance of the prover compared to existing tools are given in Section 5. We summarise our results in Section 6.

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 modal formulae, $WFF_{K_n}$, is the least set such that every $p \in P$ is in $WFF_{K_n}$; if $\varphi$ and $\psi$ are in $WFF_{K_n}$, then so are \neg\varphi, (\varphi \land \psi), and [a]\varphi for each $a \in A$. The formulae false, true, (\varphi \lor \psi), (\varphi \Rightarrow \psi), and \Diamond \varphi are introduced as the usual abbreviations for (\varphi \land \neg\varphi), \neg\text{false}, \neg\neg(\neg\varphi \land \neg\psi), (\neg\varphi \lor \psi), and \neg[\Box] \neg\varphi, respectively (where $\varphi, \psi \in WFF_{K_n}$). A literal is either a propositional symbol or its negation; the set of literals is denoted by $L$. A modal
rules can then be guided by the semantic information given by the labels and applied to a modal literal. Transformation into SNF modal clauses disjunction of literals, and expected: for a model $M$, SNF depth that

As our calculus operates on a labelled clausal normal form that is closely linked to the tree model property of Kripke models for $K_n$, we briefly overview of the semantics of $K$. A tree-like 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 $R_n$ is a binary relation on $W$ such that their union is a tree with root $w_0$, and $\pi : W \rightarrow (P \rightarrow \{\text{true, false}\})$ is a function which associates with each world $w \in W$ a truth-assignment to propositional symbols. Satisfaction of a formula at a world $w$ of a model $M$ is defined by:

- $(M, w) \models \varphi$ if, and only if, $\pi(w)(p) = \text{true}$, where $p \in P$;
- $(M, w) \models \neg \varphi$ if, and only if, $(M, w) \not\models \varphi$;
- $(M, w) \models (\varphi \land \psi)$ if, and only if, $(M, w) \models \varphi$ and $(M, w) \models \psi$;
- $(M, w) \models \Box \varphi$ if, and only if, for all $w'$, $wR_nw'$ implies $(M, w') \models \varphi$.

Let $M = (W, w_0, R_1, \ldots, R_n, \pi)$ be a model. A formula $\varphi$ is locally satisfied in $M$, denoted by $M \models_L \varphi$, if $(M, w_0) \models \varphi$. The formula $\varphi$ is locally satisfiable if there is a model $M$ such that $(M, w_0) \models \varphi$. A formula $\varphi$ is globally satisfied in $M$, if for all $w \in W$, $(M, w) \models \varphi$. We denote by 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 $\varphi$ 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 [11]). Due to this close correspondence of modal layer and modal level we use the terms interchangeably. Also, checking the global satisfiability of $\varphi$ can be reduced to checking the local satisfiability of $\varphi$ at all modal layers (up to an exponential distance from the root) of a model [31][13]. Thus, an uniform approach based on modal levels can be used to deal with both problems, as we show in the next section.

### 3 A Calculus for $K_n$

The calculus for $K_n$ presented in [13] is clausal, where clauses are labelled by the modal level at which they occur. In order to refer explicitly to modal levels, the modal language is extended with labels. We write $ml : \varphi$ to denote that $\varphi$ is true at the modal layer $ml$ in a Kripke model, where $ml \in \mathbb{N} \cup \{*\}$. By $*: \varphi$ we mean that $\varphi$ is true at all modal layers in a Kripke model. The notion of local satisfiability is extended as expected: for a model $M$, $M \models_L ml : \varphi$ if, and only if, for all worlds $w \in W$ such that depth($w$) = $ml$, we have $(M, w) \models_L \varphi$. Then, the layered normal form, called SNF$_{ml}$, is given by a conjunction of literal clauses of the form $ml : D$, where $D$ is a disjunction of literals, and modal clauses of the form $ml : l \Rightarrow l'$, where $l \in L$ and $l'$ is a modal literal. Transformation into SNF$_{ml}$ uses renaming and preserves satisfiability.

The motivation for the use of this labelled clausal normal form is that inference rules can then be guided by the semantic information given by the labels and applied to
Inference rules given in Table 1. Unification on sets of labels is defined by improving the efficiency of the proof procedure. The calculus comprises the set of inference rules, where $\text{ml} : D \lor l$ and $\text{ml} : l' \lor D'$ are first transformed into their prenex or antiprenex normal form (or one after the other) [11] and then into Negation Normal Form (NNF) or into Box Normal Form (BNF) [15]. With options $\text{nnfsimp}$ (resp. $\text{bnfsimp}$), simplification is applied to formulae in NNF (resp. BNF); with option $\text{earlymlple}$, pure literal elimination is applied at every modal level. There are then four different options that determine the normal form. In $\text{SNF}^-_{\text{ml}}$, negative literals in the scope of modal operators are renamed by propositional symbols; in $\text{SNF}^+_{\text{ml}}$, all literals in the scope of modal operators are renamed by propositional symbols. $\text{SNF}^-_{\text{ml}}$ and $\text{SNF}^+_{\text{ml}}$ are defined analogously, with positive literals being renamed by negative ones. The reuse of propositional symbols in renaming can also be controlled.

### Table 1: Inference rules

<table>
<thead>
<tr>
<th>LRES</th>
<th>MRES</th>
<th>GEN2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ml : D \lor l$</td>
<td>$ml : l_1 \Rightarrow \Box l$</td>
<td>$ml_1 : l'_1 \Rightarrow \Box l_1$</td>
</tr>
<tr>
<td>$ml' : D' \lor l$</td>
<td>$ml' : l_2 \Rightarrow \Diamond \neg l$</td>
<td>$ml_2 : l'_2 \Rightarrow \Diamond \neg l_1$</td>
</tr>
<tr>
<td>$ml : \neg l_1 \lor \neg l_2$</td>
<td>$ml : \neg l'$</td>
<td>$ml_3 : l'_3 \Rightarrow \Diamond l_2$</td>
</tr>
</tbody>
</table>

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

4 Implementation

$\text{KSP}$ is an implementation, written in C, of the calculus given in [13]. 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. one technique needs to be applied after the other, whereas most tools would apply them together), but it helps to evaluate how the different options independently contribute to achieve efficiency. In the following we give a brief overview of the main aspects of the implementation.

**Transformation to clausal form:** If the input is a set of formulae, then these formulae are first transformed into their prenex or antiprenex normal form (or one after the other) [11] and then into Negation Normal Form (NNF) or into Box Normal Form (BNF) [15]. With options $\text{nnfsimp}$ (resp. $\text{bnfsimp}$), simplification is applied to formulae in NNF (resp. BNF); with option $\text{earlymlple}$, pure literal elimination is applied at every modal level. There are then four different options that determine the normal form. In $\text{SNF}^-_{\text{ml}}$, negative literals in the scope of modal operators are renamed by propositional symbols; in $\text{SNF}^+_{\text{ml}}$, all literals in the scope of modal operators are renamed by propositional symbols. $\text{SNF}^-_{\text{ml}}$ and $\text{SNF}^+_{\text{ml}}$ are defined analogously, with positive literals being renamed by negative ones. The reuse of propositional symbols in renaming can also be controlled. In our evaluation, given in Section 5, the same propositional symbol is used for all occurrences of a formula being renamed.
Preprocessing of clauses: Self-subsumption is applied at this step if the options for forward and/or backward subsumption are set [2]. The inference rules MRES and GEN2 are also exhaustively applied at this step, that is, before the prover enters the main loop.

Main loop: The main loop is based on the given-clause algorithm implemented in Otter [10], a variation of the set of support strategy [20], a refinement which restricts the set of choices of clauses participating in a derivation step. For the classical case, a set of clauses $\Delta$ is partitioned in two sets $\Gamma$ and $\Lambda = \Delta \setminus \Gamma$, where $\Lambda$ must be satisfiable. McCune refers to $\Gamma$ as the set of support (the sos, aka passive or unprocessed set); and $\Lambda$ is called the usable (aka as active or processed set). The given clause is chosen from $\Gamma$, resolved with clauses in $\Lambda$, and moved from $\Gamma$ to $\Lambda$. Resolvents are added to $\Gamma$. For the modal calculus, the set of clauses is further partitioned according to the modal layer at which clauses are true. That is, for each modal layer $ml$ there are three sets: $\Gamma_{lit}^{ml}$, $\Lambda_{lit}^{ml}$ and $\Lambda_{mod}^{ml}$, 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, 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 $\Gamma_{lit}^{ml}$, which can be resolved with either a literal clause in $\Lambda_{lit}^{ml}$ or with a set of modal clauses in $\Lambda_{mod}^{ml-1}$. There are six options for automatically populating the usable: all negative clauses, all positive clauses, all non-negative clauses, all non-positive clauses, all clauses whose maximal literal is positive, and all clauses whose maximal literal is negative. The prover can either perform local or global reasoning.

Refinements: Besides the basic calculus with a set-of-support strategy, the user can further restrict LRES by choosing ordered clauses (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 negative + ordered resolution (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 $\{p, p \Rightarrow \Box \neg q, p \Rightarrow \Diamond s, \neg s \lor q\}$ is unsatisfiable, but as there is no negative literal clause in the set, no refutation can be found. By renaming $\neg q$ in the scope of $\Box$, we obtain the set $\{p, p \Rightarrow \square t, p \Rightarrow \Diamond s, \neg s \lor q, \neg t \lor \neg q\}$ 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$_{--}^{ml}$.

Inference rules: Besides the inference rules given in Table [7], three more inference rules are also implemented: unit resolution, which propagates unit clauses through all literal clauses and the right-hand side of modal clauses; lhs unit resolution, which propagates unit clauses through the left-hand side of modal clauses; and ires, which together with the global option, implements initial resolution and, thus, the calculus given in [12].

Redundancy elimination: Pure literal elimination can be applied globally or by modal layer. Both forward and backward subsumption are implemented. Subsumption is applied in lazy mode: a clause is tested for subsumption only when it is selected from the
sos and only against clauses in the usable. As pointed out in [17], this avoids expensive checks for clauses that might never be selected during the search of a proof.

Clause selection: There are five different heuristics for choosing a literal clause in the sos: shortest, newest, oldest, greatest maximal literal, and the smallest maximal literal.

For a comprehensive list of options, see [14], where the sources and instructions on how to install and use KṣP can be found.

5 Evaluation

We have compared KṣP with BDDTab [4], FaCT++ 1.6.3 [19], InKreSAT 1.0 [7], Spartacus 1.0 [5], and a combination of the optimised functional translation [6] with Vampire 3.0 [8]. 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 [14] consist of three collections of modal formulae:

1. The complete set of TANCS-2000 modalised random QBF (MQBF) formulae [9] complemented by the additional MQBF formulae provided by Tebbi and Kaminski [7]. 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 [2], with 56 formulae chosen from each of the 18 parameterised classes. In most previous uses of these benchmarks, only parameter values 1 to 21 were used for each class, with the result that provers were able to solve all benchmark formulae for most of the classes. Instead we have chosen the 56 parameter values so that the best current prover will not 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. Of the resulting 1008 formulae, half are satisfiable and half are unsatisfiable by construction of the benchmark classes.

3. Randomly generated 3CNF$_K$ formulae [16] 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 [16]. 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 [7] which consisted of 135 3CNF$_K$ formulae over 3 propositional symbols with modal depth 2, 4 or 6, all of which were satisfiable.

Benchmarking was performed on PCs with an Intel i7-2600 CPU @ 3.40GHz and 16GB 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.

---

3 We have excluded *SAT from the comparison as it produced incorrect results on a number of benchmark formulae.
Figure 1a compares the impact of different refinements on the performance of KSP on the MQBF collection. With plain KSP uses the rules shown in Table 1 without additional refinement, on a set of $\text{SNF}^{++}$ clauses. With cord and negative ordered, KSP applies ordered resolution and negative + ordered resolution, respectively, again on a set of $\text{SNF}^{++}$ clauses. The configuration negative uses negative resolution on a set of $\text{SNF}^{+}$ clauses, while with positive, KSP applies positive resolution on $\text{SNF}^{-}$ clauses. Irrespective of the refinement, the shortest clause is selected to perform inferences; both forward and backward subsumption are used; the lhs-unit resolution rule is applied; prenex is set; and no simplification steps are applied. The usable is populated with clauses whose maximal literal is positive, except for positive resolution where it is populated with clauses whose maximal literal is negative. The cord configuration offers the best performance. Ordered resolution restricts the applicability of the rules further than the other refinements. Not only 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 finds refutations much more quickly than with any of the other refinements.

Figure 1b compares the performance of all the provers on the MQBF collection. It shows that KSP performs better than any of the other provers. The graphs in Figure 2 offer some insight into why KSP performs well on these formulae. Each of the five graphs shows for one formula from each class how many atomic subformulae occur at different modal levels.
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 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.

Figure 3 shows the benchmarking results on the LWB and 3CNF \( K \) collections. On the LWB collection KSP performs about as well as BDDTab, FaCT++ and InKreSAT, while Spartacus performs best and the combination of the optimised functional translation with Vampire (OFT + Vampire) performs worst. Table 2 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 param-

<table>
<thead>
<tr>
<th>k_branch_n</th>
<th>BDDTab</th>
<th>FaCT++</th>
<th>InKreSAT</th>
<th>KSP (cord)</th>
<th>Spartacus</th>
<th>OFT + Vampire</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>22</td>
<td>12</td>
<td>12</td>
<td>15</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>12</td>
<td>12</td>
<td>22</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>20</td>
<td>440</td>
<td>6</td>
<td>40</td>
<td>34</td>
<td>48</td>
<td>1560</td>
</tr>
<tr>
<td>26</td>
<td>640</td>
<td>24</td>
<td>600</td>
<td>18</td>
<td>360</td>
<td>54</td>
</tr>
<tr>
<td>39</td>
<td>2400</td>
<td>42</td>
<td>2640</td>
<td>23</td>
<td>1120</td>
<td>49</td>
</tr>
<tr>
<td>42</td>
<td>2640</td>
<td>38</td>
<td>2320</td>
<td>28</td>
<td>1520</td>
<td>50</td>
</tr>
<tr>
<td>35</td>
<td>2600</td>
<td>27</td>
<td>1800</td>
<td>50</td>
<td>4500</td>
<td>5</td>
</tr>
<tr>
<td>35</td>
<td>2600</td>
<td>27</td>
<td>1800</td>
<td>51</td>
<td>5000</td>
<td>29</td>
</tr>
<tr>
<td>46</td>
<td>4000</td>
<td>43</td>
<td>3400</td>
<td>33</td>
<td>2500</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>5000</td>
<td>28</td>
<td>10000</td>
<td>56</td>
<td>50000</td>
<td>23</td>
</tr>
<tr>
<td>37</td>
<td>290</td>
<td>48</td>
<td>400</td>
<td>7</td>
<td>14</td>
<td>54</td>
</tr>
<tr>
<td>35</td>
<td>270</td>
<td>48</td>
<td>400</td>
<td>5</td>
<td>12</td>
<td>54</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>90</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>39</td>
<td>600</td>
<td>34</td>
<td>500</td>
<td>30</td>
<td>36</td>
<td>540</td>
</tr>
<tr>
<td>38</td>
<td>580</td>
<td>34</td>
<td>500</td>
<td>28</td>
<td>400</td>
<td>36</td>
</tr>
<tr>
<td>40</td>
<td>3500</td>
<td>24</td>
<td>1500</td>
<td>17</td>
<td>800</td>
<td>39</td>
</tr>
<tr>
<td>48</td>
<td>7500</td>
<td>49</td>
<td>8000</td>
<td>28</td>
<td>49</td>
<td>8000</td>
</tr>
</tbody>
</table>

Table 2: Detailed benchmarking results on LWB
eter value but fluctuates instead. As is indicated, BDDTab and InKreSAT are the best performing provers on one class each, OFT + Vampire on two, KSP on six, and Spartacus on eight 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 \( k_{\text{lin}_p} \) are unsatisfiable. For \( k_{\text{grz}_p} \), pure literal elimination can be used to reduce all formulae in this class to the same simple formula; the same is true for \( k_{\text{grz}_n} \) and \( k_{\text{lin}_n} \). Thus, these classes are tests how effectively and efficiently, if at all, a prover uses these techniques and Spartacus does best on these classes. Note that pure literal elimination has been disabled in the \textit{cord} configuration we have used for KSP. With it KSP would perform better on \( k_{\text{grz}_p} \) and \( k_{\text{grz}_n} \), but worse on other classes where this simplification has no beneficial effects.

Finally, 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.

## 6 Conclusions and Future Work

The evaluation indicates 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 and 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.

Fig. 3: Benchmarking results for LWB and 3CNF\( _K \)
References