Generating Simpler CNF Formulae for SAT-Based LTL Model Checking

Informatics Research Review

Gavin Keighren s9901726
g.keighren@sms.ed.ac.uk

Abstract. Since its introduction, Bounded Model Checking has been a motivating factor for the development of SAT solvers and considerable research has been directed towards them. However, while great advances have been made, other factors determining the overall runtime of the model-checking process have been afforded less attention. The process by which the CNF representation is obtained has a major effect on the time required to find a solution, as does the difficulty of the original propositional formula before this conversion. It has long been considered that the difficulty of a formula is directly related to its size, but it is only in recent times that ways of reducing this metric have been investigated within the framework of bounded model checking. This paper presents a comparison of some of this research, with the aim of highlighting its long overlooked importance, while noting that smaller formulae do not always require less time to solve.

1 Introduction

Model Checking is a technique for showing whether or not a model of a system satisfies particular properties. It is common that a model should satisfy properties of *liveness* (something good can always eventually happen) and *safety* (something bad should never happen). Systems are generally modelled, implicitly or explicitly, as having a collection of variables, with specific transitions determining how these variables change from one time step to the next. The states of the system correspond to every possible combination of values for those variables. That is, if the system is comprised of n Boolean variables,[1] then it will have 2^n potential states.

Properties that are to be checked are usually expressed in a temporal logic which captures the evolution of the model over different time steps. Computation Tree Logic (CTL) is used to express properties over *paths* in the model, and Linear Time Logic (LTL) is used to express properties over *states*.

Early techniques used explicit transition relations that defined a state and its successor. This was acceptable for systems that had few concurrent parts, but for systems with a large number of processes the number of states generated was too high. It was not until the introduction of symbolic model checking

[1] This is a common restriction in model checking, since finite value ranges can be represented by a series of Boolean variables.

[6] that models of a practical size could be checked. Instead of the transitions being expressed explicitly, Boolean formulae were used to capture the relationship between variables in a given state and its successor. These formulae were represented, and combined, using Bryant's ordered binary decision diagrams (OBDDs) [5]. This lead to the development of the model checker SMV by Kenneth McMillan as part of his PhD thesis from Carnegie Mellon University [16]. A comprehensive overview of model checking can be found in [9].

Biere et al. introduced Bounded Model Checking in [3] which encodes the model and the property as a Boolean satisfiabilty (SAT) problem. The SAT problem is satisfiable iff the property does not hold for a particular bound, k. That is, a satisfiable problem would correspond to the existence of a run over k time steps of the model that violated the property being checked. The application of SAT solvers to this problem meant that there was increased motivation to advance their development, since better SAT solvers could solve bigger problems in the same time. This has been one of the main areas of research with respect to the model checking process for some years now, and the state-of-the-art has recently been reviewed in [20].

Only recently have people been looking at the other factors which contribute to the runtime. It has long been assumed that the difficulty of a formula in conjunctive normal form[2] (CNF) is directly related to the number of clauses and variables in it. There is an intuitive logic to this: fewer variables mean a smaller search space, and fewer clauses mean a smaller overhead when propagating variable assignments. In practice, fewer clauses mean that the active part of the formula — those clauses which are currently affecting the search — is more likely to fit into the cache, thus dramatically reducing the time required to access them [21]. Additionally, for certain classes of property, the time required to generate the CNF formula is greater than the time spent in the solver. In such instances, the process by which the propositional formula that is subsequently converted into CNF is generated must be as fast as possible, as well as the actual conversion to CNF. The time to do so is usually measured as a function of the bound, since larger formulae are required to capture the evolution of the model over a greater number of time steps.

2 Preliminaries

In this paper the focus is on the use of LTL with past operators (PLTL) to specify properties, with the regular symbolic representation for the model (i.e. a propositional formula describing the transition relation and the initial states).

Given this framework, the two main steps involved in producing the formula to be checked are the conversion of the PLTL formula into a purely propositional formula given the bound, and the subsequent conversion of this into conjunctive normal form.

[2]The majority of SAT solvers expect the formula to be in this format.

2.1 PLTL

LTL formulae are built from propositional atoms (formulae comprising solely of Boolean connectives), with the future operators **X** (next), **F** (finally), **G** (globally), **U** (until) and **R** (releases) which have the following meanings. Given LTL formulae φ and ψ, $\mathbf{X}\varphi$ is true iff φ is true in the next time step, $\mathbf{F}\varphi$ is true iff φ is true at some future time step, $\mathbf{G}\varphi$ is true iff φ is true in all future time steps, $(\varphi\ \mathbf{U}\ \psi)$ is true iff ψ is true at some future time step and φ holds in all time steps up to that point, and $(\varphi\ \mathbf{R}\ \psi)$ is true iff ψ is true in all future time steps before φ is true. Note that if φ is never true then ψ must hold in all future time steps for $(\varphi\ \mathbf{R}\ \psi)$ to be true.

Taking π as the model, and $(\pi, i) \models \varphi$ to mean that φ holds in π at time i, the semantics of LTL can be defined recursively as follows:

$$
\begin{array}{lll}
(\pi,i) \models p & \textbf{iff} & p \text{ is a propositional atom, and holds in the state } (\pi,i) \\
(\pi,i) \models \neg\varphi & \textbf{iff} & (\pi,i) \not\models \varphi \\
(\pi,i) \models \varphi \vee \psi & \textbf{iff} & (\pi,i) \models \varphi \text{ or } (\pi,i) \models \psi \\
(\pi,i) \models \varphi \wedge \psi & \textbf{iff} & (\pi,i) \models \varphi \text{ and } (\pi,i) \models \psi \\
 & & \\
(\pi,i) \models \mathbf{X}\varphi & \textbf{iff} & (\pi,i+1) \models \varphi \\
(\pi,i) \models \mathbf{F}\varphi & \textbf{iff} & \exists j \geq i.\ (\pi,j) \models \varphi \\
(\pi,i) \models \mathbf{G}\varphi & \textbf{iff} & \forall j \geq i.\ (\pi,j) \models \varphi \\
(\pi,i) \models \varphi\mathbf{U}\psi & \textbf{iff} & \exists j \geq i.\ ((\pi,j) \models \psi \text{ and } \forall k : i \leq k < j.\ (\pi,k) \models \varphi) \\
(\pi,i) \models \varphi\mathbf{R}\psi & \textbf{iff} & \forall j \geq i.\ ((\pi,j) \models \psi \text{ or } \exists k : i \leq k < j.\ (\pi,k) \models \varphi)
\end{array}
$$

The past operators which extend LTL to PLTL are **Y** (yesterday), **Z** (similar to **Y**), **O** (once), **H** (historically), **S** (since) and **T** (trigger). These are temporal duals of the future operators and have the following meanings. At any non-initial time step, $\mathbf{Y}\varphi$ and $\mathbf{Z}\varphi$ are true iff φ was true in the previous time step. At time 0 $\mathbf{Y}\varphi$ is false whereas $\mathbf{Z}\varphi$ is true. $\mathbf{O}\varphi$ is true iff φ was true at some previous time step, $\mathbf{H}\varphi$ is true iff φ was true in all previous time steps, $(\varphi\ \mathbf{S}\ \psi)$ is true iff ψ was true at some previous time step and φ has been true in all time steps since then, and $(\varphi\ \mathbf{T}\ \psi)$ is true iff ψ holds in all previous time steps before one where φ is true. Like the future operator **R**, ψ has to hold in all previous time steps if φ has never been true for $(\varphi\ \mathbf{T}\ \psi)$ to hold.

The semantics of the past operators can be defined recursively as follows:

$$
\begin{array}{lll}
(\pi,i) \models \mathbf{Y}\varphi & \textbf{iff} & i > 0 \text{ and } (\pi,i-1) \models \varphi \\
(\pi,i) \models \mathbf{Z}\varphi & \textbf{iff} & i = 0 \text{ or } (\pi,i-1) \models \varphi \\
(\pi,i) \models \mathbf{O}\varphi & \textbf{iff} & \exists j \leq i.\ (\pi,j) \models \varphi \\
(\pi,i) \models \mathbf{H}\varphi & \textbf{iff} & \forall j \leq i.\ (\pi,j) \models \varphi \\
(\pi,i) \models \varphi\mathbf{S}\psi & \textbf{iff} & \exists j \leq i.\ ((\pi,j) \models \psi \text{ and } \forall k : j < k \leq i.\ (\pi,k) \models \varphi) \\
(\pi,i) \models \varphi\mathbf{T}\psi & \textbf{iff} & \forall j \leq i.\ ((\pi,j) \models \psi \text{ or } \exists k : j < k \leq i.\ (\pi,k) \models \varphi)
\end{array}
$$

The addition of past operators does not provide any more expressive power, but does allow for temporal properties to be given in an exponentially more succinct manner [13]. They also allow properties concerned with historical behaviour of the model to be written in a more natural form thus reducing the potential for them to contain errors.

2.2 Violating Paths

Given a property to be checked, the purpose of model checking is to look for a run, or path, which violates it. There are two types of violating paths: lasso shaped ones that capture infinite looping, and finite ones which are loop-free. Such paths are called *counter examples* as they provide a trace of a run where the property does not hold. If the violation of the property actually corresponds to a situation that you are looking for[3] then it is instead termed a *witness* as it shows the existence of what was being searched for.

Loop-free paths show the violation of properties which state that a particular thing must always hold (i.e. safety properties), since it is sufficient to give a path from an initial state to one in which the property does not hold. Lasso shaped paths, comprising of a finite initial portion and a series of states that repeat infinitely often, show the violation of properties which state that it must always be the case that a particular thing will eventually happen (i.e. liveness properties). Such a property is violated when a series in which it does not hold repeats infinitely often.

Bounded model checking places an upper limit, k, on the length of such paths. For loop-free paths this simply means that its length must be no more than k, but for lasso shaped paths, k must be at least as large as the number of steps to the second occurrence of the first repeated state in the loop. That is, the length of the finite initial portion plus the number of states in the loop must be less than k.

3 Conversion of PLTL Formulae

In this section, we look at the conversion of PLTL formulae into propositional formulae for a given bound, k. Two recent approaches to this problem are covered, the first by Cimatti et al. [8] and the other by Latvala et al. [15].

3.1 The Approach of Cimatti et al.

The conversion method presented in [8] assumes that the initial formula is in Negative Normal Form (NNF) which restricts negation to atomic variables only. Rewriting a PLTL formula into NNF can be carried out in linear time with respect to the size of the formula, and involves the repeated application of the following equivalences to push negation inwards:

$$\neg\mathbf{F}\varphi \equiv \mathbf{G}\neg\varphi \qquad \neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi \qquad \neg(\varphi\mathbf{U}\psi) \equiv \neg\varphi\mathbf{R}\neg\psi$$
$$\neg\mathbf{G}\varphi \equiv \mathbf{F}\neg\varphi \qquad\qquad\qquad \neg(\varphi\mathbf{R}\psi) \equiv \neg\varphi\mathbf{U}\neg\psi$$

$$\neg\mathbf{O}\varphi \equiv \mathbf{H}\neg\varphi \qquad \neg\mathbf{Y}\varphi \equiv \mathbf{Z}\neg\varphi \qquad \neg(\varphi\mathbf{S}\psi) \equiv \neg\varphi\mathbf{T}\neg\psi$$
$$\neg\mathbf{H}\varphi \equiv \mathbf{O}\neg\varphi \qquad \neg\mathbf{Z}\varphi \equiv \mathbf{Y}\neg\varphi \qquad \neg(\varphi\mathbf{T}\psi) \equiv \neg\varphi\mathbf{S}\neg\psi$$

$$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi \qquad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi \qquad \neg(\varphi \leftrightarrow \psi) \equiv \varphi \leftrightarrow \neg\psi$$

[3] This is the case when planning is done via model checking, and the property is the negation of what constitutes a solution.

The equivalences involving $\mathbf{Y}$ and $\mathbf{Z}$ are such, due to their semantics at time zero. Additionally, note that $\varphi \rightarrow \psi$ can be written as $\neg\varphi \vee \psi$.

The first step in their procedure is to transform the PLTL formula into Separated Normal Form (SNF) based on the procedure described in [10], which is a conjunction of implications that relate past time formulae to future time ones:

$$\mathbf{G}\left(\bigwedge_i (P_i \rightarrow F_i)\right)$$

The implications, also called *rules*, are split into four different classes referred to as start, invariant, next and eventuality rules:

$$\mathbf{start} \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{X}\bigvee_j l_j \qquad \bigwedge_i l_i \rightarrow \mathbf{F}\bigvee_j l_j$$

where **start** is true for the initial states at time zero and false otherwise. Precise transformation rules are given for all possible PLTL formulae, as well as a description of cases where the rules cannot be applied. Very little motivation is given as regards why a conversion to SNF has been used, except that the variables introduced 'can be seen as defining the transition relation of an observer automaton', and that, in the bounded case, the eventuality rules 'can be expressed with a fix-point construction' which allows for a better representation.

Two versions of the transformation rules are given for the final step of obtaining purely propositional formulae from the SNF rules. The first are for when the desired loop length is known, which produces a number of propositional formulae that is $O(n \cdot k^2)$, when all loop lengths of $0..k$ are considered. With these rules the number of variables in the encoding is $O((|\mathcal{A}| + n) \cdot k)$, where $|\mathcal{A}|$ is the number of atomic propositions in the original PLTL formula, and n is the number of temporal operators present. The second version of the rules are the result of loop independence optimisation, by reformulating them and introducing variables that determine what the loop length is (one for each possible loop length). A result of this is that the number of propositional formulae produced is $O(n \cdot k)$.

The only stage that is not covered is the conversion of propositional formulae into CNF, however they do correctly state that this step can also have a dramatic effect on how long the resulting formula takes to be processed by a SAT solver. One of the major problems with existing CNF conversion algorithms is that, in order for them to run in linear time, they introduce a considerable number of new variables.

The experimental results for this approach support their initial claim that the new encoding is superior to the usual encoding for PLTL described in [2]. It is shown that the usual encoding suffers badly for formulae with a large number of nested PLTL operators, whereas their encoding does not, a result that they attribute to its bounded nature. While for some simple properties their method is slightly slower, it does appear to be faster for more complicated instances.

A downside to the proposed encoding is that the shortest counter example, or witness, is not always found. This means that higher bounds have to be checked, although on the evidence of the test results, the runtime is still better.

3.2 The Approach of Latvala et al.

The method presented in [15] builds upon their earlier work for encoding pure LTL given in [14] to accommodate past operators. It assumes that the formula is in Positive Normal Form (PNF), where negations apply to propositional formulae only. This can be done in linear time with respect to the number of PLTL operators in the formula, but because the negations are not pushed right down to the atomic variables, it takes less time than is required to convert the formula into NNF.

Intuitively, the conversion process described involves making copies of the initial k step paths in the model. The number of copies required is determined by the maximum number of nested past operators $\delta(\varphi)$ of the formula (termed the past operator depth). The reason for this limit is that the truth of a formula in the i^{th} iteration of a loop, where $i > \delta(\varphi)$ is the same as for in iteration $\delta(\varphi)$ since it can only 'look back' as far as $\delta(\varphi)$ loops. This was originally shown in [13] and later independently in [2].

In the paper by Cimatti et al. they cited the ability to utilise the fix-point construction for the eventuality rules as a motivating factor for using SNF as an intermediary translation, however fix-points are also used in the approach by Latvala et al.

The conversion rules are presented and described, but as with the paper by Cimatti et al. they do not discuss the subsequent required conversion from propositional formulae into CNF. Copies of the model variables present in the property are made for each time step for which they are required, with k variables also being introduced to determine the loop length. Due to the loop unrolling process, the transition relation of the model also has to be unrolled by $k \cdot \delta(\varphi)$ steps. The size of the final formula is $O(|l| + k \cdot |T| + k \cdot \delta(\varphi) \cdot |\varphi|)$, where l is the formula which determines the length of the loop, T is the transition relation and φ is the property being checked. Since $\delta(\varphi)$ can be $O(|\varphi|)$, this is quadratic in the formula size in the worst case. However, it is still linear with respect to the bound which they point out is often the most important factor when searching for long counter examples.

Latvala et al. compare their new encoding with the same one as Cimatti et al. (used in NuSMV 2 [7]) due to the fact that it is currently the only PLTL conversion process which has been implemented in a published bounded model checker. The focus of their testing is on the *scalability* of the encoding and, from the results, the growth rate does appear to be linear with respect to the bound. Only the runtime appears to grow at a (marginally) faster rate which could potentially be due to the increased formula size, rather than any increased complexity. This behaviour was also exhibited for real-life examples, with the linear growth appearing to hold except for the runtime which again grows at a slightly faster rate. They also find that the usual encoding suffers badly for

formulae with a high number of nested past operators.

Having only been made aware of the paper by Cimatti et al. towards the end of their work, they provide a short comparison, stating that it is less efficient at least in cases where size of the transition relation is larger than size of the property. Additionally, the method given by Latvala et al. will often find a shorter counter example.

3.3 Discussion

Both methods successfully reduce the size of the propositional formula produced by the conversion from PLTL, which in turn reduces the overall runtime. The biggest problem with the usual encoding method is that nested past operators cause a severe blow-up in the size of the formula, and this has been eliminated by both approaches. As they tackle the problem from completely different angles, it is hard to see how the results from one could benefit the other. However, it is important that further testing with harder problem instances is carried out, thus providing a much better indication of each method's strengths and weaknesses. Furthermore, applying the two processes to the same problem instances is paramount in determining how they compare to each other.

4 CNF Conversion

In this section, we look at a method of converting a propositional formula into CNF that keeps the number of clauses to a minimum as well as introducing as few new variables as possible [12]. The authors compare it to the structure-preserving method of Plaisted and Greenbaum [19], and to the optimal encoding presented by Thierry Boy de la Tour in [4].

In the paper by Jackson and Sheridan, the starting point is a representation of Boolean formulae called Reduced Boolean Circuits (RBCs) that were proposed by Abdulla, Bjesse and Eén in [1]. An RBC is a Directed Acyclic Graph (DAG) where each vertex is either the equivalence or conjunction operator, and the incoming edge determines whether or not the sub-formula rooted at that vertex is negated or not. A consequence of this structure is that the root vertex has an incoming edge which allows for the overall formula to be represented positively or negatively. By imposing a total order on the vertices, RBCs provide a canonical form for propositional formulae. It is straight-forward to build the RBC for a propositional formula and can be done in linear time with respect to the size of the formula.

The paper presents formulations of three well-known conversion procedures such that they are defined over RBCs. Note that equivalences can be represented as a disjunction of two conjunctions, and disjunctions are just negated conjunctions (with negated operands) as shown here:

$$x \leftrightarrow y \equiv (x \wedge y) \vee (\neg x \wedge \neg y) \qquad \neg(x \leftrightarrow y) \equiv (x \wedge \neg y) \vee (\neg x \wedge y) \qquad x \vee y \equiv \neg(\neg x \wedge \neg y)$$

As such, we can, if required, only consider the conjunction operator, and whether the formula is positive or negative.

4.1 Standard CNF Conversion

The first algorithm presented does not carry out any renaming of sub-formulae, and thus does not introduce any new variables. The drawback though is that it can produce an exponential number of clauses.

Leaf vertices in the RBC (corresponding to atomic propositions) are represented by a single clause containing the variable with the same polarity of the edge that leads to it. For all other vertices, the two clause sets are combined as follows, depending on whether the edge coming into the vertex is positive or negative. If the incoming edge is positive then the result is simply the union of the two sets, if it is negative then the result is the cross-product.

4.2 Structure Preserving Conversion

The second algorithm presented is the *structure-preserving* method given in [19] where each sub-formula is renamed with a new variable. For example, in the formula $(a \wedge b \wedge c) \vee (d \wedge e \wedge f)$ the left-hand side can be renamed to give the following *equi-satisfiable* formula:

$$x_{a \wedge b \wedge c} \vee (d \wedge e \wedge f) \quad \wedge \quad x_{a \wedge b \wedge c} \leftrightarrow (a \wedge b \wedge c)$$

In fact, such an equivalence can be replaced by just an implication if the vertex representing the sub-formula only has positive or negative edges leading to it, with the direction corresponding to the polarity of those edges. The only exception is where it is a sub-formula of an equivalence node, since such formulae are actually referenced in both polarities, regardless of the incoming edges. This procedure introduces one new variable for every vertex in the RBC, which dramatically increases the potential search space of the formula produced. Furthermore, the clauses produced only ever contain two or three literals — one for the vertex of the sub-formula being renamed, and one for each of its two children.

Given RBC vertices x and y representing $(a \wedge b)$ and $(c \leftrightarrow d)$ respectively (i.e. $(a \wedge b)$ has been renamed as x and $(c \leftrightarrow d)$ has been renamed as y), this algorithm produces the following clauses:

$$x \leftrightarrow (a \wedge b) \textbf{ produces } \{(x \vee \neg a \vee \neg b) \textbf{ , } (\neg x \vee a) \textbf{ , } (\neg x \vee b)\}$$
$$y \leftrightarrow (c \leftrightarrow d) \textbf{ produces } \{(y \vee c \vee d) \textbf{ , } (y \vee \neg c \vee \neg d) \textbf{ , } (\neg y \vee c \vee \neg d) \textbf{ , } (\neg y \vee \neg c \vee d)\}$$

Note that the truth assignments of the introduced variables will always follow from a complete assignment to the variables in the original formula. Since the algorithm just visits each node in the RBC once, its runtime is $O(|V|)$ where $|V|$ is the total number of vertices.

The other algorithms only differ to this method in respect to deciding which sub-formulae should be renamed. In their case, the children of an RBC node can be sets of clauses, not just variables.

4.3 Boy de la Tour Conversion

The third procedure is the encoding given by Boy de la Tour in [4] where the renaming of sub-formulae is only carried out when it would produce less clauses than if no renaming were to be done. If a new variable is not introduced then the sub-formulae are combined as described by the standard conversion algorithm, above. For a given vertex, the number of clauses that would be produced by each option can be calculated in $O(|V|)$ time, giving the overall algorithm a runtime that is $O(|V^2|)$. While this is worse than the structure preserving algorithm, which is linear in the number of vertices, it was shown to be optimal for formulae that do not contain equivalences. Unfortunately, numbers involved in the calculations can easily become too large, making the algorithm impractical in reality. Nonnengart et al. reformulated part of the algorithm to avoid this problem [17] but Jackson and Sheridan suggest that its complexity hinders attempts to correctly implement it.

4.4 Compact Conversion

Jackson and Sheridan's algorithm[4] generates the clauses from the leaves of the RBC upwards, after having initially determined the positive and negative reference counts for the individual vertices.

Because the algorithm works from the leafs of the RBC up to the root, the number of clauses that will be produced for a given vertex can easily be determined. If no renaming is done and the vertex is a positive conjunction then number of clauses generated will just be the total generated for the two child vertices. If the vertex is a negative conjunction (i.e. a disjunction) then the result is the product of the number of clauses generated for each child. If one of the sub-formulae is renamed, then the total number of clauses that will be produced is also just the total number produced for both children.

The calculations that determine whether a sub-formula should be renamed or not assume that each vertex only has one edge coming into it. This is not the case with RBCs that share sub-formulae, so the calculations have to be modified to handle them. Sub-formulae have the potential to be referenced both positively *and* negatively, with each producing a different set of clauses. As such, there are up to four possible pairs of clause sets that can be combined for a given vertex, depending upon the connective and polarity. Additionally, each pair of clause sets can be combined with any number of other pairs. For the conversion to remain optimal, sub-formulae which are referenced more than once in the same polarity have to be renamed if more than one new clause is generated by

[4]They have termed it the *compact* conversion since it produces less clauses than the structure preserving method, and requires less code to implement than the version of Boy de la Tour's algorithm given by Nonnengart et al.

not renaming it. Otherwise, the calculation used in the non-sharing case can be used.

The algorithm requires just one pass of the RBC from bottom to top, and no extra work is required at each vertex to work out whether a sub-formulae should be renamed or not, therefore the overall runtime is $O(|V|)$.

The results presented in their paper show that, on average, the number of clauses produced is about two thirds less than with the standard conversion procedure, and about one tenth less than with the structure preserving method (where implications never replace equivalences for sub-formula referenced in only one polarity). The models used for testing are industrial benchmarks, so the results have some relevance to practical situations. It is, however, the solving time that we wish to reduce, and in a number of cases this happens, but for a few the time required actually increases. Further tests need to be carried out to determine if this difference is due to the CNF formula being harder, or if it is a result of the variable ordering heuristics used by the solver. Either way, it shows that simply reducing the number of clauses and variables in the formula does not guarantee that it will take less time to find a solution.

5 Conclusions

For bounded model checking, the number of variables in the CNF formula produced must be at least $k \cdot |M|$, where $|M|$ is the number of variables in the model[5] since each variable must be given a value at each time step to specify a unique counter example. The aim is to generate a set of clauses that allows the values of these variables to be determined as easily and quickly as possible. It is commonly thought that this means producing formulae with as few clauses and variables as possible. There is a trade-off between these two goals however, as is shown by the standard CNF conversion process — no extra variables are introduced, but there may be an exponential number of clauses. Due to the ability for modern SAT solvers to cope with millions of variables, the focus has been on simply reducing the number of clauses in the formula, the reasoning being that it will be easier to solve due to the fewer constraints present. While this tactic works in certain circumstances, it is not guaranteed to do so in all cases. As mentioned earlier, the structure-preserving algorithm only produces clauses that contain two or three literals with the introduced variables appearing in a very small number of those clauses. The compact method however, generates significantly longer clauses with each introduced variable present in a higher percentage of them. The effect that these differences have on the solving process is rarely considered, if at all, and a better understanding of them may yield better heuristics for generic CNF conversion algorithms.

The task of generating formula which are 'easier' to solve is somewhat more difficult, due in no small part to a lack of understanding about what exactly makes one representation of a formula 'harder' than another one. One reason

[5]This is not strictly true since cone of influence (COI) reduction can remove those variables that have no effect on the ones in the property being checked.

may be that the exact relationships between the variables are not captured as succinctly in the 'harder' version, resulting in portions of the search space that do not lead to a solution being cut off later than they should be. Recently, Ostrowski et al. presented a method for obtaining information about such relationships from general CNF formulae, and showed that it could dramatically reduce the time required to find a solution [18]. This suggests that it is indeed the representation of relationships between the variables that determine the difficulty of a particular formula.

Both papers presented that deal with conversion from PLTL to propositional formulae show a speed-up in the solving time, which suggests that the encodings are somehow 'better' than traditional methods. So far the only guide is that smaller (fewer connectives) is better — which also lends weight to the hypothesis that it is really due to the relationship between the variables being captured more succinctly. The SNF representation for PLTL formulae is designed to capture the relationship between model variables across different time steps, and thus appears to be a good initial step in producing simpler formulae. If this is indeed the case then the representation of the transition relation will also play an important part in how long it takes to solve the resulting formula. In the context of bounded model checking, larger and larger formulae are successively checked as the bound increases, so it is also important that the formula size grows as slowly as possible with respect it. Both methods achieve this with linear growth, improving over the apparent $O(k^3)$ growth of the algorithm given in [2] that Latvala et al. observed in their experiments.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic Reachability Analysis Based on SAT-Solvers. In S. Graf and M. I. Schwartzbach, editors, *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 411–425. Springer-Verlag, Mar. 2000.

[2] M. Benedetti and A. Cimatti. Bounded Model Checking for Past LTL. In H. Garavel and J. Hatcliff, editors, *TACAS '03: Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33. Springer-Verlag, Apr. 2003.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, Mar. 1999.

[4] T. Boy de la Tour. An Optimality Result for Clause Form Translation. *Journal of Symbolic Computation*, 14(4):283–302, Oct. 1992.

[5] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, June 1990.

[7] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Check-

ing. In E. Brinksma and K. G. Larsen, editors, *Proceedings of Computer Aided Verification: 14th International Conference (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, July 2002.

[8] A. Cimatti, M. Roveri, and D. Sheridan. Bounded Verification of Past LTL. In Hu and Martin [11], pages 245–259.

[9] E. M. Clarke, Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] A. Frisch, D. Sheridan, and T. Walsh. A Fixpoint Based Encoding for Bounded Model Checking. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD 2002)*, volume 2517 of *Lecture Notes in Computer Science*, pages 238–254. Springer-Verlag, Nov. 2002.

[11] A. J. Hu and A. K. Martin, editors. *Formal Methods in Computer-Aided Design, 5th International Conference (FMCAD 2004)*, volume 3312 of *Lecture Notes in Computer Science*. Springer-Verlag, Nov. 2004.

[12] P. Jackson and D. Sheridan. The Optimality of a Fast CNF Conversion and its Use with SAT. Technical Report APES-82-2002, APES Research Group, Mar. 2004. Available from `http://www.dcs.st-and.ac.uk/~apes/apesreports.html`.

[13] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal Logic with Forgettable Past. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 383–392. IEEE Computer Society Press, July 2002.

[14] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple Bounded LTL Model Checking. In Hu and Martin [11], pages 186–200.

[15] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is Better: Efficient Bounded Model Checking for Past LTL. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 380–395. Springer-Verlag, Jan. 2005.

[16] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, May 1992.

[17] A. Nonnengart, G. Rock, and C. Weidenback. On Generating Small Clause Normal Forms. In C. Kirchner and H. Kirchner, editors, *CADE-15: Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 397–411. Springer-Verlag, July 1998.

[18] R. Ostrowski, E. Grégoire, B. Mazure, and L. Saïs. Recovering and Exploiting Structural Knowledge from CNF Formulas. In P. van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer-Verlag, Sept. 2002.

[19] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, Sept. 1986.

[20] M. R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(2):156–173, Apr. 2005.

[21] L. Zhang and S. Malik. Cache Performance of SAT Solvers: a Case Study for Efficient Implementaion of Algorithms. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 287–298. Springer-Verlag, May 2003.