Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Just-In-Time Compilation of Knowledge Bases

Gilles Audemard*
Univ. Lille-Nord de France
CRIL/CNRS UMR8188
Lens, F-62307
audemard@cril.fr

Jean-Marie Lagniez
Institute for Formal Models
and Verification Johannes Kepler
University, AT-4040 Linz, Austria
jean-marie.lagniez@jku.at

Laurent Simon
Univ. Paris-Sud
LRI/CNRS UMR 8623
INRIA Saclay
Orsay, F-91405
simon@lri.fr

Abstract

Since the first principles of Knowledge Compilation (KC), most of the work has been focused in finding a good compilation target language in terms of compromises between compactness and expressiveness. The central idea remains unchanged in the last fifteen years: an off-line, very hard, stage, allows to "compile" the initial theory in order to guarantee (theoretically) an efficient on-line stage, on a set of predefined queries and operations. We propose a new "Just-in-Time" approach for KC. Here, any Knowledge Base (KB) will be immediately available for queries, and the effort spent on past queries will be partly amortized for future ones. To guarantee efficient answers, we rely on the tremendous progresses made in the practical solving of SAT and incremental SAT applicative problems. Even if each query may be theoretically hard, we show that our approach outperforms previous KC approaches on the set of classical problems used in the field, and allows to handle problems that are out of the scope of current approaches.

1 Introduction

In Knowledge Compilation (KC) [Selman and Kautz, 1996], the Knowledge Base (KB) is encoded in a particular language (*i.e.*, propositional logic) and then "compiled" into another one, in an *off-line* phase. This target language is supposed to accept more efficient (theoretically guaranteed) queries. In [Darwiche and Marquis, 2001; 2002], it was proposed to study algorithmic abstractions of classical needs for KC. A set of typical queries and operations on KB was proposed and studied in terms of the existence, or not, of underlying polynomial/non polynomial algorithms over the target language. This foundation work was deep and particularly influential. Most of the new approaches remain in the same framework and focus on discovering new target languages that could take a good place in it [Subbarayan *et al.*, 2007; Fargier and Marquis, 2009; Darwiche, 2011]. Over all the important progresses observed in Knowledge Reasoning since 2001, the practical solving of industrial SAT problems, with the introduction of so-called modern SAT Solvers [Moskewicz *et al.*, 2001; Eén and Sörensson, 2003a], is probably one of those having the wider impact in the field. SAT solvers are making progresses every year and new practical applications of their efficiency in other fields (like planning, BMC, Optimization, ...) give state of the art performances. An unexpected and surprising use of these solvers is their ability to be considered as NP-Complete oracles, and embedded as black boxes in problems above NP [Bradley, 2012], supporting hundreds or even thousands calls in one run. This is called *incremental* SAT solving: a SAT solver is typically run many times on a set of very close instances (a few clauses can be added or removed between calls).

We propose in this paper a new approach for Knowledge Compilation, made possible by the progresses reported above. One of the key point of our approach is based on the powerful capability of SAT Solvers to learn important clauses, and to reuse them for future queries thanks to incremental SAT solving. In our approach, called "Just-in-Time" Compilation, there are no off-line/on-line phases. Knowledge Bases are immediately operational, and users can query/manipulate them immediately. This more reactive framework will also allow us to add new queries on KB, like clauses removal or explanation-based answers, which are not yet taken into account in previous Compilation languages. Of course, queries and operations can no more be guaranteed to be polynomial. However, we will rely on "practical" guarantee of efficiency and we will show how previous queries/operations will be amortized, up to some extend, by speeding-up future queries. As we report in the experimental section, in most of the cases, queries are solved in zero seconds (three digits precision) on a large family of previous problems used in the field. We will base our approach on `glucose` [Audemard and Simon, 2009] a `minisat`-based [Eén and Sörensson, 2003a] awarded and publicly available SAT Solver. This solver was chosen for its ability to manage *interesting* clauses. If we want to reuse past learnt clauses, then this seems to be a crucial choice.

2 Knowledge Base Compilation

In a time when many works were focusing on BDD representations and Prime Implicants/Implicates generations, it

*This work has been partially supported by FWF, NFN Grant S11408-N23 (RiSE), an EC FEDER grant, by CNRS and OSEO, under the ISI project "Pajero".

was proposed in [Darwiche and Marquis, 2001; 2002] to study the hardness (Polynomial or not) of most important queries/operations to ask a KB for. This idea was already underlying the work of [Del Val, 1994], who proposed to focus on Unit-Complete compilation. In [Darwiche and Marquis, 2002], however, this idea was pushed forward and a number of propositional fragments were studied w.r.t their succinctness and their theoretical efficiency on a set of typical queries/operations that could be mandatory for typical A.I. problems. They studied a number of NNF (Negation Normal Form) descendants (*i.e.* DNNF (Decomposable NNF), CNF, DNF, d-DNNF (determinist DNNF), ROBDD (Binary Decision Diagrams), SDD (Sentential Decision Diagrams, [Darwiche, 2011], ...) and draw a very useful map of their respective succinctness, but also the set of polynomial queries/operations each of them allows. We can notice here that direct (and incomparable) descendants of NNF were DNNF, CNF and DNF. Moreover, given the lack of polynomial queries/operations guaranteed on the CNF fragment, it has mostly been left apart in studies until now.

Let us recall here the main queries and operations proposed in the above papers. Given I_1 and I_2 instances of a compilation form F, c a clause and t a term, the following queries were introduced in [Darwiche and Marquis, 2002]. *Consistency* (**CO**: $I_1 \models \bot$?), *Validity* (**VA**: $I_1 \models \top$?), *Clausal Entailment* (**CE**: $I_1 \models c$?), *Implicant* (**IM**: $t \models I_1$?), *Equivalence* (**EQ**: $I_1 \equiv I_2$?), *Sentential Entailment* (**SE**: $I_1 \models I_2$?), *Model Counting* (**CT**: counting the number of models of I_1), *Model Enumeration* (**ME**: enumerating the models of I_1). It is important to notice that **ME** is supported by the compilation language F if each successive model can be printed in polynomial time (we do not require the language to print all the models in polynomial time, of course).

They also introduced typical KB operations, that take a set (Bounded or not) of KBs in a given language and apply one of the following operations on them: *Conditioning* (**CD**: let t be a term, $\Sigma_{|t}$ is the formula obtained by replacing each variable x of Σ by $\top$ (resp. $\bot$) if x (resp. $\neg x$) is a positive (resp. negative) literal of t), *Forgetting* (**FO**: let X be a subset of variables of Σ, $\exists X.\Sigma$ is the formula s.t., for any formula α that do not mention any variable from X, we have $\Sigma \models \alpha$ exactly when $\exists X.\Sigma \models \alpha$), *Conjunction* ($\wedge$**C**), *Bounded Conjunction* ($\wedge$**BC**:), *Disjunction* ($\vee$**C**), *Bounded Disjunction* ($\vee$**BC**), and *Negation* ($\neg$**C**).

A first couple of intriguing results were published in [Subbarayan *et al.*, 2007] and then [Fargier and Marquis, 2009]. These two papers reported theoretical and experimental opposite studies of tree-of-BDD, a propositional fragment of choice for KC. ToBDD-based compilations was experimentally showing good compilation/query times on the classical set of benchmarks used so far. However, it was shown that these queries were NP-Hard.

3 Just-in-Time Compilation by Incremental SAT Solving

3.1 Limits of KC approaches

Despite its fundamental importance in Knowledge Representation, KC has some limits and drawbacks. First of all, the off-line/on-line paradigm allows only a limited reactive use knowledge bases, i.e. remove/deactivate any initial knowledge once it has been compiled, or backtracking on any **FO** operation performed before is hard to provide (revision is indeed performed by using adequate operations on the compiled KB, but it has only been considered theoretically, to the best of our knowledge, and simple clause removal, for instance, is not easily taken into account). This may be crucial, for instance, during the KB construction, where distinct users may want to measure the impact of forgetting variables, or removing a partial set of initial clauses. KC does not address problems where a number of users have distinct KBs in conjunction with a global, common, KB. Explanation-based reasoning is also another application that is not yet taken into account. The relative independence between the initial language of the KB and the compiled language does not immediately allow this. Moreover, the theoretical view of queries/operations complexity may have only a limited impact on practical performances obtained on real-world problems. For instance, even a quadratic complexity may fail in practice to handle instances of million of variables or on huge compiled theories. It is thus important to check the proposed approaches against challenging problems to complete the global picture of KC.

3.2 Incremental SAT Solving

So-called "modern SAT solvers" are based on the *conflict driven clause learning* paradigm (CDCL) [Moskewicz *et al.*, 2001]. These solvers were initially introduced as an extension of the DPLL algorithm [Davis and Putnam, 1960; Davis *et al.*, 1962], with a powerful conflict clause learning [Silva and Sakallah, 1996; Zhang *et al.*, 2001] scheme. Nowadays, it is acknowledged that they must be described as a mix between backtrack search and plain resolution engines. They integrate a number of effective techniques including clause learning, highly dynamic variable ordering heuristic [Moskewicz *et al.*, 2001], polarity heuristic [Pipatsrisawat and Darwiche, 2007], clause deletion strategy [Goldberg and Novikov, 2002; Audemard and Simon, 2009] and search restarts [Huang, 2007] (see [Marques-Silva *et al.*, 2009] for a detailed description of CDCL solvers).

It is well known that CDCL solvers are particularly well suited for solving huge industrial problems (see `www.satcompetition.org`). However, a new and surprising use of these solvers has emerged in the last years. It is called *incremental SAT Solving* [Nadel and Ryvchin, 2012]. The main goal of the incremental SAT solving is to be able to reuse as much informations as possible between successive SAT calls. For instance, the final state of variable ordering, polarity cache, and clause deletion/restarts strategies can be easily saved for the next call. However, it is clear that, for learned clauses, it is important to be able to detect which clauses were obtained thanks to which initial clause, to be able to reuse only sound informations. This can be made possible by simply working with *assumptions*. Assumptions in SAT solvers were already proposed in early versions of `Minisat` [Eén and Sörensson, 2003a] and are daily used in many distinct applications (*s.t.* bounded model checking [Eén and Sörensson, 2003b], extraction of a minimal unsatisfiable

set [Nadel, 2010] ...). An assumption $\mathcal{A}$ is defined as a set of literals $\{a_1, a_2, \ldots, a_n\}$ considered as unit clauses by the solver but not explicitly added to the formula Σ (there is no simplification of the formula, the underlying **CD** operation is only performed by branching first on these literals). Therefore, the SAT solver (called now by $solve(\Sigma, \mathcal{A})$) tries to find an interpretation that satisfies all the clauses, as well as the unit literals a_i. Additionally, if there is no such an interpretation, the set of assumptions used in the final UNSAT proof can be returned as an explanation, and can possibly be minimized with an additional effort.

As an illustration, let us focus the extraction of a *minimal unsatisfiable set* (MUS) [Oh *et al.*, 2004]. In this approach, a unique assumption is added to each clause (positively) in order to activate or not the clause in the database. Then, the SAT solver is incrementally called with most of these additional literals, called *selectors* here, forced to false. When a sub-problem is checked for consistency, all learnt clauses will contain additional selector variables representing the set of initial clauses needed to derive them. The solver can remove any initial clause, and any of its derived learnt clauses, by simply assigning its selector to true. Furthermore, when the formula is UNSAT, the final conflict can be analysed to produce a sufficient set of clauses that imply inconsistency.

3.3 Principles of Just-in-Time compilation

The progresses observed in incremental SAT solving, associated with their powerful learning mechanisms, allow to propose a new paradigm for KC. Queries are NP-Hard, but experimentally guaranteed to be efficient enough by a powerful SAT solver. The "Just-in-Time" part of the compilation will rely on reusing as much previous learnt clauses as possible. This is essential for amortizing the cost of past queries, as we will see in section 5.

Keeping KBs in CNF

The propositional fragment we use is simply CNF, enriched by learnt clauses and a set of judicious selectors and assumption variables. More precisely, Σ is split in two sets, *Hard* clauses Σ_H and *Soft* clauses Σ_S. A unique selector variable is added to each soft clause. Hard clauses are supposed to remain in the KB in any case, and Soft clauses can be removed on the fly. Explanations will also be given in terms of Soft clauses only. Given the mechanisms of CDCL, additional clauses (Hard or Soft) can be added afterwards without any special care (SAT solvers supports the addition/removal of any variable, including a possible new selector to be added to a new Soft clause).

Querying while ensuring amortization

All the queries are built on top of incremental SAT calls. For instance, **CE** of c can be answered by a call to $solve(\Sigma \wedge \neg c)$. However, if we want to amortize it, we need to add the negated literals of c as assumptions. Thus, any learnt clause during this call can be kept or removed (when the solver will trigger a clauses database cleaning). **CE** will thus be a simple call to $solve(\Sigma, \{l | l \in \neg c\})$, the second argument of which is the set of additional assumptions to consider for this call.

Additionally (this is classical when working with assumptions), the $solve$ function can return the set of assumption literals associated with the inconsistency, if it was returned. This gives the user the opportunity to consider which part of the KB is sufficient to imply the given clause. It is even possible to minimize this set, by considering the same algorithms used in Minimal UNSAT Core extraction [Oh *et al.*, 2004]. We did not adress this minimization problem here, but, depending on the queries, it should be easy to adapt to our problem (also based on multiple incremental SAT solving calls).

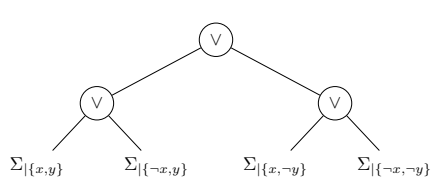
Based on the experimental results of sec. 5, our approach can be considered as supporting **CO**, **VA**, **CE**, **IM**, **SE** (depending on the sentence length), and **ME**. The following queries seems to be harder to fulfill: **EQ** and **CT**. However, we will give an algorithm for **CT** that gives good results on problems with few models, and which is anytime. **CT** is a very difficult task, dominating all the other queries we could ask a KB for. To ensure a good query time on it, decomposition-based methods remain the state of the art.

Implicit operations on KBs

Our main idea for considering operations is to preserve the initial KB by handling them implicitly only. Operations, when possible, will be taken into account during query time only. For instance, performing a Conditioning **CD** is a very easy task for an incremental SAT Solver. The principle here is simply to have a set of assigned variables to branch on before any other variable during any call to $solve()$. For this, we need to use, once again, a special set of assumptions, but for conditioned literals only.

The Forgetting operation **FO** is however more difficult to handle. On CNF formulas, removing a variable is usually performed by resolution. It is the reasoning principle underlying the first SAT algorithm [Davis and Putnam, 1960] (and also in [Dechter and Rish, 1994]). However, this operation is in general very hard and may quickly produce intractable KBs sizes. Here, the trick is to handle it intentionally, by maintaining in the SAT solver the set $\mathcal{F}$ of "forgotten" variables by the user. Intuitively, each time a model or an implicant will be found, the forgotten variables will be removed on the fly. More precisely, let us define $\Sigma' \equiv \mathbf{FO}(\Sigma, \mathcal{F})$ as the result of forgetting all the variables of $\mathcal{F}$ in Σ. As defined in [Lang *et al.*, 2003] $\mathbf{FO}(\Sigma, \ell) \equiv \Sigma_{|\ell} \vee \Sigma_{|\neg\ell}$ and $\mathbf{FO}(\Sigma, \mathcal{F} \cup \{\ell\}) \equiv \mathbf{FO}(\Sigma_{|\ell}, \mathcal{F}) \vee \mathbf{FO}(\Sigma_{|\neg\ell}, \mathcal{F})$. This recursive definition can be captured by a *forgetting tree* where the models of $\mathbf{FO}(\Sigma, \mathcal{F})$ are the disjunction of the models of formulas at its leaves. Each path of this tree is a complete assignment of all the variables of $\mathcal{F}$, as illustrated below.

Example 1 *Let us consider a CNF formula Σ and $\mathcal{F} = \{x, y\}$ the set of forgetting variables. One possible forgetting tree for FO$(\Sigma, \mathcal{F})$ can be:*



Given this tree, it is easy to check that intentional forgetting does not interfere **CO** (if we cannot find a model for Σ, then Σ' has no model too), **VA** (if all the models are satisfied for Σ, their projection on Σ' are also satisfying Σ'), **CE** (if $\Sigma \models c$ then $\exists x.\Sigma \models c$, if $x \notin c$). For **IM**, we can notice that, if $t \models \Sigma$, then $t \models \Sigma|_x$ and $t \models \Sigma_{\neg x}$, and thus $t \models \exists x.\Sigma$. However, the other way is not as direct. If we want to check that $t \models \exists x.\Sigma$, we need to extend t to t' s.t. $t' \models \Sigma$. For **ME**, it is easy to filter out literals from $\mathcal{F}$ while printing models. However, model counting (**CT**) is more tricky (see algorithm 2). Given the fact that **CE** is easily supported, we will propose to check all clauses during **SE**. At last, we do not claim to efficiently support **EQ** even if we think that our SAT-based solution could give some very interesting results.

Now, if it is clear that $\wedge$**C** and $\wedge$**BC** are supported, we cannot yet provide interesting solutions for $\vee$**C** and $\vee$**BC**. Also, $\neg$**C** must not be considered as supported.

4 Incremental SAT-Querying Algorithms

Before introducing the different queries of our framework, let us recall some notations. $\Sigma = \Sigma_H \wedge \Sigma_S$ is the input CNF formula . $\mathcal{A}$ is the set of assumptions containing both conditioned variables and selectors used to activate or not soft clauses. $\mathcal{F}$ is the set of forgotten variables. Then, queries are performed on the resulting formula, *i.e.* $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$ instead of Σ.

4.1 Checking Entailment

Clause Entailment (CE) and Sentential Entailment (SE)

CE can simply be encoded by the following call to a SAT solver: $solve(\Sigma, \{\ell | \neg\ell \in c\} \cup \mathcal{A})$, where c is the clause to check and $\mathcal{A}$ the set of current assumptions. When the answer is UNSAT (c is implied by $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$), the call can return the set of literals from $\mathcal{A}$ that were used for deriving the inconsistency.

It is not unusual to know in advance the sequence of queries to be asked sequentially. A possible extension to our **CE** algorithm could be to use any model found for a query i (when c_i is not implied) to remove pending queries built on a clause c_j falsified by this interpretation.

We propose a simple approach to **SE**, based on the previous **CE** algorithm. This approach tests each clause of Φ, one by one, in order to know if $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F}) \models \Phi$. This approach looks particularly naive but, as it is shown in the experimental section, on all the current available KC problems, our approach will be able to check many **CE**s in a few seconds, which makes this approach a surprisingly very competitive practical solution.

Equivalence (EQ)

For the equivalence checking (**EQ**) it is easy to first check that both formulas are built on the same set of variables, then to test the implication in both direction ($\Sigma \equiv \Psi$ iff $\Sigma \models \Psi$ and $\Psi \models \Sigma$). Notice that here, the incremental SAT solving is only available in one direction (*e.g.* if Σ is the formula compiled just in time so far, then to know if $\Phi \models \Sigma$ we must start a new incremental SAT solving).

4.2 Checking Models

In this section, one of the main difficulties is to handle the implicit **FO** operation and to work under assumptions.

Implicant (IM)

A term t is an implicant of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$ if it can be extended to a model for $\Sigma_{|\mathcal{A}}$ over the variables of $\mathcal{F}$, *i.e.* we can find an assignment $\mathcal{I}$ of variables from $\mathcal{F}$ such that $t \wedge \mathcal{I} \models \Sigma_{|\mathcal{A}}$. $\mathcal{I}$ can be viewed as a branch in the forgetting tree defined earlier. This principle allows us to propose Algorithm 1, which first checks (line 1) that t is not contradictory with current assumptions, then (lines 3-4), builds the formula Σ' by removing from Σ all the clauses already satisfied by t. The SAT call (line 5) allows to check that t can indeed be extended to a model over the variables of $\mathcal{F}$.

Algorithm 1: Implicant Checking (**IM**)

Input: Σ: CNF, $\mathcal{A}$: assumpt., $\mathcal{F}$: forgotten var, t: term
Output: true if $t \models \mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$, false otherwise

1 **if** $\exists \ell \in t$ *s.t.* $\neg\ell \in \mathcal{A}$ **then return** false;
2 $\Sigma' \leftarrow \emptyset$;
3 **foreach** $c \in \Sigma_{|\mathcal{A}}$ **do**
4 **if** $t \cap c = \emptyset$ **then** $\Sigma' \leftarrow \Sigma' \cup \{\bigvee_{\ell \in c | var(\ell) \in \mathcal{F}} \ell\}$;
5 **return** $solve(\Sigma') \neq \emptyset$;

Models Counting (CT) and Models Enumeration (ME)

Model counting is a very hard task dominated by decomposition methods. We could have embedded state of the art CNF model counters [Sang *et al.*, 2004], but we rather propose here to study how far a simple incremental SAT based counting method could go. The idea is to search a model, to reduce it to a smaller implicant by removing unuseful literals and then add its negation to the database, in order to prevent any of the models it covers to be counted twice ($reduce(\Sigma, \mathcal{I})$ reduces the implicant $\mathcal{I}$ on Σ by returning a prime implicant that can be extended to $\mathcal{I}$). Once again, their is a small difficulty when considering forgotten variables.

Our method is based on the enumeration of reduced implicants, that only contain necessary assignments. Each time a reduced implicant $\mathcal{I}'$ is found, a "blocking clause" is added to Ω in order to prevent any model covered by it to be found or counted again. This technique prevents our reducing mechanism to produce prime implicants of the initial formula, but is sufficient to produce an implicant covering. More precisely, Algorithm 2 starts by setting the number of models to 0 and Ω to $\emptyset$. a is a fresh variable used to mark clauses to be deleted afterwards. Then, while there exists an interpretation $\mathcal{I}$ of $(\Sigma_{|\mathcal{A}\cup\{a\}} \wedge \Omega)$ (*i.e.* Ω does not cover every models of $\Sigma_{|\mathcal{A}\cup\{a\}}$), a path $\mathcal{I}_{\mathcal{F}}$ in the forgetting tree of $\mathbf{FO}(\Sigma_{|\mathcal{A}\cup\{a\}}, \mathcal{F})$ is built (line 6). Afterwards, a prime implicant $\mathcal{I}'$ of the formula on the leaf carried by $\mathcal{I}_{\mathcal{F}}$ is computed. Since all interpretations of this formula is an interpretation of $\mathbf{FO}(\Sigma_{|\mathcal{A}\cup\{a\}}, \mathcal{F})$, $\mathcal{I}'$ is also an implicant of $\mathbf{FO}(\Sigma_{|\mathcal{A}\cup\{a\}}, \mathcal{F})$. Then, the number of new models covered by $\mathcal{I}'$ are added to $nbModels$ and a blocking clause is added.

Algorithm 2: Model Counting (**CT**)

Input: Σ: CNF, $\mathcal{A}$: assumptions, $\mathcal{F}$: forgot variables
Output: the number of models of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$

1 $nbModels \leftarrow 0$; $\Omega \leftarrow \emptyset$;
2 a a literal not present in Σ;
3 **while** $(\Sigma_{|\mathcal{A}\cup\{a\}} \wedge \Omega)$ *is SAT* **do**
4 $\quad \mathcal{I} \leftarrow solve(\Sigma \wedge \Omega, \mathcal{A} \cup \{a\})$;
5 $\quad \mathcal{I}_{\mathcal{F}} \leftarrow \{\ell \in \mathcal{I} | var(\ell) \in \mathcal{F}\}$;
6 $\quad \mathcal{I}' \leftarrow reduce((\Sigma \wedge \Omega \wedge a \wedge (\bigwedge_{\ell \in \mathcal{A}\cup\mathcal{I}_{\mathcal{F}}} \ell))), \mathcal{I})$;
7 $\quad \Omega \leftarrow \Omega \wedge ((\bigvee_{\ell \in \mathcal{I}'\setminus\mathcal{I}_{\mathcal{F}}} \neg\ell) \vee \neg a)$;
8 $\quad nbModels \leftarrow nbModels + 2^{|Var(\Sigma)|-|\mathcal{I}'|}$;
9 remove from Σ the set of clauses which contain a;
10 **return** $nbModels$;

4.3 Implicants Covering (IC)

Proposing an algorithm for computing implicants (or prime implicants) is a frequent request by industrials. Algorithm 3 computes an implicant cover of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$. The method depicted in this algorithm differs from Algorithm 2 by the way it handles clauses in Ω when reducing the implicant. First (lines 4-5), the restriction $\mathcal{I}_{\mathcal{F}}$ over $\mathcal{F}$ of an implicant $\mathcal{I}$ is computed. Then the implicant is reduced, but literals from $\mathcal{A}$ and $\mathcal{F}$ are forced to be kept (line 6). Here, $\mathcal{I}'$ is an implicant of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, F)$ (Ω is not used line 6), and it can thus be added to the final result. Then, line 7, Ω is updated to ensure that the algorithm terminates.

Algorithm 3: Implicant Cover (**IC**)

Input: Σ: CNF, $\mathcal{A}$: assumptions, $\mathcal{F}$: forgot variables
Output: $\mathcal{PI}$ a implicants cover of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, F)$

1 $\mathcal{PI} \leftarrow \emptyset$; $\Omega \leftarrow \emptyset$;
2 a a literal not present in Σ;
3 **while** $(\Sigma \wedge \Omega \wedge (\bigwedge_{\ell \in \mathcal{A}} \ell) \wedge a)$ *is SAT* **do**
4 $\quad \mathcal{I} \leftarrow solve(\Sigma \wedge \Omega, \mathcal{A} \cup \{a\})$;
5 $\quad \mathcal{I}_{\mathcal{F}} \leftarrow \{\ell \in \mathcal{I} | var(\ell) \in \mathcal{F}\}$;
6 $\quad \mathcal{I}' \leftarrow reduce(\Sigma \wedge (\bigwedge_{\ell \in \mathcal{A}\cup\mathcal{I}_{\mathcal{F}}} \ell) \wedge a, \mathcal{I})$;
7 $\quad \Omega \leftarrow \Omega \wedge ((\bigvee_{\ell \in \mathcal{I}'} \neg\ell) \vee \neg a)$;
8 $\quad \mathcal{PI} \leftarrow \mathcal{PI} \cup \{\mathcal{I}'\}$;
9 remove from Σ the set of clauses which contain a;
10 **return** $\mathcal{PI}$;

When $\mathcal{F} = \emptyset$, implicants returned by Algorithm 3 are guaranteed to be prime (of $\Sigma_{|\mathcal{A}}$). We can also notice that it is still possible to compute a prime implicants covering of $\mathbf{FO}(\Sigma_{|\mathcal{A}}, \mathcal{F})$ by iteratively applying **IM** to reduce $\mathcal{I}'$.

5 Experimental Evidences

We propose in this section some experiments on our tool, called `Jit` and based on the SAT solver `glucose`, on different benchmarks and queries. All tests of this paper are done on a Intel XEON X5550 quad-core 2.66 GHz with32Gb RAM. The CPU time limit is set to one hour. Our tool, with its source and exhaustive and detailed results, will be publicly available.

5.1 Classical but very easy problems

Classical KC Problems

When we wanted to test our approach, we set up a first experiment using exactly the protocol used in [Subbarayan *et al.*, 2007]. However, under the same conditions (see reference for details), our SAT-based approach, without any off-line phase, solved all the queries (on all the problems) in 0.00s (three digits). In many cases, even after 10,000 queries, less than a thousand conflicts are necessary (total). This demonstrates the paradigm shift of our approach over previous approaches and even ToBDD ones (the latter needs a thousand seconds to compile all the problems, and a few "hard" problems need 0.30s, median time, for queries). Of course, this shows only how well suited is our approach for **CE** and **IM** queries, and these queries are very close to simple SAT checking. However, we can make three conclusions here. (1) We should consider new families of problems for KC ; (2) Instances that are decomposable in practice seem trivial for SAT solvers ; (3) We need harder problems to account for amortized Just-in-Time compilation.

Classical Configurations Problems

Configuration problems are typical A.I. problems of industrial importance. In these problems, the user is interactively choosing the polarity among the set of unassigned variables, and we need to guarantee that at least one configured product can be found after each of his possible choice. To do so, it is necessary to produce all implied literals, according to the set of current conditioned variables, in order to remove these literals from possible user choices. It is also necessary to let the user go back to a previous choice, if needed. For assigning variables, we can use our **CD** operator, that allows to unset any conditioned variable, in any order (we just need to remove the assigned literal from the set $\mathcal{A}$). We implemented the algorithm proposed in [Sinz *et al.*, 2003] and tested it on a classical set of car/truck configuration problems from Renault, publicly available. We used the *order encoding* [Tamura *et al.*, 2006] for translating the initial CSP problem into SAT. Each run simulated a loop of user choice/implied literals computation until we end up with a final car configuration. However, once again, all the problems were trivial for SAT solvers. All the queries are performed in constant time, 0.00s in all the cases.

5.2 Harder instances for KC

We decided, in this section, to use a subset of satisfiable benchmarks from the SAT 2011 competition, which contains very hard problems for SAT. Our experimental framework is the following. We took 50 benchmarks solved in less than 100 seconds by `glucose`. We performed 10,000 random **CE** queries with at most 100 literals for each of them. For 35 benchmarks, `Jit` is able to answer the 10,000 queries in less than one hour (corresponding to an average of 0.3 seconds per queries) whereas ToBDD [Subbarayan *et al.*, 2007] fails for 44 instances at the compilation stage, within a timeout of one hour (some instances triggered memory out problems too).

Amortization of Just-in-Time compilation

One of the important claims of our work is the concept of amortization of past queries. The more queries, the faster

Instance			Incremental				No Incremental			ToBDD
	#V	#C	time	avg	unit lits	conflicts	time	avg	conflicts	Time
AProVE11-02	106,815	552,819	2,088	0.2	4,652	1,169	TO		$> 5{,}000$	–
AProVE11-12	17,914	147,360	14	0.001	13,945	85	403	0.4	4,000	10
12pipe_bug6_q0	81,184	4,547,858	1,046	0.1	8,987	4,371	TO		$>15{,}000$	–
SAT_dat.k80_04	42,375	262,415	101	0.01	12,744	3,288	TO		$>13{,}000$	–
smtlib-qfbv-aigs-bin...	42,608	212,741	482	0.04	0	2,805	730	0.7	1,042	–

Table 1: For each instance, we report the number of variables and clauses (#V, #C), the time to achieve 10,000 queries, the average time for one query, the number of unit literals learnt during the search and the total number of conflicts (in thousands). We also report the time, average time and conflicts for non incremental SAT and the compilation time for ToBDD if it successes.

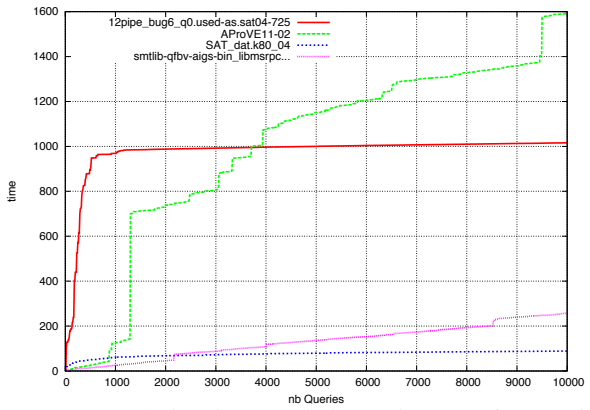


Figure 1: Cumulative time needed (x-axis) to perform a given number of queries (y-axis) on some SAT instances.

new ones should be answered. Fortunately, we can positively answer to our initial idea, as soon as we consider hard enough problems. This is highlighted in Figure 1. We report, on a set of characteristic instances, the number of queries performed (x-axis) and the total time needed to answer all of them (y-axis). The amortization is clear for the instance `12pipe`... The first 500 queries are difficult (2s by query) but, after that, the answer is given almost instantaneously. The instance `AProve11-02` has also an interesting behavior. Some queries are very difficult (take a look at the peak around 1200 queries) but here again, the amortization pays. Instance `SAT_dat.k80` is a typical easy problem. The last example is the instance `smtlib`. Here, each query is very easy, but the amortization is not clear. One reason for this is that no implied literal is proven during the different calls.

Table 1 provides details for the instances considered above. This table also compares our incremental approach with a non incremental SAT solver (each query is applied on a new solver instance) and a ToBDD [Subbarayan *et al.*, 2007] approach. Once again, (we observed the same tendency on all the problems we tried), this clearly demonstrates the power of our approach on a set of hard problems. We also report an instance `AproVE11-12` where ToBDD does not fail, but once again, it appears that this instance is trivial for our approach.

5.3 Model Counting

We conclude with some experiments on model counting. We compare `Jit` with `cachet` [Sang *et al.*, 2004]. Table 2 shows results on a few instances, due to lack of space. However, all experiments will be available on the web and we believe that our selection represents a pretty fair picture of `Jit` strenghts and limits on this task. First of all, `Jit` is quite efficient on problems with few models and clearly inefficient on problems with many. On such problems, decomposition methods, like SDD (see [Darwiche, 2011]) are clearly the best choices. Column `Jit+` shows results on model counting after 1,000 random queries: Just-in-Time compilation can also improve performances on such queries.

instance	#models	`Jit`	`Jit+`	`cachet`
cnt10	1	14.491	0.02	TO
bw-large-d	106	0.738	0.22	148
ais12	1328	480.4	420.3	TO
hanoï5	1	0.283	0.0001	65.57
logistic-b	$>10^{23}$	TO	TO	6.92
iscas89-s1488	$>10^{5}$	5.86	6.05	0.02
bmc-ibm-1	$>10^{302}$	TO	TO	17.36

Table 2: Model Counting results

6 Conclusion

We presented a new concept for KC, called Just-in-Time Compilation. The progresses made in the practical solving of SAT problems, and the development of incremental SAT solvers have made this concept possible in practice. We experimentally reported impressive results of our approach on the set of classical problems from the KC domain, and on a set of configuration problems. On these problem, we reported 0.00s query time. It seems that, in order to be decomposable in practice, problems are trivial for SAT engines. We also demonstrated how well founded is the idea of amortization of past queries efforts, by showing how incremental SAT solving improves the performances of a set of direct (non incremental) SAT queries by orders of magnitude.

Our approach has however some limitations, for instance when it is crucial to be able to count models. On this last task, we believe that more traditional KC approaches are still the best, even if we obtained interesting results. Of course, this work opens many new possible improvements. For instance, we can notice that it is always possible to use a CNF encoding of any other propositional fragment, by using the well-known Tseitin transformation (additional variables can be introduced). Thus, it is also possible to "compile" a subset of the CNF, by replacing its clauses by a set of clauses encoding the DNNF of the removed clauses, for instance, in order to improve SAT solvers performances on particularly hard problems for queries.

References

[Audemard and Simon, 2009] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.

[Bradley, 2012] A. Bradley. IC3 and beyond: Incremental, inductive verification. In *proceedings of CAV*, page 4, 2012.

[Darwiche and Marquis, 2001] A. Darwiche and P. Marquis. A perspective on knowledge compilation. In *proceedings of IJCAI*, pages 175–182, 2001.

[Darwiche and Marquis, 2002] A. Darwiche and P. Marquis. A knowledge compilation map. *J. of AI Research*, pages 229–264, 2002.

[Darwiche, 2011] A. Darwiche. SDD: a new canonical representation of propositional knowledge bases. In *proceedings of IJCAI*, pages 819–826, 2011.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, pages 201–215, 1960.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 1962.

[Dechter and Rish, 1994] R. Dechter and I. Rish. Directional resolution: the davis-putnam procedure revisited. In *proceedings of KR*, pages 134–145, 1994.

[Del Val, 1994] A. Del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In *proceedings of KR*, pages 551–561, 1994.

[Eén and Sörensson, 2003a] N. Eén and N. Sörensson. An Extensible SAT-solver. In *proceedings of SAT*, pages 333–336, 2003.

[Eén and Sörensson, 2003b] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543 – 560, 2003.

[Fargier and Marquis, 2009] H. Fargier and P. Marquis. Knowledge compilation properties of Trees-of-BDDs revisited. In *proceedings of IJCAI*, pages 772–777, 2009.

[Goldberg and Novikov, 2002] Evguenii I. Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *proceedings of DATE*, pages 142–149, 2002.

[Huang, 2007] J. Huang. The effect of restarts on the efficiency of clause learning. In *proceedings of IJCAI*, pages 2318–2323, 2007.

[Lang *et al.*, 2003] J. Lang, P. Liberatore, and P. Marquis. Propositional independence: formula-variable independence and forgetting. *J. Artif. Int. Res.*, pages 391–443, 2003.

[Marques-Silva *et al.*, 2009] J. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press, 2009.

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Conor, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *proceedings of DAC*, 2001.

[Nadel and Ryvchin, 2012] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *proceedings of SAT*, 2012.

[Nadel, 2010] A. Nadel. Boosting minimal unsatisfiable core extraction. In *proceedings of FMCAD*, pages 221–229, 2010.

[Oh *et al.*, 2004] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, and I.L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conference*, pages 518–523, 2004.

[Pipatsrisawat and Darwiche, 2007] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *proceedings of SAT*, pages 294–299, 2007.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H.A., Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *proceedings of SAT*, 2004.

[Selman and Kautz, 1996] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[Silva and Sakallah, 1996] J. P. Marques Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *proceedings of CAD*, pages 220–227, 1996.

[Sinz *et al.*, 2003] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.

[Subbarayan *et al.*, 2007] S. Subbarayan, L. Bordeaux, and Y. Hamadi. Knowledge compilation properties of Tree-of-BDDs. In *proceedings of AAAI*, 2007.

[Tamura *et al.*, 2006] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. In *proceedings of CP*, pages 590–603, 2006.

[Zhang *et al.*, 2001] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *proceedings of CAD*, pages 279–285, 2001.