On Checking Kripke Models for Modal Logic K

Jean-Marie Lagniez, Daniel Le Berre, Tiago de Lima, Valentin Montmirail

On Checking Kripke Models for Modal Logic K

Jean-Marie Lagniez Daniel Le Berre Tiago de Lima Valentin Montmirail

CRIL, Univ. Artois and CNRS, F62300 Lens, France
{lagniez,leberre,delima,montmirail}@cril.fr

Abstract

This article presents our work toward a rigorous experimental comparison of state-of-the-art solvers for the resolution of the satisfiability of formulae in modal logic K. Our aim is to provide a pragmatic way to verify the answers provided by those solvers. For this purpose, we propose a certificate format and a checker to validate Kripke models for modal logic K. We present some experimental results using efficient solvers modified to incorporate this verification step. We have been able to validate at least one certificate for 67 percent of the satisfiable problems, which provides a set of benchmark with independently checked solution which can be used to validate new solvers. We discuss the limits of our approach in the last part of this article.

1 Introduction

The practical evaluation of running systems played a key role in the success of SAT solvers for classical propositional logic [2, 18, 30]. When the latter became efficient enough to solve "real problems", even more improvements were observed, thanks to the huge interest on that new technology. We believe that such success can be repeated for many other systems. In this paper, we are interested in making it happen for satisfiability in modal logic K.

There are however a few conditions for that. First, there is a need for a common input format. For SAT, such format was provided thanks to the Second Dimacs Challenge [5]. While both a CNF and a generic format were proposed in 1993, only the CNF format is currently supported by SAT solvers. Second, it is important to check the answers produced by a solver. There are several reasons why a solver could return an incorrect answer. It may contain bugs on corner cases such as trivial answers, very large files, very deep formulae, etc. It might also be the case that the solver does not read properly the input.

In the early days of the development of SAT solvers, the only kind of answer that could be checked was SAT: a model was given to a third party tool which could independently check that the formula was indeed satisfied by that model [30]. More recently [16], checking UNSAT answers became also possible in practice for many solvers: during its computation, the system writes in a log file some basic steps of the solver. This log file is then analysed and the answer can be verified.

For more complex logics, it is not always possible to check the answers produced by the solvers: take QBF for instance, i.e. propositional logic plus quantifiers on variables. QBF solvers answers can hardly be verified. One would need to gather policies instead of a single model. While much effort has been devoted to produce certificates for QBF, it is still an open challenge [24]. Our work is in line with the one being done for QBF. We aim at verifying, when possible, the answers provided by solvers for the satisfiability of modal logic K formulae, and building a library of benchmarks with verified answers to foster the development of new solvers. Therefore, we are faced with two problems: producing a certificate on the solver side and checking it using an independent tool. In this paper, we address both problems and present some experimental results with verified answers.

In: P. Fontaine, S. Schulz, J. Urban (eds.): Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning (PAAR 2016), Coimbra, Portugal, 02-07-2016, published at http://ceur-ws.org

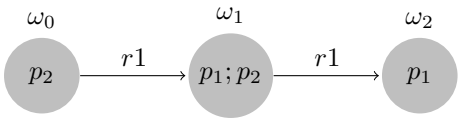


Figure 1: A Kripke Model.

The remainder of the paper is organised as follows. Section 2 presents the basic modal logic K, its syntax and semantics. Section 3 presents the algorithm used for the verification of the certificates. Section 4 presents the I/O formats we propose for the certificates. In Section 5 we present the settings of our experimental evaluation whose results are presented and discussed in the subsequent section. The final section draws some conclusions and provides some followup research directions.

2 Preliminaries: Modal Logic K

Definition 2.1. The language of Modal Logic K (or simply K) is the language of classical propositional logic extended with 2 unary operators: $\Box$ and $\Diamond$.

A formula of the form $\Box\varphi$ (*box phi*) means φ is necessarily true. A formula of the form $\Diamond\varphi$ (*diamond phi*) means φ is possibly true.

Definition 2.2 (Modal Depth)**.** The modal depth of a formula φ in the language K, noted $\mathrm{md}(\varphi)$, is defined by (where $\oplus \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$):

$$\begin{aligned} \mathrm{md}(p) = \mathrm{md}(\top) = \mathrm{md}(\bot) &= 0 \\ \mathrm{md}(\neg\varphi) &= \mathrm{md}(\varphi) \\ \mathrm{md}(\varphi \oplus \psi) &= \max(\mathrm{md}(\varphi), \mathrm{md}(\psi)) \\ \mathrm{md}(\Box\varphi) = \mathrm{md}(\Diamond\varphi) &= 1 + \mathrm{md}(\varphi) \end{aligned}$$

For example, $\mathrm{md}(\Box(p_1 \vee \Diamond p_2 \vee \Diamond p_3)) = 2$. The language K is interpreted using Kripke semantics [20], which uses Kripke models.

Definition 2.3 (Kripke Model)**.** Let a non-empty countable set of propositional variables P be given. A Kripke model is a triplet $M = \langle W, R, \mathcal{I}\rangle$, where: W is a non-empty set of possible worlds; R is a binary relation on W (called accessibility relation); and $\mathcal{I}$ is a function which associates, to each $p \in P$, the set of possible worlds from W where p is true.

Example 2.1. Let $M = \langle W, R, \mathcal{I}\rangle$, where: $W = \{\omega_0, \omega_1, \omega_2\}$, $R = \{(\omega_0, \omega_1), (\omega_1, \omega_2)\}$, $\mathcal{I} = \{(p_1, \{\omega_1, \omega_2\}), (p_2, \{\omega_0, \omega_1\})\}$. A graphical representation of M is given in Figure 1.

Definition 2.4 (Pointed Kripke Model)**.** A pointed Kripke model is a pair $\langle M, \omega_0\rangle$, where M is a Kripke model and ω_0 is a possible world in W.

In the rest of the paper, we will simply use "model" to denote a "pointed Kripke model".

Definition 2.5 (Satisfaction Relation)**.** The satisfaction relation $\vDash$ between formulae and models is recursively defined as follows (the semantics of the operators $\top$, $\bot$, $\vee$, $\rightarrow$ and $\leftrightarrow$ is defined as usual):

$M, \omega \vDash p$ iff $\omega \in \mathcal{I}(p)$
$M, \omega \vDash \neg\varphi$ iff $M, \omega \nvDash \varphi$
$M, \omega \vDash \varphi \wedge \psi$ iff $M, \omega \vDash \varphi$ and $M, \omega \vDash \psi$
$M, \omega \vDash \Box\varphi$ iff for all v if $(\omega, v) \in R$ then $M, v \vDash \varphi$
$M, \omega \vDash \Diamond\varphi$ iff there exists v such that $(\omega, v) \in R$ and $M, v \vDash \varphi$

Definition 2.6 (Satisfiability)**.** A formula φ is satisfiable in K if and only if there exists a model $\langle M, \omega_0\rangle$ that satisfies φ.

Definition 2.7 (Validity)**.** A formula φ is valid in K if and only if every model $\langle M, \omega_0\rangle$ satisfies φ.

Definition 2.8 (K-Satisfiability Problem)**.** Let φ be a formula in the language K. The K-satisfiability problem (K-SAT) is the problem of answering YES or NO to the question "Is φ satisfiable?".

As one might expect, software trying to solve this problem may try to find a model that satisfies the formula given as input. If such model is found, the software can answer YES (or SAT) and provides the model to justify its answer. However, the size of the model may be exponential in the size of the input. Indeed, K-SAT is PSPACE-complete [14, 21]. In this paper, we are interested in the practical aspects of model verification i.e., to know in practice how big are such models, and how many of them could be checked.

3 Checking satisfiable answers

As mentioned in the introduction, it is important to be able to verify the answers given by the K-SAT solvers being evaluated. Ideally, when a K-SAT solver answers SAT, it should also provide a certificate, a model. Such an answer could thus be independently verified to check that it satisfies the input formula.

The method used to verify certificates amounts to what is commonly called Model Checking [6]. Several approaches to modal logic model checking can be found in the literature [7, 10, 27]. However, to the best of our knowledge, those existing checkers are designed for logics that are different from K (often Temporal Logic) and therefore are not easily adaptable to the task of checking a modal logic K model.

The algorithm we use is based on Algorithm 1, with a reasonable effort in the implementation to make it efficient enough to check the certificates produced in our experimental evaluation in reasonable time (less than 300s). The code is written in C++ and accessible online[1].

Algorithm 1: check(φ,M,ω_i)

Data: φ: a formula, M : a model, ω_i : a world
Result: *true* if $M, \omega_i \vDash \varphi$, $false$ otherwise

```
1  begin
2      if (φ = □ψ) then
3          for each ω_j successor of ω_i do
4              if (not check(ψ, M, ω_j)) then
5                  return false
6          return true
7      if (φ = ◇ψ) then
8          for each ω_j successor of ω_i do
9              if (check(ψ, M, ω_j)) then
10                 return true
11         return false
12     if (φ = (ψ ∧ ϕ)) then
13         return (check(ψ, M, ω_i) ∧ check(ϕ, M, ω_i))
14     if (φ = (ψ ∨ ϕ)) then
15         return (check(ψ, M, ω_i) ∨ check(ϕ, M, ω_i))
16     if (φ = ¬ψ) then return (not check(ψ, M, ω_i))
17     if (φ = p_j) then return (M[ω_i] contains p_j)
```

The recursive function *check*() is called with the model provided by the K-SAT solver. Its correctness is easily verified, as it implements each clause of Definition 2.5. Nonetheless, we still had to optimise the procedure. To see why, assume the following input formula $\varphi = \Diamond p_1 \wedge \Diamond p_2 \wedge \cdots \wedge \Diamond p_n$. An efficient way to generate a model for this formula is to start with a possible world ω_0 and create a new accessible world ω_i for each sub-formula $\Diamond p_i$ of φ. This is indeed what some of the solvers we tested do.

Unfortunately, the procedure in Algorithm 1 is very inefficient to check that a model generated in this way indeed satisfies φ. This is because, for each conjunct $\Diamond p_i$ of φ, the procedure will check if each possible world ω_i satisfies p_i: first, it takes the first conjunct $\Diamond p_1$ and successfully checks p_1 against ω_1; then, it takes $\Diamond p_2$ and

[1] `http://www.cril.univ-artois.fr/~montmirail/mdk-verifier`

checks p_2 against ω_1, and fails; it checks p_2 against ω_2 and succeed; and so on. In the i-th iteration, it checks p_i against i possible worlds before succeeding. It is clear that the model checking procedure takes much more time to finish than the satisfiability procedure itself. Indeed, the naive algorithm could not verify (in a reasonable time) the majority of the models proposed as solution.

To minimise this problem, we used a kind of caching. On the i-th iteration, the procedure does not check only p_i against ω_i, but also all sub-formulae of φ, and then stores the results on ω_i. When it comes back to ω_0, it does not need to explore ω_i again. On the iteration $i+1$, if necessary, it goes directly to the unexplored worlds, thus starting from ω_{i+1}.

The second optimisation deals with formulae of the form $\Diamond\Diamond\ldots\Diamond p_1$. Here, the satisfiability method will create a long chain of possible worlds. Instead of exploring them one by one, the optimised algorithm stores the length of those chains of "empty possible worlds", and jumps directly to the last possible world when necessary.

4 Input and Output Formats

There exists several different input formats for modal logic. Among them, we can cite ALC, used by *SAT [11]. For example, the formula $((p \to \Diamond q) \wedge \Box q)$ is written in ALC as `(AND (IMP C0 (SOME R0 C1)) (ALL R0 C1))`. This format is purely textual, functional, but uses quantifiers (`SOME`, `ALL`) to express modal operators, which can be confusing. Another (very similar) example is KRSS [25]. The same formula can be written in KRSS as `(((not C0) or (some R0 C1)) and (all R0 C1)))`. This format, in addition, lacks symbols for implication and equivalence. As a third example, we can cite LWB [17]. The same formula is written in LWB as `begin (p1 => dia(p2)) & box(p2) end`. Unfortunately, this format does not allow for the most natural future extension of our approach, namely, the representation of multiple modalities (multiple agents).

4.1 The Input format InToHyLo

We decided to use InToHyLo as input format. It is used, for example, by the solvers InKreSAT [19] and Spartacus [13]. The formula $((p \to \Diamond q) \wedge \Box q)$ is written in InToHyLo as `((p1 -> <r1>p2) & [r1]p2 )`. We believe that this format is easy to read and it is also easily adaptable for multiple modalities.

Definition 4.1 (InToHyLo Language)**.** The InToHyLo Language is defined by the following Backus-Naur Form grammar, where identifiers (`id`) are numerical sequences.

⟨*file*⟩ ::= '`begin`' ⟨*fml*⟩ '`end`'

⟨*fml*⟩ ::= '`(`' ⟨*fml*⟩ '`)`'
| '`true`' | '`false`' | '`p`'⟨*id*⟩ | '`~`' ⟨*fml*⟩
| '`<r`' ⟨*id*⟩ '`>`' ⟨*fml*⟩ | '`[r`' ⟨*id*⟩ '`]`' ⟨*fml*⟩
| ⟨*fml*⟩ '`&`' ⟨*fml*⟩ | ⟨*fml*⟩ '`|`' ⟨*fml*⟩
| ⟨*fml*⟩ '`->`' ⟨*fml*⟩ | ⟨*fml*⟩ '`<->`' ⟨*fml*⟩

Unary operators have the highest precedence in InToHyLo. The precedence of the binary operators is the following: `&, |, ->, <->`.

4.2 The Output Format Flat Kripke Model

In order to be able to check answers produced by the solvers, we also propose an output format to represent Kripke models. This should be seen as an exchange format between softwares, and not something written directly by a human, in the spirit of the DIMACS CNF format [5].

Below, we have a representation of the model in Figure 1 in the proposed format.

```
2 3 1 2
-1 2 0
1 2 0
1 -2 0
r1 w0 w1
r1 w1 w2
```

The first line contains exactly four integers separated by spaces. They provide respectively the number of propositional variables, the number of possible worlds, the number of relations (which will be useful in the future for multi-modal problems), and finally the number of edges in the model. In the subsequent lines, each propositional variable is represented by a positive integer. For example, if there exists 3 propositional variables in the model, then they are named 1, 2 and 3. The second line corresponds to the valuation of the first possible world (i.e., ω_0). The third line corresponds to the valuation of the second possible world (ω_1), and so on. Each of these lines follows the DIMACS format [5]: the integer i means that the propositional variable i is assigned to *true* in that world and $-i$ means that the propositional variable i is assigned to

false in that world. There must be exactly one valuation line for each possible world in the model. Each of these lines must end with a 0. After that, the connections between worlds are provided. Each line has the format: `rI wJ wK`, where `I`, `J` and `K` are integers. The first one represents the I-th relation of the model, the second one is the J-th world and the third one is the K-th world. Each one of these lines means that ω_K is reachable from ω_J in the accessibility relation number I. There must be exactly one such line for each edge of the model.

5 Evaluation settings

5.1 Solvers

There are numerous solvers for modal logic K, developed in the last two decade. Some of the earlier solvers have the ability to output a model: SPASS [33] provides the branches open in every world or LWB [15] which provide an image representing a Kripke model. However, those solvers are no longer "state-of-the-art" in terms of runtime (see e.g. [29, 1] for a comparison with more recent solvers considered here). We decided to consider all but one of the solvers used in [19], which is, to the best of our knowledge, the most recent paper comparing modal logic K solvers. FaCT++ [32] (v1.6.1), an established reasoner for the web ontology language OWL 2 DL, is missing in our experiments because it is a Tableau solver like Spartacus generally outperformed by Spartacus according to [13].

Note that our goal in this work was not to perform a comparison of all existing solvers because we needed to modify those solvers ourselves, but to show that it is possible to check the answers of different solvers using an independent tool. We hope it will encourage other authors to allow their solvers to be able to output the model in our Flat Kripke Model format.

5.1.1 Km2SAT

Km2SAT [29] translates modal logic formulae into an equisatisfiable CNF formula which can be solved by any SAT solver. We used Minisat 2.2.0 [8] in our experiments. If the formula is unsatisfiable, then the original formula is unsatisfiable. If the formula is satisfiable, the satisfying assignment found can be transformed into a model. In such case, we only need to interpret the assignment and output it using our proposed format. Unfortunately, Km2SAT, by default, may modify the input formula before applying the translation into CNF. Such optimisation preserves the satisfiability of the formula, but prevents us, in some cases, to retrieve a model for the original input.

Km2SAT reads formulae in the LWB format. Thus, we had to modify the original solver to allow it to read formulae in InToHyLo format. Such modification is obviously a threat to its correctness. We did our best to make sure that the modifications on the input format did not impact the solver itself. To that end, we ran the solver on examples using the LWB format supported by Km2SAT. Then, we used the Fast Transformation Tool (ftt) embedded with the software Spartacus [13] to transform those LWB problems into InToHyLo problems and we checked if the modified version of Km2SAT returned the exact same models. This was performed on the 240 instances of TANCS-2000-modKSSS, on the 80 instances of TANCS-2000-modkLadn, and on the 259 instances of LWB solved by the modified Km2SAT. In all cases we obtained the same result.

5.1.2 *SAT

*SAT [11] (v1.3) is a reasoner for the description logic $\mathcal{ALC}$. *SAT implements a modal extension of the Davis-Putnam procedure. But, because *SAT features several decision procedures of classical modal logics, it can perfectly be used as a K modal logic solver.

First of all, *SAT reads the ALC format. So, we had to modify the input parser in order to make this solver read InToHyLo files. Because this threatens its correctness, we used the Fast Transformation Tool to translate InToHyLo formatted benchmarks into ALC and then ran the original version of *SAT on the translated benchmarks. In all cases we obtained the same result.

Second, the interpretation of the result of *SAT is more challenging than for Km2SAT. Indeed, *SAT uses the SAT solver in an interleaved and incremental approach: the SAT solver is called many times, since it drives the verification of the tableau rules. As such, analyzing a single SAT answer does not allow in general to retrieve a complete model. For that reason, many of the SAT answers provided by *SAT could not be verified.

*SAT uses the SAT solver SATO [34] as backend. It was designed in the late nineties when only "small" CNF where considered. The solver accepts by default CNF with up to 10K variables and no more than 256 literals per clause. SATO may report unexpected results if those limits are enforced. We increased the variables boundary

to 30K and added some code to abort the process if the generated CNF contains more than 30K variables or 256 literals in a clause.

5.1.3 InKreSAT

InKreSAT [19] is a prover for the modal logics K, T, K4, and S4. InKreSAT reduces a tableau based approach for the modal satisfiability problem to a series of Boolean satisfiability problems. By proceeding incrementally, it interleaves translation steps with calls to the SAT solver and uses the feedback provided by the SAT solver to guide the translation. InKreSAT has very good results in term of speed when solving modal logic problems. However, we could not output a model each time it answers SAT. The modifications needed to perform this task require a deep knowledge of the solver and important changes that would make the modified solver quite different from the original one. As such, we decided to simply identify the branches that are true in the implicit tableau method. In some cases, that information is sufficient to extract a model. In other cases, InKreSAT sets a non-atomic branch as true (eg.: $\Diamond(p_1 \wedge p_2) \vee \Box p_2$). Unless a few more tableaux rules are performed, it is not possible to generate a model with all the information. In the latter case, we choose to display the incomplete model.

5.1.4 Spartacus

Spartacus [13] is a tableau prover for hybrid multimodal logic with global modalities and reflexive and transitive relations. This solver has a module that displays what we first thought was a model. It is quite often a model, but in some cases some parts of the model are "missing". After some discussion with the authors of Spartacus, we realised that, in fact, Spartacus produces a representation of an open saturated tableau, which indicates the existence of a model but is not always a complete Kripke model. We output that open saturated tableau using the flat Kripke model format. This explains why in some cases, the certificate provided by Spartacus cannot be verified by our checker.

5.2 Experimental settings

The solvers ran on a cluster of identical computers with two processors Intel XEON E5-2643 - 4 cores - 3.3 GHz running CentOS 6.0 x86_64 with 32 GB of memory. Each solver was given 4 cores for its execution, and a timeout of 900s to solve each benchmark with a memory limit of 15500 MB using the tool runsolver [26]. The results are given per family of benchmarks. For each family, we provide the number of benchmarks for which the model provided by at least one solver could be independently checked in the Verified SAT column. We were able to verify globally 36% of the whole problems, and 67% of the SAT answers. Any new solver which would answer differently on such verified problems could be considered incorrect. We believe that this is important information for anyone willing to develop a new modal logic K solver (or to evaluate an existing one). Note that we did not find any discrepancy in the results: no solver answered UNSAT while another answered SAT. So, we consider all the solvers here as correct, even if all answers cannot be verified.

6 Results

In the following tables, bold numbers denote the highest number of problems solved, numbers between parenthesis denote the number of benchmarks for which the solver is the fastest, and numbers between square brackets (when the sub-category is mixing SAT/UNSAT problems) denote the number of SAT answers, i.e. candidates for verification.

6.1 3CNF_K

The Randomly Generated 3CNF_K formulae [12] have four parameters: the modal depth (d); the number of clauses (m); the number of propositional variables (n); and the probability that a disjunction occurring at a degree inferior to d is purely propositional (p).

For each depth d, the problems consists of 9 formulae with $m = \{30, 60, 90, 120, 150\}$, respectively. Parameter n is always equals to 3 and p is always equals to 0.00%. See [12] for details.

d	#	**Km2SAT**	***SAT**	**InKreSAT**	**Spartacus**	**Verified SAT**
2	45	**45**	33	**45**	42	26 / 45 (57.78%)
4	45	9/MO	20	12	**45**	**45 / 45** (100.0%)
6	45	0/MO	7	8	**45**	**45 / 45** (100.0%)
total	135	54 (0)	60 (2)	65 (13)	**132** (120)	116 / 135 (85.92%)

Table 1: 3CNF_K Results (d = modal depth)

All the problems are satisfiable, and all of them have been solved by at least one solver. Unfortunately, since we are not able to provide a certificate for all solvers, we were able to verify only the benchmarks for modal depth greater than 2, which amounts to 85.92% of the problems. One may wonder why Spartacus solves problems when the modal depth is increasing, but does not perform as well as InKreSAT and Km2SAT when the modal depth is equal to 2. Basically, Spartacus did not manage to solve 3 instances with parameters m=150 and d=2. Those problems with many clauses and a small modal depth are closer to a SAT problem than a modal-SAT problem for which Spartacus is designed. This is also why InKreSAT and Km2SAT (which are SAT-based) performed so well. On the smaller modal depth, only half of the answers are verified despite several solvers being able to decide the satisfiability of the benchmarks. This is the consequence of Spartacus being the only provider of verified answers on the whole set of benchmarks and that the saturated open tableau it generates is not a full Kripke model on small modal depth. *SAT did not perform as good as the other SAT based systems mainly because it calls more often its SAT solver. For instance, for the problem "c090v03p00d02s02", *SAT called 30,179,676 times its SAT solver while InKreSAT called Minisat 15 times and Km2SAT called Minisat only once. Note that we also experimented *SAT with Minisat: it does not help. The issue is really the number of SAT calls, not the solver. The models returned by *SAT are surprisingly very small in number of worlds (compared to the other solvers), but we cannot verify them because we do not have all the information needed to assign a truth value to a given variable in a given world.

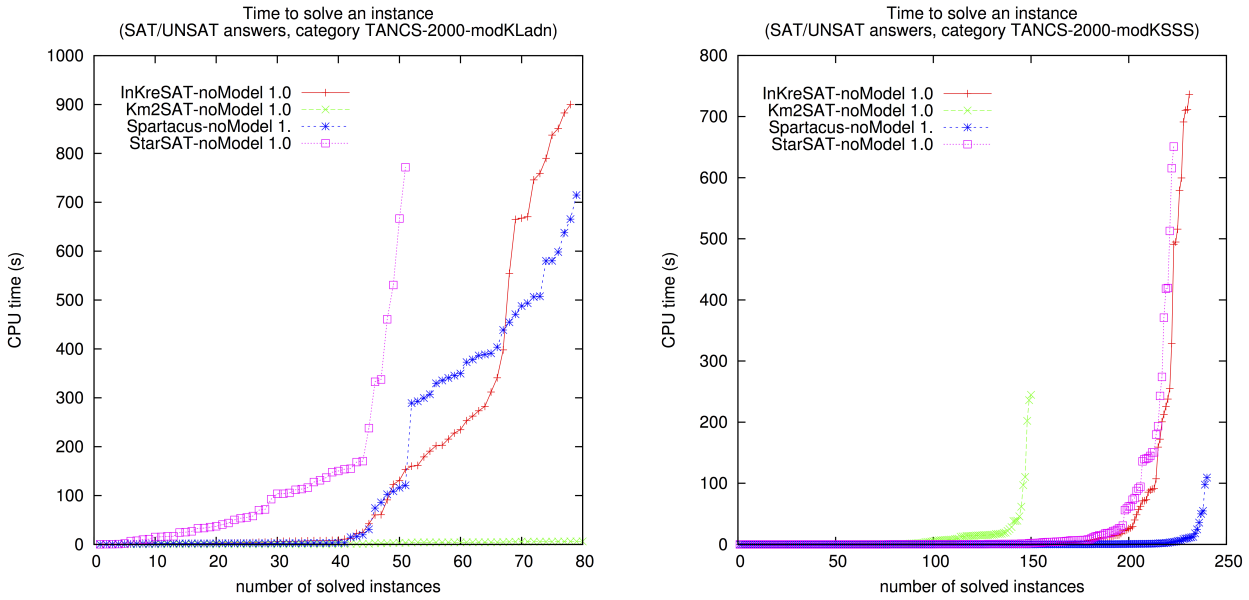
6.2 Randomly Generated modalized MQBF formulae

n,a	#	**Km2SAT**	***SAT**	**InKreSAT**	**Spartacus**	**Verified SAT**
4,4	40	MO	**40** [**18**]	**40** [**18**]	**40** [**18**]	11 / 18 (61.11%)
4,6	40	MO	**40** [**20**]	39 [19]	**40** [**20**]	9 / 20 (45.00%)
8,4	40	MO	**40** [**31**]	29 [22]	37 [28]	6 / 31 (19.35%)
8,6	40	MO	31 [30]	16 [16]	**34** [31]	0 / 33 (00.00%)
16,4	40	MO	26 [25]	17 [16]	**29** [**28**]	0 / 28 (00.00%)
16,6	40	MO	25 [25]	16 [16]	**29** [**29**]	0 / 29 (00.00%)
total	240	MO	202 (41)	157 (1)	**209** (173)	26 / 159 (16.35%)

Table 2: qbfMS

Randomly generated modalized MQBF formulae [22] are randomly generated QBF formulae translated to modal logic K. QBF with m clauses, alternation depth equal to a (the number of times that we changed the quantifier in the formula), with at most n variables per alternation. For each clause 4 different variables are randomly generated and each is negated with probability 0.5. The first and the third variables are existentially quantified, whereas the second and the fourth variables are universally quantified. A translation close to Schmidt-Schauß and Smolka's reduction of QBF validity into ALC satisfiability [28] is used. The different values of the parameters for this category of benchmarks are as follows: $m \in \{10, 20, 30, 40, 50\}$ denotes the number of clauses; $n \in \{4, 8, 16\}$ denotes the number of variables; $a \in \{4, 6\}$ denotes the alternation depth. For each triplet (a, n, m) we have 8 problems, so the whole family is composed of 240 problems. As already explained in the beginning of this section, there is a possibility for a solver to do a memory-out. This is what happened to Km2SAT in the whole category. Contrariwise to the previous benchmark families, very few satisfiable benchmarks from qbfMS could be verified. Those formulae contain basically 1 variable, a lot of operators (modal and Boolean) and a very big modal depth. The issue with such "toy-problems" is that it is very difficult to verify a model for it. For this purpose, we put a time-out on our checker of 300s. If we did not manage to verify the solution, we consider the SAT answers as unchecked. It happened only in this category, 103 times for Spartacus and 38 times for *SAT.

6.3 TANCS-2000 benchmarks for K Results



(a) Runtime distribution on modKLadn

(b) Runtime distribution on modKSSS

Figure 2: Runtime distribution for TANCS 2000

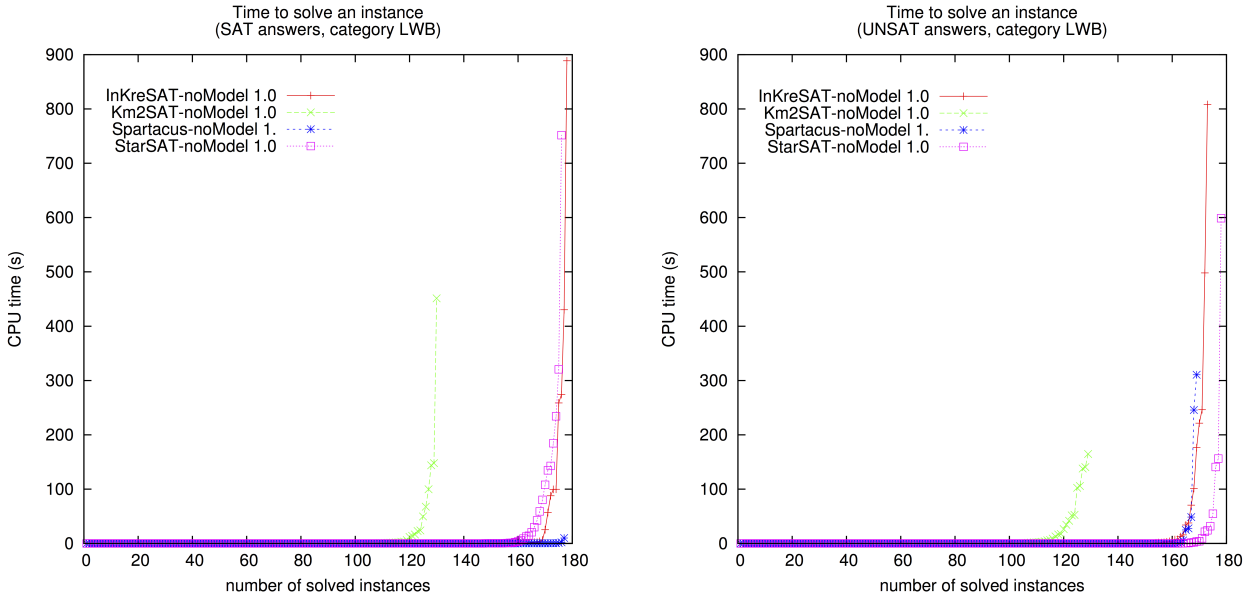
The TANCS-2000 Unbounded Modal QBF [23] benchmarks are generated the same way as the previous ones except that a single relation in modal logic K is replaced by a sequence of back-forth-back relations, in a clever way to avoid making most formulae unsatisfiable. These QBF formulae were originally converted to K using 3 different techniques of increasing hardness. We ran only the sub-categories modKSSS (Schmidt-Schauss-Smolka translation, easy) and modKLadn (Ladner translation, medium) because the third one (Halpern translation, the hardest) was not available in Spartacus benchmark archive (nor in the results presented in [13]).

n,a	#	**Km2SAT**	***SAT**	**InKreSAT**	**Spartacus**	**Verified SAT**
4,4	40	**40** [**17**]	**40** [**17**]	**40** [**17**]	**40** [**17**]	**17 / 17** (100%)
4,6	40	**40** [**25**]	**40** [**25**]	**40** [**25**]	**40** [**25**]	23 / 25 (92.00%)
8,4	40	**40** [**26**]	**40** [**26**]	**40** [**26**]	**40** [**26**]	**26 / 26** (100%)
8,6	40	14/MO [14]	38 [35]	37 [34]	**40** [**37**]	18 / 37 (48.64%)
16,4	40	8/MO [8]	33 [33]	36 [36]	**40** [**40**]	21 / 40 (52.50%)
16,6	40	8/MO [8]	32 [32]	38 [38]	**40** [**40**]	15 / 40 (37.50%)
total	240	150 (2)	223 (13)	231 (1)	**240** (226)	120/ 185 (64.86%)
4,4	40	**40** [**19**]	**40** [**19**]	**40** [**19**]	**40** [**19**]	**19 / 19** (100%)
4,6	40	**40** [**24**]	11 [6]	38 [22]	39 [23]	**24 / 24** (100%)
total	80	**80** (80)	51 (0)	78 (0)	79 (0)	**43 / 43** (100%)

Table 3: Upper: modKSSS — Lower: modKLadn

Spartacus outperforms the other solvers on modKSSS benchmarks. The bigger the values of n and a, the more difficult it is to verify the model provided by the solver. It is interesting to note that on the modKLadn benchmarks, Km2SAT performs much better than the other solvers, which can be seen in the runtime distribution of the solvers in Figure 2a. Here, the approach benefits from the advances on SAT solvers. Note that for modKSSS, Km2SAT simply cannot generate the CNF within our memory limit when it does not solve those benchmarks.

6.4 Tableaux'98 Results



(a) Runtime distribution on *_n

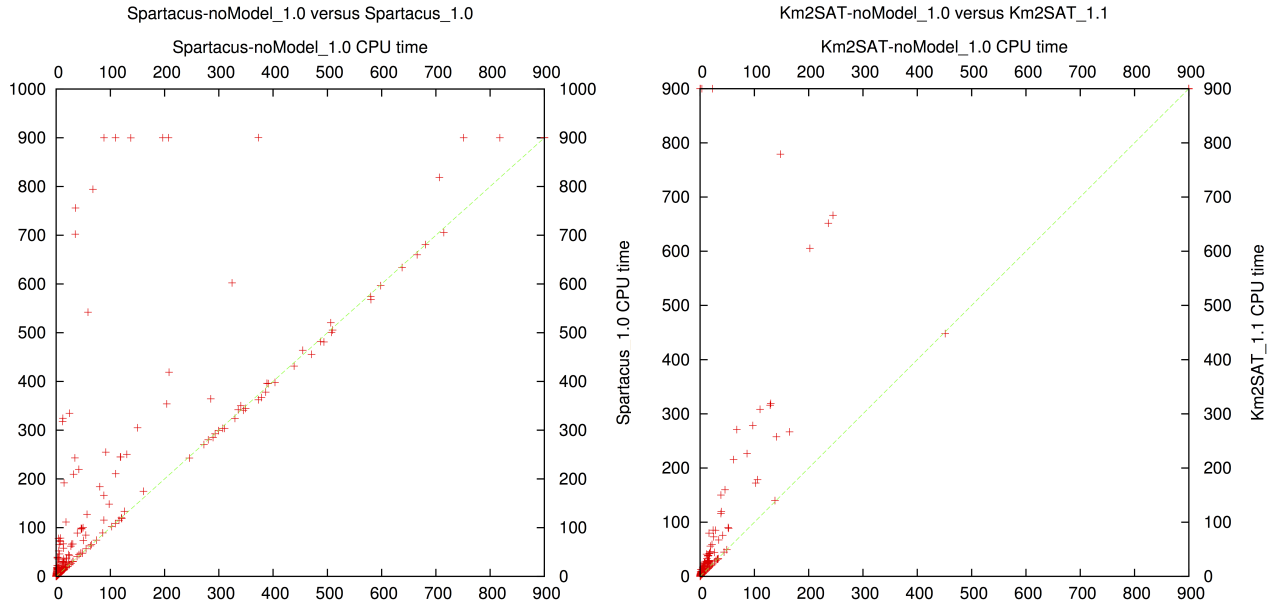
(b) Runtime distribution on *_p

Figure 3: Runtime distribution for Tableaux'98

name	#	**Km2SAT**	***SAT**	**InKreSAT**	**Spartacus**	**Verified SAT**
branch_n	17	5/MO	**12**	11	9	5 / 12 (41.66%)
dump_n	21	**21**	**21**	**21**	**21**	**21** / **21** (100.0%)
grz_n	21	**21**	**21**	**21**	**21**	**21** / **21** (100.0%)
d4_n	21	6/MO	**21**	**21**	**21**	19 / 21 (90.47%)
lin_n	21	**21**	**21**	**21**	**21**	**21** / **21** (100.0%)
path_n	21	8/MO	**21**	20	**21**	**21** / **21** (100.0%)
t4p_n	21	6/MO	**21**	**21**	**21**	**21** / **21** (100.0%)
ph_n	21	**21**	*17*	**21**	**21**	**21** / **21** (100.0%)
poly_n	21	**21**	**21**	**21**	**21**	**21** / **21** (100.0%)
branch_p	21	5/MO	**21**	20	13	
dump_p	21	20/MO	**21**	**21**	**21**	
grz_p	21	**21**	**21**	**21**	**21**	
d4_p	21	11/MO	**21**	**21**	**21**	
lin_p	21	**21**	**21**	**21**	**21**	
path_p	21	9/MO	**21**	17	**21**	
ph_p	21	**10**	**10**	**10**	9	
poly_p	21	**21**	**21**	**21**	**21**	
t4p_p	21	11/MO	**21**	**21**	**21**	
total	374	259 (2)	**354** (281)	351 (6)	346 (53)	171 / 184 (92.93%)

Table 4: Tableaux'98 Benchmarks for K

We have also a set of the Tableaux'98 benchmarks suite for K [3, 4]. The precise definition of how the formulae are created is available in [14]. It is important to notice that all the sub-categories finishing with "_p" are sub-categories of 21 UNSAT problems. Because we are unable to certify UNSAT answers, we display and count only the verification on SAT answers in this category. On those benchmarks, *SAT performs slightly better than the other solvers overall despite not performing well compared to the other solvers for the benchmarks "ph_n". After a detailed analysis of the solver on those benchmarks, we discovered that the clauses generated by *SAT were too long for SATO (limited to 256 literals). Note also that the "ph_p" benchmarks correspond to



(a) Spartacus with vs without model output

(b) Km2SAT with vs without model output

Figure 4: overhead of our modifications

hard combinatorial UNSAT Pigeon-Hole problems [31], which explains why they are difficult for all solvers. And finally, all the "_n" are almost fully Verified SAT except "branch_n". Let take a closer look to the Kripke model returned for this category. The formula contains sub-formulae looking like $(\Diamond(x \wedge y \wedge z) \wedge \Diamond(x \wedge y \wedge \neg z))$ and the accessibility relation between worlds must be a tree. Such formula requires 2 accessible worlds: one satisfying z and another satisfying $\neg z$. However Spartacus only provides one of them in its open tableau. The same kind of incomplete models are returned by *SAT and InKreSAT. Only Km2SAT manages to provide complete Kripke models on this category, but unfortunately, Km2SAT solves only 5/17 problems, so we have only 5/17 verified solutions.

6.5 On the importance of the SAT solvers

3 out of 4 solvers evaluated here are SAT-based. InKreSAT is tightly coupled with Minisat 2.2.0 [8], it is as such very difficult for us to use another backend SAT solver. As such, we used Minisat 2.2.0 with Km2SAT solver as well. There are nowadays better SAT solvers (Glucose, Lingeling). Does it make any difference in our context? After some experimentation, we could only obtain a small speed gain by using Glucose instead of Minisat with Km2SAT. This gain is obtained in the category 3CNF where Glucose is much faster than Minisat. It is worth noting that 754 out of the 766 times Km2SAT does not answer, it is because the CNF cannot be generated with the available amount of memory, not because the CNF cannot be solved. We also tried to plug Minisat 2.2.0 in *SAT instead of SATO. Indeed, *SAT is very efficient despite being coupled to a SAT solver created in the nineties. One would expect that it would get even better with a more recent SAT solver like Minisat or Glucose. Unfortunately, the developers of *SAT made the code deeply linked to SATO for efficiency reasons (they share the same data structures to store the CNF for instance). While it is possible to plug any recent SAT solver using a file based approach, it is really inefficient compared to the tight integration with SATO. Note that the CNF produced by *SAT are in many cases quite simple to solve, and do not require a much sophisticated SAT solver. One interesting research direction would be to design a *SAT like algorithm taking into account the current incremental capabilities of SAT solvers.

6.6 Overhead of model production

We had to modify the solvers not only to produce the Kripke model but also to bookkeep information to be able to produce such model. As such, the performance of the solver may be affected by those modifications. We compared the runtime of the original solvers against the modified versions, to evaluate the overhead induced by our modifications. The difference in runtime may be important, as in Figure 4a, representing the runtime of

Spartacus in its original version (no-model) versus the version providing a model. All point above the diagonal denote benchmarks for which the modified version of Spartacus takes longer than the original version. All the points at y=900 correspond to benchmarks that the original solver can solve (answer SAT) but for which the modified solver cannot produce the Kripke model within the timeout. The difference can be quite significant for Spartacus because we rely on an existing debug ouput in the solver (i.e. non optimized) to generate the certificate. There is certainly room for improvement here.

6.7 Verification of SAT answers per solver

We were able to check globally 67% of satisfiable problems for which at least one solver provided a Kripke model. However, while looking at Table 5, two solvers are the main providers of those verified answers.

	#SAT	#Verified	Uniquely Verified
Km2SAT	330	168	39/476
***SAT**	584	6	0/476
InKreSAT	573	26	0/476
Spartacus	696	443	352/476
Global	708	476	391/476

Table 5: Statistics about the verification

Spartacus provided the most important number of certificates both because it is quite efficient and because it was designed to provide an open tableau model for debugging purposes, which is often sufficient to build a Kripke model. Km2SAT provides the remaining certificates because we simply have to interpret the answer provided by the SAT solver. While such model is not always a model of the original formula but of a simplified one, in practice it works in half of the cases.

6.8 Challenging models

The biggest Kripke model that we manage to verify was a model returned by Km2SAT on the instance modKSSS-C20-V8-D6.7. The Kripke model contained 10,618,391 worlds and 10,618,390 edges for 56 vars. It could be verified in 23,68s. Behind this Kripke model, Km2SAT generated a SAT formula with 108,700,724 variables and 122,697,799 clauses. It took 273.620 seconds to generate this file of 3 GB plus the mapping file of 1 GB bytes. Then this SAT problem was solved in 87.049 seconds. Then, it took 291 seconds to parse the SAT solution and the mapping to finally generate this Kripke model. The smallest Kripke model that we did not manage to verify under 300s was a model returned by *SAT on the instance modKSSS-K4-C10-V16-D4.4. The Kripke model contained 81 worlds and 2792 edges for 80 variables. This model is shown in Figure 5.

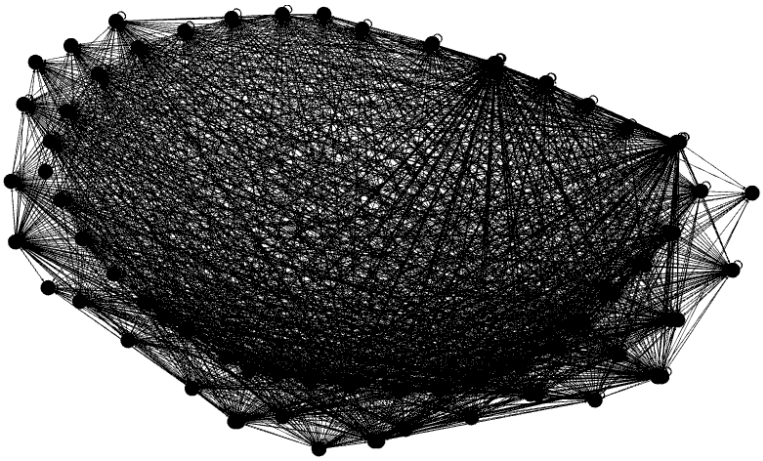


Figure 5: Unverified model from *SAT

7 Conclusion and perspective

In this paper, we studied the feasibility to verify in practice Kripke models returned by a selection of existing solvers on a selection of existing benchmarks for modal logic K. For that purpose, we designed the Flat Kripke Model format to represent those models and built an independent model checker for modal logic K. We modified several solvers for modal logic K to provide such models and ran them on a wide range of existing benchmarks. We have been able to verify 67 percent of all the satisfiable benchmarks, which demonstrates that in practice, we can check the majority of SAT answers on current benchmarks. Our results also showed that it is a non obvious task to modify existing solvers to output a Kripke model if the solver was not designed for that in the first place: by providing a specific textual format and a checker to the community, we would like to encourage solver designers to provide the ability to produce models in that format. We plan to increase the number of checkable SAT answers by both improving the output of the models in efficient solvers such as Spartacus and Km2SAT, and improving the checker itself. A next logical step would be to study the feasibility to validate UNSAT answers, in the spirit of what is already done for SAT, based on the previous work on UNSAT proofs in the modal logic K [9].

Acknowledgements

The authors thank Olivier Roussel for providing support for running the experiments. Part of this work was supported by the French Ministry for Higher Education and Research and the Nord-Pas de Calais Regional Council through the "Contrat de Plan Etat Région (CPER)" and by an EC FEDER grant.

References

[1] C. Areces and M. de Rijke. Computational modal logic. 2003.

[2] A. Balint, A. Belov, M. Järvisalo, and C. Sinz. Overview and analysis of the SAT Challenge 2012 solver competition. *Artif. Intell.*, 223:120–155, 2015.

[3] P. Balsiger and A. Heuerding. Comparison of theorem provers for modal logics : introduction and summary. In *Autom. Reasoning with Analytic Tableaux and Related Methods*, pages 25–26. Springer, 1998.

[4] P. Balsiger, A. Heuerding, and S. Schwendimann. A Benchmark Method for the Propositional Modal Logics K, KT, S4. *J. Autom. Reasoning*, 24(3):297–317, 2000.

[5] D. Challenge. Satisfiability: Suggested Format. *DIMACS Challenge. DIMACS*, 1993.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[7] L. A. Dennis, M. Fisher, N. Lincoln, A. Lisitsa, and S. M. Veres. Practical Verification of Decision-Making in Agent-Based Autonomous Systems. *CoRR*, abs/1310.2431, 2013.

[8] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Selected Revised Papers*, pages 502–518, 2003.

[9] P. Enjalbert and L. F. del Cerro. Modal Resolution in Clausal Form. *Theor. Comput. Sci.*, 65(1):1–33, 1989.

[10] O. Gasquet, A. Herzig, B. Said, and F. Schwarzentruber. *Kripke's Worlds - An Introduction to Modal Logics via Tableaux*. Studies in Universal Logic. Birkhäuser, 2014.

[11] E. Giunchiglia and A. Tacchella. System description: *SAT: A platform for the development of modal decision procedures. In *Automated Deduction - CADE-17 Proceedings*, pages 291–296, 2000.

[12] E. Giunchiglia, A. Tacchella, and F. Giunchiglia. SAT-Based Decision Procedures for Classical Modal Logics. *J. Automated Reasoning*, 28(2):143–171, 2002.

[13] D. Götzmann, M. Kaminski, and G. Smolka. Spartacus: A Tableau Prover for Hybrid Logic. *ENTCS*, 262:127–139, 2010.

[14] J. Y. Halpern and Y. Moses. A Guide to Completeness and Complexity for Modal Logics of Knowledge and Belief. *Artificial Intelligence*, 54(2):319–379, 1992.

[15] A. Heuerding, G. Jäger, S. Schwendimann, and M. Seyfried. Propositional logics on the computer. In *Theorem Proving with Analytic Tableaux and Related Methods*, pages 310–323. Springer, 1995.

[16] M. Heule, W. A. H. Jr., and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD 2013.*, pages 181–188, 2013.

[17] G. Jaeger, P. Balsiger, A. Heuerding, and S. Schwendiman. LWB 1.1 Manual. *Unpublished*, 1997.

[18] M. Järvisalo, D. L. Berre, O. Roussel, and L. Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1), 2012.

[19] M. Kaminski and T. Tebbi. InKreSAT: Modal Reasoning via Incremental Reduction to SAT. In *Automated Deduction - CADE-24 Proceedings*, pages 436–442, 2013.

[20] S. A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[21] R. E. Ladner. The computational complexity of provability in systems of modal propositional logic. *SIAM Journal of Computing*, 1977.

[22] F. Massacci. Design and Results of the Tableaux-99 Non-classical (Modal) Systems Comparison. In *Autom. Reasoning, International Conf., '99 Proceedings*, pages 14–18, 1999.

[23] F. Massacci and F. M. Donini. Design and Results of TANCS-2000 Non-classical (Modal) Systems Comparison. In *Autom. Reasoning., International Conf., 2000 Proceedings*, pages 52–56, 2000.

[24] M. Narizzano, C. Peschiera, L. Pulina, and A. Tacchella. Evaluating and certifying QBFs: A comparison of state-of-the-art tools. *AI Communications*, 22(4):191–210, 2009.

[25] P. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification from the KRSS group of the ARPA knowledge sharing effort, 1993.

[26] O. Roussel. Controlling a solver execution with the runsolver tool system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

[27] A. Saffidine. Minimal proof search for modal logic K model checking. *CoRR*, abs/1207.1832, 2012.

[28] M. Schmidt-Schauß and G. Smolka. Attributive Concept Descriptions with Complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[29] R. Sebastiani and M. Vescovi. Automated reasoning in modal and description logics via SAT encoding: the case study of k(m)/alc-satisfiability. *J. Artif. Intell. Res. (JAIR)*, 35:343–389, 2009.

[30] L. Simon, D. Berre, and E. A. Hirsch. The SAT2002 competition. *Annals of Mathematics and Artificial Intelligence*, 43(1):307–342, 2004.

[31] W. A. Trybulec. Pigeon hole principle. *JFM*, 2(199):0, 1990.

[32] D. Tsarkov and I. Horrocks. FACT++ Description Logic Reasoner: System Description. In *IJCAR 2006 Proceedings*, pages 292–297, 2006.

[33] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass Version 2.0. In *Aut. Deduction CADE18*, pages 275–279. Springer, 2002.

[34] H. Zhang. SATO: An Efficient Propositional Prover. In *Automated Deduction - CADE-14 Proceedings*, pages 272–275, 1997.