Propositional Simplification With BDDs and SAT Solvers

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. We show how LCF-style interactive theorem provers might use BDD engines and SAT solvers to perform normalization, simplification of terms and theorems, and assist with interactive proof. The treatment builds on recent work integrating SAT solvers as non-trusted decision procedures for LCF-style theorem provers. We limit ourselves to propositional logic, but briefly note that the results may be lifted to more expressive logics.

1 Introduction

Interactive theorem provers like PVS [21], HOL4 [10] or Isabelle [22] traditionally support rich specification logics. Automation for these logics is however difficult, and proving a non-trivial theorem usually requires manual guidance by an expert user. Automatic proof procedures on the other hand, while often designed for simpler logics, have become increasingly powerful over the past few years. New algorithms, improved heuristics and faster hardware allow interesting theorems to be proved with little or no human interaction, sometimes within seconds.

By integrating automated provers with interactive systems, we can preserve the richness of our specification logic and at the same time increase the degree of automation [24]. However, to ensure that a potential bug in the integration with the external proof procedure does not render the whole system unsound, theorems in LCF-style [8] provers can be derived only through a small fixed kernel of inference rules. Therefore it is not sufficient for the automated prover to return whether a formula is provable, but it must also generate the actual proof, expressed (or expressible) in terms of the interactive system's inference rules. HOL4, Isabelle and HOL Light are well known LCF-style provers. PVS, on the other hand, is not implemented in an LCF-style manner, except at a very high level. The Coq interactive theorem prover [13] is not technically LCF-style, but follows the "de Bruijn criterion" of verifying proofs using a proof checker that in spirit is much like an LCF-style kernel.

Formal verification is an important application area of interactive theorem proving. Problems in verification can often be reduced to Boolean satisfiability (SAT) and so the performance of an interactive prover on propositional problems may be of significant practical importance. Binary decision diagrams (BDDs) [2] and SAT solvers [5, 19] are powerful proof methods for propositional logic and it is natural to use them as proof engines within interactive provers.

Recent work [6, 29] showed how LCF-style interactive provers could use SAT solvers as non-trusted decision procedures for propositional logic. We now build on this work to show how, for propositional logic, LCF-style interactive theorem provers might use BDD engines and SAT solvers to perform normalisation, simplification of terms and theorems, and assist with interactive proof, without having to trust the external procedures. The treatment is tool independent, and assumes only that the interactive prover is expressive enough to suppose quantification over pure Boolean formulas.

The next section gives a brief account of the relevant aspects of normal forms, BDDs and SAT solvers, to keep the paper self-contained. In §3 and §4, we look at normalisation and simplification respectively. We end with a look at previous related work and some concluding remarks.

2 Preliminaries

We use $\vdash t$ to denote that t is a theorem in the object logic, i.e, the logic of the interactive prover. We reserve the words "iff" and "implies" for logical equivalence and implication in our proofs, and use $\Leftrightarrow$ and $\Rightarrow$ to denote their respective counterparts in the object logic. Quantification binds weaker than $\Leftrightarrow$, which binds weaker than $\Rightarrow$. Propositional truth is denoted by $\top$ and falsity by $\bot$. We use $\equiv$ to denote syntactic equivalence in the object logic. All other notation is standard.

A *literal* is either an atomic proposition or its negation. A *clause* is a disjunction of literals. A *monomial* is a conjunction of literals. Since both conjunction and disjunction are associate-commutative (AC), clauses and monomials can also be interpreted as sets of literals. If a literal occurs in a set, then we abuse notation and assume its underlying proposition also occurs in the set.

2.1 Normal Forms

A term is in *disjunctive normal form* (DNF) if it is a disjunction of monomials, and in *conjunctive normal form* (CNF) if it is a conjunction of clauses. Any propositional term t can be transformed into a logically equivalent term in DNF or CNF, denoted by $DNF(t)$ and $CNF(t)$ respectively. Again, by AC, $DNF(t)$ and $CNF(t)$ can also be interpreted as sets of sets of literals, and we overload the notation accordingly. We will switch back and forth between the term and set interpretations, as convenience dictates.

Computing normal forms is important in automated reasoning for many reasons, most to do with term rewriting theory. For our purposes, they are also important as many proof procedures accept input terms only in some normal form, e.g., resolution based provers use CNF, or have to compute normal forms internally, e.g., some quantifier elimination methods use DNF.

SAT solvers require their input term to be in CNF. Any term can be transformed to CNF, but the result can be exponentially larger than the original term. To avoid this, *definitional* CNF [26] introduces extra propositions as placeholders for subterms of the original problem. We use $dCNF(t)$ to denote the definitional CNF of t, and note that it can also be interpreted as a set. We will use *dCNF* extensively in what follows, so we give a short description here.

The best way to understand the basic idea is by example. Consider the term t given by

$$\neg((((p \Rightarrow q) \Rightarrow p) \Rightarrow p)$$

The first preprocessing step is to rewrite away the $\Rightarrow$ operators, using the identity $\vdash p \Rightarrow q \Leftrightarrow \neg p \vee q$, and the second preprocessing step is to then push all negations inwards. Having done this, we obtain

$$((p \wedge \neg q) \vee p) \wedge \neg p$$

Now we proceed bottom up, introducing abbreviations for each subterm in the form of fresh propositions v_0, v_1 etc. We set $v_0 \Leftrightarrow p \wedge \neg q$ and obtain

$$(v_0 \Leftrightarrow p \wedge \neg q) \wedge (v_0 \vee p) \wedge \neg p$$

and then introduce v_1 to get

$$(v_0 \Leftrightarrow p \wedge \neg q) \wedge (v_1 \Leftrightarrow v_0 \vee p) \wedge v_1 \wedge \neg p$$

at which point it is easy to see that the term can be made into CNF by applying a standard CNF conversion to each abbreviation. We can now see that the negations were moved inwards to avoid needlessly introducing definitional variables. There are other possible optimizations but we will stick with this simple procedure for now.

The term $dCNF(t)$ is not logically equivalent to t, since there are valuations for the *definitional variables* v_i that can disrupt an otherwise satisfying assignment to the variables of t. However, it is equisatisfiable. This fact is expressed by the theorem

$$\vdash t \Leftrightarrow \exists \bar{v} \in V.dCNF(t) \tag{1}$$

where V is the set of all the definitional variables and $\bar{v}$ indicates that the quantification is over all $v \in V$. The definitional CNF procedure can be augmented to produce this theorem automatically. Henceforth, we reserve the identifier prefix v to refer to the definitional variables only. The equivalence that introduces the abbreviation for each v_i will be referred to as the *definition of* v_i, and the right-hand side of the definition, i.e., the term v_i is abbreviating, will be denoted by $\hat{v_i}$.

2.2 BDDs

Binary decision diagrams (BDDs) are data structures for representing Boolean formulas and Boolean operations on them. For instance, the BDD of the term $p \Rightarrow q$ is given in Figure 1. Dotted arcs indicate a valuation of $\bot$ and solid arcs a valuation of $\top$ to the parent node. A path from the root to the 1 node indicates

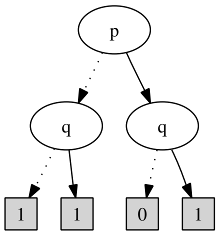

Fig. 1. Example BDD for $p \Rightarrow q$

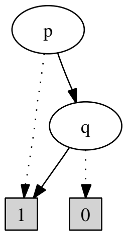

Fig. 2. Reduced Ordered Version

an assignment that makes the formula true, and a path to the 0 node, a falsifying assignment.

The representation can be made canonical by establishing an ordering on the variables, and can be made efficient by removing redundant arcs and nodes. The *reduced ordered* BDD corresponding to that of Figure 1 is given in Figure 2. When we say BDD, we mean the ordered and reduced version.

BDDs are canonical up to variable ordering, and have efficient counterparts for all Boolean operations, including quantification. In theory, the problem is NP-complete. In practice, BDDs can often achieve very compact representations. We need not say any more about them. The interested reader may consult [1].

It turns out that representing BDDs efficiently in an LCF style prover causes too high a performance penalty (see §5 for details). Therefore, we assume that the results of BDD operations by themselves cannot produce theorems in the object logic.

2.3 SAT Solvers

SAT solvers are efficient algorithms for testing Boolean satisfiability. A SAT solver will accept a Boolean term in CNF and return a satisfying assignment to its variables. If the term is unsatisfiable, the SAT solver will return a resolution refutation proof from the clauses of the input CNF term. Not all SAT solvers can produce this proof, but some can [31], and have been integrated with interactive theorem provers with (mostly) a reasonable slowdown [6, 29]. We assume we have access to such an integration. Thus, the result of a SAT solver can be represented as a theorem in the object logic. This short description is sufficient for our purposes. A tutorial introduction to resolution based SAT solvers is available [18].

3 Normalization

Our first contribution is to term normalization. Normalization means reducing a term to its normal form. This is traditionally done by rewriting with a set of identities. When computing normal forms in LCF-style theorem provers, we further require that the normalization is done by proof, in effect requiring a theorem that the term is logically equivalent to, or in the case of definitional CNF, equisatisfiable with, the obtained normal form. The requirement of proof generation causes a slowdown. Moreover, if we are not careful, a rewrite based transform can generate large normal forms.

We can instead exploit the compact term representation and speed of BDDs and SAT solvers. The solution is straightforward. To generate the DNF of a term t, we build the BDD of t, and then just read off the set of all satisfying assignments. Each assignment is a monomial and the disjunction t' of all the assignments is thus in DNF. Further, it is satisfiable iff t is. The required theorem can be obtained by using the SAT solver to check that $\neg(t \Leftrightarrow t')$ is unsatisfiable, giving $\vdash t \Leftrightarrow t'$.

Similarly, the CNF of a term can be obtained by reading off all falsifying assignments. The DNF term t' thus obtained is logically equivalent to $\neg t$. Then applying negation to t' and driving all negations inwards to the atoms, we obtain a term that is in CNF, and equivalent to t. Once again, the SAT solver can be used to obtain the required theorem.

For CNF terms generated in this manner, the redundancy removal algorithm in BDDs guarantees that the clauses are subsumption free, i.e., no clause implies another. Similarly, for DNF terms, no satisfying assignment is a subset of another. This contributes towards keeping the normalised terms small.

This might give the impression that any transform on propositional formulas can be implemented by computing the desired result using an efficient external engine and then confirming the result with a SAT solver. While possible in theory, it may not always work in practice. Many such transforms have sub-exponential worst-case or average-case behaviour, and converting the problem to SAT may not help. Also, this approach has a high overhead of external procedure calls, where the interface is often via disk files. If the transform is done several times on small formulas, the overhead may dominate the benefit.

We compared our method of generating normal forms for CNF generation, with the built-in CNF conversion present in the HOL4 theorem prover, on randomly generated propositional terms of various sizes. The results were inconclusive. On even small terms (say, 15 variables and 300 connectives), both methods ran out of memory. This is expected since CNF terms can become exponentially large and interactive theorem provers are not engineered for efficient storage of large clausal terms. In particular, our procedure ran out of memory during the reading off of the CNF from the BDD. On smaller terms, our method was faster in general, but not by much. However there were certain cases where it was much slower. We put this down to an unfortunate variable ordering for the BDD, since as yet we make no effort to find a good one, and to the fact that the SAT solver we use [19] is not tuned for random problems.

BDDs do not scale up as well as SAT solvers, in the sense that as the number of variables increases, the space requirements for storing BDDs can become infeasible. In typical interactive proof, users are unlikely to be using such large terms. Our aim however is to support better automation, and automatic methods may well operate on large terms. It is worthwhile looking for a way to perform normalization using SAT solvers only.

The idea (already well known) is to use the SAT solver to generate all satisfying assignments for a given term t. A simple way of doing this is by the addition of *blocking clauses*. The method works as follows:

1. Let S be the set of known satisfying assignments; set $S = \emptyset$
2. Send t to the SAT solver
3. If the SAT solver returns unsatisfiable, return S
4. Otherwise we have a satisfying assignment σ.
5. Set S to $S \cup \{\sigma\}$
6. Form the conjunction of $\neg\sigma$ with the previous input to the solver and send that to the SAT solver
7. Go to step 3

Note that σ is a monomial and so $\neg\sigma$ is a clause, so that $t \wedge \neg\sigma$ is valid input to the solver. By adding $\neg\sigma$ to the term, we ensure that the SAT solver cannot return satisfying assignments already in S. Indeed, the solver will not return assignments that are supersets of any known assignments. Subsets may be returned, but the set S can be kept irredundant by adapting well known techniques [4, 30].

This is easily seen to terminate. By the end, we have all satisfying assignments, and can derive the DNF of t by forming $\bigvee S$. It is not too hard to change the steps above so that we can derive CNF instead. This is of course a very expensive way to do normalization, and really is only useful for large terms, i.e., thousands of variables. A rough estimate of the work required can be arrived at by using a stochastic DNF solution counting method (see Chapter 28 of [27]).

Using blocking clauses as above is not the best approach because it forces the solver to redo the search from scratch. Many SAT solvers have *incremental* search capability, where the information learnt from previous searches is retained and is applicable so long as the new problem term is an extended version of the previous one. In fact, specialised algorithms for enumerating all solutions do even better [11]. At the moment we are not aware of any integration of these with interactive provers.

If anything, the lesson from this work is that, for the generation of large normal forms in interactive provers, space complexity is a far more serious problem than time complexity. The sizes that can be handled are enough for most interactive proof however, so the work in §4.3 for instance, is of practical use.

4 Simplification

If we restrict ourselves to the propositional structure of a term, simplification usually means reducing the size, or the depth, or both, of the term. Normalization, for instance, eliminates depth altogether (modulo AC), and may often result in a smaller term as well. The downside to normalization as a vehicle for simplification is that, by definition, it destroys the structure of the term in question, and with it, any intuition that the human using the theorem prover may have had about the term. Therefore, simplification by rewriting is typically preferred during interactive proof.

Our second contribution is to show how useful simplification can sometimes be accomplished using BDDs and SAT solvers. An important consideration is to do this simplification without the kind of mangling that normalization produces. There are several applicable scenarios, which we now consider.

4.1 Theorems

Suppose we have a propositional term t, and we wish to check whether or not it is a tautology. This can be done by computing $dCNF(\neg t)$ and asking a SAT solver if that term is unsatisfiable. If so, let V be the set of definitional variables appearing in $dCNF(\neg t)$, as in §2.1, and we have

$$\begin{array}{ll} & \vdash dCNF(\neg t) \Rightarrow \bot \text{ from the SAT solver supplied refutation} \\ \text{iff} & \vdash \forall \bar{v} \in V.dCNF(\neg t) \Rightarrow \bot \\ \text{iff} & \vdash (\exists \bar{v} \in V.dCNF(\neg t) \Rightarrow \bot \\ \text{iff} & \vdash (\exists \bar{v} \in V.dCNF(\neg t) \Leftrightarrow \bot \\ \text{iff} & \vdash \neg t \Leftrightarrow \bot \text{ by (1)} \end{array}$$

and we can conclude $\vdash t$.

It is rarely the case that every single clause of $dCNF(\neg t)$ is used in the SAT solver's refutation proof. The subset of clauses that does get used is called the unsatisfiable *core*. There are algorithms that attempt to find smaller cores [7] given a refutation proof. We can use smaller cores to deduce $\vdash s$ rather than $\vdash t$, where $\vdash s \Leftrightarrow t$ but s is simpler than t. We now show how to construct s.

Suppose we have obtained a core D, so $\vdash D \Rightarrow \bot$. Now $D \subseteq dCNF(\neg t)$ by definition (of a core). If $D = dCNF(\neg t)$ then there is no simplification, so $s \equiv t$.

Otherwise, we impose an ordering relation $\prec$ on V, such that $v \prec v'$ iff v occurs in $\hat{v'}$, i.e., in the right-hand side of the definition of v'. Let $\prec^+$ be the transitive closure of $\prec$. Let $\preceq = \prec \cup \equiv$ and let $\preceq^+$ be the transitive closure of $\preceq$. Then s is constructed as follows:

1. Let $V' = \{v \in V \cap \bigcup_{d \in D} d | \forall v' \in V \cap \bigcup_{d \in D} d.v \preceq^+ v' \Rightarrow v \equiv v'\}$
2. Let $V'' = \{v \in V | \exists v' \in V'.v \prec^+ v'\}$
3. Let $D' = D \cup \{c \in \mathit{dCNF}(\neg t) | \exists v \in (V' \cup V'').v \in c\}$. Since $\vdash D \Rightarrow \perp$, we have $\vdash D' \Rightarrow \perp$.
4. Let $D'' = D'[\hat{v}/v \in V''][\top/v \in V']$, so $\vdash D'' \Rightarrow \perp$ also. In D'' we also explicitly reverse the CNF expansions of the definitions of the $v' \in V'$.
5. Simplify away the $v \in V$ in D'' to obtain D''' and set $s \equiv \neg D'''$. At this point $\vdash \neg s \Rightarrow \perp$.

To elaborate a little, V' is the set of the maximal (w.r.t. $\prec^+$) $v \in V$ that occur in D. For each such v, D' contains all the clauses of $\mathit{dCNF}(\neg t)$ containing a variable that was $\preceq^+ v$. V'' is the set of $v \in V$ that are strictly below any $v' \in V'$. The implicit strategy is to in effect collect together the clauses comprising the definitions of each $v \in V' \cup V''$ and reverse the per-abbreviation CNF conversion applied during the definitional CNF computation outlined in §2.1, though we do not explicitly do this reversal except for the $v' \in V'$. First, we replace each occurrence in D' of $v'' \in V''$ by its definition. Next, each maximal $v' \in V'$, is replaced by $\top$. These replacements give us D''. Now since each $v'' \in V''$ has been substituted into its own definition, the clauses corresponding to its definition can be simplified away. Also, since each maximal $v' \in V'$ occurs only on the left-hand side of its defining equivalence which has been explicitly reconstructed, replacing it by $\top$ and simplifying converts that equivalence into $\hat{v'}$ except that it has been expanded out fully. This gives us D''', which is a conjunction of single literals and the fully expanded right-hand sides of the definitions of the maximal $v \in V'$. Effectively, D''' is the term after $\neg t$ was negated and preprocessed but before definitional CNF was applied, less some top-level structure of t. Then we get s by negating D'''.

Intuitively, each $v \in V$ represents a subterm of t. Any such v remaining in the core are clearly pertinent to the truth value of t. Any $v' \prec^+ v$ also cannot be ignored since each v is dependent on their definition. Roughly speaking, we collect together all these variables and back-substitute their definitions into the core, in an attempt to resurrect the structure of t which is implicitly encoded in the definitions. We need to do a little more to better recover the structure, but before we introduce those complications let us first establish the soundness of the basic idea.

Proposition 1. $\vdash s \Leftrightarrow t$
Proof The theorem $\vdash D' \Rightarrow \perp$ at the end of step 3 above is easily seen to be correct. Note that the occurrence of any $v \in V$ in D' is implicitly universally quantified, so the substitutions in step 4 are just instantiations, preserving equivalence. Thus step 5 correctly concludes that $\vdash \neg s \Rightarrow \perp$. Now, we have $\vdash D \Rightarrow \perp$ from the SAT solver's refutation proof. Then $\vdash \neg s \Leftrightarrow D$, using $\forall t.t \Rightarrow \perp \Leftrightarrow (t \Leftrightarrow \perp)$. Now $D \subseteq \mathit{dCNF}(\neg t)$, so $\vdash \mathit{dCNF}(\neg t) \Rightarrow D$. But $\vdash D \Rightarrow \perp$, so we have $\vdash \mathit{dCNF}(\neg t) \Leftrightarrow \neg s$ by transitivity of $\Leftrightarrow$. Finally, $\vdash \neg t \Leftrightarrow \exists \overline{v} \in V.\mathit{dCNF}(\neg t)$ by the definitional CNF construction, so we get $\vdash \neg t \Leftrightarrow \exists \overline{v} \in V.\neg s$. No $v \in V$ occurs in s since $V' \cup V'' \subseteq V$ is the set of definitional variables occurring in D

but these are all substituted away in step 4. So we conclude $\vdash \neg t \Leftrightarrow \neg s$ and the required result follows. □

Syntactically, s does not quite follow the structure of t yet, mainly because we have ignored the preprocessing steps of definitional CNF such as rewriting away $\Rightarrow$ operators and moving negations inwards. These preserve equivalence however, and can be reversed by storing suitable information for each subterm. Therefore these operations too can be reversed without affecting Proposition 1. The details are uninteresting. We now present an example, before making some concluding remarks. The example is rather contrived, but we want a small example that provokes the "right" behaviour.

Example Let t be then term $(((p \Rightarrow q) \Rightarrow p) \Rightarrow p) \vee t'$ where $t' \Rightarrow\perp$ but t' is complicated enough that it is beyond the ability of the interactive prover's simplifier to prove that. We further assume that the prover's native simplifier cannot prove $\vdash ((p \Rightarrow q) \Rightarrow p) \Rightarrow p$ either.[1] We use the work already done in §2.1 and obtain

$$\vdash dCNF(\neg t) \Leftrightarrow (v_0 \Leftrightarrow p \wedge \neg q) \wedge (v_1 \Leftrightarrow (v_0 \vee p) \wedge v_1 \wedge \neg p \wedge dCNF(\neg t')$$

We are not interested in what happens to t', and apply per-abbreviation CNF to get

$$\begin{aligned}\vdash dCNF(\neg t) \Leftrightarrow & (v_0 \vee \neg p \vee q) \wedge (\neg v_0 \vee \neg q) \wedge (\neg v_0 \vee p) \wedge \\ & (\neg v_1 \vee v_0 \vee p) \wedge (v_1 \vee \neg v_0) \wedge (v_1 \vee \neg p) \\ & \wedge v_1 \wedge \neg p \wedge dCNF(\neg t')\end{aligned}$$

At this point, $V = \{v_0, v_1\}$, ignoring the definitional variables occurring in $dCNF(\neg t')$. All we need to know about the latter is that they are incomparable with any in V w.r.t. $\prec$.

Now $dCNF(\neg t')$ is satisfiable, so the SAT solver is forced to use the rest of $dCNF(\neg t)$ to show unsatisfiability. The reader may confirm that if $D = \{\neg v_0 \vee p, \neg v_1 \vee v_0 \vee p, v_1, \neg p\}$ under the set interpretation, then $D \Rightarrow\perp$ (recall D is a CNF term). Then $V' = \{v_1\}$ and $V'' = \{v_0\}$. Collecting together the needed clauses, we see that

$$\begin{aligned}D' = \{ & v_0 \vee \neg p \vee q, \neg v_0 \vee \neg q, \neg v_0 \vee p, \\ & \neg v_1 \vee v_0 \vee p, v_1 \vee \neg v_0, v_1 \vee \neg p, \\ & v_1, \neg p\}\end{aligned}$$

and of course $\vdash D' \Rightarrow\perp$. We explicitly reverse the CNF expansion of the definition of v_1, to obtain

$$\begin{aligned}D' = \{ & v_0 \vee \neg p \vee q, \neg v_0 \vee \neg q, \neg v_0 \vee p, \\ & v_1 \Leftrightarrow v_0 \vee p, \\ & v_1, \neg p\}\end{aligned}$$

[1] If it can, we just replace with a more complex term. It is certainly beyond the HOL4 simplifier, which is our behind-the-scenes test bed.

Now substituting v_0 by its definition and v_1 by $\top$, and simplifying just enough to remove the v_i, we get $\vdash D''' \Rightarrow \bot$ which looks like

$$\vdash (((p \land \neg q) \lor p) \land \neg p) \Rightarrow \bot$$

and finally reversing the preprocessing step (which also adds back the top-level negation that was added when t was negated prior to applying definitional CNF) we get $\vdash \neg s \Rightarrow \bot$ which looks like

$$\vdash \neg(((p \Rightarrow q) \Rightarrow p) \Rightarrow p) \Rightarrow \bot$$

We can now follow the reasoning in the proof above to conclude that

$$\vdash t \Leftrightarrow (((p \Rightarrow q) \Rightarrow p) \Rightarrow p)$$

□

It is clear that this method did simplify $\vdash t$, and that this kind of simplification is likely beyond the reach of the rewriting-based simplifiers currently in use in interactive theorem provers. But by now it should also be suspected that such clean simplification cannot always be achieved. For instance, the current method is crude in the sense that any subterm that is top-level conjunctive (in a general sense) cannot be simplified in this manner, because the SAT solver would then have to prove unsatisfiability of a disjunctive term and so will likely use clauses from both top-level subterms of that subterm. This is not so bad since disjunctions and implications are not affected, and equivalences and conjunctions can be split on their conjunctive structure and proved separately, but it does impose a limit on usability.

We currently make no effort to simplify away unused definitions (i.e., definitions whose definitional variables did not occur in the core) that fall below (w.r.t. $\prec$) a maximal used definition. The intuition suggests this may not be possible, at least not without considering the SAT solver proof directly.

Finally, whenever the set V' is disconnected in the sense that the subterms corresponding to the definitional variables in V' are not connected by some operator in the original term, some of the top-level structure of t is necessarily lost and replaced by a flat conjunctive structure, deteriorating to more or less CNF in the worst case. We plan to address these shortcomings.

4.2 Terms

We have shown how the derivation of small unsatisfiable cores can help simplify theorems. As it stands, the method cannot be used to simplify terms that are not theorems, i.e., given a term t, prove $\vdash t \Leftrightarrow s$ where s is in some sense a simpler term. However, with a minor adaptation, almost the same process can sometimes succeed in doing so.

This time we convert t to definitional CNF without negating it first. This results in a theorem $\vdash t \Leftrightarrow \exists \bar{v} \in V.dCNF(t)$. Now we build the BDD of $dCNF(t)$ and read off the CNF structure of the BDD as outlined in §3, obtaining a CNF term which we shall call D. At this point, we use a SAT solver to confirm that $\vdash \mathit{dCNF}(t) \Leftrightarrow D$.

Intuitively, our hope is that the redundancy removal in the BDD will have the same effect as finding a smaller unsatisfiable core in the previous section. However, since t is not a theorem, the results we can achieve are slightly different.

Using the two theorems we have, we calculate as follows

$$\begin{array}{ll} & \vdash t \Leftrightarrow \exists \bar{v} \in V.D \text{ by transitivity of } \Leftrightarrow \\ \text{iff} & \vdash (\exists \bar{v} \in V.D) \Rightarrow t \\ \text{iff} & \vdash \forall \bar{v} \in V.D \Rightarrow t \end{array}$$

and then instantiate the $v \in V$ (which do not occur in t) and reverse construct the resulting term as in the previous section, to obtain a term s. It follows that $\vdash s \Rightarrow t$.

Now we check using a SAT solver whether $\vdash t \Rightarrow s$. If so, we have $\vdash t \Leftrightarrow s$ and we have successfully simplified t. If not, we have failed in the simplification attempt, but the resulting scenario has an application described in the next section.

We note in passing that this method can dispense with BDDs altogether by using the SAT solver based solution enumeration method described towards the end of §3. As then, this alternative should be considered for large terms only.

4.3 Goal-directed Proof

If the simplification attempt in the previous section fails, we have a term s such that $\vdash s \Rightarrow t$ and $\vdash \neg(t \Rightarrow s)$. In other words s is a stronger proposition. In theory, finding a stronger proposition is trivial: $\bot$ is the strongest proposition. In practice, finding a stronger proposition that retains some of the structure of the original term finds an obvious application in interactive proof.

In most interactive theorem provers, a proof begins by setting up as a *goal* the term that we hope to show is a theorem. Proof then proceeds in a "backwards" manner, by reducing the goal to simpler subgoals which we hope eventually to reduce to axioms (or ground rules) of the object logic. The state of a proof is represented by the outstanding subgoals, each of which can be represented as a two-sided sequent $\Gamma \vdash t$, where t is the subgoal and Γ is the set of assumptions to that subgoal. This is known as goal-directed proof. Describing this in any detail will take us too far afield. The interested reader may consult [10] which contains several examples of this style, or for that matter any tutorial introduction to a higher-order interactive prover.

One very useful rule of inference that is found in practically every logic is modus ponens, i.e.,

$$\frac{\Gamma \vdash p \qquad \Delta \vdash p \Rightarrow q}{\Gamma \cup \Delta \vdash q}$$

in the propositional two-sided incarnation. The backwards equivalent of this rule is effectively that if the goal is $\Gamma \vdash q$ and we have a theorem $\Delta \vdash p \Rightarrow q$ such that $\Delta \subseteq \Gamma$, then the goal can be reduced to $\Gamma \vdash p$.

We can think of our theorem $\vdash s \Rightarrow t$ as $\emptyset \vdash s \Rightarrow t$. Hence, given a propositional goal t, one way of simplifying it is to find a stronger but simpler term s. Note that if we had succeeded in our simplification attempt in the previous section, $\vdash t \Leftrightarrow s$ can also be used to reduce t. The point here is that even in the event of failure, the result that we do have is still of some use. In fact, the structure of s fits nicely into the scheme since it is just a conjunction of subterms of t, and so can be naturally split into subgoals.

5 Related Work

There is a large body of work on the use of BDDs in interactive provers. One of the earliest results combined higher-order logic with BDDs for symbolic trajectory evaluation [14]. A little later, temporal symbolic model checking was done in PVS [23]. These integrations trusted the underlying BDD engines. Around the same time, a serious attempt at using BDDs in an LCF-style manner [12] reported an approximate 100x slowdown. Later, a larger project added BDDs to the Coq theorem prover [28] and reported similar slowdowns, except that the faster programs were themselves extracted by reflection from the Coq representation, and could thus said to have higher assurance. The penalty for checking BDD proofs has thus more or less ensured that BDDs are not used internally by LCF-style theorem provers, in a non-trusted manner. There have of course been trusted integrations of BDDs with LCF-style provers [9], as well as verifications of aspects of BDD algorithms in such provers [17, 20].

This does not rule out the use of BDDs in interactive provers in general. BDDs are used in the ACL2 prover [16] to help with conditional rewriting (a BDD can be thought of as a nested conditional) and for deciding equality on bit vectors (see ACL2 System Documentation). The PVS theorem prover uses BDDs for propositional simplification [3]. This was in fact the inspiration for our work. Roughly speaking, when invoked on a goal with propositional structure, it uses BDDs to obtain the CNF of the goal, which is then used to split the goal into subgoals. Similar functionality can now easily be added to LCF-style provers using the method of §3, since the propositional structure of interactive goals is typically manageable by a BDD.

Integrations of SAT solvers with interactive provers has a shorter history. The integration is trivial for the case where the solver returns a satisfying assignment: we simply substitute the assignments into the input term and check that the resulting ground term evaluates to $\top$. This can be done efficiently. LCF-style integration of the unsatisfiability case had to wait for the arrival of proof producing SAT solvers [31]. The first such integrations were reported relatively recently [6, 29]. All LCF-style integrations that we know of so far, use the SAT solvers as one-shot decision procedures. Trusted integrations go further, such as the integration of PVS with the Yices Satisfiability-module-theories (SMT) solver. Work on LCF-style integrations with SMT solvers is underway [6].

6 Conclusions

We have shown how BDDs and SAT solvers can be used for fast normalisation, simplification of terms and theorems, and assistance with interactive proof. Even though we have restricted ourselves to propositional logic, the results can be applied to the propositional structure of more expressive logics via Skolemization, as is done in PVS. The results can also be extended directly to use SMT solvers [25] rather than SAT solvers, using the alternative solutions that avoid BDDs, and should allow us to do simplification in combinations of decidable theories.

Using BDDs to generate normal forms and checking the result is an obvious next step once a non-trusted proof producing SAT solver is available. To the best of our knowledge however, exploiting the non-occurrence of definitional variables in the results of BDDs and SAT solvers for the purposes of simplification, has not been done before. Perhaps because of this, the treatment has an ad hoc feel to it, and many opportunities for optimization exist.

We plan to use SAT solvers and BDDs in a more fine grained manner, perhaps in conjunction with the rewriting system of the theorem prover, as is done in ACL2. We expect to customize SAT solvers and unsatisfiable core finders, for instance to give special treatment to definitional variables. We also hope to generate proofs directly from BDDs, possibly using some of the techniques of [15]. These plans will form the initial steps for future research.

References

1. Henrik Reif Andersen. An introduction to binary decision diagrams. Available from `http://www.itu.dk/people/hra/bdd97.ps`, October 1997.
2. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
3. David Cyrluk, S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD)*, volume 901 of *LNCS*, pages 203–222. Springer, 1994.
4. Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
5. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
6. Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.
7. Roman Gershman, Maya Koifman, and Ofer Strichman. Deriving small unsatisfiable cores with dominators. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *LNCS*, pages 109–122. Springer, 2006.
8. M. J. C. Gordon. From LCF to HOL: a short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language and interaction: essays in honour of Robin Milner*, pages 169–185. MIT Press, 2000.

9. M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
10. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
11. Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions SAT solver and its application for reachability analysis. In Alan J. Hu and Andrew K. Martin, editors, *FMCAD*, volume 3312 of *LNCS*, pages 275–289. Springer, 2004.
12. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.
13. Gerard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 7.2. Technical Report RT-0256, INRIA, February 2002.
14. Jeffrey J. Joyce and Carl-Johan H. Seger. The HOL-Voss system : Model checking inside a general-purpose theorem prover. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 185–198. Springer, 1993.
15. T. Jussila, C. Sinz, and A. Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *9th Intl. Conf. on Theory and Applications of Satisfiability Testing*, LNCS. Springer, 2006. To appear.
16. Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.
17. Sava Krstic and John Matthews. Verifying BDD algorithms through monadic interpretation. In *VMCAI*, LNCS, pages 182–195. Springer, 2002.
18. David G. Mitchell. A SAT solver primer. In *EATCS Bulletin*, volume 85 of *The Logic in Computer Science Column*, pages 112–133. EATCS, February 2005.
19. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.
20. Veronika Ortner and Norbert Schirmer. Verification of BDD normalization. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 261–277. Springer, 2005.
21. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, jun 1992. Tool URL :`http://pvs.csl.sri.com`.
22. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
23. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking and automated proof checking. In Pierre Wolper, editor, *Proceedings of Computer Aided Verification*, volume 939 of *LNCS*, pages 84–97. Springer-Verlag, 1995.
24. Natarajan Shankar. Using decision procedures with a higher-order logic. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 5–26. Springer, 2001.
25. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *LNCS*, pages 500–504. Springer, July 2002.

26. G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation Of Reasoning: Classical Papers On Computational Logic, Vol. II, 1967-1970*, pages 466–483. Springer, 1983. Also in *Structures in Constructive Mathematics and Mathematical Logic Part II*, ed. A. O. Slisenko, 1968, pp. 115–125.
27. Vijay Vazirani. *Approximation Algorithms.* Springer, 2001.
28. Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Proc. 6th Asian Computing Science Conference (ASIAN'2000), Penang, Malaysia, Nov. 2000*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000.
29. Tjark Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. Technical Report PRG-RR-05-02, Oxford University Computing Laboratory, August 2005. TPHOLs - Track B.
30. Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *LNCS*, pages 482–489. Springer, 2005.
31. Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885. IEEE Computer Society, 2003.