Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers

Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama

Massachusetts Institute of Technology, Cambridge, MA, USA
{jinala,rohitsingh,asolar}@csail.mit.edu

The theory of bit-vectors in SMT solvers is very important for many applications due to its ability to faithfully model the behavior of machine instructions. A crucial step in solving bit-vector formulas is the translation from high-level bit-vector terms down to low-level boolean formulas that can be efficiently mapped to CNF clauses and fed into a SAT solver. In this paper, we demonstrate how a combination of program synthesis and machine learning technology can be used to automatically generate code to perform this translation in a way that is tailored to particular problem domains. Using this technique, the paper shows that we can improve upon the basic encoding strategy used by CVC4 (a state of the art SMT solver) and automatically generate variants of the solver tailored to different domains of problems represented in the bit-vector benchmark suite from SMT-COMP 2015.

1 Introduction

SMT solvers are at the heart of a number of software engineering tools ranging from automatic test generators [48,43,41] to deterministic replay tools [16], just to name two applications among many others [11,25,52]. Of particular importance to these applications is the theory of bit-vectors, which is widely used [17,26,42,46] because of its ability to faithfully represent the full range of machine arithmetic.

One of the most important steps in a bit-vector solver is the mapping of high-level bit-vector terms down to low-level CNF clauses that can be fed to a SAT solver—a process often referred to as *bit-blasting*. One approach to bit-blasting is to use the known efficient encodings for simpler boolean terms (such as AND or XOR) and compose them to generate CNF clauses for complex terms [49]. This approach can have a huge impact on the performance of the solver [39,38], but generally, it relies on having optimal encodings for the simpler terms, and even then it does not guarantee any kind of optimality of the overall encoding.

In this paper, we propose OptCNF, a new approach to automatically generate the code that converts high-level bit-vector terms into low-level CNF clauses. In addition to the obvious benefits of having the code automatically generated instead of having to write it by hand, OptCNF has three novel aspects that together significantly improve the quality of the overall encoding: (a) OptCNF uses synthesis technology to automatically generate efficient encodings from high-level formulas to CNF (b) OptCNF relies on auto-tuning to choose encodings that produce the best results for problems from a given domain. (c) OptCNF identifies commonly occurring clusters of terms in a given domain and focuses on finding optimal encodings for such clusters.

The synthesis of encodings balances optimality among three criteria: number of clauses, number of variables and propagation completeness. The propagation completeness requirement has been proposed as an important criterion in order for the encoded constraints to solve efficiently in the SAT solver [10]. Modern SAT solvers rely heavily on unit propagation to infer the values of variables without having to search for them. Propagation completeness means that if a given partial assignment implies that another unassigned variable should have a particular value, then the solver should be able to discover this value through unit propagation alone. Prior work has demonstrated the synthesis of propagation complete encodings for terms involving a small number of variables [12]. OptCNF, however, is more flexible and is able to produce propagation complete encodings even for relatively large bit-vector terms by taking advantage of high-level hypothesis about the structure of the encoding (See Section 2).

In practice, however, propagation completeness does not *always* improve the performance of an encoding. For certain classes of problems, for example, the additional unit propagations caused by a propagation complete encoding can actually slow the solver down. Similarly, there is often a trade-off between the number of auxiliary variables and the number of clauses used by an encoding; for some problems having more variables but fewer clauses can be better, but for other problems, having fewer variables at the expense of more clauses can be better. In order to cope with this variability, OptCNF uses auto-tuning to make choices about which encodings are best for problems from a particular domain. Prior work has demonstrated the value of tuning solver parameters in order to achieve optimal performance for problems from particular domains [32], but ours is the first work we know of where auto-tuning is used to make high-level decisions about how to encode particular terms (see Section 5).

Finally, OptCNF is able to better leverage its ability to synthesize optimal encodings by focusing on larger clusters of terms, as opposed to focusing on individual bit-vector operations independently. Given a corpus of sample problems from a domain, OptCNF is able to identify common recurring patterns in the formulas from those problems and then generate specialized encodings for those patterns.

Figure 1 shows how these ideas come together as OptCNF. The input to OptCNF is a collection of formulas represented as DAGs (Directed Acyclic Graphs) extracted from a set of benchmarks from a given problem domain. OptCNF samples these DAGs to extract representative clusters of terms—what the figure refers to as patterns. OptCNF then leverages Sketch synthesis system [45] to synthesize "optimal" encodings for those patterns and generates C++ code for the encodings that can be linked with a modified version of CVC4 solver [6]. The auto-generated code contains a set of switches to turn different encodings on or off. Finally, the auto-tuner searches for the optimal configuration of those switches in order to produce the best performing domain-specific version of CVC4.

Our evaluation shows that the resulting domain-specific encodings are able to significantly improve the performance of CVC4 when run in eager bit-blasting mode. Using OptCNF, we generated a separate solver for 7 different domains represented in the quantifier-free bit-vector benchmarks from the SMT-COMP 15 benchmark suite [7]; using these specialized solvers on their respective domains, we were able to solve 83 problems from the test set (see Section 6) that CVC4 could not solve.

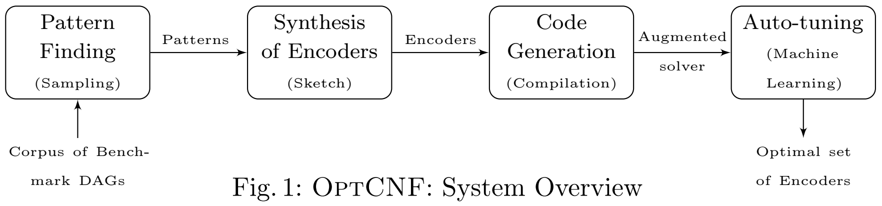
Pattern Finding (Sampling)
Patterns
Synthesis of Encoders (Sketch)
Encoders
Code Generation (Compilation)
Augmented solver
Auto-tuning (Machine Learning)
Corpus of Benchmark DAGs
Optimal set of Encoders

Fig. 1: OptCNF: System Overview

2 Synthesis of Encoders

Previous work [12] has attacked the problem of generating optimal propagation complete encodings for a given term by starting with an initial encoding and then exhaustively checking for violations of propagation completeness and incrementally adding more clauses to fix these violations. The resulting propagation complete encoding is then minimized to produce an equivalent but smaller encoding. Our approach to generating encodings is quite different because it relies on program synthesis technology, allowing us to symbolically search for an encoding based on a formal specification. An important advantage of our approach is flexibility. In particular, it allows us to generate *encoders* that generate encodings at solver run-time from terms that have parameters that will only be known at run-time (for example, the bit-width for a bit-vector operation).

OptCNF frames the task of generating these encoders as a Syntax Guided Synthesis problem (SyGuS) [2]. A SyGuS problem is a combination of a template or grammar that represents the space of the candidate solutions and a specification that constrains the solution. The goal of a SyGuS solver is to find a candidate in the template that satisfies the specification. The two components, template and specification, are very crucial in determining the scalability of the problem. Here, we first describe the templates that represent the space of CNF encoders for booleans and bit-vector terms. Then, we formalize the correctness and the optimality specification that constraints the template. Finally, we describe an efficient but equivalent specification that makes the SyGus synthesis problem more scalable.

2.1 CNF Encoders and Templates

The encoders generated by OptCNF work in two passes. Given a formula to be encoded into SAT, OptCNF first identifies terms for which it has learned to generate CNF constraints and replaces them by special placeholder operators N_i. Then, the pass that would normally have generated low-level constraints from the bit-vector terms is extended to recognize these placeholder operators and generate the specialized constraints for them.

The pass that identifies the known terms, and the scaffolding that iterates through the different operators in a DAG representation of the formula and identifies the placeholder nodes are all produced using relatively straightforward code-generation techniques. The synthesis problem focuses on the code that executes when one of these placeholder nodes is found. This is the encoder code that generates the CNF encoding for a previously identified term T.

$t \equiv and(x, or(y,z))$

```
clause({x,~t})
clause({~x,~y,t})
clause({~x,~z,t})
clause({y,z,~t})
```

$t \equiv bvAND_N(x, bvOR_N(y,z))$

```
for i from 1 to N:
 clause({x[i],~t[i]})
 clause({~x[i],~y[i],t[i]})
 clause({~x[i],~z[i],t[i]})
 clause({y[i],z[i],~t[i]})
```

$t \equiv bvEQ_N(x,y)$

```
t1 = true
for i from 1 to N:
 t2 = i == N ? t : newVar
 clause({x[i],y[i],~t1,t2})
 clause({x[i],~y[i],~t2})
 clause({~x[i],y[i],~t2})
 clause({~x[i],~y[i],~t1,t2})
 clause({t1,~t2})
 t1 = t2
```

Fig. 2: Encoders for three different kinds of terms

The term T for which OPTCNF is generating an encoding is known at synthesis time, so OPTCNF can choose a template or a set of templates for this code depending on the properties of T. Figure 2 illustrates the three different kind of terms and the encodings that represent the terms. If T is not parametric—for example if it is just a collection of boolean operators—then the encoder just needs to generate a fixed set of clauses corresponding to the constraint represented by T, and the template will reflect that. On the other hand, many terms will be parameterized by bit-widths, so the encoder will have to produce clauses in one or more loops.

For bit-vector terms, which are parametric on the bit-width of their different operators, we differentiate between two different kinds – bit-parallel and non bit-parallel. Bit-parallel terms are those that are composed entirely of operations, such as bitwise AND, OR or XOR, where there is no dependency from one column of the bit-vector to another. For these kinds of terms, generating the encoding for a single column and then enumerating them over all columns will still preserve optimality. Hence, it is sufficient to just synthesize the encoding for the boolean term that represents operations in a single column. This is, however, not the case for all bit-vector terms.

Terms involving bitwise PLUS, for example, cannot be dealt in the same way because there are dependencies that flow from one column to another. These operations can still be represented as a loop of encodings, but there will be auxiliary variables that are threaded from one iteration of the loop to another. Figure 3 shows one such template for a bit-vector formula involving two bit-vector inputs of size N (taken as a parameter) and outputs another bit-vector of size N. For each column in the bit-vectors, the template calls encode_column which is another template for explicit encodings, but this template can be instantiated with variables specific to loop iteration. This template has one auxiliary variable per column. Every column has an incoming auxiliary variable (a constant for the first column) which carries information from the previous columns and an outgoing auxiliary variable that carries information forward. This same template represents multiple formulas depending on how the encode_column template is instantiated. For example, this same template is used to generate encodings for both bitwise PLUS and bitwise MINUS operations.

The templates in OptCNF are all written in the Sketch language, which allows us to leverage the Sketch synthesis engine for the synthesis problem. A template in Sketch is a piece of code with integer and boolean holes to represent the set of candidate solutions that the synthesizer should consider. The standard template for an encoding is a list of clauses with holes representing the number of clauses, and the length and the literals present in each clause. We significantly reduce the size of the search space by enforcing an order among the literals in each clause and among clauses themselves and thus, eliminating symmetries. This canonical representation captures any general CNF encoding, but it does not impose any structure on the clauses. We found that this model is scalable enough for boolean formulas that expand into a small number of CNF clauses (about 20 to 30). But, in order to deal with bigger formulas like bit-vector operations, we need to represent the search space using loops to capture the structure.

OptCNF has a library of templates for different kinds of input types, output types and number of auxiliary variables per column. When running Sketch on a term, OptCNF runs different instances of Sketch with a different template and chooses the one that provides the best encoding (based on heuristics like number of clauses and number of auxiliary variables). Due to the scalability limits of Sketch, OptCNF can currently only synthesize encodings for non bit-parallel terms that have at most two input bit-vectors, at most two auxiliary variables per column and no nested loops in the template.

```
Lit [N] encode(Lit [N] mval, Lit [N] fval ) {
    Lit [N] out = newVar(N)   /* creates an array of out literals */
    Lit [N] aux = newVar(N)   /* creates an array of auxiliary literals */
    /* Specialize the first column */
    encode_column(mval[1], fval [1], out [1], const?, aux [1])
    for i in 2 to N:
      encode_column(mval[i], fval [ i ], out[ i ] aux[i−1], aux[ i ])
    return out
}
```

Fig. 3: Template for a bitwise operation on two bit-vectors (with one auxiliary variable per column)

2.2 Problem formulation

In addition to the template, the other important component of a SyGus problem is the specification. Unlike the templates, which are very different for parameterized and non-parameterized terms, the specifications for both are actually very similar; the only difference is that for parameterized terms, the parameters must be threaded through to all the relevant predicates. Therefore, the rest of the section will omit these bit-width parameters in the interest of clarity.

A term $T(in)$ can be represented by a predicate $P(in, out)$ defined as $P(in, out) \Leftrightarrow out = T(in)$. For notational convenience, we will just write $P(x)$, where x is understood to be a vector containing both the input and the output variables. The goal is to generate an alternative representation of the predicate in terms of CNF clauses $C(x)$.

Definition 1 (Correctness Specification). *A set of CNF clauses "represents" a boolean predicate P iff $P(x) \Leftrightarrow C(x)$.*

In addition to the correctness specification, however, we want to ensure propagation completeness which needs to be defined in terms of the behavior of the encoding under *partial* assignments. A partial assignment σ maps every variable to one of $\{true,\ false,\ \top\}$ where $\top$ indicates that the value has not been assigned by the solver and could be *true* or *false*. A partial assignment can be understood as the set of all complete assignments that are consistent with the partial assignment. Therefore, it is standard to define a partial order among partial assignments as:

$$\sigma \sqsupseteq \sigma' \iff \forall i.\ \sigma(x_i) \neq \top \Rightarrow \sigma'(x_i) = \sigma(x_i)$$

We generalize the predicate to be a function from partial assignments to the set $\{true,\ false,\ \top\}$, and define $P(\sigma) = \top$ for any partial assignment where some variable x_i is set to $\top$.

Definition 2. *We define the following predicates on partial assignments:*

$$\begin{aligned}
complete(\sigma) &\equiv \forall i.\ \sigma(x_i) \neq \top \\
satisfiable(\sigma, P) &\equiv \exists \sigma' \sqsubseteq \sigma.\ P(\sigma') \ = \ true \\
unsatisfiable(\sigma, P) &\equiv \forall \sigma' \sqsubseteq \sigma.\ P(\sigma') \ \neq true \\
forces(\sigma, P, x_i, b) &\equiv (\sigma' = extend(\sigma, x_i, \neg b)) \ \Rightarrow \ unsatisfiable(\sigma', P) \\
maypropagate(\sigma, P) &\equiv \exists i, b.\ forces(\sigma, P, x_i, b)
\end{aligned}$$

Where $extend(\sigma, x_i, b)$ is defined as extending an assignment with $\sigma(x_i) = \top$ to one where variable x_i has value b. The predicate $maypropagate(\sigma, P)$ indicates that the partial assignment σ forces the value of some currently unassigned variable.

Lemma 1. *The forces() predicate has the following property.*

$$\begin{aligned}
&forces(\sigma, P, \hat{x_i}, \hat{b}) \ \wedge \ \sigma' = extend(\sigma, \hat{x_i}, \hat{b}) \\
&\quad \Rightarrow \forall_{(x_i, b) \neq (\hat{x_i}, \hat{b})} \ forces(\sigma, P, x_i, b) \Rightarrow forces(\sigma', P, x_i, b)
\end{aligned}$$

This means that if P and a partial assignment σ force $\hat{x_i}$ to take a particular value $\hat{b}$, then any other variable that was also forced by σ and P will also be forced after extending the assignment with $\sigma(\hat{x_i}) = \hat{b}$.

Lemma 2. *Another important property of forces() is the following.*

$$\begin{aligned}&\mathit{satisfiable}(\sigma, P) \;\wedge\; \mathit{forces}(\sigma, P, \hat{x_i}, \hat{b}) \;\wedge\; \sigma' = \mathit{extend}(\sigma, \hat{x_i}, \hat{b})\\ &\qquad\qquad\Rightarrow \mathit{satisfiable}(\sigma', P)\end{aligned}$$

This means that if P and a partial assignment σ force $\hat{x_i}$ to take a particular value, then after extending the assignment with $\sigma(\hat{x_i}) = \hat{b}$, the new assignment is still satisfiable.

A clause c can be applied to a partial assignment as well, resulting in a value $c(\sigma) \in \{\mathit{true}, \mathit{false}, \mu, \top\}$. A clause is unit ($\mu$) if one of the literals in the clause has an unknown value and all others are $false$. A CNF encoding is a collection of clauses C. $C(\sigma)$ can either be *true* if $c(\sigma) = \mathit{true}$ for all the clauses, *false* if $c(\sigma) = \mathit{false}$ for at least one of the clauses, μ if σ makes at least one clause unit (and σ does not falsify any others), or $\top$ if none of the above. Thus, the result of applying C to a partial assignment helps identify the case when at least one of the clauses is a unit clause, and it is, therefore, possible to propagate further assignments. This is useful in describing unit propagation.

Definition 3 (UP). *The function UP captures the unit propagation in SAT solvers. We say that C propagates σ to $\mathit{UP}(C, \sigma)$ under unit propagation according to the following rules:*
1. *if $C(\sigma) \neq \mu$, then $\mathit{UP}(C, \sigma) = \sigma$.*
2. *else, $C(\sigma)$ has a unit clause. If the unit clause forces $\sigma(x_i) = b$, then $\mathit{UP}(C, \sigma) = \mathit{UP}(C, \sigma')$ where $\sigma' = \mathit{extend}(\sigma, x_i, b)$.*

The definitions above give rise to an important lemma.

Lemma 3. *A set of CNF clauses C "represents" a boolean predicate P iff it satisfies the following two conditions:*

$$\begin{aligned}&1.\ \mathit{satisfiable}(\sigma, P) \;\Rightarrow\; C(\mathit{UP}(C,\ \sigma)) \;\neq\; \mathit{false}\\ &2.\ \mathit{unsatisfiable}(\sigma, P) \;\Rightarrow\; C(\mathit{UP}(C,\ \sigma)) \;\neq\; \mathit{true}\end{aligned} \tag{2.1}$$

i.e. if an assignment can be extended to a satisfiable assignment for P, then unit propagation should not lead to a contradiction. And similarly, if an assignment (possibly partial) already contradicts P, then unit propagation should not lead to a satisfiable assignment for the CNF clauses.

With the definitions above, we can now state the requirement for propagation completeness.

Definition 4 (Propagation Completeness). *C is a set of propagation complete CNF clauses representing P if C "represents" P and*

$$\begin{aligned}&\forall\sigma.\ \mathit{satisfiable}(\sigma, P)\\ &\Rightarrow \forall\ x_i, b_i.\ (\mathit{forces}(\sigma, P, x_i, b_i) \Rightarrow\ \ \mathit{UP}(C, \sigma) \sqsubseteq \mathit{extend}(\sigma, x_i, b_i))\end{aligned} \tag{2.2}$$

In other words, if a partial assignment can be completed into a satisfying assignment, and if there are unassigned variables x_i that if set to $\neg b_i$ would make the partial assignment unsatisfiable, then unit propagation must set all such x_i to b_i.

2.3 Synthesis-friendly propagation completeness

The above definition captures the notion of propagation complete encodings, but it is unsuitable as a specification for synthesis because the recursive definition of *UP* essentially defines a small SAT solver, making it too complex for a state of the art synthesizer. Instead, OptCNF relies on an equivalent but simpler specification that does not require implementing a SAT solver. The idea is that instead of thinking in terms of full unit propagation, we now verify propagation only one step at a time. Specifically, the claim is that the following three rules guarantee propagation completeness.

$$\begin{array}{l} 1.\forall\sigma.\ \mathit{satisfiable}(\sigma, P)\ \Rightarrow\ C(\sigma) \neq \mathit{false} \\ 2.\forall\sigma.\ \mathit{maypropagate}(\sigma, P)\ \Rightarrow C(\sigma) = \mu \\ 3.\forall\sigma.\ \mathit{unsatisfiable}(\sigma, P)\ \Rightarrow\ C(\sigma) = \mathit{false}\ \vee\ C(\sigma) = \mu \end{array} \tag{2.3}$$

Theorem 1. *Formula* (2.3) $\iff$ *Correctness* $\wedge$ *Formula* (2.2)

Proof: Formula (2.3) $\Rightarrow$ Correctness
This follows directly from Formula (2.3), because when σ is complete, $\mathit{satisfiable}(\sigma, P)$ implies $P(\sigma) = \mathit{true}$ and similarly, $\mathit{unsatisfiable}(\sigma, P)$ implies $P(\sigma) = \mathit{false}$.

Proof: Formula (2.3) $\Rightarrow$ Formula (2.2)
This can be proved by induction on the number of times σ can be extended before it fails $\mathit{maypropagate}(\sigma, P)$. For the base case, $\neg\mathit{maypropagate}(\sigma, P)$, (2.2) is vacuously satisfied because $\mathit{forces}()$ fails for all variables. For the inductive case, $\mathit{maypropagate}(\sigma, P)$, $C(\sigma) = \mu$ (by 2.3-2). Let $\sigma' = \mathit{extend}(\sigma, \hat{x_i}, \hat{b})$ which is obtained by propagating the unit clause in $C(\sigma)$. Note that $\mathit{UP}(C, \sigma) = \mathit{UP}(C, \sigma')$ by Definition 3. Applying Lemma 2 tells us that $\mathit{satisfiable}(\sigma', P)$, so applying the inductive hypothesis together with Lemma 1, we can prove the inductive case.

Proof: Correctness $\wedge$ Formula (2.2) $\Rightarrow$ Formula (2.3)
First, we use the fact that correctness is equivalent to Formula (2.1). If $\mathit{satisfiable}(\sigma, P)$, then $C(\mathit{UP}(C, \sigma)) \neq \mathit{false}$ and this implies $C(\sigma) \neq \mathit{false}$.
If σ can be propagated, then $\exists x_i, b.\ \mathit{forces}(\sigma, P, x_i, b)$. And hence, $\mathit{UP}(C, \sigma) \neq \sigma$ and this implies $C(\sigma) = \mu$.

If $\mathit{unsatisfiable}(\sigma, P)$, then let $\sigma' \sqsupset \sigma$ be the maximal satisfying subset of σ i.e. σ' is satisfiable and $\forall\sigma' \sqsupset \sigma'' \sqsupset \sigma.\ \sigma''$ is unsatisfiable. Then, $C(\sigma') = \mu$ and since σ' is maximal subset, $C(\sigma) = \mathit{false}\ \vee\ C(\sigma) = \mu$.

2.4 Introducing Auxiliary Variables

In some cases, the encoding C will involve auxiliary variables t_i in addition to the variables x_i, in such cases, we write $C((x, t))$. In that case, the correctness specification must be generalized to

$$\forall x. \quad P(x) \iff \exists t.\ C((x, t))$$

Similarly, the conditions in Formula (2.3) generalize to the conditions below.

$$\begin{aligned}&1.\forall\sigma.\ \mathit{satisfiable}(\sigma, P) \ \Rightarrow\ \exists\sigma_t.\ C((\sigma,\ \sigma_t)) \neq \mathit{false}\\&2.\forall\sigma,\sigma_t.\ \mathit{maypropagate}(\sigma, P)\ \wedge\ C((\sigma,\ \sigma_t)) \neq \mathit{false}\ \Rightarrow\ C((\sigma,\ \sigma_t)) = \mu\\&3.\forall\sigma,\sigma_t.\ \mathit{unsatisfiable}(\sigma, P)\ \Rightarrow\ C((\sigma,\ \sigma_t)) = \mathit{false}\ \vee\ C((\sigma,\ \sigma_t)) = \mu\end{aligned} \quad (2.4)$$

The proof for this has a similar structure to the previous proof. Basically, once we establish the first rule above, auxiliary variables can be treated just as the other variables in P. It should be noted that this specification is more complex than Formula (2.3) because of the existential quantifier in the R.H.S of rule 1. The CEGIS algorithm employed by solvers like SKETCH is designed to deal with the outer universal quantifiers, but cannot handle inner existential quantifiers. Hence, this existential quantifier should be translated into an explicit loop over all auxiliary assignments, which makes the synthesis problem hard. In practice, we found that this overhead is not significant when the number of auxiliaries used in the encodings is low.

2.5 Clause Minimization

Another important optimality criterion for the encodings is the clause minimization. If there are two propagation complete encodings having different number of clauses representing the same predicate, then the encoding with the lower number of clauses is preferred. OPTCNF relies on binary search to find an encoding with an optimal number of clauses. This requires solving a logarithmic number of synthesis problems to generate a single encoding, which has proven to be reasonably efficient in practice.

2.6 Guarantees of the synthesized solution

When the formula is a boolean term or a bit-parallel term, SKETCH performs full verification and hence, the output is guaranteed to be correct and propagation complete. When the input formula is a non bit-parallel bit-vector term, OPTCNF does bounded verification on the size of the bit-width parameters. The correctness specification is easier to verify than the propagation completeness requirement, so OPTCNF allows the user to separately specify the checking bounds for both specifications. In our experiments, we check correctness for all inputs up to 6-bits and propagation completeness for up to 3-bits. Beyond these bounds, OPTCNF relies on verifying the output (sat/unsat) of the solver on all the benchmarks used in our experiments to provide confidence on the correctness of the synthesized encodings. We did not encounter a single instance where OPTCNF resulted in an incorrect output.

3 Pattern Finding

In this phase, we identify commonly occurring patterns in the formulas arising from a given domain. For this, we build on prior work on representative sampling from DAG-based representations of formulas ([44]). The original sampling work on which we build takes as input a size k and produces a representative sample of all sub-terms of size k that appear in the corpus. When $k = 1$, for example, the process will return a sample of all the operations that appear in the corpus; the frequency with which a given operation appears in the sample will be approximately the same as the frequency with which it appears in the corpus. When sampling with higher values of k, the sampling process takes into account the fact that some operations are commutative, but not others.

Given a corpus, OptCNF collects representative samples for values of $k \leq 5$ for bit-parallel formulas and $k \leq 3$ for non bit-parallel formulas. The upper bounds are determined by the capabilities of our encoding synthesis algorithm, which is unable to generate encodings for larger terms.

4 Encoder Code Generation

OptCNF uses CVC4 as the target solver and generates the code for implementing the synthesized encoders in two phases: (1) Pattern matching in the decreasing order of the pattern size and (2) Extending the existing encoding phase in CVC4. OptCNF generates code for a straight-forward pattern matching phase while handling symmetries by enumerating all equivalent permutations of patterns with commutative operations. The generated code for augmenting CVC4 implements the synthesized encoder for each matched pattern and provides a command-line interface for switching them on or off individually.

However, there is scope for optimizing this code by implementing: (1) fast pattern matching that reuses common terms in the matched patterns (2) caching and reusing newly generated literals in the encoding phase (3) reduction in number of function calls in the generated code and (4) simplifying the encodings for patterns with constant inputs. Even without these optimizations, we are able to show significant improvement in CVC4's performance on certain domains (Section 6).

5 Auto-tuning Encoders

For each domain, we use OpenTuner [3] to auto-tune the set of encoders (one for each pattern) obtained from the synthesis phase according to a performance metric based on the number of benchmark problems solved and the time taken to solve them. The evaluation function (**fopt**) to be optimized takes as input a set of encoders to be used and returns a real number. The number is the sum of all the times taken by the benchmarks to solve; for any benchmarks that time out, their time is counted as the timeout bound times two. The auto-tuner tries to minimize this value by trying out various subsets of encoders provided to it as input while learning a model of the dependence of **fopt** on the selection of encoders.

6 Evaluation

We extend CVC4 solver (ranked 2 in the bit-vector category of SMT-COMP 2015 [8]) with synthesized encoders for each domain and evaluate the impact on its performance. Each generated solver is evaluated on the non-incremental quantifier free bit-vector (QF_BV) benchmark suite from SMT-COMP 2015. This benchmark suite consists of 26320 benchmarks that are grouped into 36 sub-categories. In most cases, these sub-categories represent a particular domain of problems–for example, the log−slicing category represents benchmarks that verify bit-vector translation from operations like addition and multiplication to a set of base operations. Some other sub-categories like asp are themselves a collection of benchmarks from multiple different sources. We treat these sub-categories as domains irrespective of whether they really represent a single domain.

6.1 Experimental Setup

OptCNF generates a domain specific solver in four stages:

1. Randomly sampling 10% of the benchmarks from the domain and running CVC4 to collect all the formulas just before they are encoded to SAT.
2. Pattern finding (Section 3) on these formulas and filtering the terms based on capabilities of the synthesis phase of OptCNF.
3. Translation of each term to multiple SyGus problems one for each possible template that is suitable for the type and the size of the term. For problems involving non bit-parallel terms, OptCNF uses Sketch with 4 cores to parallelize the clause minimization algorithm (Section 2.5). All other problems use a single core. Each problem is also given a timeout of 3 hours.
4. Augmenting CVC4 code with the generated encoders (Section 4) and auto-tuning to find a subset of encoders that improve the performance (Section 5).

Different parts of OptCNF system were run on different machines. Pattern finding and synthesis of encoders were run on a machine with forty 2.4 GHz Intel Xeon processors and 96 GB RAM. For auto-tuning, we used a private cluster running OpenStack with parallelism of 150 on 75 virtual machines each with 4 cores and 8GB RAM of processing power. Finally, the performance experiment evaluating the solvers on QF_BV benchmarks was run on the StarExec [47] cluster infrastructure with a timeout of 900 seconds and a memory limit of 200 GB (similar to the resources used for the SMT competition).

6.2 Domains and Benchmarks

We generate a total of 7 domain-specific solvers and a general solver which is obtained by using the entire QF_BV benchmark suite for pattern finding and synthesis. For the general solver, we enable all the generated encoders and do not auto-tune them. The 7 domains are chosen from the 36 categories in QF_BV. We chose these categories based on the criteria that the number of benchmarks in the domain is at least 20 and the average run-time is significant enough to see an improvement. The solvers for these domains are referred by their category name.

6.3 Experiments

We focus on the following questions: (1) Can OptCNF generate domain-specific solvers in reasonable amount of time? (2) How does the performance of the domain-specific optimal solvers generated by OptCNF compare to CVC4? (3) How domain-specific are the encoders generated by OptCNF?

Time taken to generate optimal encoders Table 1 shows the number of generated (gen) and selected (sel) encoders (selected after auto-tuning, differentiated by the type of patterns), and, the total time taken to synthesize these encoders (both cpu time and clock time). In addition to this, Pattern Finding was run for an hour per domain and Auto-tuning was run for 7.5 hours per domain. In total, OptCNF was able to generate domain-specific encoders in $10-22$ hours per domain which is a reasonable amount of time as compared to a software engineer implementing and debugging encoders in a solver.

Table 1: Encoder statistics and Sketch running times

Domain	# boolean		# bit parallel		# non bit parallel		Total patterns		Synthesis time	
	gen	sel	gen	sel	gen	sel	gen	sel	(cpu hrs)	(clock hrs)
general	336	336	334	334	12	12	682	682	497	17
asp	29	22	0	0	4	3	33	25	8	2
brummayerbiere2	66	0	12	7	2	2	80	9	16	2
brummayerbiere3	35	0	13	3	5	3	53	6	15	3
bruttomesso	21	4	1	0	1	0	23	4	5	2
float	272	17	294	18	3	0	569	35	360	13
log-slicing	19	0	86	60	5	5	110	65	49	4
mcm	13	3	2	1	4	1	19	5	7	2

Impact of domain-specific solvers: With the exception of the general solver, all the other solvers are auto-tuned to select a subset of the generated encodings that improves the performance. For all domains except asp and bruttomesso, the training set for auto-tuning contains 50% benchmarks chosen randomly from the domain. For these domains, we perform 2-fold cross-validation i.e. we swap training/test sets and run auto-tuning again. For asp and bruttomesso, the training set contains only 20% benchmarks due to resource constraints for auto-tuning resulting from them having a large number of benchmarks. For these two domains, we run auto-tuning again for approximating cross-validation with another disjoint training set that contains 20% benchmarks form the domain.

We compare the performance of the domain-specific solvers (auto-tuned on the first training set) with the general solver and CVC4 in Table 2. Only the benchmarks from the first test set are considered for evaluation in the table. The best-performing solver for every domain is marked as bold. The auto-tuned solver solves 83 benchmarks more than CVC4 in total. For all domains, the domain-specific solvers outperform CVC4. The domain-specific solvers auto-tuned on the second training set for each domain also outperform CVC4 and solve 73 more benchmarks on their corresponding test sets (the details are omitted due to lack of space).

Table 2 also presents the performance of the Boolector solver (the best bit-vector solver in SMT-COMP'15) on the same test set benchmarks for reference. CVC4 is already better than Boolector on two domains (mcm, float) and OptCNF improves it slightly further. On one domain (log−slicing), CVC4 is notably worse than Boolector, but OptCNF makes it outperform Boolector. In addition, OptCNF significantly bridges the gap between CVC4 and Boolector on the mcm domain.

Table 2: Performance comparison: Domain-specific, general and CVC4 solvers on 7 categories of QF_BV benchmark suite (first training set)

	CVC4		general		Domain-Specific		Boolector	
Benchmark category	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
asp (365)	240	32652.8	238	33291.8	**288**	**34971.5**	308	29821.6
brummayerbiere2 (33)	28	1202.8	24	1653.2	**29**	**1691.0**	33	1371.2
brummayerbiere3 (40)	23	1165.2	23	2239.4	**24**	**1272.1**	32	1760.7
bruttomesso (676)	623	32880.8	604	35808.6	**623**	**32840.2**	774	8461.1
float (62)	59	4015.9	55	3599.6	**60**	**4395.5**	58	6152.9
log-slicing (79)	33	12636.1	57	17290.6	**62**	**21115.4**	53	9534.8
mcm (61)	40	3933.9	38	3355.0	**43**	**4193.0**	39	8333.1
	1046	88487.5	1039	97238.2	**1129**	**100479.8**	1297	65435.4

The run-times for benchmarks from domains where we did not perform auto-tuning can be found in the appendix. The general solver performs better on some domains but not the others, and, slightly worse than CVC4 overall. In all cases where we performed auto-tuning, the domain-specific solvers beat the general solver (Table 2). Two scatter plots showing the performance of CVC4 versus general and the domain-specific solvers on these 7 domains can be found in Figure 4. It is evident from the graphs that the domain-specific solvers reduce the number of negative points (in the upper left triangle) thereby improving the performance when compared to CVC4 overall.

Domain specificity: We ran each domain-specific solver (obtained from the first training set) on all the other domains and the results are summarized in Table 3. The best performing result for each domain is marked as bold and the results that are worse than CVC4 are underlined. 5 out of 7 of the domains

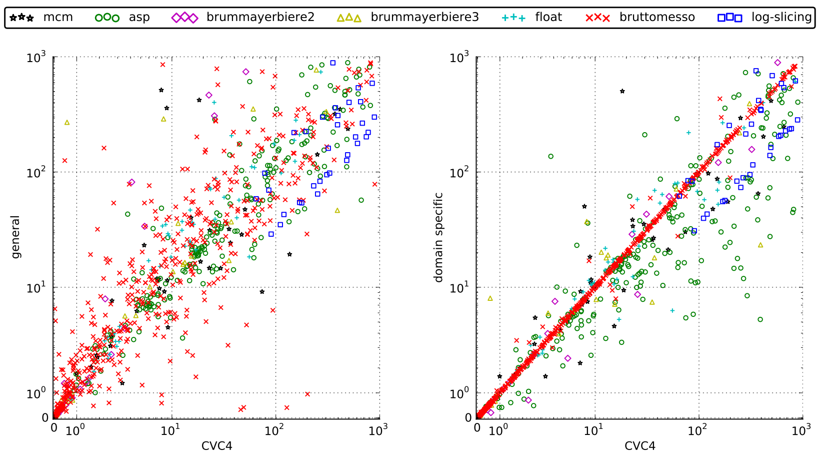

Fig. 4: Scatter plots showing run-times (log scale) for different solvers on the 7 domains

are very domain-specific; the solvers that are tuned specially for them perform significantly better than all the other solvers. In some cases, using one solver on another domain makes it worse than CVC4. However, mcm domain has a solver optimal for other two domains performing almost identical to their respective solvers.

Table 3: Cross-domain performance

solver →	asp		brummayerbiere2		brummayerbiere3		bruttomesso		float		log-slicing		mcm	
domain ↓	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
asp	**288**	**34971.5**	227	28173.5	253	34061.6	240	33118.9	236	29491.3	227	28230.5	255	35159.5
brummayerbiere2	28	786.9	**29**	**1691.0**	29	2363.0	29	2174.8	29	1804.2	29	1705.3	28	1706.6
brummayerbiere3	22	1206.7	22	1149.3	24	1272.1	23	1169.2	23	1410.1	22	911.3	**25**	**1945.6**
bruttomesso	606	37216.1	609	38744.1	623	32809.8	623	32840.1	623	32867.5	607	37164.7	**623**	**32683.5**
float	57	1650.8	57	2179.3	60	4853.1	59	3599.5	**60**	**4395.5**	57	1832.4	59	4100.9
log-slicing	58	20816.6	59	20125.7	35	12955.7	35	14640.7	32	11796.1	**62**	**21115.4**	36	14021.6
mcm	38	4301.6	40	3413.1	39	3411.2	41	3940.7	39	3759.5	39	5313.0	**43**	**4193.0**

7 Related Work

A recent paper [12] on automatically generating propagation complete encodings is the closest to this work. Encodings generated through OptCNF are propagation complete and OptCNF also minimizes the number of clauses across the template being used for the encoder similar to [12]. But, OptCNF is different in two important ways: (1) Instead of encodings, OptCNF generates *encoders* which produce encodings at run-time (enabled by program synthesis) (2) The generated encoders are specialized for a particular domain (enabled by pattern finding and auto-tuning).

Different notions of propagation strength of encodings have been considered in both Knowledge Compilation [18] (e.g. unit-refutation completeness [20] and its generalizations [27,28,29]) and Constraint Programming [5,13] communities. Propagation complete encodings (PCEs) have been established [10] to be "well-posed" for a SAT solver's deduction mechanism, which provides a tractable reasoning on the constraints. [10] reduces the problem of generating PCEs to iteratively solving QBF formulas whereas OptCNF relies on CEGIS based program synthesis [45] to generate encoders producing PCEs at run-time. There has also been some recent work on using SAT solvers for enumeration of prime implicants in the Knowledge Compilation community [27,28]. In Constraint Programming, Generalized Arc-Consistency (GAC) [5] is connected to propagation completeness and has been adopted in SAT [24] but is usually only enforced on input/output variables and not on auxiliary variables which provides a weaker notion of propagation strength as compared to PCEs. [9] shows that certain global constraints can require exponential sized formulas for PCEs. In our work, we do not encounter this issue since we consider only small patterns as constraints.

Reducing the size of the CNF encodings derived from SAT formulas has been shown to be an effective way of optimizing SAT solvers [23,15,30,22,40,51]. There has been a lot of work on optimal encodings for specific kinds of constraints like cardinality constraints [1], sequence constraints [13], verification of microprocessors [51]. There is also some work on logic minimization techniques like Beaver [37]. But, to our knowledge, we are the first ones to generate domain specific encodings that are propagation complete and minimal for multiple challenging domains using program synthesis technology.

OptCNF can be extended to other SMT solvers besides CVC4 such as Z3 [19], Beaver [37], Boolector [14] and Yices [21]. In Beaver and Boolector, intermediate data structures like And-Inverter graphs (AIGs) are employed and are later on transformed to CNF efficiently. Consequently, they have numerous optimizations on the AIG representation before translating it to CNF. Applying OptCNF directly to such solvers can override these optimizations and hence, requires more work. These solvers can also use lazy bit-blasting strategy as opposed to eager bit-blasting that we use in our experiments. OptCNF can be extended to solvers employing lazy bit-blasting by using the generated encodings at the time of bit-blasting.

Finally, algorithm configuration [34,4,33], an active area of research in artificial intelligence, has been used in generation of encodings for Planning Domain Models [50] and improving CSP solving by searching for optimal solver choices and the different encodings for the CSP constraints [31]. It has also been shown to be successful for tuning parameters for SAT solvers [36]. Unlike OpenTuner [3], where the optimization function is a black-box, algorithm configuration can use the structure of certain types of functions and employ additional heuristics [35,36] to optimize them.

8 Conclusion

In this paper, we presented a technique to generate propagation complete CNF encoders for bit-vector terms. We combined it with machine learning based techniques namely pattern finding and auto-tuning to generate domain-specific solvers. Our evaluation showed that this technique can significantly improve CVC4, a state of the art SMT solver, on the domains represented in the bit-vector benchmark suite from SMT-COMP 2015.

Acknowledgments: This research was partially supported by NSF award #1139056 (ExCAPE) and by DARPA MUSE award #FA8750-14-2-0270.

References

1. I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, chapter A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints, pages 80–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
2. R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 40:1–25, 2015.
3. J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In J. N. Amaral and J. Torrellas, editors, *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 303–316. ACM, 2014.
4. C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In I. P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2009.
5. F. Bacchus. Gac via unit propagation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, pages 133–147, Berlin, Heidelberg, 2007. Springer-Verlag.
6. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
7. C. Barrett, M. Deters, L. Moura, A. Oliveras, and A. Stump. 6 years of smt-comp. *Journal of Automated Reasoning*, 50(3):243–277, 2012.
8. C. W. Barrett, L. de Moura, and A. Stump. Smt-comp: Satisfiability modulo theories competition. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer, 2005. `http://smtcomp.sourceforge.net/2016/`.

9. C. Bessiere, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In C. Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 412–418, 2009.
10. L. Bordeaux and J. Marques-Silva. Knowledge compilation with empowerment. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'12, pages 612–624, Berlin, Heidelberg, 2012. Springer-Verlag.
11. E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press.
12. M. Brain, L. Hadarean, D. Kroening, and R. Martins. Automatic generation of propagation complete sat encodings. In *Verification, Model Checking, and Abstract Interpretation*, pages 536–556. Springer, 2016.
13. S. Brand, N. Narodytska, C. Quimper, P. J. Stuckey, and T. Walsh. Encodings of the sequence constraint. In C. Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 2007.
14. R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.
15. B. Chambers, P. Manolios, and D. Vroon. Faster sat solving with better cnf generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1590–1595, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.
16. A. Cheung, A. Solar-Lezama, and S. Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145, New York, NY, USA, 2011. ACM.
17. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 236–250, Berlin, Heidelberg, 2010. Springer-Verlag.
18. A. Darwiche and P. Marquis. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.
19. L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
20. A. del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94). Bonn, Germany, May 24-27, 1994.*, pages 551–561. Morgan Kaufmann, 1994.
21. B. Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer, 2014.

22. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, SAT'05, pages 61–75, Berlin, Heidelberg, 2005. Springer-Verlag.
23. N. Een, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up sat. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, SAT'07, pages 272–286, Berlin, Heidelberg, 2007. Springer-Verlag.
24. I. P. Gent. Arc consistency in SAT. In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.
25. P. Godefroid. Test generation using symbolic execution. In D. D'Souza, T. Kavitha, and J. Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPIcs*, pages 24–33. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
26. S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 120–135, Berlin, Heidelberg, 2009. Springer-Verlag.
27. M. Gwynne and O. Kullmann. Generalising and unifying SLUR and unit-refutation completeness. In P. van Emde Boas, F. C. A. Groen, G. F. Italiano, J. R. Nawrocki, and H. Sack, editors, *SOFSEM 2013: Theory and Practice of Computer Science, 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 26-31, 2013. Proceedings*, volume 7741 of *Lecture Notes in Computer Science*, pages 220–232. Springer, 2013.
28. M. Gwynne and O. Kullmann. Towards a theory of good SAT representations. *CoRR*, abs/1302.4421, 2013.
29. M. Gwynne and O. Kullmann. Generalising unit-refutation completeness and SLUR via nested input resolution. *J. Autom. Reasoning*, 52(1):31–65, 2014.
30. M. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2010.
31. B. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In H. Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming - 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings*, volume 8451 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2014.
32. F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the Formal Methods in Computer Aided Design*, FMCAD '07, pages 27–34, Washington, DC, USA, 2007. IEEE Computer Society.
33. F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In C. A. C. Coello, editor, *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer, 2011.
34. F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.*, 36(1):267–306, Sept. 2009.

35. F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1152–1157. AAAI Press, 2007.
36. F. Hutter, M. T. Lindauer, A. Balint, S. Bayless, H. H. Hoos, and K. Leyton-Brown. The configurable SAT solver challenge (CSSC). *CoRR*, abs/1505.01221, 2015.
37. S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 668–674, Berlin, Heidelberg, 2009. Springer-Verlag.
38. N. Manthey, M. Heule, and A. Biere. Automated reencoding of boolean formulas. In A. Biere, A. Nahir, and T. E. J. Vos, editors, *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2012.
39. R. Martins, V. M. Manquinho, and I. Lynce. Exploiting cardinality encodings in parallel maximum satisfiability. In *IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011*, pages 313–320. IEEE Computer Society, 2011.
40. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.
41. C. Nguyen, H. Yoshida, M. R. Prasad, I. Ghosh, and K. Sen. Generating Succinct Test Cases Using Don't Care Analysis. In *Proceedings of the Eighth IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.
42. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.
43. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.
44. R. Singh and A. Solar-Lezama. Automatic generation of formula simplifiers based on conditional rewrite rules,arxiv:1602.07285, 2016.
45. A. Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS Dept., UC Berkeley, 2008.
46. S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.
47. A. Stump, G. Sutcliffe, and C. Tinelli. Introducing starexec: a cross-community infrastructure for logic solving. In V. Klebanov, B. Beckert, A. Biere, and G. Sutcliffe, editors, *COMPARE*, volume 873 of *CEUR Workshop Proceedings*, page 2. CEUR-WS.org, 2012.
48. H. Tanno, X. Zhang, T. Hoshino, and K. Sen. Tesma and catg: Automated test generation tools for models of enterprise applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 717–720, Piscataway, NJ, USA, 2015. IEEE Press.
49. G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

50. M. Vallati, F. Hutter, L. Chrpa, and T. L. McCluskey. On the effective configuration of planning domain models. In Q. Yang and M. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1704–1711. AAAI Press, 2015.

51. M. N. Velev. Efficient translation of boolean formulas to cnf in formal verification of microprocessors. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 310–315, Piscataway, NJ, USA, 2004. IEEE Press.

52. X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. A differential approach to undefined behavior detection. *Commun. ACM*, 59(3):99–106, 2016.

9 Appendix

Table 4 shows the performance comparison between domain specific solvers that is auto-tuned on the second training set and CVC4. These are evaluated on the corresponding second test sets and hence, are not directly comparable to Table 2.

Table 4: Performance comparison: Domain-specific, general and CVC4 solvers on 7 categories of QF_BV benchmark suite (second training set)

	CVC4		general		Domain-Specific		Boolector	
Benchmark category	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
asp (365)	237	36330.9	228	33828.7	**273**	**37212.5**	300	31661.7
brummayerbiere2 (32)	26	1030.6	21	1885.6	**29**	**3390.7**	31	999.5
brummayerbiere3 (39)	17	1653.4	16	2333.7	**18**	**2465.4**	27	668.4
bruttomesso (676)	**621**	**30642.0**	610	35967.9	620	31492.5	774	7832.0
float (62)	53	3829.2	53	7670.0	**54**	**4086.5**	49	6462.2
log-slicing (79)	29	9340.1	57	17955.2	**58**	**17465.5**	60	11098.1
mcm (61)	39	3159.6	39	6382.2	**43**	**4274.5**	40	9379.3
	1022	85985.8	1024	106023.3	**1095**	**100387.6**	1281	68101.2

Table 5 shows the performance comparison of CVC4 with general solver on domains we didn't run auto-tuning on. Here, we eliminate rows where the performance of both the solvers were similar.

Table 5: Performance comparison between general optimal solver and CVC4 on the other domains of QF_BV benchmarks

	CVC4		general		Boolector	
Benchmark category	solved	time (s)	solved	time (s)	solved	time (s)
VS3 (10)	**2**	**742.2**	0	0.0	3	434.9
uclid (416)	**416**	**1625.3**	416	1981.6	416	450.9
tacas07 (5)	5	1257.0	**5**	**831.1**	5	251.3
stp_samples (426)	**424**	**72.4**	424	182.3	426	9.9
spear (15)	**12**	**251.8**	12	786.1	12	128.0
sage (22390)	**22390**	**6225.7**	22390	7683.3	22390	3690.3
brummayerbiere (52)	39	2611.4	**39**	**1714.7**	41	448.4
bmc-bv (135)	135	520.2	**135**	**473.9**	135	51.9
fft (16)	**8**	**886.2**	7	41.0	9	597.1
calypto (16)	9	2.99	**11**	**985.1**	15	1447.8
	23440	14195.2	**23339**	**14679.1**	23452	7510.5

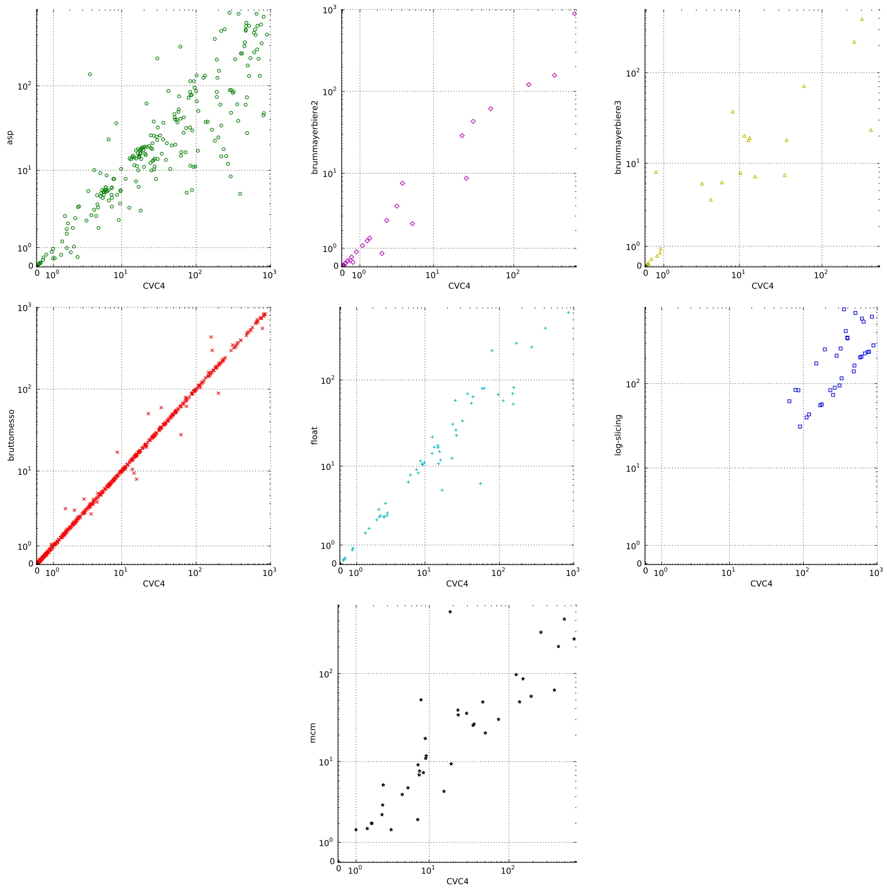

Fig. 5: Scatter plots for performance comparison between domain specific solvers and CVC4 for each domain