Bit-Precise Reasoning with Affine Functions

Neil Kettle and Andy King*

Portcullis Computer Security Limited, Pinner, HA5 2EX, UK

Abstract

The class of affine Boolean functions is rich enough to express constant bits and dependencies between different bits of different words. For example, the function $(x_0) \wedge (\neg y_1) \wedge (x_4 \iff y_7) \wedge (x_5 \iff \neg y_9)$ is affine and expresses the invariant that the low bit (bit 0) of the variable x is true, that bit 1 of y is false, that the bits 4 and 7 of x and y coincide whereas bits 5 and 9 of x and y differ. This class of Boolean function is amenable to bit-precise reasoning since it satisfies strong chain properties which bound the number of times a system of semantic fixpoint equations need to be reapplied when reasoning about loops. This paper address the key problem of abstracting an arbitrary Boolean function to either a general affine function or a so-called affine function of width 2, when the function is represented as an ROBDD. Novel algorithms are presented for this task: one that manipulates Boolean vectors and another which is inspired by anti-unification. The speed and precision of both algorithms are compared on benchmark circuits, to draw conclusions on the tractability of affine abstraction.

1 Introduction

In the context of bounded model checking, it is now commonplace to reason about the semantics of basic-blocks with equations over bit-vectors. At this level of abstraction, the wrap-around nature of finite integer arithmetic can be modeled, and even pointers can be supported [6, section 2.3]. An alternative to exploring loops to a fixed depth, is to apply abstract interpretation techniques and compute bit-level summaries that enable all paths through the program to be systematically considered. Bit-value inference is an exemplar of this thread of work [5, 18], in which forward and backward data-flow analyses are deployed to discover constant and redundant bits within program variables. These analyses refine constant propagation analysis and live variables analysis [14] to the bit-level, by describing the state of individual bits at a given program point using ternary logic: 0, 1 or $\frac{1}{2}$ (for unknown or irrelevant bits). The state of a variable at a program point is then represented as a bit-vector $\vec{b} = \langle b_0, \ldots, b_{31} \rangle$ where $b_i \in \{0, 1, \frac{1}{2}\}$, and by extension, the state of set of n variables at a point is described by a concatenated sequence $\vec{b}_1 \cdots \vec{b}_n$. Although useful in their own right, such bit-vectors can be combined with ranges [10] in a mutually beneficial way. For instance, if it is the known that a (whole) variable takes a value in $[0, 17]$ and has a bit-level description of $\langle 0, \frac{1}{2}, \ldots, \frac{1}{2} \rangle$, then the bit-vector can be refined to $\langle 0, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 0, \ldots, 0 \rangle$. Moreover, the interval can be pruned to $[0, 16]$ since the vector specifies that the variable is a multiple of 2. Such a combined analysis can be used to reason about the bit-width of operands (by discovering constant and unused bits) and thereby exploit sub-word parallelism [15].

*Andy King is on secondment from the University of Kent, Canterbury, CT2 7NF, UK

1.1 Two classes of affine Boolean function

Part of the attraction of reasoning about bits using ternary bit-vectors is that they induce a strong bound on the number of times that a bit-vector can be weakened. For instance, the 8-bit vector $\langle 0,1,0,0,1,0,1,1\rangle$ can be weakened in the sequence (ascending chain) $\langle 0,1,0,\frac{1}{2},1,0,1,1\rangle$, $\langle 0,\frac{1}{2},0,\frac{1}{2},1,0,1,1\rangle$,..., $\langle \frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2},\frac{1}{2}\rangle$. The length of the longest ascending chain for an 8-bit ternary vector is 8 and, in general, is n for an n-bit vector. The significance of this is that it enforces a strong bound on the number of times that the forward and backward fixpoint equations can be reapplied when performing the analysis: the number of iterations of the analysis is linear in the number of program variables (assuming words have fixed size). One natural question is how this abstract domain for bit-level analysis can be enriched, without compromising this complexity bound. This paper proposes two sub-classes of Boolean function for this task: the domain of affine Boolean functions, and the sub-domain of affine functions of width two [8, section 4.3]. The latter domain consists of (implicitly conjoined) systems of equations of the syntactic form: x_i, $\neg x_i$, $x_i \iff y_j$, $x_i \oplus y_j$ where $\oplus$ denotes exclusive-or and x_i and y_j the values of bit i and j of the variables x and y respectively. Thus, if the ternary bit-vector $\langle 0,\frac{1}{2},\frac{1}{2},\frac{1}{2},1,0,\frac{1}{2},1\rangle$ describes the state of an 8-bit variable x, then the state could also be represented as the formula $\neg x_0 \wedge x_4 \wedge \neg x_5 \wedge x_7$. Moreover, the two variable formulae $x_i \iff y_j$ and $x_i \oplus y_j$ can additionally express equivalences and disequalities between pairs of bits, possibly in different variables. This class of Boolean formulae is reminiscent of the class of weakly relational domains, such as the Octagon domain [13], which expresses systems of two variable numeric constraints of the form $\pm x \pm y \leq c$ where $c \in \mathbb{Q}$. As with Octagons, we shall see that this sub-domain of affine formulae is computationally attractive for the purposes of analysis.

The most general class of affine formulae are those that can be represented as a conjunction of equations of the form $x_i \oplus y_j \oplus \ldots \oplus z_k \iff c$ where $c \in \{0,1\}$. Since $(x_i \iff y_j)$ holds iff $x_i \oplus y_j \iff 0$ holds, it follows that the class includes all affine formulae of width 2, as well as including additional formulae such as $x_3 \oplus (y_5 \iff z_7)$. The number of satisfying assignments to any affine formula is always a power of 2 [19] and since the minimal and maximal number of assignments for Boolean functions over n propositional variables is 0 and 2^n respectively, it follows that the length of the longest chain of affine formulae is merely n. Thus relaxing ternary bit-vectors to affine formulae, whether width 2 or not, does not increase the number of iterations required to solve fixpoint equations.

1.2 Abstracting ROBDDs to affine functions

Appendix C of [4] provides transfer functions for ternary bit-vectors for both forwards and backwards analysis, for operations such as shifting, bit operations, arithmetic operations, etc. Bit-blasting [6] provides an alternative to specifying a transfer function for each concrete operation. Instead, the semantics of an operation, or even a sequence of operations, can be expressed relationally as a Boolean function, say f, that encodes the relationship between a vector of input bits, say $\vec{x}$, and a vector of output bits, say $\vec{x}'$. If an input state on $\vec{x}$ is described by a function f_i, then the output state on $\vec{x}'$ can be derived by computing $f_i \wedge f$, and then projecting $f_i \wedge f$ onto $\vec{x}'$. Since ROBDDs support projection [3], they provide a natural way to implement this tactic and thereby construct a bit-precise forward analysis. However, the class of arbitrary Boolean functions contains exponentially long ascending chains (consider a sequence of functions where each function is derived from its predecessor by adding a single

assignment). Therefore the (recursive) fixpoint equations that characterise a loop may need to be re-evaluated an exponentially large number of times. This computational problem can be finessed by relaxing an ROBDD that cuts a loop (see [2] for a discussion of loop cutting) to an ROBDD that represents an affine formula. More precisely, the problem is to compute the best affine abstraction of a function f represented as an ROBDD that is itself defined over a set of variables X. This amounts to computing an ROBDD that represents the function $g = \wedge\{a \in \mathrm{Aff}_X \mid f \models a\}$ where Aff_X denotes the set of all affine Boolean functions that can be defined over X. The function g is affine since the conjunction of two affine formulae is itself affine. The problem of computing the best affine abstraction of width 2 is formulated by merely replacing the class Aff_X with $\mathrm{Aff2}_X$, where the latter denotes the set of affine formulae of width 2 that can be defined over X. This paper focusses on the problem of computing these abstractions for functions represented as ROBDDs, as this is a necessary step for preserving a strong bound on the number iterations of a bit-precise analysis.

2 Preliminaries

2.1 Boolean Functions

A Boolean function is a mapping $f : \mathit{Bool}^n \to \mathit{Bool}$ where $\mathit{Bool} = \{0, 1\}$ that is conventionally written as a propositional formula defined over a totally ordered set of variables $X = \{x_1, \ldots, x_n\}$. For instance, $x_1 \vee x_2$ represents the dyadic function $\{\langle 0,0\rangle \mapsto 0, \langle 0,1\rangle \mapsto 1, \langle 1,0\rangle \mapsto 1, \langle 1,1\rangle \mapsto 1\}$. The set of propositional formulae over X is denoted Bool_X and henceforth functions and formulae will be used interchangeably. We define the set of satisfying assignments of a function f as the mapping $\mathit{model}_X(f) : \mathit{Bool}_X \to \wp(\mathit{Bool}^n)$ such that $\mathit{model}_X(f) = \{\langle b_1, \ldots, b_n\rangle \mid f(b_1, \ldots, b_n) = 1\}$ where $\wp$ denotes the power-set operator. For example, if $X = \{x_1, x_2, x_3\}$ then $\mathit{model}_X(x_1 \wedge (x_2 \to x_3)) = \{\langle 1,0,0\rangle, \langle 1,0,1\rangle, \langle 1,1,1\rangle\}$. One Boolean function f_1 entails another f_2, denoted $f_1 \models f_2$ iff $\mathit{model}_X(f_1) \subseteq \mathit{model}_X(f_2)$. The structure $\langle \mathit{Bool}_X, \models, \vee, \wedge, 0, 1\rangle$ is a finite lattice where 0 and 1 abbreviate the Boolean functions $\lambda\vec{b}.0$ and $\lambda\vec{b}.1$ respectively and $\vec{b} \in \mathit{Bool}^n$. A chain of Boolean functions C is a set $C \subseteq \mathit{Bool}_X$ such that either $f \models f'$ or $f' \models f$ for all $f, f' \in C$. The Shannon cofactor of a Boolean function f w.r.t. a variable x_i and a Boolean constant b is defined by $f|_{x_i \leftarrow b} = f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$. Finally, we denote existential quantification w.r.t. a variable x_i by $\exists_{x_i}(f)$ which can be computed using Schröder elimination, that is, by $\exists_{x_i}(f) = f|_{x_i \leftarrow 0} \vee f|_{x_i \leftarrow 1}$.

2.2 Binary Decision Diagrams

A Binary Decision Diagram (BDD) [3] is a rooted directed acyclic graph where each internal node is labelled with a variable x_i. Each internal node has one successor node connected via an edge labelled 0, and another successor connected via an edge labelled 1. An external (leaf) node is represented by one of two nodes labelled with the Boolean constants 0 or 1. The Boolean function represented by a BDD can be evaluated for a given variable assignment by traversing the graph from the root, taking the 1 edge at a node when the variable is assigned to 1 and the 0 edge when the variable is assigned to 0. The external node reached in this traversal indicates the value of the Boolean function for the assignment. Observe that each sub-BDD of a BDD also itself represents a Boolean function.

```
1:  procedure affine2(input: an ROBDD f over x_1, ..., x_n)
2:  begin
3:      i := index(f)
4:      if f|_{x_i←1} = 1 ∧ f|_{x_i←0} = 0 then return ⟨i, 1, 0⟩ :: ϵ
5:      else if f|_{x_i←1} = 0 ∧ f|_{x_i←0} = 1 then return ⟨i, 0, 1⟩ :: ϵ
6:      else if f|_{x_i←0} = 1 then return ϵ
7:      else if f|_{x_i←1} = 1 then return ϵ
8:      else if f|_{x_i←0} = 0 then return ⟨i, 1, 0⟩ :: affine2(f|_{x_i←1})
9:      else if f|_{x_i←1} = 0 then return ⟨i, 0, 1⟩ :: affine2(f|_{x_i←0})
10:     else
11:     begin
12:         l_1 := ⟨i, 0, 1⟩ :: affine2(f|_{x_i←0})
13:         l_2 := ⟨i, 1, 0⟩ :: affine2(f|_{x_i←1})
14:         return antiunify(l_1, l_2)
15:     end
16: end
```

Figure 1: Affine abstraction of width 2 of an ROBDD

An ROBDD is a BDD that is restricted, as follows, to induce a canonical representation. Firstly, the label of a node x_i is less than the label x_j of any internal node reachable via its successors, that is, $i < j$. Secondly, there exists no sub-ROBDD that is rooted at a node labeled with x_i that represents a function f such that $f|_{x_i \leftarrow 0} = f|_{x_i \leftarrow 1}$. Thirdly, there are no two nodes have that identical successor nodes and are labeled with the same variable.

3 Abstracting an ROBDD with an affine function of width 2

$\mathrm{Aff2}_X$ has been previously studied [8] because it represents the bijective sub-class of Aff_X. A function f is bijective iff whenever f possesses the satisfying assignments $\vec{x}, \vec{y}, \vec{z}$ then it also satisfied by $(\vec{x} \wedge \vec{y}) \vee (\vec{x} \wedge \vec{z}) \vee (\vec{y} \wedge \vec{z})$. More pragmatically, these functions represent the affine sub-class that can expressed as conjunctions of two variable constraints [8, lemma 4.9]. This is pertinent to analysis because such functions can be represented a list of tuples. Moreover, if the lists l_1 and l_2 represent two affine formulae f_1 and f_2 then (rather surprisingly) the affine abstraction of width 2 of $f_1 \vee f_2$ can be found by applying anti-unification to l_1 and l_2.

3.1 Primer on anti-unification

Anti-unification is better known in machine learning [16] than program analysis, and thus to make the paper self-contained, we recall that anti-unification of two terms t_1 and t_2, denoted $\mathrm{antiunify}(t_1, t_2)$, computes a term t that is the most specific generalisation of t_1 and t_2. For instance, $\mathrm{antiunify}(f(a,b,b), f(a,c,c)) = f(a,X,X)$ (rather than $f(a,X,Y)$). The term $f(a,X,X)$ (and renamings such as $f(a,Y,Y)$ and $f(a,Z,Z)$) are the most specific terms that can be instantiated to obtain $f(a,b,b)$ and $f(a,c,c)$. More formally, if $t = \mathrm{antiunify}(t_1, t_2)$, then there exists substitutions θ_1 and θ_2 such that $\theta_1(t) = t_1$ and $\theta_2(t) = t_2$ (t generalises t_1 and t_1). Moreover, if t' is a term such that $\theta'_1(t') = t_1$ and $\theta'_2(t') = t_2$ then $\vartheta(t) = t'$ for some substitution ϑ (t is the most specific generalisation of t_1 and t_2).

3.2 Anti-unification and affine abstraction for width 2

The relevance of anti-unification becomes more clear when the relationship between affine functions and lists of triples is spelt out. To illustrate, consider the function $(x_1 \iff x_2) \wedge \neg x_3$ over the (ordered) variables x_1, x_2, x_3. Observe that this affine function of width 2 can be represented as a list term, namely $l = \langle 1, A, B\rangle :: \langle 2, A, B\rangle :: \langle 3, 0, 1\rangle$, where A and B are free variables. The first position in a triple denotes the variable index. The second and third positions indicate the polarity of a variable in any satisfying assignment, and express two variable affine equations using the free variables. For instance, $\langle 3, 0, 1\rangle$ indicates that $\neg x_3$ holds: the positive position is 0 and the negative position is 1, indicating that x_3 is 0 in all satisfying assignments in the function represented by l. Taken together, the triples $\langle 1, A, B\rangle$ and $\langle 2, A, B\rangle$ indicate the truth values of x_1 and x_2 coincide in all satisfying assignments, hence $(x_1 \iff x_2)$ holds. Furthermore, the lists $\langle 1, A, B\rangle :: \langle 2, 1, 0\rangle :: \langle 3, B, A\rangle$ and $\langle 1, A, B\rangle :: \langle 2, A, B\rangle :: \langle 4, B, A\rangle$ encode $(x_1 \oplus x_3) \wedge x_2$ and $(x_1 \iff x_2) \wedge (x_2 \oplus x_4)$ respectively. The interpretation of the empty list ϵ is the vacuous affine function *true*. More generally, the following function specifies how a list l can be interpreted as an affine formula:

$$\text{decode}(l) = \begin{cases} \mathit{true} & \text{if } l = \epsilon \\ x_i \wedge \text{decode}(l)' & \text{if } l = \langle i, 1, 0\rangle :: l' \\ \neg x_i \wedge \text{decode}(l') & \text{if } l = \langle i, 0, 1\rangle :: l' \\ \text{decode}(l[A \mapsto 0, B \mapsto 1]) \vee \text{decode}(l[A \mapsto 1, B \mapsto 0]) & \text{if } l = \langle i, A, B\rangle :: l' \end{cases}$$

Now consider the two affine formulae $f_1 = x_2 \wedge x_4 \wedge \neg x_5$ and $f_2 = \neg x_2 \wedge x_3 \wedge \neg x_4 \wedge x_5$ that are represented by $l_1 = \langle 2, 1, 0\rangle :: \langle 4, 1, 0\rangle :: \langle 5, 0, 1\rangle$ and $l_2 = \langle 2, 0, 1\rangle :: \langle 3, 1, 0\rangle :: \langle 4, 0, 1\rangle :: \langle 5, 1, 0\rangle$ respectively, that is, $\text{decode}(l_1) = f_1$ and $\text{decode}(l_2) = f_2$. The attraction of the encoding is that $l = \text{antiunify}(l_1, l_2)$ yields an affine abstraction of $f_1 \vee f_2$. Specifically $l = \langle 2, A, B\rangle :: \langle 4, A, B\rangle :: \langle 5, B, A\rangle$ and $\text{decode}(l) = (x_2 \iff x_4) \wedge (x_4 \oplus x_5)$.

3.3 An divide-and-conquer abstraction algorithm based on anti-unification

This trick suggests a divide-and-conquer approach for computing the affine abstraction of width 2 for an ROBDD. To abstract a sub-ROBDD f that is labeled by x_i, compute the abstraction of the sub-ROBDDs for the co-factors $f|_{x_i \leftarrow 0}$ and $f|_{x_i \leftarrow 1}$, where the abstractions as represented themselves as lists l_1 and l_2. Then extend l_1 and l_2 to represent the affine formulae $(\neg x_i) \wedge f|_{x_i \leftarrow 0}$ and $(x_i) \wedge f|_{x_i \leftarrow 1}$ to give l_1' and l_2'. Next, apply anti-unification to l_1' and l_2' to obtain l which can then be reinterpreted as the affine abstraction of f.

The algorithm is represented in Figure 1 applies this technique to compute a list of triples that represents the affine abstraction of width 2 of an ROBDD. The operation $\mathsf{index}(f)$ returns the index of the variable that labels the root of the ROBDD f. Figure 2 illustrates a run of the algorithm for an ROBDD that represents $f = (x_1 \wedge (x_2 \vee x_3)) \wedge (x_2 \iff x_4) \wedge (x_2 \iff \neg x_5)$. The result of the algorithm is the list $\langle 1, 1, 0\rangle :: \langle 2, A, B\rangle :: \langle 4, A, B\rangle :: \langle 5, B, A\rangle$ that encodes $g = x_1 \wedge (x_2 \iff x_4) \wedge (x_4 \oplus x_5)$. Observe $f \models g$ but g is additionally satisfied by $\langle 1, 0, 0, 0, 1\rangle$.

To reason about complexity, observe that the length of any list will not exceed n where n is the number of variables in the ROBDD. Then, if antiunify is implemented using an AVL tree [16], each call to antiunify resides in $O(n \log n)$. ROBDD memoisation [3] ensures that the number of times anti-unification is invoked does not exceed the number of nodes in the ROBDD, denoted $|f|$. Hence the overall complexity of the abstraction algorithm is $O(|f| n \log n)$. The polynomial complexity stems from the fact that the intermediate affine abstractions, namely those for $f|_{x_i \leftarrow 0}$, $f|_{x_i \leftarrow 1}$, $(\neg x_i) \wedge f|_{x_i \leftarrow 0}$ and $(x_i) \wedge f|_{x_i \leftarrow 1}$, are not themselves

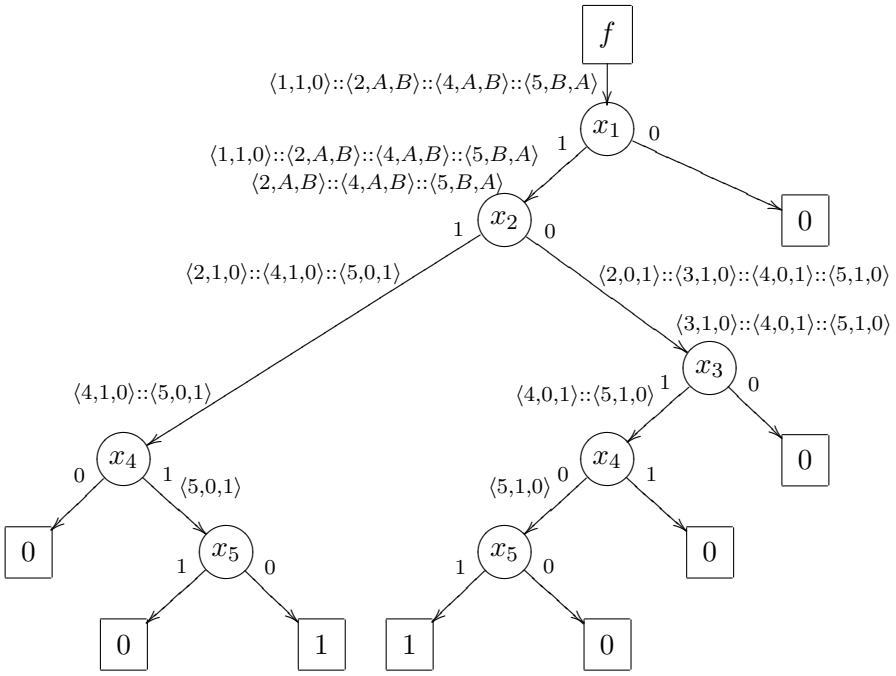

Figure 2: Computing an abstraction for $f = (x_1 \wedge (x_2 \vee x_3)) \wedge (x_2 \iff x_4) \wedge (x_4 \iff \neg x_5)$

represented as ROBDDs. Such an approach would compromise the polynomial complexity because even a single logical operation is quadratic in the size of its input ROBDDs. (Note that for brevity the abstraction algorithm ignores the post-processing step of converting the list back into an ROBDD. However, this step is straightforward and, in fact, resides in $O(n)$.)

4 Abstracting an ROBDD with a (general) affine function

As long as ago as Schaefer [19], it was observed that affine Boolean functions could be put into a triangular form by applying Gaussian elimination. Moreover, the (general) affine abstraction of a Boolean function can be derived by triangularising its set of satisfying assignments. This tactic is then lifted to ROBDDs to compute an (unrestricted) affine abstraction.

4.1 Computing the generators of a set of vectors

Algorithm 3 fleshes out the idea of Schaefer [19] and generates a set of Boolean vectors S' in a triangular form from S. S' is triangular in the sense that if $\vec{x}_1, \vec{x}_2 \in S'$ then $\mathsf{leading}(\vec{x}_1) \neq \mathsf{leading}(\vec{x}_2)$, where $\mathsf{leading}(\vec{x})$ returns -1 if $\vec{x} = \vec{0}$ and the position of the first non-zero element of $\vec{x}$ otherwise. When S is the set of satisfying assignments of a function f of n variables, then S' is set of assignments whose affine combination spans the affine abstraction of f. Put another way, if $S' = \{\vec{x}_1, \ldots, \vec{x}_k\}$ and $y_1, \ldots y_k \in \mathit{Bool}$ such that $(y_1 \oplus \ldots \oplus y_k) \iff 1$ then $(y_1 \wedge \vec{x}_1) \oplus \ldots \oplus (y_k \wedge \vec{x}_k)$ is a satisfying assignment of the affine abstraction of f where $y \wedge \langle x_1, \ldots, x_m \rangle = \langle y \wedge x_1, \ldots, y \wedge x_m \rangle$. Since the number of vectors in S' cannot exceed n, this gives a compact representation of an affine abstraction as a so-called set of generators. The algorithm resides in $O(n^3)$ just like the classical form of Gaussian elimination.

```
1:  procedure triangular(input: S ⊆ Bool^n)
2:  begin
3:      t := ∅
4:      for all (x⃗ ∈ S)
5:      begin
6:          i := leading(x⃗)
7:          while (i ≥ 0 ∧ i ∈ dom(t))
8:          begin
9:              x⃗ := x⃗ ⊕ t(i)
10:             i := leading(x⃗)
11:         end
12:         t := t[i ↦ x⃗]
13:     end
14:     return {t(i) | i ∈ dom(t)}
15: end
```

Figure 3: Triangularisation of a set of Boolean vectors

```
1:  procedure triangular(input: an ROBDD f over the variables x_1, ..., x_n)
2:  begin
3:      return triangular(f, 1, n)
4:  end
5:
6:  procedure triangular(input: an ROBDD f, integers i, k)
7:  begin
8:      if f = true then
9:      begin
10:         S := {0⃗_{k-i+1}}
11:         for all (i ≤ ℓ ≤ k) S := S ∪ {0⃗_{ℓ-i} · 1 · 0⃗_{k-ℓ}}
12:         return S
13:     end
14:     else if f = false then return ∅
15:     else
16:     begin
17:         j := index(f)
18:         f_0 := f|_{x_j ← 0}
19:         f_1 := f|_{x_j ← 1}
20:         S := {0⃗_{j-i}}
21:         for all (i ≤ ℓ ≤ j - 1) S := S ∪ {0⃗_{ℓ-i} · 1 · 0⃗_{j-1-ℓ}}
22:         S_0 := {x⃗ · 0 · y⃗ | x⃗ ∈ S ∧ y⃗ ∈ triangular(f_0, j + 1, k)}
23:         S_1 := {x⃗ · 1 · y⃗ | x⃗ ∈ S ∧ y⃗ ∈ triangular(f_1, j + 1, k)}
24:         return triangular(S_0 ∪ S_1)
25:     end
26: end
```

Figure 4: Triangularisation of a set of vectors represented by an ROBDD

4.2 Computing the generators of an ROBDD

Algorithm 4 develops this idea to compute a set of generators that describe the affine abstraction of an ROBDD. In the presentation of the algorithm, $\vec{0}_n$ denotes the zero vector of length n so that, for instance, $\vec{0}_2 = \langle 0, 0\rangle$ and $\vec{0}_0 = \epsilon$ where ϵ indicates the empty vector. The intuition behind the algorithm is that a sub-ROBDD f which is labeled by x_i, can be abstracted by computing generator sets S_1 and S_2 for its co-factors $f|_{x_i \leftarrow 0}$ and $f|_{x_i \leftarrow 1}$. Then sets $S'_0 = \{0 \cdot \vec{x} \mid \vec{x} \in S_0\}$ and $S'_1 = \{1 \cdot \vec{x} \mid \vec{x} \in S_1\}$ are computed where $\cdot$ denotes vector concatenation. Next $S'_0 \cup S'_1$ is triangularised to ensure that the number of generators does not exceed the number of variables in the ROBDD. The complexity in the algorithm stems from the corner cases that relate, among other things, to the fact that the labels that arise down a branch are not necessarily consecutive (though they are always increasing). With memoisation, the overall complexity of the algorithm is $O(|f|n^3)$.

4.3 Reinterpreting the generators of an ROBDD

Although a set of generators can indeed represent the affine abstraction of an ROBDD, it is not immediately obvious how such a set can be applied to construct the ROBDD which is the affine abstraction. In fact, rather conveniently, the triangularisation algorithm of figure 3 can be reapplied in a post-processing step that translates the generators into an ROBDD. This stems from the way Gaussian elimination can be used to invert a matrix. To illustrate the post-processing algorithm of figure 5, let f be an ROBDD that represents the function $(x_1 \wedge x_4 \wedge \neg x_5 \wedge (x_2 \iff x_3)) \vee (\neg x_1 \wedge \neg x_4 \wedge ((x_2 \iff x_3) \iff \neg x_5)))$. Then the satisfying assignments of f correspond to the following set of vectors: $\langle 0, 0, 0, 0, 0\rangle$, $\langle 0, 0, 1, 0, 1\rangle$, $\langle 0, 1, 0, 0, 1\rangle$, $\langle 0, 1, 1, 0, 0\rangle$, $\langle 1, 0, 0, 1, 0\rangle$, $\langle 1, 1, 1, 1, 0\rangle$. At line 3 of the post-processing algorithm, the call triangular(f) will return the set $\{\vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4\}$ where $\vec{x}_1 = \langle 1, 0, 0, 1, 0\rangle$, $\vec{x}_2 = \langle 0, 1, 1, 0, 0\rangle$, $\vec{x}_3 = \langle 0, 0, 1, 0, 1\rangle$ and $\vec{x}_4 = \langle 0, 0, 0, 0, 0\rangle$. These are the generators of f. The loop that starts at line 5 will then construct the set of vectors $S = \{\vec{y}_1, \ldots \vec{y}_5\}$ where

$$
\begin{array}{lcllcl}
\vec{y}_1 & = & \langle 1,0,0,0,1,0,0,0,0\rangle & \vec{z}_1 & = & \langle 1,0,0,0,1,0,0,0,0\rangle \\
\vec{y}_2 & = & \langle 0,1,0,0,0,1,0,0,0\rangle & \vec{z}_2 & = & \langle 0,1,0,0,0,1,0,0,0\rangle \\
\vec{y}_3 & = & \langle 0,1,1,0,0,0,1,0,0\rangle & \vec{z}_3 & = & \langle 0,0,1,0,0,1,1,0,0\rangle \\
\vec{y}_4 & = & \langle 1,0,0,0,0,0,0,1,0\rangle & \vec{z}_4 & = & \langle 0,0,0,0,1,0,0,1,0\rangle \\
\vec{y}_5 & = & \langle 0,0,1,0,0,0,0,0,1\rangle & \vec{z}_5 & = & \langle 0,0,0,0,0,1,1,0,1\rangle
\end{array}
$$

These vectors correspond to a set of simultaneous equations that, when solved, yield the affine equations that define the affine abstraction of f. The call to triangular(S) to line 12 produces the set of vectors $\{\vec{z}_1, \ldots \vec{z}_5\}$, stated above, that gives a solved form to these equations. The first three of these vectors will be disregarded at line 14 since $\mathsf{leading}(\vec{z}_1) = 1$, $\mathsf{leading}(\vec{z}_2) = 2$ and $\mathsf{leading}(\vec{z}_3) = 3$. However, the remaining two vectors of the solved form can be directly interpreted as affine equations. Indeed, the loop commencing at line 15 will generate two affine equations $h_1 = (x_1 \oplus x_4) \iff 0$ and $h_1 = (x_2 \oplus x_3 \oplus x_5) \iff 0$, from which the ROBDD $g = h_1 \wedge h_2$ is constructed. The satisfying assignments of g correspond to the following set of vectors: $\langle 0, 0, 0, 0, 0\rangle$, $\langle 0, 0, 1, 0, 1\rangle$, $\langle 0, 1, 0, 0, 1\rangle$, $\langle 0, 1, 1, 0, 0\rangle$, $\langle 1, 0, 0, 1, 0\rangle$, $\langle 1, 0, 1, 1, 1\rangle$, $\langle 1, 1, 0, 1, 1\rangle$ and $\langle 1, 1, 1, 1, 0\rangle$. Thus g possesses two satisfying assignments, namely $\langle 1, 0, 1, 1, 1\rangle$ and $\langle 1, 1, 0, 1, 1\rangle$, that the original ROBDD does not.

1: procedure affine(input: an ROBDD f over $x_1, \ldots, x_n$)
2: begin
3: $\{\vec{x}_1, \ldots, \vec{x}_i\} :=$ triangular(f)
4: $S := \emptyset$
5: for all $(1 \leq k \leq n)$
6: begin
7: $\vec{y} := \epsilon$
8: for all $(1 \leq \ell \leq n)$ $\vec{y} := \vec{y} \cdot \pi_k(\vec{x}_\ell)$
9: $S := S \cup \{\vec{y} \cdot \vec{0}_{k-1} \cdot 1 \cdot \vec{0}_{n-k}\}$
10: end
11: $g := 1$
12: for all $(\vec{z} \in$ triangular(S))
13: begin
14: if leading($\vec{z}$) $> i$ then
15: begin
16: $h := 0$
17: for all $(i < \ell \leq i + n)$ if $\pi_\ell(\vec{z}) = true$ then $h := h \oplus x_{\ell - i}$
18: end
19: $g := g \wedge (h \iff 0)$
20: end
21: return g
22: end

Figure 5: Affine abstraction of an ROBDD

5 Experimental Results

The abstraction algorithms have differing complexities, reflecting the fact that algorithm 5 will generate an ROBDD possessing no more (and possibly fewer) satisfying assignments than that produced by algorithm 1. Since these operations represent just one component of an analysis (albeit a non-trivial one) it is important that these operations scale to ROBDDs with large numbers of nodes. This is because expressing the semantics of an operation in a relational fashion will inevitably require large numbers of propositional variables: typically 32 for each integer variable.

Table 1 presents the results of some timing and precision experiments for some standard benchmark circuits (although these circuits do not arise in analysis, the functions are large, structured, and permit the experiments to be reproduced). The first four columns of the table give, respectively, the circuit name, number of input variables, number of defined functions (outputs) and the number of internal ROBDD nodes across all outputs (disregarding sharing between outputs). The remaining four columns are split into two pairs, the first of these pairs of columns records the total number of individual formulae as discovered by algorithm 1 and the time (in seconds) required. The second pair of columns replicates the results for algorithm 5. The experiments were performed on an Intel P3 1.73GHZ PC, equipped with 2GB of RAM, running Linux using the CUDD package [22]. However, even with 2GB, algorithm 5 failed to complete for s9234.1 and C3540 benchmarks, due to lack of memory.

Circuit	# var	# func	$\Sigma\|G\|$	$\Sigma\|E\|$	Time (s)	$\Sigma\|A\|$	Time (s)
alu4	14	8	1099	0	0.01	93	0.02
apex1	45	45	3000	115	0.01	636	0.37
apex2	39	3	1771	3	0.01	104	0.24
apex3	54	50	1661	217	0.01	399	0.68
dalu	75	16	5128	0	0.01	211	0.32
des	256	245	15209	61	0.01	2530	0.58
frg1	28	3	200	1	0.01	32	0.02
frg2	143	139	6679	218	0.01	1596	0.37
k2	256	245	3029	115	0.01	695	0.42
mm4a	19	16	529	28	0.01	111	0.01
mm9a	39	36	40878	45	0.01	486	4.38
mm9b	38	35	520690	43	0.04	701	36.69
pair	173	137	118066	90	0.20	2504	24.60
rot	135	107	13565	97	0.01	1160	3.57
too_large	38	3	2312	3	0.01	92	0.28
s4863	153	104	126988	179	0.01	651	6.36
s9234.1	247	250	4434504	122	0.61	-	-
C1908	33	25	30832	0	0.01	749	8.56
C3540	50	22	4618194	4	0.81	-	-
C432	36	7	32151	0	0.01	224	3.19
C499	41	32	110675	0	0.02	1312	48.01
C880	60	26	600998	33	0.06	392	332.36

Table 1: Experimental Results

6 Discussion

It should be noted that bit-blasting also provides a way to realise a backward analysis for discovering irrelevant bits. Suppose again that an ROBDD f encodes the relationship between the input bits $\vec{x}$ and the output bits $\vec{x}'$. Suppose also that $Y' \subseteq \vec{x}'$ is a set of output variables that are known to be irrelevant. This information can propagated backwards by projecting Y' out of f to obtain g. If $y \in \vec{x}$ then y is irrelevant if $g|_{y \leftarrow 0} = g|_{y \leftarrow 1}$. The set $Y = \{y \in \vec{x} \mid g|_{y \leftarrow 0} = g|_{y \leftarrow 1}\}$ then represents the input bits that are irrelevant. This technique can be refined because if the input state over $\vec{x}$ is known to be f_o, then Y can be computed by $Y = \{y \in \vec{x} \mid g_y|_{y \leftarrow 0} = g_y|_{y \leftarrow 1}\}$ where $g_y = \exists_y(f_o) \wedge \exists_{Y'}(f')$. For instance, if $f = (u' \iff u) \wedge (v' \iff u \wedge v)$, $\vec{x} = \langle u, v \rangle$, $\vec{x}' = \langle u', v' \rangle$ and neither u' nor v' are known to be irrelevant, then the input variable v can still be detected as being irrelevant if it is additionally known that $f_o = (\neg u)$. Thus, as suggested by Budiu and Goldstein [4], backward analysis should ideally be applied in conjunction with (after) forward analysis.

Although ROBDDs are the natural workhorse for equivalence checking, SAT techniques are potentially more robust and flexible [9]. The above analysis can be reformulated using SAT because $x_i \in Y$ if $g(\vec{x}_0) \oplus g(\vec{x}_1)$ is unsatisfiable where $\vec{x}_b = \vec{x}[x_i \mapsto b]$. Moreover, if $x_j \in Y'$ this test can be relaxed by checking whether $g(\vec{x}_0) \oplus g(\vec{x}_2)$ is unsatisfiable where $\vec{x}_2 = \vec{x}_1[x_j \mapsto x'_j]$ and x'_j is a fresh variable. This motivates reformulating the forward analysis using SAT, and certainly simple equivalences are detectable using cutpoint techniques [9].

7 Related Work

Interestingly, the literature already suggests an approach for approximating an ROBDD with an affine function, albeit a very restricted one. Bagnara and Schachte propose an $O(|f|n^2)$ algorithm [1] for finding all pairs of propositional variables x and y such that $f \models (x \iff y)$ where n and $|f|$ are the number of variables and nodes in the ROBDD f. This information is then used to remove all equivalences from the ROBDD so that they can be stored separately, so as to achieve a more memory-dense representation.

The abstraction algorithms presented in this paper share an important property with the ROBDD widening of Kettle, King and Strzemecki [12]. A widening [7] is an approximation that is applied to curtail the growth of an abstraction. In the context of an ROBDD [12], the idea is to systematically add models so as to construct an ROBDD that can be represented more compactly. Unlike earlier techniques [17, 21] that are informed only by the syntactic structure of the ROBDD, the widening, and affine approximation algorithms presented in this paper, are insensitive to the variable ordering.

Affine formulae also arise in knowledge compilation where the idea is to abstract a knowledge base to either a core or an envelope [20]. These are more tractable knowledge bases that contain, respectively, fewer and more solutions. Zanuttini [23] suggests using (lower- and upper-) affine functions for the core and envelope, that are represented as sets of binary vectors, and mentions that such sets can be minimised by applying Gaussian elimination. Very recently, Henshall *et al.* [11] have developed an elegant algorithm for computing the affine envelope of an ROBDD, that uses ROBDDs for the intermediate results. On random formulae, the envelope of a 24 variable ROBDD can be computed in approximately 15 secs on a Solaris 9 machine, equipped with two 2.8GHz Intel Xeon CPUs and 4GB of memory.

8 Conclusions

The paper has investigated two classes of Boolean function that can only admit chains who length does not exceed the number of underlying variables. Novel algorithms have been presented for abstracting ROBDDs to formulae in these classes: one algorithm inspired by anti-unification and the other by Gaussian elimination.

Acknowledgements This work was funded, in part, by EPSRC grant EP/F012896 and a Royal Society Industrial Fellowship that has enabled Andy King to visit Portcullis Computer Security. We thank Paul Docherty for suggesting obfuscation and Laurent Mauborgne for spotting the connection between affine functions of width 2 and weakly relational domains.

References

[1] R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*. In *Algebraic Methodology and Software Technology*, volume 1548 of *LNCS*, pages 471–485. Springer-Verlag, 1999.

[2] F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *International Conference on Formal Methods in Programming and their Applications*, number 735 in LNCS, pages 128–141. Springer-Verlag, 1993.

[3] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[4] M. Budiu and S. C. Goldstein. Bitvalue Inference: Detecting and Exploiting Narrow Bitwidth Computations. Technical Report CMU-CS-00-141, Carnegie Mellon University, PA 15213, 2000.

[5] M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein. Bitvalue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Sixth International Euro-Par Conference on Parallel Processing*, volume 1900 of *LNCS*, pages 969–979. Springer-Verlag, 2000.

[6] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2–3):105–127, 2004.

[7] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295. Springer-Verlag, 1992.

[8] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Monographs On Discrete Mathematics And Applications. SIAM, 2001. ISBN 0-89871-479-6.

[9] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for Combinational Equivalence Checking. In *Conference on Design, Automation and Test in Europe*, pages 141–121. ACM, 2001.

[10] W. Harrison III. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, 1977.

[11] K. Henshall, P. Schachte, H. Søndergaard, and L. Whiting. Binary Decision Diagrams for Affine Approximation, April 2008. `http://arxiv.org/abs/0804.0066`.

[12] N. Kettle, A. King, and T. Strzemecki. Widening ROBDDs with Prime Implicants. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 105–119. Springer-Verlag, 2006.

[13] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[14] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[15] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[16] G. Plotkin. A Note on Inductive Generalisation. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[17] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and Decomposition of Binary Decision Diagrams. In *Design Automation Conference*, pages 445–450. IEEE, 1998.

[18] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, Cambridge, MA, USA, May 1994.

[19] T. J. Schaefer. The Complexity of Satisfiability Problems. In *ACM Symposium on Theory of Computing*, pages 216–226. ACM Press, 1978.

[20] B. Selman and H. Kautz. Knowledge Compilation and Theory Approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[21] T. R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Electronics Research Laboratory, 1996.

[22] F. Somenzi. CUDD Package, Release 2.4.1. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[23] B. Zanuttini. Approximating Propositional Knowledge with Affine Formulas. In *European Conference on Artificial Intelligence*, pages 287–291. IOS Press, 1992.