Automatic Generation of Propagation Complete SAT Encodings

Martin Brain[1], Liana Hadarean[1], Daniel Kroening[1], and Ruben Martins[2]

[1] Department of Computer Science, University of Oxford, UK
`first.last@cs.ox.ac.uk`
[2] University of Texas at Austin, USA
`rmartins@cs.utexas.edu`

Abstract. Almost all applications of SAT solvers generate Boolean formulae from higher level expression graphs by encoding the semantics of each operation or relation into propositional logic. All non-trivial relations have many different possible encodings and the encoding used can have a major effect on the performance of the system. This paper gives an abstract satisfaction based formalisation of one aspect of encoding quality, the *propagation strength*, and shows that propagation complete SAT encodings can be modelled by our formalism and automatically computed for key operations. This allows a more rigorous approach to designing encodings as well as improved performance.

1 Introduction

Almost all industrial applications of SAT solvers translate from a higher level language into propositional logic. Many of these translations are *modular* in the sense that each sub-expression is encoded into a set of clauses whose structure is independent of how the expression is used. For example, an SMT solver can use the same template to generate clauses for every occurrence of a 64-bit multiplication operation.

For most non-trivial expressions, there are many different encodings available. For example, there are several ways to encode cardinality constraints [4,37,1]. These may use different clauses and possibly introduce auxiliary variables to simplify and compact the encodings. The choice of encoding can have a significant impact on the performance of the solver [35]. This difference can be large enough that identifying a bad encoding from the CNF it generates and then replacing it with a better one *within the SAT solver* can give a net improvement in solver performance [34]. Despite the importance of choosing a good encoding there remain open questions about why some encodings perform better than others. A common rule of thumb is that smaller encodings (primarily in terms of number clauses but also in the number of variables) are preferable. For some kinds of encoding, for example cardinality constraints, arc consistency [24] is regarded to be a desirable property. Another desirable property is being propagation complete [11]. Encodings with this property are considered extremely important since constraint solvers can benefit from the increase in inference power. However, its use is not yet wide spread in encoding design within the SMT community.

These issues are particularly relevant in encodings of bit-vector and floating-point operations. Often the only way to tell if an encoding might be better than another is to implement it and then compare system level performance on a 'representative' set of benchmarks. Furthermore, the encodings commonly used are frequently literal translations of circuits designs used to implement these operations in hardware. These designs were created to minimise signal propagation delay, to reduce area or for power and layout concerns. It is not clear why a multiplier hardware design with low cycle count should give a good encoding from bit-vector logic to CNF.

This paper advances both the theory and practice of the creation of encodings through the following contributions:

- Section 3 uses and extends the framework of abstract satisfiability [19] to formalise one aspect of encoding quality: *propagating strength*. We show that propagation complete encodings [11] are modelled by our framework and can serve as a basis for comparing encodings.
- An algorithm is given in Section 4 which can be used to determine if an encoding is propagation complete, strengthen it if it is not or generate a propagation complete encoding from scratch with and without auxiliary variables.
- In Section 5 we show that using our propagation complete encodings improves the performance of the CVC4 SMT solver on a wide range of bit-vector benchmarks.

2 Abstract Satisfaction

The *abstract satisfaction* framework [19] uses the language of abstract interpretation to characterise and understand the key components in a SAT solver [17]. One advantage of this viewpoint is that it is largely independent of the concrete domain that is being searched (sets of assignments) or the abstract domain used to represent information about the search (partial assignments). This allows the CDCL algorithm to be generalised [18] and applied to a range of other domains [27,20,12]. Another important feature of the abstract satisfaction framework is that it allows the *representation of a problem* and the *effects of reasoning* to be cleanly formalised. As we show later, this allows us to characterise propagation algorithms, such as unit propagation, as a map from representation to effect. In this section we recall some background results required to formalise this idea.

The foundation of abstract interpretation is using an *abstract domain* to perform approximate reasoning about a *concrete domain*. This requires a relation between the two domains; with Galois connections being one of the simplest and most popular choices.

Definition 1. *Let* $(C, \subseteq)$ *and* $(A, \sqsubseteq)$ *be sets with partial orders. The pair* $(\alpha : C \to A, \gamma : A \to C)$ *form a* Galois connection *if:*

$$\forall c \in C, a \in A \,.\, \alpha(c) \sqsubseteq a \Leftrightarrow c \subseteq \gamma(a)$$

C *is referred to as the* concrete domain *and* A *is the* abstract domain. *It is sometimes useful to use an equivalent definition of Galois connection:* α *and* γ *are monotone and*

$$\forall c \in C \,.\, c \subseteq \gamma(\alpha(c)) \qquad \forall a \in A \,.\, \alpha(\gamma(a)) \sqsubseteq a$$

If, additionally, $\gamma \circ \alpha = id$*, then the pair is referred to as a* Galois insertion *and each element of the concrete domain has one or more representations in the abstract domain.*

Given the domain that we want to reason about and the abstraction that will be used to perform the reasoning, the next step is to characterise the reasoning as *transformers*.

Definition 2. *A* concrete transformer *is a monotonic function* $f : C \to C$*. Many of the transformers of interest are* extensive*,* reductive *or* idempotent*, respectively defined as:*

$$\forall c \in C \centerdot c \subseteq f(c) \qquad \forall c \in C \centerdot f(c) \subseteq c \qquad f \circ f = f$$

A function that is extensive, monotonic and idempotent is referred to as an upper closure *while a reductive, monotonic, idempotent function is referred to as a* lower closure*.*

Finally, we will need a means of approximating the transformer on the abstract domain using an *abstract transformer*. This gives a key result: the space of abstract transformers (for a given concrete transformer) forms a lattice with a unique *best abstract transformer*.

Definition 3. *Given a transformer* f *on* C*,* $f_o : A \to A$ *is an* (over-approximate) abstract transformer *if:*

$$\forall a \in A \centerdot \alpha(f(\gamma(a))) \subseteq f_o(a)$$

Proposition 1. *Given a reductive transformer* f *on a lattice* $(C, \subseteq)$*, the set of abstract transformers on lattice* $(A, \sqsubseteq)$ *form a lattice with the bottom element, referred to as the* best abstract transformer*, is equal to:*

$$\alpha \circ f \circ \gamma$$

3 Characterising Propagating Strength

While the framework we introduce in this section generalizes to other domains, we will focus on CNF encodings targeting CDCL-style SAT solvers [9]. We only consider unit propagation, but other propagation algorithms, such as generalised unit propagation [33], can be treated in the same way.

A number of attributes can be used for evaluating encodings. Some of these are algorithmic such as how much information it can propagate, how it affects the quality of learnt clauses, how it interacts with the branching heuristic or what effect it has on preprocessing. Others are more implementation-oriented: how many variables it uses, how many clauses it contains and how many are binary, ternary, how quickly it propagates, etc. In this work we will be characterising one of the major algorithmic properties: the amount of information that can be propagated.

Informally, this can be thought of as the proportion of facts that are true (with respect to the current partial assignment and encoding) that can be proven with unit propagation. If E is an encoding, l is a literal and p is a partial assignment expressed as a conjunct of all of the assigned literals, then it is the degree to which:

$$\mathsf{p} \wedge \mathsf{E} \models l \quad \text{implies} \quad \mathsf{p} \wedge \mathsf{E} \vdash_{up} l,$$

where $\models$ represents logical entailment and $\vdash_{up}$ stands for unit propagation.

We formalise this intuition using the viewpoint of abstract satisfaction. Figure 1 gives a visual summary of the formalisation; the key steps are:

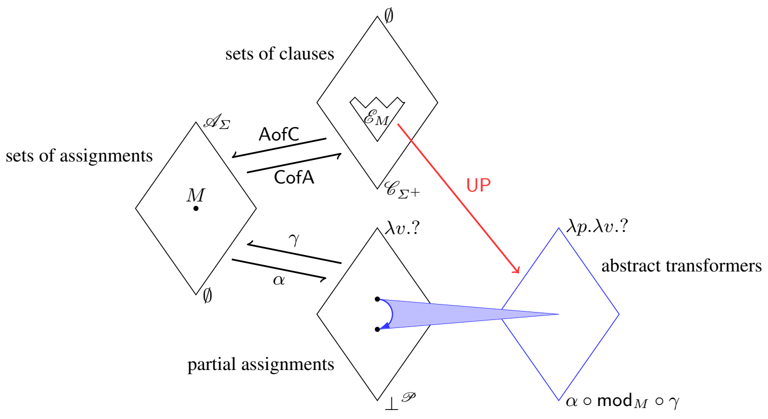
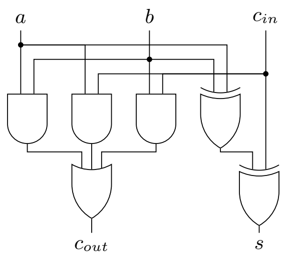


Fig. 1: A graphical presentation of the results in Section 3

a	b	c_{in}	c_{out}	s
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

(a) Truth table



(b) A basic adder circuit

Fig. 2: A nest of adders

1. Present syntax as an abstraction of semantics and define the space of encodings of a set of assignments as a substructure of the syntax lattice (Subsection 3.1).
2. Show that partial assignments, the information about possible models that is manipulated during the search, is also an abstraction of the semantics (Subsection 3.2).
3. Express the *effects of reasoning* as abstract transformers and characterise propagation algorithms such as unit propagation as *maps from representations of a problem to the effects of reasoning* (Subsection 3.3).

3.1 Syntax and Semantics

We first fix a set of *variable names* Σ. This will include the 'input' and 'output' bits of the encoding, plus any auxiliaries. Let Σ^+ be the set of *literals* constructed from these variables (i.e. $\Sigma^+ = \{v | v \in \Sigma\} \cup \{\neg v | v \in \Sigma\}$). For simplicity we will assume double negation is always simplified $\neg\neg v = v$.

A *clause* is a disjunction of one or more literals. For convenience we will identify clauses with the set of literals they contain. A clause is a *tautology* if it contains a literal and its negation. Let $\mathscr{C}_{\Sigma^+}$ be the set of non-tautological clauses which can be constructed from Σ^+. We identify sets of clauses with their conjunction. Let $2^{\mathscr{C}_{\Sigma^+}}$ denote the powerset of $\mathscr{C}_{\Sigma^+}$ and note that it forms a complete lattice ordered by $\supseteq$. For convenience we will pick $\emptyset$ to be the top element and $\mathscr{C}_{\Sigma^+}$ to be the bottom.

Example 1. We will use a full adder as a running example. Figure 2 shows one possible circuit that can be used to implement a full adder as well as the truth table for the input which gives the 8 possible satisfying assignments of the formula. In this case $\Sigma = \{a, b, c_{in}, s, c_{out}\}$ so $\Sigma^+ = \{a, b, c_{in}, s, c_{out}, \neg a, \neg b, \neg c_{in}, \neg s, \neg c_{out}\}$. Thus $\{a\}$, $\{b, \neg a\}$, $\{s, \neg s\}$ are clauses and only the last is a tautology. Also $\mathscr{C}_{\Sigma^+} = \{\emptyset, \{a\}, \{b\}, \{\neg a\}, \{a, b\}, \{a, \neg b\}, \ldots\}$.

An *assignment* is a map from Σ to $\{\top, \bot\}$ and the set of all assignments is denoted by $\mathscr{A}_\Sigma$. Similarly $2^{\mathscr{A}_\Sigma}$ forms a powerset lattice. Following usual convention (and the opposite of the syntax lattice), the top element will be $\mathscr{A}_\Sigma$ and $\emptyset$ the bottom. With a slight abuse of notation, we use assignments to give literals values: $\mathsf{x}(\neg a) = \neg \mathsf{x}(a)$.

The *models* relation, denoted using an infix $\models$, is a relationship between $\mathscr{A}_\Sigma$ and $\mathscr{C}_{\Sigma^+}$, defined as follows:

$$\mathsf{x} \models c \Leftrightarrow \exists l \in c \,\centerdot\, \mathsf{x}(l) = \top$$

An assignment is a *model* of a set of clauses if the models relation holds for all of the clauses in the set.

Example 2. An assignment for the full adder example would be:
$\mathsf{x} = \{(a, \top), (b, \top), (c_{in}, \bot), (c_{out}, \top), (s, \bot)\}$.
From this we can see that $\mathsf{x} \models \{a\}$ and $\mathsf{x} \models \{a, \neg c_{in}, c_{out}\}$ but $\mathsf{x} \not\models \{\neg b, \neg a\}$. So x is a model of $\{\{a\}, \{a, \neg c_{in}, c_{out}\}\}$.

This relation gives maps $\mathsf{AofC} : 2^{\mathscr{C}_{\Sigma^+}} \to 2^{\mathscr{A}_\Sigma}$ and $\mathsf{CofA} : 2^{\mathscr{A}_\Sigma} \to 2^{\mathscr{C}_{\Sigma^+}}$:

$$\mathsf{AofC}(C) = \{\mathsf{x} \in \mathscr{A}_\Sigma | \forall c \in C \,\centerdot\, \mathsf{x} \models c\}$$
$$\mathsf{CofA}(A) = \{c \in \mathscr{C}_{\Sigma^+} | \forall \mathsf{x} \in A \,\centerdot\, \mathsf{x} \models c\}$$

$\mathsf{AofC}(C)$ is the set of assignments which are models of C, while $\mathsf{CofA}(A)$ is all of the clauses that are consistent with all of the assignments in A. Both maps are monotonic, $\mathsf{AofC}(\mathsf{CofA}(A)) = A$ and $\mathsf{CofA}(\mathsf{AofC}(C)) \supseteq C$ so they form a *Galois insertion* between $2^{\mathscr{A}_\Sigma}$ and $2^{\mathscr{C}_{\Sigma^+}}$. *A set of clauses is a representation, or abstraction, of its set of models.*

Example 3. Given $C = \{\{a, \neg b\}, \{\neg a\}\}$, the set of all models of C is $\mathsf{AofC}(C) = \{\mathsf{y} : \Sigma \to \{\top, \bot\} | \mathsf{y}(a) = \bot \wedge \mathsf{y}(b) = \bot\}$. Conversely, $\mathsf{CofA}(\{\mathsf{x}\}) = \{\{a\}, \{a, b\}, \{a, \neg b\}, \ldots\}$ is the set containing all of the clauses consistent with the assignment x from Example 2. When multiple assignments are given this is all of the clauses that are consistent with *all* of the assignments.

In the SAT field, similar Galois connections to the one presented in this section have been studied in [32]. Although we have presented this result with Boolean valuations (the "concrete" domain) and CNF (the "abstract" domain), the construction is much

$\{\neg a, \neg b, \neg c_{in}, \neg c_{out}, s\}$	$\{\neg a, \neg b, \neg c_{in}, c_{out}, \neg s\}$	$\{\neg a, \neg b, \neg c_{in}, c_{out}, s\}$
$\{\neg a, \neg b, c_{in}, \neg c_{out}, \neg s\}$	$\{\neg a, \neg b, c_{in}, c_{out}, \neg s\}$	$\{\neg a, \neg b, c_{in}, c_{out}, s\}$
$\{\neg a, b, \neg c_{in}, \neg c_{out}, \neg s\}$	$\{\neg a, b, \neg c_{in}, c_{out}, \neg s\}$	$\{\neg a, b, \neg c_{in}, c_{out}, s\}$
$\{\neg a, b, c_{in}, \neg c_{out}, \neg s\}$	$\{\neg a, b, c_{in}, \neg c_{out}, s\}$	$\{\neg a, b, c_{in}, c_{out}, s\}$
$\{a, \neg b, \neg c_{in}, \neg c_{out}, \neg s\}$	$\{a, \neg b, \neg c_{in}, c_{out}, \neg s\}$	$\{a, \neg b, \neg c_{in}, c_{out}, s\}$
$\{a, \neg b, c_{in}, \neg c_{out}, \neg s\}$	$\{a, \neg b, c_{in}, \neg c_{out}, s\}$	$\{a, \neg b, c_{in}, c_{out}, s\}$
$\{a, b, \neg c_{in}, \neg c_{out}, \neg s\}$	$\{a, b, \neg c_{in}, \neg c_{out}, s\}$	$\{a, b, \neg c_{in}, c_{out}, s\}$
$\{a, b, c_{in}, \neg c_{out}, \neg s\}$	$\{a, b, c_{in}, \neg c_{out}, s\}$	$\{a, b, c_{in}, c_{out}, \neg s\}$

(a) Naïve truth table encoding

$\{\neg a, \neg b, c_{in}, \neg s\}$	$\{\neg a, b, \neg c_{in}, \neg s\}$	$\{a, \neg b, \neg c_{in}, \neg s\}$	$\{a, b, c_{in}, \neg s\}$
$\{\neg a, \neg b, \neg c_{in}, s\}$	$\{\neg a, b, c_{in}, s\}$	$\{a, b, \neg c_{in}, s\}$	$\{a, \neg b, c_{in}, s\}$
$\{\neg a, \neg b, c_{out}\}$	$\{\neg a, \neg c_{in}, c_{out}\}$	$\{\neg b, \neg c_{in}, c_{out}\}$	
$\{a, b, \neg c_{out}\}$	$\{a, c_{in}, \neg c_{out}\}$	$\{b, c_{in}, \neg c_{out}\}$	

(b) Eén and Sörensson's basic encoding

$\{c_{in}, \neg s, \neg c_{out}\}$	$\{a, \neg s, \neg c_{out}\}$	$\{b, \neg s, \neg c_{out}\}$	$\{a, b, c_{in}, \neg s\}$
$\{\neg a, \neg b, \neg c_{in}, s\}$	$\{\neg c_{in}, s, c_{out}\}$	$\{\neg a, s, c_{out}\}$	$\{\neg b, s, c_{out}\}$
$\{\neg a, \neg b, c_{out}\}$	$\{\neg a, \neg c_{in}, c_{out}\}$	$\{\neg b, \neg c_{in}, c_{out}\}$	
$\{a, b, \neg c_{out}\}$	$\{a, c_{in}, \neg c_{out}\}$	$\{b, c_{in}, \neg c_{out}\}$	

(c) A propagation complete encoding

Fig. 3: A nest of adder encodings

more general and can be applied to SMT, CSP, ASP, etc. For more discussion of the Galois connection between syntax and semantics, see [21].

Given a set of assignments $M \subset \mathscr{A}_{\Sigma}$, an *encoding (of M)* is any set of clauses $C \subset \mathscr{C}_{\Sigma^+}$ such that $\mathsf{AofC}(C) = M$. We shall denote the set of encodings of M as $\mathscr{E}_M = \{C \subset \mathscr{C}_{\Sigma^+} | \mathsf{AofC}(C) = M\}$. If C and D are both encodings (of the same set of models), then so is $C \cup D$; this is the basis for redundant encodings in CSP. It also implies that the encodings of a set of models form a meet semi-lattice with a minimum element, $\mathsf{CofA}(M)$, the most verbose encoding. There can be multiple, incomparable, least verbose encodings. For example if $M = \emptyset$, then $\{a, \neg a\}$ is a least verbose encoding (as there are no proper subsets which are encodings), but so is $\{b, \neg b\}$. This notion of encoding has been studied is the SAT field (e.g. [23]) and has recently been formalised as a formula that has the same satisfying assignments as the set of assignments of a given specification [26].

Example 4. Continuing our example of a full adder, let M be the set of eight models described by the truth table in Figure 2a. There are many possible encodings, some of which are given in Figure 3. All of these are subsets of $\mathsf{CofA}(M)$, all the clauses consistent with M, in effect, the 'theory' of the full adder. However, not every subset of $\mathsf{CofA}(M)$ is an encoding, as they are required to have the same models as M. Possible encodings include the naive encoding (Figure 3a) in which all full assignments that are not models are removed, the basic encoding given by [23] (Figure 3b) and a propagation complete encoding (Figure 3c). Notice that the first two encodings are not propagation complete.

To formally define propagation strength, we will need a notion of what kind of information we are propagating and to relate the encoding to the action of propagation.

3.2 Representing Information During Search

Some propositional logic tools, such as BDDs, represent sets of models directly. For solving SAT problems this is not really viable — as soon as you have a model that you could represent, you have solved the problem. Thus SAT algorithms need a way of representing partial information about models. For example if an encoding contains the clause $\{\neg a\}$ then the SAT solver needs a way of recording "there are no models that assign a to $\top$". The most common approach is to use *partial assignments*.

Following [17] we characterise a partial assignment over Σ ($\mathscr{P}_\Sigma$ denotes the set of all of them) as an abstraction of $2^{\mathscr{A}_\Sigma}$. Partial assignments are maps from Σ to $\{\top, ?, \bot\}$, where ? denotes an unknown or unassigned variable. They can be ordered by:

$$\mathsf{p} \sqsubseteq \mathsf{q} \Leftrightarrow \forall v \in \Sigma . \mathsf{q}(v) \neq ? \Rightarrow \mathsf{p}(v) = \mathsf{q}(v)$$

Allowing an additional 'contradiction' partial assignment, $\bot^{\mathscr{P}}$, ordered below all other partial assignments, makes $\mathscr{P}_\Sigma$ a complete lattice, where $\top^{\mathscr{P}} = \lambda v.?$ is the partial assignment that does not assign any variables. The discussion below generalises to other abstractions; we use partial assignments as they are a popular and simple choice.

Example 5. In our running example, p and q are partial assignments:

$$\mathsf{p} = \{(a, ?), (b, \bot), (c_{in}, \bot), (s, \top), (c_{out}, \top)\}$$
$$\mathsf{q} = \{(a, ?), (b, ?), (c_{in}, ?), (s, \top), (c_{out}, \top)\}$$

with $\mathsf{p} \sqsubseteq \mathsf{q}$ because where q assigns a variable to $\top$ or $\bot$, p agrees.

To use $\mathscr{P}_\Sigma$ as an abstraction of $2^{\mathscr{A}_\Sigma}$, we need to define a Galois connection between them. Let $\alpha : 2^{\mathscr{A}_\Sigma} \rightarrow \mathscr{P}_\Sigma$ denote the map from models to the most complete partial assignment that is consistent with all of them and $\gamma : \mathscr{P}_\Sigma \rightarrow 2^{\mathscr{A}_\Sigma}$ denote the map from a partial assignment to the set of models that is consistent with it:

$$\alpha(A) = \bigsqcup_{\mathsf{x} \in A} x \qquad \gamma(\mathsf{p}) = \{\mathsf{x} \in \mathscr{A}_\Sigma | \forall v \in \Sigma \mathbin{.} \mathsf{p}(v) \neq ? \Rightarrow \mathsf{x}(v) = \mathsf{p}(v)\}$$

Example 6. Let $\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3$ and x_4 be (full) assignments:

$$\mathsf{x}_1 = \{(a, \top), (b, \top), (c_{in}, \bot), (s, \bot), (c_{out}, \top)\}$$
$$\mathsf{x}_2 = \{(a, \top), (b, \bot), (c_{in}, \top), (s, \bot), (c_{out}, \top)\}$$
$$\mathsf{x}_3 = \{(a, \top), (b, \top), (c_{in}, \top), (s, \bot), (c_{out}, \top)\}$$
$$\mathsf{x}_4 = \{(a, \top), (b, \bot), (c_{in}, \bot), (s, \bot), (c_{out}, \top)\}$$

then:

$$\alpha(\{\mathsf{x}_1, \mathsf{x}_2\}) = \{(a, \top), (b, ?), (c_{in}, ?), (s, \bot), (c_{out}, \top)\}$$
$$\gamma(\alpha(\{\mathsf{x}_1, \mathsf{x}_2\})) = \{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4\}$$

3.3 Effects of Reasoning

Having defined partial assignments as the 'units' of information that propagation uses, the next step is to formalize what kind of reasoning we are performing. In a SAT solver the role of reasoning is to add to a partial assignment p (i.e., reduce the set of assignments that is being considered) that is consistent with all of the models in $\gamma(\mathsf{p})$. Formally, this is expressed in two steps: a *models transformer* on the concrete domain, $2^{\mathscr{A}_\Sigma}$, which captures the kind of reasoning that we are approximating and abstract transformers on $\mathscr{P}_\Sigma$, which express the actual changes to the partial assignments.

In slight variation from [18] we define the *models transformer*, $\mathsf{mod}_M : 2^{\mathscr{A}_\Sigma} \to 2^{\mathscr{A}_\Sigma}$, as parameterised by a set of assignments rather than a formula:

$$\mathsf{mod}_M(A) = M \cap A$$

This is a downward closure function on $2^{\mathscr{A}_\Sigma}$ and expresses the ideal reasoning, or, conversely, the limit of what is sound.

Example 7. In the full adder example, let M be the set of assignments described in the truth table in Figure 2a. If $A = \{\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4\}$, then $\mathsf{mod}_M(A) = \{\mathsf{x}_1, \mathsf{x}_2\}$ as these are the only two assignments in A that are also models of the full adder.

As $2^{\mathscr{A}_\Sigma}$ is not directly representable for problems of significant size, we use $\mathscr{P}_\Sigma$. Likewise, we cannot directly implement mod_M so instead we must use over-approximate transformers on $\mathscr{P}_\Sigma$. Let $\mathscr{T}_{\mathsf{mod}_M}$ denote the set of abstract transformers that over-approximate mod_M and recall from Proposition 1 that they can be ordered point-wise to form a lattice with id as the top element and $\alpha \circ \mathsf{mod}_M \circ \gamma$ as the bottom. The *effect* of a sound propagator or other form of reasoning should be an abstract transformer, as they soundly add to partial assignment.

The final link is to connect the encoding used to the effect of reasoning. To do this we consider the unit propagation algorithm as a map from $\mathsf{UP} : \mathscr{E}_M \to (\mathscr{P}_\Sigma \to \mathscr{P}_\Sigma)$ that uses a set of clauses to add assignments to a partial assignment.

Definition 4. *Let* $\mathsf{up} : \mathscr{C}_{\Sigma+} \to (\mathscr{P}_\Sigma \to \mathscr{P}_\Sigma)$ *map clauses to functions on partial assignments.*

$$assign(l) = \lambda k. \begin{cases} \top & k = l \\ \bot & k = \neg l \\ ? & \textit{otherwise} \end{cases}$$

$$\mathsf{up}(c) = \lambda p. \begin{cases} p \sqcap assign(l) & \exists l \in c \,.\, p(l) = ? \wedge \forall k \in c \,.\, k \neq l \Rightarrow p(k) = \bot \\ p & \textit{otherwise} \end{cases}$$

Define UP *as the (greatest) fix-point of applying* $\mathsf{up}(c)$ *for each clause in the encoding:*

$$\mathsf{UP}(C)(p) = \mathit{GFP}\left(\lambda q.p \sqcap (\prod_{c \in C} \mathsf{up}(c)(q))\right)$$

Example 8. Given the set C clauses in Figure 3b we have:

$$\mathsf{UP}(C)(\{(a,\top),(b,\top),(c_{in},\top),(s,?),(c_{out},?)\}) = \\ \{(a,\top),(b,\top),(c_{in},\top),(s,\top),(c_{out},\top)\})$$

as the clause $\{\neg a, \neg b, c_{out}\}$ assigns c_{out} to $\top$ and $\{\neg a, \neg b, \neg c_{in}, s\}$ assigns s to $\top$.

Formalised in this manner, UP has a number of useful order-theoretic properties:

Proposition 2. *Given $C, D \subset \mathscr{C}_{\Sigma^+}$, $\mathsf{UP}(C_i)$ is a closure function as:*

$$\mathsf{UP}(C) \leqslant id \qquad C \leqslant D \implies \mathsf{UP}(C) \leqslant \mathsf{UP}(D) \qquad \mathsf{UP}(C) \circ \mathsf{UP}(C) = \mathsf{UP}(C)$$

Note that UP is neither injective ($\mathsf{up}(\{\{a\},\{b\}\}) = \mathsf{up}(\{\{a\},\{b\},\{\neg a, b\}\})$) nor surjective. Furthermore, UP does not preserve meets (well defined on encodings) or joins (partially defined on encodings, fully defined on supersets of a given encoding). A propagation algorithm that preserves joins would give a Galois connection between supersets of an encoding and abstract transformers, thus giving a unique, minimal encoding required to give a certain amount of inference.

The final step is to show that the closure functions given by $\mathsf{UP}(C)$ are abstract transformers and that they include the best abstract transformer.

Theorem 1. *Let $M \in 2^{\mathscr{A}_\Sigma}$ be a set of assignments then:*

$$\{\mathsf{UP}(C) | C \in \mathscr{E}_M\} \subseteq \mathscr{T}_{\mathsf{mod}_M} \qquad \mathsf{UP}(\mathsf{CofA}(M)) = \alpha \circ \mathsf{mod}_M \circ \gamma$$

Thus an encoding $C \in \mathscr{E}_M$ is a propagation complete encoding *(PCE) [11] when:*

$$\mathsf{UP}(C) = \alpha \circ \mathsf{mod}_M \circ \gamma$$

Propagation complete encodings (PCEs) are not unique and there may be many, incomparable PCEs. One goal of encoding design can be the creation of PCEs with other desirable properties, such as using a minimal number of clauses or auxiliary variables. As with clauses, assignments and partial assignments, the discussion above is more general than unit propagation alone. Using our abstract satisfaction framework we can model PCEs. In the next section we present an algorithm for automatically generating PCEs.

4 Generating Propagation Complete Encodings

The previous section defined the notion of propagation complete encodings (PCEs) within our framework. Next, we present an algorithm (Algorithm 1) that can be used to determine if an encoding is propagation complete, strengthen it if not, and generate a PCE that is equisatisfiable to a reference encoding. Algorithm 1 takes as input a set of variables Σ that will serve as the encoding vocabulary, an initial encoding E_0 and a reference encoding $\mathsf{E}_{\mathsf{Ref}}$ (over a vocabulary including Σ), such that $\mathsf{AofC}(\mathsf{E}_{\mathsf{Ref}}) = M$. Note that, if $\mathsf{E}_0 = \emptyset$, then the algorithm will build a PCE over Σ from scratch that is equisatisfiable to $\mathsf{E}_{\mathsf{Ref}}$. In practice $\mathsf{E}_0 = \emptyset$, and $\mathsf{E}_{\mathsf{Ref}}$ can be any encoding of the circuit.

Algorithm 1: Generating a propagation complete encoding of a CNF formula

```
   Input: $\langle \Sigma, \mathsf{E}_0, \mathsf{E}_{\mathsf{Ref}} \rangle$
1  $\mathsf{E} \leftarrow \mathsf{E}_0$
2  PQ.push ($\lambda v.?$)
3  while not PQ.empty () do
       // $\forall q_1, q_2 \in \mathsf{PQ} \,.\, \mathsf{UP(E)}(q_1) \neq \mathsf{UP(E)}(q_2)$ and $\perp^{\mathscr{P}} \notin \mathsf{PQ}$
4      pa $\leftarrow$ PQ.pop ()
5      foreach $v \in \{x | x \in \Sigma$ and $\mathsf{UP(E)(pa)}(v) = ?\}$ do
6          foreach $l \in \{v, \neg v\}$ do
7              $\mathsf{pa}' \leftarrow \mathsf{pa} \sqcap assign(l)$
8              if SATSolver ($\mathsf{E}_{\mathsf{Ref}}$, $\mathsf{pa}'$) = sat then
9                  PQ.push ($\mathsf{pa}'$)
10             else
11                 $\mathsf{E} \leftarrow \mathsf{E} \cup \{\neg \mathsf{MUS}(\mathsf{pa}')\}$
12                 PQ.compact ()
       // $\mathsf{UP(E)(pa)} = (\alpha \circ \mathsf{mod}_M \circ \gamma)(\mathsf{pa})$
13 return E
```

The algorithm traverses the fix-points of the best abstract transformer $\alpha \circ \mathsf{mod}_M \circ \gamma$, i.e. partial assignments where no new facts can be deduced. To achieve this, the algorithm uses a priority queue (PQ) of partial assignments sorted by partial assignment size. For each element of PQ, we examine the variables v that unit propagation cannot infer from E and pa (line 5). We then check if the reference encoding $\mathsf{E}_{\mathsf{Ref}}$, along with the current partial assignment pa logically entail either v or $\neg v$. This check is done via a call to a SAT oracle at line 8 (in our implementation this is a call to a CDCL SAT solver). If the query returns sat, the variable is not entailed and the extended partial assignment is added to the queue. Otherwise, l was a missed propagation and the encoding is strengthened by adding a clause that blocks the partial assignment pa.[3]

If two partial assignments q_1 and q_2 unit propagate the same literals ($\mathsf{UP(E)(q_1)} = \mathsf{UP(E)(q_2)}$) we only need to explore extensions of one of them. Therefore, the push operation on line 9 only adds pa' to PQ if $\forall \mathsf{q} \in \mathsf{PQ} \,.\, \mathsf{UP(E)(q)} \neq \mathsf{UP(E)(pa)}$. In other words we cache assignments that become equal when extended by unit propagation. Because we are potentially strengthening the encoding E with each iteration of the for-loop the amount of information unit propagation can infer from E increases. The PQ.compact call on line 12 iterates over the queue elements and removes queue elements that UP-extend to the same partial assignment. This ensures the invariant at the beginning of the while-loop. Furthermore, at the end of the while loop the current encoding E is strong enough to unit propagate all literals entailed from pa. The continuous strengthening of E also reduces the number of unassigned variables explored at line 5.

The algorithm is not always guaranteed to generate subset-minimal encodings. The order in which the partial assignments is considered may lead to the learning of redundant clauses. A clause c is redundant w.r.t. a PCE E_{PC} if for all literals $l \in c$ unit propagation can infer l from $\mathsf{E}_{\mathsf{PC}} \backslash c$ assuming the negation of the other literals $\neg(c \backslash \{l\})$.

[3] As an optimization we add the negation of the minimal unsatisfiable core of $\neg \mathsf{pa}'$: $\mathsf{MUS}(\mathsf{pa}')$.

Algorithm 2: Greedy algorithm for introducing auxiliary variables

```
1  $\mathsf{E} \leftarrow \mathsf{genPCE}\,(\mathsf{E}_0,\, \mathsf{E}_{\mathsf{ref}},\, \Sigma)$
2  while $\mathsf{Aux} \neq \emptyset$ do
3      $\mathsf{best} \leftarrow \mathsf{undef}$
4      foreach $\mathsf{aux} \in \mathsf{Aux}$ do
5          $\mathsf{E}' \leftarrow \mathsf{genPCE}\,(\mathsf{E}_0,\, \mathsf{E}_{\mathsf{ref}} \wedge \mathsf{Def}\,(\mathsf{aux}),\, \Sigma \cup \{\mathsf{aux}\})$
6          if $|\mathsf{E}'| < |\mathsf{E}|$ then
7              $\mathsf{E} \leftarrow \mathsf{E}'$
8              $\mathsf{best} \leftarrow \mathsf{aux}$
9      if $\mathsf{best} = \mathsf{undef}$ then return $\mathsf{E}$
10     $\Sigma \leftarrow \Sigma \cup \{\mathsf{best}\}$
11     $\mathsf{E}_{\mathsf{ref}} \leftarrow \mathsf{E}_{\mathsf{ref}} \wedge \mathsf{Def}\,(\mathsf{best})$
12     $\mathsf{Aux} \leftarrow \mathsf{Aux} \setminus \{\mathsf{best}\}$
13 return $\mathsf{E}$
```

For example, in the presence of a chain of implications, $v_1 \Rightarrow v_2 \Rightarrow \ldots \Rightarrow v_k$, the algorithm may learn the redundant clause $c = \{\neg v_1, v_k\}$. Note that c is redundant since $v_1 \wedge (\mathsf{E}_{\mathsf{PC}} \setminus c) \vdash_{up} v_k$ and $\neg v_k \wedge (\mathsf{E}_{\mathsf{PC}} \setminus c) \vdash_{up} \neg v_1$. For this reason, after running Algorithm 1 we use the minimisation procedure described in [11] to remove redundant clauses while maintaining propagation completeness.

Auxiliary variables. The algorithm we described so far only works on a fixed vocabulary Σ consisting of the input and output variables of the encoding. For certain operators, there no polynomially-sized CNF encodings if we restrict ourselves to the input/output variables only. For this reason, we extended our algorithm to further reduce the size of the encoding while maintaining propagation completeness by heuristically adding auxiliary variables. Given a set of auxiliary variables Aux, we extend the reference encoding $\mathsf{E}_{\mathsf{Ref}}$ by adding the definitional clauses $\mathsf{Def}(\mathsf{aux})$ for each auxiliary variable $\mathsf{aux} \in \mathsf{Aux}$: $\mathsf{Def}(\mathsf{aux}) \wedge \mathsf{E}_{\mathsf{Ref}}$. For example, to introduce an auxiliary variable $a \equiv x \wedge y$ for inputs x, y, we add the clauses corresponding to the formula $a \Leftrightarrow (x \wedge y)$ to $\mathsf{E}_{\mathsf{Ref}}$ and run Algorithm 1 on $\Sigma \cup \{a\}$.

We implemented a greedy algorithm that iteratively repeats this process as shown in Algorithm 2. We denote by genPCE the procedure of generating a propagation complete encoding from a reference encoding given in Algorithm 1. We denote by $|\mathsf{E}|$ the size of an encoding as the number of clauses. The algorithm takes as input a reference encoding $\mathsf{E}_{\mathsf{Ref}}$, a fixed alphabet Σ as well as a set of auxiliary variables Aux. It initially computes the PCE over the input/output variables Σ. For each auxiliary variable aux in the current set of auxiliary variables, it computes the PCE over the alphabet $\Sigma \cup \{\mathsf{aux}\}$ from reference encoding $\mathsf{E}_{\mathsf{Ref}} \wedge \mathsf{Def}(\mathsf{aux})$, where $\mathsf{Def}(\mathsf{aux})$ is the set of definitional clauses for aux. It then chooses the auxiliary variable best that minimises the encoding the most, and adds it to the reference encoding. The process is repeated on the remaining auxiliary variables $\mathsf{Aux} \setminus \{\mathsf{aux}\}$ until no minimisation is achieved. Note that this is a greedy algorithm, and does not guarantee finding a minimal size encoding w.r.t. the given auxiliary variables. For the set of potential auxiliary variables Aux, we generate Boolean combinations over the input/output variables up to a limited depth.

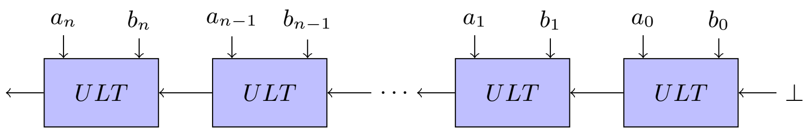


Fig. 4: Composition of encoding primitives to build a n-bit less than comparator.

As a heuristic, we also add to the set Aux the auxiliary variables used by the reference encoding.

Generating propagation complete encodings. Algorithm 1 solves an inherently hard problem and may call a SAT solver an exponential number of times. It is intended to be used as a tool to support encoding design rather than generating complete encodings.

To explore the feasibility of generating PCEs, we analysed the propagation completeness of encodings used in the CVC4 SMT solver [5]. CVC4 uses small circuit primitives to build more complex encodings of word-level bit-vector operators. Figure 4 shows an example of how small circuits for unsigned less than ($a < b$) primitives can be composed to build a more complex encoding to compare n-bit bit-vectors. Each unsigned less than comparator (ULT) has three input bits (a, b, r) and one output bit (o). There are different ways that this primitive can be encoded into CNF. A possible PCE is: $\{\{o, \neg b, a\}, \{o, \neg b, \neg r\}, \{a, \neg r, o\}, \{\neg o, b, \neg a\}, \{\neg o, r, \neg a\}, \{\neg o, r, b\}\}$. If r has value $\perp$, then o will be $\top$ iff $a < b$. Otherwise, if r has value $\top$, then o will be $\top$ iff $a \leq b$. This structure allows the ULTs to be chained together to form an n-bit PCE for the ULT comparator. A similar construction can be done for other encoding primitives and is common in circuit design. For example, full-adders can be chained to form a ripple-carry adder. Note that, if the encoding primitives are not PC, then their composition will not be PC. However, the converse does not necessarily hold.

Table 1 shows the size of the encodings generated by Algorithm 1 and by introducing auxiliary variables compared to the size of the reference encoding $\mathsf{E}_{\mathsf{Ref}}$, starting with an empty initial encoding E_0. As encoding primitives (*prim*), we have considered the if-then-else operator (ite-gadget), an unsigned less than comparator (ult-gadget), a signed less than comparator (slt-gadget), the full-adder (full-add), adder with base 4 (full-add-base4), bit-count circuits (bc3to2, bc7to3), 2x2 multiplication circuit (mult2), and multiplication by a constant (mult-const3, mult-const5, mult-const7). These encoding primitives are then composed (*comp*) to build n-bit bit-vector operators.

These experiments were run on Intel Xeon X5667 processors (3.00 GHz) running Fedora 20 with a timeout of 3 hours and a memory limit of 32 GB. In case of timeout of the greedy algorithm, we present the smallest encoding found until the timeout. The reference encodings used were the default implementations in CVC4. From the encoding primitives presented in Table 1, ite is the only encoding primitive that is propagation complete in CVC4. This scenario is not restricted to CVC4, and most state-of-the-art SMT solvers do not build PCEs (see section 5 for further details).

For small primitives our algorithms can easily find PCEs with small size even when restricting the set of variables to inputs and outputs. For more complex circuits, as mult-4bit, the PCE can be much larger than the non-PCE. When generating PCEs with

Table 1: Generation of PCEs for small encoding *prim*itives and their *comp*osition

		Original enc.		PCE			PCE w/ aux. vars		
Benchmark	Type	#Vars	#Cls	#Vars	#Cls	#time (s)	#Vars	#Cls	#time (s)
ite-gadget	prim	4	6	4	6	<0.01	4	6	<0.01
ult-gadget	prim	5	10	4	6	<0.01	4	6	<0.01
slt-gadget	prim	4	6	4	6	<0.01	4	6	<0.01
full-add	prim	8	17	5	14	<0.01	5	14	<0.01
full-add-base4	prim	33	74	10	120	0.31	12	86	140.40
bc3to2	prim	20	46	8	76	0.03	10	57	5.32
bc7to3	prim	27	68	10	254	0.49	14	136	769.50
mult2	prim	30	66	8	19	0.02	8	19	0.50
mult-const3	prim	16	33	6	20	<0.01	6	20	0.03
mult-const5	prim	25	50	9	24	0.01	9	24	0.21
mult-const7	prim	38	105	9	32	0.01	9	32	0.62
ult-6bit	comp	33	68	13	158	27.73	15	38	timeout
add-3bit	comp	18	39	9	96	0.09	11	29	10.05
add-4bit	comp	26	58	12	336	3.89	15	43	1,607.50
bc3to2-3bit	comp	38	78	12	1,536	11.72	16	69	timeout
mult-4bit	comp	36	97	12	670	5.26	12	670	298.95

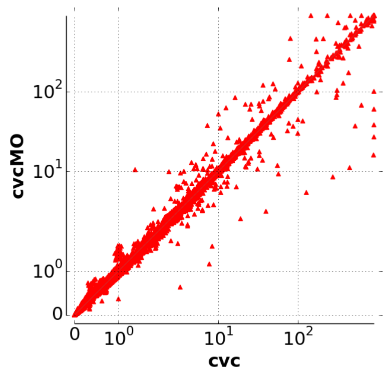
Σ containing auxiliary variables, we can obtain considerably smaller encodings. For example, for the addition operator `add-4bit` the number of clauses decreased from 336 to 43 by only adding 3 auxiliary variables. In this case, the auxiliary variables that are added by our greedy algorithm correspond to the carry bits from the chained adders. Note that the PCE for `add-4bit` formed by chaining the propagation complete full-adder results in an encoding with 20 variables and 60 clauses, which has a similar size to the PCE found by our greedy algorithm.

Even though the algorithm can take a considerable amount of time to find small PCEs with auxiliary variables, our goal is not to apply such algorithm to large formulae but only to find PCEs of primitives. This process is done once, offline. Afterwards, the encoding primitives can be chained together to form larger encodings for any bit-width. We verified with our algorithm that for small bit-widths the composition of PCEs for adders and comparators is propagation complete, while for the multiplier is not. We conjecture that the existence of a reasonably-sized propagation complete multiplier is unlikely, as this would help to efficiently solve hard factorization problems.

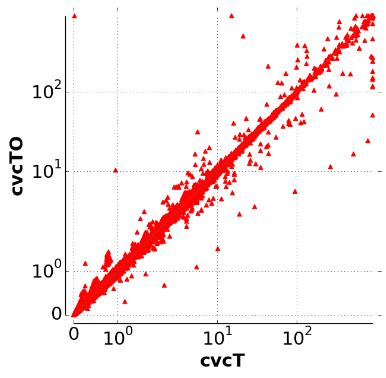
5 Experimental Evaluation

To explore the impact of propagation strength on performance, we implemented the PCE primitives generated in Section 4 in the CVC4 SMT solver [5]. CVC4 is a competitive solver that ranked 2nd in the 2015 SMT-COMP bit-vector division. We instrumented the solver's bit-blasting procedure to use the primitives to build more complex encodings of word-level bit-vector operators.

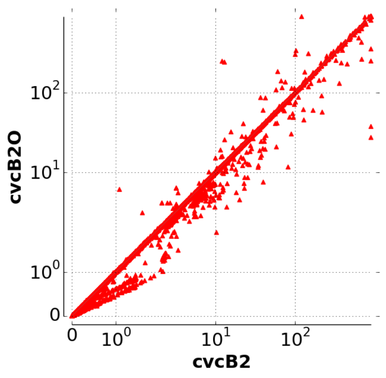
We focused on the following bit-vector operators: comparison, addition and multiplication. The rest of the bit-vector operations were either already propagation complete (e.g. bitwise and), or could be expressed in terms of other operations. We implemented n-bit circuits using the primitives described in Section 4. For addition, we used the prop-



(a) Shift-add multiplier



(b) Tree reduction multiplier



(c) Blocking multiplier

Fig. 5: The impact of using PC primitives in various kinds of multiplication circuits

agation complete full-adder (cvcAO) and for comparison the ult-gadget and slt-gadget (cvcLO). For multiplication we implemented variants that use PC primitives: shift-add multiplication (cvc vs cvcMO), tree reduction (cvcT vs cvcTO) and multiplication by blocking (cvcB2 vs cvcB2O). We append O to the solver's name to denote that the propagation complete sub-circuits are enabled. All implementations of multiplications that use propagation complete sub-circuits use the PC full-adder for adding the partial products, while blocking multiplication also uses the propagation complete 2 by 2 multiplication sub-circuit mult2.

We used 31066 quantifier-free bit-vector benchmarks from SMT-LIB v2.0 [6]. Experiments in this section were run on the StarExec [38] cluster infrastructure on Intel Xeon E5-2609 processors (2.40 GHz) running Red Hat Enterprise Linux Workstation release 6.3 (Santiago) with a timeout of 900 seconds and a memory limit of 200 GB.

Figure 5 quantifies the impact of the PC components in the various kinds of multiplication circuits we implemented. The scatter plots are on the entire 31066 set of

Table 2: Comparison of performance of propagation complete encodings in CVC4

	cvc		cvcMO		cvcAMO		cvcALMO	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
VS3 (11)	**1**	**145.3**	0	0.0	0	0.0	0	0.0
bmc-bv (135)	**135**	**641.4**	135	665.6	135	669.7	134	527.0
brummayerbiere2 (65)	57	2866.9	62	2852.9	62	2870.2	**62**	**2847.9**
brummayerbiere3 (79)	40	3570.4	39	3527.6	40	3369.3	**44**	**5546.3**
bruttomesso (64)	38	3143.2	38	3145.7	**41**	**4876.7**	41	4904.5
calypto (23)	**8**	**5.5**	8	6.5	8	7.6	8	6.3
fft (23)	**8**	**981.4**	8	1053.9	7	59.1	7	190.7
float (213)	159	14550.6	158	13233.2	158	12004.4	**159**	**12190.9**
log-slicing (208)	67	23828.9	66	23426.5	70	27087.5	**70**	**26871.2**
mcm (186)	78	8674.2	78	8646.3	**80**	**7387.8**	80	8182.2
rubik (7)	6	623.2	6	619.0	6	615.8	**7**	**1371.7**
spear (1695)	1690	28046.5	1690	28603.8	**1690**	**22731.4**	1690	23357.9
	2287	87077.4	2288	85780.9	2297	81679.5	**2302**	**85996.6**

benchmarks, and are on a log-scale. Each point is a benchmark, and the x and y-axis represent the time (seconds) taken by CVC4 to solve the benchmark with the given configuration. Using the propagation complete primitives (cvcMO, cvcTO and cvcB2O) consistently improves performance over their default implementations. Although the performance improvement is not dramatic, we believe it is consistent enough to show that propagation strength is an important characteristic of encodings. Since cvcMO had the best performance between multiplication circuits that use propagation complete subcircuits, we considered this encoding for further evaluation.

Table 2 gives the number of problems solved and the time taken to solve them for CVC4 without propagation complete primitives (cvc) and with propagation complete primitives, namely: shift-add multiplier (cvcMO); shift-add multiplier and full-adder (cvcAMO); and shift-add multiplier, full-adder and comparison (cvcALMO). Due to space constraints we removed rows where the number of problems solved by all configurations was the same (see Appendix for full table). Table 2 shows that adding each PC primitives increases performance, with the configuration using PC primitives for addition, comparison and multiplication (cvcALMO) solving the most.

We believe this improvement is not limited to CVC4 but will translate to other solvers as well. We examined the source code of other competitive SMT solvers, such as boolector [13], stp2 [28], yices2 [22] and z3 [15], and their implementation of addition is not propagation complete. Therefore, although the notion of propagation complete encodings is not new, it is not widely applied to solver encoding design. Preliminary results from implementing the PC full-adder in the CBMC model-checker [14] also showed an improved performance. The improvement is also not limited to constraint solvers that use CDCL SAT solvers but is also expected for look-ahead SAT solvers [29]. These solvers are geared towards propagation and are even more likely to take advantage of the increased inference power than CDCL SAT solvers.

We have shown that the propagation complete encoding primitives our algorithm generated can be used to build encodings of bit-vector operators in an SMT solver. The results are promising considering we are only strengthening a small part of the overall problem. Furthermore the propagation complete encodings have been automatically generated from scratch, while the existing encodings had been optimized by hand. This highlights the importance of propagation complete encoding in encoding design and that our proposed framework can help practitioners improve encodings.

6 Related Work

The notion of propagation strength has been explored under various names such as unit refutation complete [16] and propagation complete encodings (PCEs) [11] in AI knowledge compilation. A formula is unit refutation complete [16] iff any of its implicates can be refuted by unit propagation. Here we refer to refutation as being the process of proving the implication $\mathsf{E} \models l$ by proving $\mathsf{E} \wedge \neg l \models \bot$. Bordeaux et al. [10] consider variations of unit refutation complete encodings, such as its disjunctive closure and a superset of unit refutation complete encodings where variables can be existentially quantified and unit refutation concerns only implications from free variables. Gwynne and Kullmann [25] introduce a general hierarchy of CNF problems based on "propagation hardness" and generalise the notion of unit refutation complete encodings.

PCEs are a proper subset of refutation complete encodings [25] and have been introduced by Bordeaux and Marques-Silva [11] for finding encodings where only using unit propagation suffices to deduce all the literals that are logically valid. The authors reduce the problem of generating PCEs to iteratively solving QBF formulas. We consider PCEs using an abstract satisfaction framework and rely on a SAT solver's efficient UP routine to check whether a clause is empowering. Since QBF is a PSPACE-complete problem, it is unclear that the approach from [11] would scale better than ours. Because [11] has no implementation that we are aware of, we cannot compare against them. Their framework can also support adding auxiliary variables to PCEs but this approach was not explored by the authors. Our approach supports generating encodings over a limited alphabet of auxiliary variables and includes an implementation and extensive experimental results that show performance gains. The work in [2] shows that checking whether a clause is *empowering* (it is entailed by the given CNF formula and it increases the propagation power of the formula) is co-NP complete. It also shows the existence of operations that have only exponential PCEs. This supports our targeting of small encoding primitives as opposed to n-bit circuits which is likely intractable.

Propagation completeness has also been considered in CSP (e.g. [3,8]) because of its connection to Domain Consistency, also known as Generalised Arc Consistency (GAC): when a constraint is encoded into SAT over some Finite-Domain variables, if the encoding of the constraint is propagation complete, then unit propagation on the SAT encoding effectively finds the same implications as Domain Consistency. In CSP it is common to consider GAC over procedural propagators [3] of specific constraints. Propagators can also be decomposed into primitive constraints that can be translated to SAT [8]. GAC has been adopted in SAT [24] and many encodings have this property [4,37,1]. However, GAC is usually only enforced on input/output variables and not on auxiliary variables. PCEs consider a stronger notion of propagation strength since GAC is enforced on both input/output variables as well as on auxiliary variables.

Trevor Hansen's PhD [28] (independently) touches on many of the techniques we have used. He considers both 'bit-blasting' encodings and forward propagators (algorithms that implement abstract transformers directly), but treats these as independent approaches, omitting the link we show in Section 3. Although he tests the propagators for propagation completeness and even generates clauses to improve the propagators, he does not use this approach to generate complete encodings, nor does he perform minimisation. The SMT solver Beaver [30] also computes pre-synthesised templates for bit-vectors operators which are optimised offline using logic synthesis tools such as the ABC logic synthesis engine [7]. However, these templates are only computed for predefined bit-widths and are not PC. Hansen makes use of Reps' et al. [36] work on computing best abstract transformers via a lifted version of Stalmarck's algorithm. Algorithm 1 similarly uses breadth-first traversal, but the key difference is in how and when the algorithms are used. In [36] and most applications of their work [31], the *result* of the best abstract transformer is computed on-line as part of a search. We compute an *encoding* that gives the best abstract transformer off-line as part of solver development.

7 Conclusion

By using the abstract satisfaction framework we can characterise the space of encodings, the effects of reasoning and the link between them. Propagation complete encodings allow an increase of inference power that can be exploited by CDCL SAT solvers. We have showed that these encodings are captured by our abstract satisfaction formalism which allows us to reason about them and their extensions (Section 3). It is possible to compute subset-minimal propagation complete encodings and for various key operations these are tractably computable and often smaller than conventional encodings. For more complex encodings, we have shown that greedily introducing auxiliary variables can generate significantly smaller propagation complete encodings (Section 4). Implementing these in the CVC4 SMT solver gives performance improvements across a wide range of benchmarks (Section 5). It is hoped that this work will contribute to a more theoretically rigorous approach to encoding design.

Linking encodings to abstract transformers has many possible applications. Abstract transformers are functions on ordered sets and are therefore partially ordered. This gives a way of comparing the propagation strength of different encodings or investigating the effects of pre and in-processing techniques. This is particularly important as for certain operators there are no polynomially sized PCEs. A quantitative measure of propagation strength is a useful practical alternative. Proof-theoretic measures can be expressed as properties of the syntactic representation lattice, for example proof length becomes path length. Likewise solver run-time is bounded by the length of paths in $\mathsf{UP}(2^{\mathscr{C}_{\Sigma+}})$. Finally, the abstract satisfaction viewpoint provides a means of exploring many interesting questions about composition of encodings and when they preserve propagation strength.

Acknowledgments: This research was partially supported by ERC project 280053 (CPROVER) and by DARPA MUSE award #FA8750-14-2-0270. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

1. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality Networks: a theoretical and empirical study. Constraints 16(2), 195–221 (2011)
2. Babka, M., Balyo, T., Čepek, O., Gurskỳ, Š., Kučera, P., Vlček, V.: Complexity issues related to propagation completeness. Artificial Intelligence 203, 19–34 (2013)
3. Bacchus, F.: GAC Via Unit Propagation. In: International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 133–147. Springer (2007)
4. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. In: International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 2833, pp. 108–122. Springer (2003)
5. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: International Conference on Computer Aided Verification. LNCS, vol. 6806. Springer (2011)
6. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: Workshop on Satisfiability Modulo Theories (2010)
7. Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification, Release 70930, `http://www.eecs.berkeley.edu/~alanmi/abc`
8. Bessiere, C., Katsirelos, G., Narodytska, N., Walsh, T.: Circuit Complexity and Decompositions of Global Constraints. In: International Joint Conference on Artificial Intelligence. pp. 412–418. AAAI Press (2009)
9. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
10. Bordeaux, L., Janota, M., Marques-Silva, J., Marquis, P.: On Unit-Refutation Complete Formulae with Existentially Quantified Variables. In: Principles of Knowledge Representation and Reasoning. AAAI Press (2012)
11. Bordeaux, L., Marques-Silva, J.: Knowledge Compilation with Empowerment. In: International Conference on Current Trends in Theory and Practice of Computer Science. LNCS, vol. 7147, pp. 612–624. Springer (2012)
12. Brain, M., D'Silva, V., Haller, L., Griggio, A., Kroening, D.: An Abstract Interpretation of DPLL(T). In: International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 7737, pp. 455–475. Springer (2013)
13. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 5505, pp. 174–177. Springer (2009)
14. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2988, pp. 168–176. Springer (2004)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 4963, pp. 337–340. Springer (2008)
16. Del Val, A.: Tractable Databases: How to Make Propositional Unit Resolution Complete Through Compilation. In: Principles of Knowledge Representation and Reasoning. pp. 551–561. Morgan Kaufmann (1994)
17. D'Silva, V., Haller, L., Kroening, D.: Satisfiability Solvers Are Static Analysers. In: International Static Analysis Symposium. LNCS, vol. 7460, pp. 317–333. Springer (2012)
18. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: Symposium on Principles of Programming Languages. pp. 143–154. ACM (2013)
19. D'Silva, V., Haller, L., Kroening, D.: Abstract satisfaction. In: Symposium on Principles of Programming Languages. pp. 139–150. ACM (2014)

20. D'Silva, V., Haller, L., Kroening, D., Tautschnig, M.: Numeric Bounds Analysis with Conflict-Driven Learning. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7214, pp. 48–63. Springer (2012)
21. D'Silva, V., Kroening, D.: Abstraction of Syntax. In: International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 7737, pp. 396–413. Springer (2013)
22. Dutertre, B.: Yices 2.2. In: Computer Aided Verification. LNCS, vol. 8559, pp. 737–744. Springer (2014)
23. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2(1-4), 1–26 (2006)
24. Gent, I.P.: Arc consistency in sat. In: European Conference on Artificial Intelligence. pp. 121–125. IOS Press (2002)
25. Gwynne, M., Kullmann, O.: Generalising unit-refutation completeness and slur via nested input resolution. Journal of Automated Reasoning 52(1), 31–65 (2014)
26. Gwynne, M., Kullmann, O.: On SAT Representations of XOR Constraints. In: Language and Automata Theory and Applications. LNCS, vol. 8370, pp. 409–420. Springer (2014)
27. Haller, L., Griggio, A., Brain, M., Kroening, D.: Deciding floating-point logic with systematic abstraction. In: Formal Methods in Computer-Aided Design. pp. 131–140. IEEE (2012)
28. Hansen, T.: A Constraint Solver and its Application to Machine Code Test Generation. Ph.D. thesis, University of Melbourne (2012)
29. Heule, M., van Maaren, H.: Look-Ahead Based SAT Solvers. In: Handbook of Satisfiability, pp. 155–184. IOS Press (2009)
30. Jha, S., Limaye, R., Seshia, S.A.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: International Conference on Computer Aided Verification. LNCS, vol. 5643, pp. 668–674. Springer (2009)
31. King, A., Søndergaard, H.: Automatic Abstraction for Congruences. In: International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 5944, pp. 197–213. Springer (2010)
32. Kleine Büning, H., Kullmann, O.: Minimal Unsatisfiability and Autarkies. In: Handbook of Satisfiability, pp. 339–401. IOS Press (2009)
33. Kullmann, O.: Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. Annals of Mathematics and Artificial Intelligence 40(3-4), 303–352 (2004)
34. Manthey, N., Heule, M., Biere, A.: Automated Reencoding of Boolean Formulas. LNCS, vol. 7857, pp. 102–117. Springer (2013)
35. Martins, R., Manquinho, V., Lynce, I.: Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In: International Conference on Tools with Artificial Intelligence. pp. 313–320. IEEE Computer Society (2011)
36. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: International Conference on Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 2937, pp. 252–266. Springer (2004)
37. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 3709, pp. 827–831. Springer (2005)
38. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a Cross-Community Infrastructure for Logic Solving. In: International Joint Conference on Automated Reasoning. LNCS, vol. 8562, pp. 367–373. Springer (2014)