Successful SAT Encoding Techniques

Magnus Björk

25th July 2009

Abstract

This article identifies good practices for SAT encodings by analysing interviews with a number of well known SAT experts. The purpose is both to determine the confidence in different encoding strategies, by analysing whether there is consensus among the experts or not, as well as bringing out hidden knowledge to SAT users.

There is consensus that encoding techniques usually have a dramatic impact on the efficiency of the SAT solver, that it often takes much work to find a good encoding, and that the size of an encoding is only very loosely related to the hardness of finding a solution. Topics where the interviewees disagree include the feasibility of including arithmetics in SAT problems and whether to formulate problems as clauses or circuits.

The article describes a number of strategies that are good in different situations, such as different ways to represent numbers and how to use incrementality.

1 Introduction

1.1 Background and Approach

During the last decades, SAT solvers have been successfully used to solve a multitude of real world problems. The boolean satisfiability problem, SAT, is so central that there are devoted conferences, journals and competitions organized on regular basis. The applications range over diverse subjects like configuration management, formal verification of hardware and software, first order theorem proving, and power estimations for digital circuits. A number of powerful open source SAT solvers are available for anybody to use.

These SAT solvers are typically easy to operate, and since the input language is very small it often does not take long for a beginner to start to use a SAT solver for simple problems. However, as problems grow, so does the runtime of the solver, and soon the user has to make decisions about how to encode the problem which may have impact on the solver's ability to find a solution.

[0]The work for this article was mainly conducted during the author's employment at Chalmers University of Technology

Finding a SAT encoding for a given problem may be easy, but finding an efficient encoding is much harder.

Learning what constitutes good SAT encodings is a threshold for SAT users to overcome. Literature searches suggest that there are little material published to help users with this. Papers describing encoding techniques are often hard to find, since are typically categorized under a specific application, even though they contain more general tecniques. Some good starting points for reading about SAT encodings are [MS08], [Pre09], [CESS08], [ES05] and [Eén07].

The hypothesis for this article is that there is a lot of hidden knowledge among the experts. The purpose of the article is to make that knowledge public. The approach chosen is to interview a number of experts about SAT encodings, and then summarize their answers. Furthermore, the answers from the interviewees are compared, to see whether there is agreement. Thereby, it should be possible to identify whether a topic is well understood, or open for interpretation. The article is intended to be written in such a way that also people with little experience of SAT can benefit from it.

1.2 The Interviewees

A list of suggested interviewees were put together, containing people whose names are often seen in connection with SAT (authors of related papers or program committee members for related conferences and journals). These people were contacted, and slightly less than half agreed to participate in the study. The ones who agreed are (in alphabetical order):

- **Jason Baumgartner, Bob Kanzelman, and Hari Mony** are members of IBM EDA's formal verification research and development team, that has been using SAT for a long time: notably for combinational equivalence checking (using IBM's "Verity"), as well as property checking and sequential equivalence checking (using IBM's "SixthSense"). Baumgartner is the technical lead of IBM EDA's formal verification efforts. Kanzelman is the primary owner of the SAT solver used by these tools. Mony has worked extensively on the efficient integration of this SAT solver within SixthSense.

- **Armin Biere** is a professor at the Johannes Kepler University in Linz, Austria. He has published numerous SAT-related papers, organized both SAT competitions and conferences, and is an editor for the Journal on Satisfiability, Boolean Modeling and Computation (JSAT) and of the Handbook of Satisfiability. Biere has written the SAT solver PicoSat.

- **Koen Claessen** is an associate professor at Chalmers University of Technology in Gothenburg, Sweden, and has also worked for both Jasper Design Automation and Prover Technology. Claessen has used SAT solvers in many projects, including the first order theorem prover Equinox [Cla05] and model finder Paradox [CS03].

- **Joao Marques-Silva** is a professor at University College Dublin, Ireland. Apart from publishing many SAT papers, he is an editor for JSAT and has organized e.g. the SAT 2007 conference.
- **Robert Nieuwenhuis** is a professor at the Technical University of Catalonia in Barcelona, Spain. He has participated in the development of the Barcelogic SAT solver, and has applied SAT in many problems. Nieuwenhuis has also organized several related conferences, and is an editor of the Journal of Automated Reasoning.
- **Niklas Sörensson** developed MiniSat together with Niklas Een, mainly during their PhD studies at Chalmers University of Technology, and has applied SAT for various problems, including model checking, temporal induction, and model finding.

1.3 SAT Algorithms Represented

The large majority of the interviewees use various *conflict driven, clause learning* (CDCL) SAT solvers. These are often referred to as *DPLL* style solvers after [DP60] and [DLL62]. Even though CDCL is fundamentally different from DPLL, the name DPLL is normally used for CDCL for historical reasons. MiniSat, PicoSat, and BarceLogic are all examples of CDCL SAT solvers. Baumgartner et al use a hybrid circuit SAT solver, that operates directly on circuits [1] (or rather and-inverter graphs), and records learned information as clauses.

It is not within the scope of this article to go into the details of the SAT algorithms, but good introductions can be found for example in [Eén07] and [CESS08].

2 Fundamentals of SAT Encodings

2.1 Importance of Good Encoding

There is consensus that the encoding have big impact on the runtime of the SAT solver. Ultimately, the quality of the encoding often determines whether the problem is solvable or not. Only very easy problem can be solved with naive encodings. Several interviewees say that finding a good encoding is a hard task that takes much time. Furthermore, there do not seem to be any techniques that work well for all problems, so different approaches have to be tried for each problem.

Several interviewees say that automated pre-simplification techniques can give you some advantage, especially for naive encodings. However, automated techniques cannot match carefully crafted encodings, so spending time on the encoding is important, according to the interviewees. Baumgartner et al point

[1] In this context, a circuit is a formula with subformula sharing, also known as a DAG (Directed Acyclic Graph).

out that since their main application is hardware verification, much of their encoding is already given, but they believe strongly in powerful pre-simplification techniques.

2.2 How to Find a Good Encoding

The interviewees agree that finding a good encoding of a problem usually takes much effort. Different techniques work well in different situations, so it is hard to know how to encode a certain problem until various strategies have been tried out. Several of the experts suggest starting with a very simple encoding. If it happens to work, then you're fine, and have not wasted effort. If it does not work, you have still usually learned something about the problem, which is important.

Improving the encoding is often an iterative process of trial and error — pick something that can be encoded in several ways or that is obviously poorly encoded in the current encoding, make another encoding and observe how this affects the run time. Keep the changes that have a positive impact, and throw away the negative ones.

Claessen suggests some points to think about to get started: What information do you want to get out of the solution? What decisions do you need to make to find a solution? What important information is there in your input? The things you come up with are good candidates for variables in your encoding. Then use constraints to relate the variables to each other.

Marques-Silva stresses that to find a good encoding, it is essential to know the problem domain well. He mentions, for example, that if solutions have a known lower bound, then including that as a constraint in the encoding can improve the efficiency substantially; then the SAT solver does not waste time looking for solutions that are smaller than the bound. Or if it is known that a problem has many symmetrical solutions, then one can add constraints that forbid some of them, which improves efficiency in some cases. Marques-Silva takes as an example his work on haplotype inference [LMS06] where the first encoding was slower than anything that existed at the time. The encoding was then improved mainly by learning about the problem domain and using that knowledge in the encoding. After three month, the encoding had been improved so much that solutions were obtained orders of magnitudes faster than with previously known algorithms.

2.3 Different SAT Solvers and Algorithms

Most interviewees have seen variation in how different solvers handle identical problems, but there does not seem to be any good guidelines for how to select an appropriate solver for a given problem. Baumgartner et al say that they have seen significant performance differences between different solvers on the same problem class, and even on different problems in the same class. They are currently investigating this topic further.

So, although there are differences, it is hard to pick a solver based on the problem class. Claessen says that it is important to know the basic propagation behaviour of the solver, and produce encodings that take advantage of that behaviour.

Most of the interviewees use conflict driven clause learning solvers, and say that they usually work very well for diverse applications. Biere and Marques-Silva mention two very special applications where such solvers are known to perform poorly: pigeonhole problems, and cryptography problems. But as long as one is not trying to solve such a problem, standard off-the-shelf solvers seem to be a good choice.

One obvious difference between solvers is whether they are based on clauses or circuits. Among the interviewees, Baumgartner et al are the only ones using a circuit based solver, and claim that it works very well for hardware verification problems, where the problems are given in circuit representation. They also point out that circuit based solvers can easily integrate with circuit-based rewriting transformation without complex two-way conversions, and have advantages for incrementality. Rewrite according to mail? Sörensson, on the other hand, prefers the CNF representation, since it gives more opportunities for refactoring independently of the circuit structure. Claessen mentions that the CNF based solver MiniSat [ES03b] has won CircuitSAT competitions[2], and does not think that there are convincing arguments for circuit based solvers.

2.4 Propagation Behaviour

Each theorem proving method has some notion of propagation — that is, the simplest way of deriving facts. For DPLL/CDCL based solvers this is unit propagation. It is important to formulate SAT problems in such a way that propagation works as efficiently as possible.

Consider, for instance an if-then-else construction[3] (also called a *mux*): $ITE(I,T,E)$ assumes the value of T if I is true, and the value of E if I is false. The expression $x = ITE(I,T,E)$, can be represented by the following clause set: $\{(\neg I|\neg T|x),(\neg I|T|\neg x),(I|\neg E|x),(I|E|\neg x)\}$.

Although this representation is complete, it is not optimal for unit propagation. Assume that both T and E are known to be true. Unit propagation then removes the clauses containing positive occurrences of T and E, and removes $\neg T$ and $\neg E$ from the clauses in which they occur. Thus, the following two clauses remain: $\{(\neg I|x),(I|x)\}$. It is easy to see that these two clauses together imply that x is true, but it cannot be derived by unit propagation. However, if the two clauses $(\neg T|\neg E|x)$ and $(T|E|\neg x)$ are added to the original representation, unit propagation is powerful enough to derive x from T and E (and $\neg x$ from $\neg T$ and $\neg E$).

Another similar example is the half adder, where extra clauses can be added to propagate that if the sum and carry are both low, then both inputs are also

[2] MiniSat++ (which uses MiniSat as a back end) won the AIG track of the SAT-Race 2008. See http://baldur.iti.uka.de/sat-race-2008/.

[3] Thanks to Sörensson and Claessen for this example.

low. These examples highlight cases where bigger encodings may be more efficient than small ones, supporting the stance that problem hardness is only loosely correlated to problem size. With the extra clauses, the mentioned encodings are said to be *arc consistent*. This means that all information that can be deduced from partial assignments in the original circuits can be deduced with unit propagation in the encoding under the same assignments.

Sörensson remarks that intuitively, adding redundant clauses like this is often good, but perhaps not always. In case it causes the problem to grow significantly, it may not be worth the effort. For maximal benefit, one should try to understand whether the situation that the redundant clauses address is likely to happen. Sörensson also points out that the redundant clause $(\neg T|\neg E|x)$ above is actually a resolvent of the two original clauses $(\neg I|\neg T|x)$ and $(I|\neg E|x)$, with respect to I. During preprocessing, it is possible to add resolvents to any pairwise resolvable clauses; the hard part is to know whether it is beneficial or will just swamp the solver with irrelevant clauses. Sörensson hints that resolution of two clauses $(x|C_1)$ and $(\neg x|C_2)$, where C_1 and C_2 are sets of literals, is potentially beneficial for unit propagation if and only if C_1 and C_2 are not disjoint (in the if-then-else example, x occurs in both original clauses).

3 Clauses and Circuits

3.1 Thinking in Circuits or Clauses

With CNF based solvers, one have the option of first formulating the problem as circuits that are converted to CNF, or to express the problem as clauses immediately. There is no consensus about which of these approaches to chose, and it seems to be a matter of taste. Nieuwenhuis is the only interviewee who says he uses CNF exclusively. Biere and Marques-Silva think circuits are generally preferable, since the circuit representation is often easier to connect to the problem domain. Claessen suggests choosing the representation that feels most comfortable, and points out that constraints are often conveniently expressed as clauses. He also mentions that MiniSat now has accompanying libraries for circuit translation that lets the user decide when the actual translation occurs, which makes it easier to mix representations. Sörensson also suggests using the most convenient representation, but in cases where there is no obvious difference he recommends clauses, since it gives precise control of the coding.

3.2 Translating Circuits to CNF

Translating circuits into logically equivalent clause sets is subject to an exponential blowup, and thereby not feasible. Instead, circuits are usually translated into clause sets where new variables are introduced for individual circuit nodes. There is consensus that these extra variables impose a much smaller overhead than the exponential blowup in size. Normally, the encodings are done in such a way that a satisfying assignment to the clause set is also a satisfying assignment

to the original circuit.

The most well known CNF translation was introduced by Tseitin [Tse68]. For each non-atomic subformula A of the original problem, Tseitin's algorithm invents a corresponding variable x_A. Thereby, the problem can be partitioned so each clause only relates neighbours in the circuit. For example, disjunctions are translated as $\mathit{Tseitin}(A \vee B) = \mathit{Tseitin}(A) \cup \mathit{Tseitin}(B) \cup \{(x_{A\vee B}|\neg x_A), (x_{A\vee B}|\neg x_B), (\neg x_{A\vee B}|x_A|x_B)\}$. On the top level, a formula A is represented by the clause set $\{(x_A)\} \cup \mathit{Tseitin}(A)$. Some simple improvements to the algorithm include recognizing larger conjunctions and disjunctions, as well as eliminating negations by keeping track of whether each variable represents a certain subcircuit or its negation.

Two decades after Tseitin's invention, Plaisted and Greenbaum presented a modified algorithm that essentially produces a subset of Tseitin's representation [PG86]. Plaisted and Greenbaum noticed that by keeping track of polarities, one can remove large parts of the Tseitin translation. For instance, if $A \wedge B$ is satisfiable, then both A and B must be satisfiable. This is represented by the two clauses $(\neg x^+_{A\wedge B}|x^+_A)$ and $(\neg x^+_{A\wedge B}|x^+_B)$. Intuitively the variable x^+_A encodes that A is satisfiable, while x^-_A means that $\neg A$ is satisfiable. Conversely, if $\neg(A \wedge B)$ is satisfiable, then one of $\neg A$ or $\neg B$ is satisfiable, which is represented by $(\neg x^-_{A\wedge B}|x^-_A|x^-_B)$. Plaisted and Greenbaum's algorithm produces a smaller encoding than Tseitin's, but it is not certain whether it is more efficient for SAT solving [Eén07], since it may have worse propagation behaviour.

More recent work includes the methods by Boy de la Tour [dlT92], Jackson and Sheridan [JS04], and Manolis and Vroon [MV07]. All of these methods create intermediate representations with different set of connectives, and apply transformation rules to simplify the intermediate representation. Een, Mishchenko, and Sörensson applied logic mapping techniques normally used for hardware synthesis to produce efficient CNF representation [EMS07].

The interviewees agree that Tseitin is usually good enough, and a good choice for a first try. Several interviewees mention that intermediate representations are useful: Biere and Claessen prefer and-inverter graphs, AIGs, and Marques-Silva prefers reduced boolean circuits, RBCs [ABE00]. Neither of these claim that one representation is better than the other. Claessen notes that more connectives in immediate representations does not imply more expressivity, since more complex structures like MUXes can easily be found in e.g. AIGs using local pattern matching.

Most interviewees mention that Plaisted-Greenbaum approaches can improve the coding somewhat. Biere remarks that Plaisted-Greenbaum is hard to combine with incrementality (see below). Baumgartner et al do not have an opinion on CNF translation techniques, since their solver uses the circuit representation internally.

3.3 Problem Size versus Problem Hardness

It is well known that SAT is NP-complete, which means that the runtime of the SAT solver is potentially exponential in the number of variables in the problem.

So, in theoretical results, there seem to be a strong connection between the size and the hardness of a problem. However, SAT would be practically useless if actual run times were anywhere close to the exponential upper bounds, and experience suggests that it is often much shorter. With this in mind, it is interesting whether the theoretical connection between problem size and problem hardness also holds in practice.

The interviewees agree that there is only a very weak correlation between size and hardness in practical SAT. As an example, Marques-Silva brings up the *smallest unsolved problem* prize in the SAT-competition, which in 2007 was won by a problem containing only 117 variables in 244 clauses. At the same time, there are problems with millions of variables that are efficiently solved. Marques-Silva mentions, on the other hand, that for a fixed problem domain, there is a stronger correlation between problem size and hardness. Biere brings up that a more important factor is the number of primary inputs.

Nieuwenhuis says that while the correlation between number of variables and hardness is very weak, there is a less weak correlation between number of clauses and hardness. This is motivated by that the number of clauses influences the speed of the unit propagation. There is consensus that other factors, such as arc consistency, are much more important than various measurements of problem size.

Older systems often had problems with large problems because of representations; for instance, Biere reported on a SAT solver that could only handle 32k variables, which forced him to change the representation in an earlier BMC experiment. Such limits are rarely a problem in modern solvers.

3.4 Clause Sizes

For inexperienced SAT users, it may be hard to see if the size of clauses have an impact on the problem. Should one avoid large clauses or strive for large clauses, or is it irrelevant? Most interviewees think clause size is an irrelevant factor. If a large clause is natural (such as in a onehot-constraint), then there is no point in avoiding it. Sometimes one hear claims about large or small clauses being preferable (large because it makes the watched literal strategy more efficient, and small because it means higher implicativity). However, as pointed out by Sörensson, usually one does not have a choice. The consensus is to not spend effort on clause size.

3.5 Horn Clauses

A clause is called a Horn clause if it contains at most one positive literal. It is well known that problems consisting exclusively of Horn clauses can be solved in polynomial time. Of course, this also implies that not all problems can be formulated with only Horn clauses. First order theorem provers often have special techniques to handle Horn clauses, so in first order theorem proving it can be a good idea to strive for Horn clauses, even if the encoding is only partially Horn.

For SAT solving, however, the interviewees agree that Horn clauses are mostly irrelevant. If, indeed, the problem is completely Horn, then any SAT solver will solve it quickly. However, no current SAT solver exploits Horn clauses, so if a problem is not 100% Horn, then it is pointless to increase the proportion of Horn clauses.

3.6 Cone of Influence

The cone of influence (COI) of a node in a circuit is the subcircuit that influences the particular node. The COI can be identified simply by tracing backwards from the node, including inputs, inputs of inputs, and so on. Naturally, supplying SAT solvers with circuitry that is not in the COI of the proof goal may slow down the solver. On the other hand, the strategy of keeping track of variable activities means that variables outside of the COI tend to be picked rarely for branching, and thereby the effect may be small.

The interviewees agree that SAT solvers usually efficiently detect the relevant information in problems, but think in general that it is worth the effort to isolate the COI. Biere says that irrelevant information usually does not prevent the solver from finishing, but can slow it down by as much as an order of magnitude. Marques-Silva says that although it usually does not matter much, it is easy to think of cases where it could make a very big difference (such as a huge XOR tree outside of the COI), so he recommends COI reduction. Baumgartner et al note that in hardware verification, COI reduction is usually quite simple (since they use circuit representations), and in many cases it matters much — especially in equivalence checking, where the COI of any particular equivalence checking sub-goal is typically much smaller than the whole circuit. Sörensson also recommends isolating the COI, but points out that it may be hard to do precisely in incremental settings, such as BMC.

4 More Techniques

4.1 Using Incrementality

Traditionally, SAT problems have been encoded in DIMACS[4] files, which are handed to the solver. After termination of the solver, the result is read back and analyzed by the calling program. Thus, the actual SAT solving is an isolated and atomic event. In contrast, many modern SAT solvers offer sophisticated APIs, allowing the calling program to interact with the solver, possibly even maintaining several separate solvers and feed information between them. One particularly useful feature that requires an API is incrementality.

Incrementality is the ability to add more clauses to the problem after a satisfying assignment has already been found. This means that the solver retains its state between the runs, including learned conflict clauses, variable activities,

[4]There does not seem to be an official definition of the DIMACS format. However, web searches easily come up with many unofficial descriptions of this very simple format.

and watched literals. Therefore, using incrementality is usually much more efficient than restarting the solver.

Adding new clauses between iterations is simple to do with the API. Removing clauses is usually not directly supported, but can be done using the *solve under assumptions* function. Checking the satisfiability of $A \cup B$ and thereafter of $A \cup C$ with an incremental solver (where A, B, and C are clause sets) can be done as follows: first add all clauses of A to the solver state. Then add all clauses of B, but add a fresh variable x to all those clauses. Thereafter one solves the current set under the assumption $\neg x$. When that terminates, all clauses from C are added, and the solver is invoked under the assumption x.

There are several situations where incrementality can be useful. The most well known is probably bounded model checking, BMC [ES03a], where a sequential circuit is iteratively unrolled until a counter example of the property to prove has been found, or an iteration limit is reached. Since large portions of the problem formulation is retained between iterations, it pays off to use an iterative approach. Sörensson points out that for BMC, one can choose between unrolling the circuit both backwards and forwards. Although cone of influence reduction is easier in backwards BMC, one usually does forward BMC since that enables constant propagation (which often leads very far, according to Sörensson) and structural hashing (which means finding common subformulas). In BMC with induction [SSS00], one usually does the base case forwards and the step backwards, unless they are done simultaneously in the same environment, in which case backwards unrolling seems better, according to Sörensson.

Claessen points out a less obvious application of incrementality: lazily adding constraints to a problem. For instance in BMC with induction [SSS00], one needs to express that n states are unique. This uniqueness constraint grows quickly and can make the SAT solving slow.

A more efficient way of handling this problem is to ignore the uniqueness constraint in the initial SAT solver invocation. Counter examples are then analyzed, and if it turns out that they do not respect the uniqueness property, a subset of the uniqueness constraint is added which only rules out the current counter example. This must often be repeated a few times, but normally only a relatively small subset of the full uniqueness constraint is needed. The total run-time of all the iterations is then usually much shorter than doing a single run with the full constraint.

Other applications where incrementality has been successfully used is QBF solving, first order theorem proving [Cla05] and model finding [CS03].

4.2 Representing Discrete Sets, such as States and Natural Numbers

In many situations, one have to represent sets of discrete values, such as states or integers. There are at least three common approaches for coding such values:

- **Onehot encoding:** Introduce a boolean variable x_n for each possible value n, which is true if and only if the represented value equals n. This

encoding usually requires both an *at least one* constraint (a single clause containing all x_n) and an *at most one* constraint (a quadratic number of 2-clauses).

- **Unary encoding:** This is a close relative of onehot encoding, except that each variable x_n is true if and only if the represented value is larger than or equal to n. This encoding usually requires a linear number of 2-clauses constraining that x_{i+1} implies x_i.
- **Binary encoding:** Using $\lceil \log_2 n \rceil$ variables, where n is the number of possible values, each value is represented using a unique assignment of the variables. In the case of natural numbers, normal binary number encoding is typically used.

Claessen uses all of these encodings in Paradox [CS03], and has more information. The onehot encoding, Claessen says, is well suited if the domain is small. It is not good for arithmetics, since defining operators takes a quadratic number of clauses. Both Claessen and Biere mention that the binary encoding works well for larger domains and even some arithmetic, but poorly for small domains.

An encoding that Claessen has found useful in Paradox is the combination of onehot and binary (see figure 1). This combination works out well, since expressing the *at least one* property is trivial in onehot, while mapping the onehot encoding to the binary encoding implies *at most one*, with good propagation behaviour. With this dual encoding, one can refer to either encoding in other properties, according to which is most convenient. For instance, a state transition function can use the onehot encoding, while arithmetic operations can use the binary encoding. Thanks to Claessen for disclosing this previously unpublished encoding.

Onehot	**Onehot+binary**
$o_0 \vee o_1 \vee o_2 \vee o_3$	$o_0 \vee o_1 \vee o_2 \vee o_3$
$\bigwedge_{i \neq j} \neg o_i \vee \neg o_j$	$o_0 \rightarrow \neg b_0 \wedge \neg b_1$
	$o_1 \rightarrow \ \ b_0 \wedge \neg b_1$
	$o_2 \rightarrow \neg b_0\wedge \ \ b_1$
	$o_3 \rightarrow \ \ b_0 \wedge \ \ b_1$

Figure 1: Onehot encoding versus onehot+binary encoding for a domain of size four. The *at least one* constraint is common for the two encodings, and consists of one linearly sized clause. Onehot encoding needs an *at most one* constraint with a quadratic number of 2-clauses. Onehot+binary uses a mapping from onehot to binary with $n \log_2 n$ 2-clauses, which also implies *at most one* with arc-consistency. Each implication $o_i \rightarrow b_1 \wedge \ldots \wedge b_m$ translates to m 2-clauses, e.g.: $(\neg o_1 | b_0)$, $(\neg o_1 | \neg b_1)$.

Other variants are possible as well. Claessen and Biere both mention mixed radix representation [ES05] where numbers may be encoded, for instance, in decimal notation with each digit onehot encoded.

4.3 Adding Arithmetics

Performing arithmetic operations in SAT seems to be hard but possible. Sörensson says that most attempts work pretty poorly, but people do it anyway. Marques-Silva is doubtful and suggests that problems containing integers are probably better handled by SMT than by SAT. Biere and Claessen, however, have successfully used arithmetics in some examples. Both think that the binary representation is preferable for arithmetics.

Claessen remarks that the topic is not well understood; there are many networks for addition and multiplication in the hardware community, but it is not clear which works well in SAT. Hardware networks are optimized for circuit depth, which is irrelevant for SAT. Claessen says that it is hard to know what matters for SAT, so lacking understanding one chooses the smallest encoding.

A practical detail that Claessen brings up is the importance of normalizing arithmetic expressions before encoding. For instance, one should make sure that the expressions $a \times b$ and $b \times a$ do not both occur in the same problem, but replace one with the other. Biere reports that functional encodings usually work better than relational ones.

Sörensson mentions a particular kind of constraints: $max_k(x_1, \ldots, x_n)$ which is true if and only if at most k of the n variables x_i are true. This can be useful in arithmetics with unary representations: a sum exceeds a particular value if the number of true variables in the unary representations of all numbers exceeds the value. Sörensson reports that in this case, simply sorting all unary variables and checking the value of variable number $n - k$ yields good results [ES05]. In this application, most sorting networks result in arc consistent encodings - that is, as soon as an input variable obtains a value, then this value is unit propagated to the output, resulting in a shrinking gap of undetermined output variables. There are, however, other situations where sorting is inappropriate: for instance if a number of integers are to be sorted (instead of single bits as in the previous case), then virtually all integers must be fully known before the position of any of the integers can be determined.

4.4 Tweaking Solver Parameters

Most solvers have a number of parameters that can be changed. Such parameters may be restart frequency, variable decay factor, and initial variable activities. The question is whether the user is supposed to adjust these parameters to suit the problem, or if it is better to leave them untouched.

Most of the experts agree that tweaking solver parameters can have positive impact on the solver run time. However, it is also agreed that finding better values for parameters is a hard problem, and the general view seems to be that the effort is better spent on the encoding. Nieuwenhuis claims that tweaking

branching heuristics and variable activities usually only makes things worse. Nieuwenhuis and Biere have both had some success with changing restart frequencies for some classes of problems — Biere has published a paper on the topic [Bie08]. Of course, whether to change parameters also depends on what kind of parameters that the SAT solver offers.

Claessen, Marques-Silva, and Sörensson advice against parameter tweaking. Marques-Silva remarks that the SAT solver developers are usually more able than users to find good values for parameters, and they continuously fine tune the default parameters. Claessen indicate that effects of parameter changes are instable, and therefore not good to use in the longer run.

Nieuwenhuis and Biere mention a paper that suggests using techniques that randomly search for local optima [HBHH07]. However, Biere claims that in his own experiments, employing the described techniques has had negative impact on the efficiency.

Among the interviewees, Baumgartner et al are most positive towards parameter tweaking. They point out that it is not possible to do tweaking for every problem, unless one exploits parallel processing. However, they claim that broad categorizations can be made: for instance, if a problem is expected to be satisfiable, then the theorem prover should be biased toward assignments that satisfy clauses quickly. On the other hand, if the problem is expected to be UNSAT or very difficult SAT, then assignments that result in conflicts are more desirable.

Sörensson mentions that one also have to consider what kind of parameters the sat solver offers. For SAT solvers that offer many possibilities for the user to give information about the problem it makes more sense to use the parameters. If the solver only offers a few parameters with uncertain consequences, then one should probably not use them.

5 Other Gems Picked up During the Interviews

The techniques discussed so far have been answers to questions asked during the interviews. Naturally, some interviewees mentioned other related techniques. Since these topics have only been discussed in few interviews it is not possible to determine whether there is consensus about them. However, they seem so useful that it would be a shame not to include them in the article.

5.1 Symmetry Removals

Claessen and Marques-Silva both mention symmetry removals. If it is known that a problem can have many solutions that are variants of each other then it is often beneficial to add constraints that forbid some of the solutions to prune the search space. An example is when it is known that the order of certain inputs does not matter — then one can add a constraint forcing them to be sorted. If it is necessary to be able to find all solutions, then one can usually

write a function that takes one solution and returns all symmetrical solutions, including the ones that were forbidden by the encoding.

5.2 Order of Variable Indexes

Biere points out that the order of variable indices actually matters. In early SAT competitions, all problems were randomly scrambled, which turned out to have a strongly negative impact on all solvers. Normally, the runtime worsened between factors 10 and 100. Apparently, solvers benefit from related variables having close indices, perhaps because it guides selection of branch variables.

Variables are usually appropriately ordered by virtue of how encodings are done, for instance in a depth first traversal of a circuit, neighbour nodes usually have close indices. However, one should be aware of this phenomenon, so that one does not try to come up with something like a global variable ordering. Being faithful to the circuit structure in clausification is therefore a good strategy.

5.3 Solutions Close to All Zero

When branching, SAT solvers usually try setting variables to false first. During his tutorial at FMCAD 2007 [Eén07], Niklas Een pointed out that for existing benchmarks, always choosing false first is more successful than always choosing true first. Biere suggests that this can be exploited: if encodings are formulated so that solutions will be close to the all-zero assignment, then SAT solvers should be able to find them sooner.

6 Conclusions

The common points picked up during the different interviews is that the encoding does have a big impact on the efficiency of the SAT solver, that finding a good encoding takes much effort, and that encoding quality does not depend much on easily measured properties like size or number of variables. The interviewees usually suggest starting with a simple encoding which is iteratively improved.

It is well known from before that people have different opinions about whether to use clauses of circuits internally in SAT solvers. It turned out in the interviews that even when one has decided to use a clause based solver, it does not seem to matter much whether one uses clauses immediately or formulates the problem as circuits that are translated to circuits. For this translation, the simple Tseitin encoding is quite highly esteemed, even though much work has gone into developing more sophisticated translations. At least there is consensus that Tseitin is a good starting point.

There is consensus that problem hardness and problem size is only very loosely related, although if one fix the problem domain, then size may have an impact on the hardness. The size of individual clauses does not seem to

matter much either. There is consensus that the proportion of horn clauses in an encoding is mainly irrelevant for SAT.

Knowledge about different techniques is certainly helpful for coming up with a better encoding, but there are few techniques that always work well. In other words, many different techniques may have to be tested for each problem.

7 Future Work

The topic of SAT encodings is very large, and an article like this can merely scratch the surface of it. More effort should be spent bringing out hidden knowledge from the experts. One useful project would be to create a larger source of information, perhaps a wiki, where numerous encoding techniques can be described and discussed; a sort of encyclopedia for SAT encodings.

Another useful project would be to create a library (in the sense of a programming language library) of encoding generators. This would allow SAT users to swiftly create and try out different encodings for well known problems (such as arithmetics).

8 Acknowledgements

Many thanks to the interviewees, who made this article possible. Thanks also to Niklas Sörensson and Nicholas Smallbone for their proofreading of the article.

References

[ABE00] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén, *Symbolic reachability analysis based on SAT-solvers*, Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS'00 (Springer Verlag, ed.), LNCS, vol. 1785, 2000, pp. 411–425.

[Bie08] Armin Biere, *Adaptive Restart Strategies for Conflict Driven SAT Solvers*, Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, LNCS, vol. 4996, Springer, 2008.

[CESS08] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson, *SAT-solving in practice*, 9th International Workshop on Discrete Event Systems, WODES 2008, IEEE, 2008, pp. 61–67.

[Cla05] Koen Claessen, *Equinox, a new theorem prover for full first-order logic with equality*, Presentation at Dagstuhl Seminar 05431 on Deduction and Applications, October 2005.

[CS03] Koen Claessen and Niklas Sörensson, *New techniques that improve MACE-style model finding*, Proc. of Workshop on Model Computation (MODEL), 2003.

[DLL62] M. Davis, G. Logemann, and D. Loveland, *A Machine Program for Theorem-Proving*, Communications of the ACM **5** (1962), 394–397.

[dlT92] Thierry Boy de la Tour, *An optimality result for clause form translation*, J. Symb. Comput. **14** (1992), no. 4, 283–302.

[DP60] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*, JACM **7** (1960), no. 3, 201–215.

[EMS07] Niklas Eén, Alan Mishchenko, and Niklas Sörensson, *Applying Logic Synthesis for Speeding Up SAT*, Proceedings of Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, LNCS, vol. 4501, Springer, 2007, pp. 272–286.

[ES03a] Niklas Eén and Niklas Sörensson, *Temporal induction by incremental SAT solving*, Electr. Notes Theor. Comput. Sci. **89** (2003), no. 4.

[ES03b] Niklas Eén and Niklas Sörensson, *An Extensible SAT-solver*, Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, LNCS, vol. 2919, Springer, 2003, pp. 502–518.

[ES05] Niklas Een and Niklas Sörensson, *Translating Pseudo-Boolean Constraints into SAT*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2005), 1–26.

[Eén07] Niklas Eén, *Practical SAT - a tutorial on applied satisfiability solving*, Invited talk, FMCAD, November 2007, Slides available on http://minisat.se/Papers.html.

[HBHH07] Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu, *Boosting verification by automatic tuning of decision procedures*, Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, IEEE Computer Society, 2007, pp. 27–35.

[JS04] Paul Jackson and Daniel Sheridan, *Clause form conversions for boolean circuits*, Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Revised Selected Papers, LNCS, vol. 3542, Springer, 2004, pp. 183–198.

[LMS06] I. Lynce and J. Marques-Silva, *Efficient haplotype inference with boolean satisfiability*, National Conference on Artificial Intelligence (AAAI), 2006.

[MS08] Joao Marques-Silva, *Practical applications of boolean satisfiability*, Workshop on Discrete Event Systems (WODES'08), May 2008, http://eprints.ecs.soton.ac.uk/15340/.

[MV07] Panagiotis Manolios and Daron Vroon, *Efficient circuit to cnf conversion*, Proceedings of Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, vol. 4501, Springer, 2007, pp. 4–9.

[PG86] D. Plaisted and S. Greenbaum, *A Structure-Preserving Clause Form Translation*, Journal of Symbolic Computation, vol. 2, Elsevier, 1986.

[Pre09] Steven Prestwich, *CNF Encodings*, Handbook of Satisfiability (Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds.), IOS Press, 2009, pp. 75–97.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck, *Checking Safety Properties Using Induction and a SAT-Solver*, Formal Method in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA (Warren A. Hunt Jr. and Steven D. Johnson, eds.), Lecture Notes in Computer Science, vol. 1954, Springer, 2000, pp. 108–125.

[Tse68] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, Studies in Constructive Mathematics and Mathematical Logics, vol. 2, 1968, Reprinted in Siekmann and Wrightson (eds), Automation of Reasoning Vol 2, 1983 (Springer-Verlag).