Structural Search for RTL with Predicate Learning

G. Parthasarathy M. K. Iyer K.-T. Cheng F. Brewer
Dept of Electrical and Computer Engineering
University of California – Santa Barbara
Santa Barbara, CA

ABSTRACT

We present an efficient search strategy for satisfiability checking on circuits represented at the register-transfer-level (RTL). We use the RTL circuit structure by extending concepts from classic automatic test-pattern generation (ATPG) algorithms and interval-arithmetic to guide the search process. We extend the idea of Boolean recursive learning on predicate logic in the RTL using Boolean and interval constraint propagation in the control and data-path of the circuit. This is used as a pre-processing step to derive relations between predicate logic signals that are used to augment the search. We demonstrate experimentally that these methods provide significant improvement over current techniques on sample benchmarks.

Categories and Subject Descriptors: B.5 [Register Tranfer Level Implementation] – *verification, testing*

General Terms: Algorithms, Verification

Keywords: Interval Arithmetic, Learning, Predicate Abstraction, Satisfiability.

1. INTRODUCTION

Checking the *satisfiability* (SAT) of complex first-order logic formulas is a frequent problem when verifying hardware designs described at the *register-transfer level* (RTL). There have been significant improvements in Boolean SAT solver run-time performance and capacity over the last decade. Consequently, the most popular method of solving a satisfiability problem on RTL is to use a Boolean SAT solver on its Boolean translation. Unfortunately, these solvers show poor performance for RTL circuits with large data-paths [2]. This has spurred research into devising algorithms that can operate on the native problem with performance comparable or better than Boolean SAT engines. One promising method is to combine decision theories for Boolean and integer arithmetic so that the problem can be solved at the RTL level. In this paper, we propose a technique that takes advantage of circuit structure to improve performance of one such algorithm – *hybrid DPLL* (HDPLL) [12].

Related Work. There has been considerable activity in trying to integrate decision procedures for the Boolean and integer domains into a *combined decision procedure* (CDP), resulting in tools such as the *Cooperating Validity Checker* (CVC) [16], the *Integrated Canonizer and Solver* (ICS) [5], and UCLID [15]. The authors of [12] used *constraint propagation* (CP) to integrate Boolean SAT and Fourier-Motzkin elimination (FME) [3, 13] in a *Hybrid-DPLL* decision procedure. They augmented this with conflict-based learning over the combined decision procedure to bound the combined search space. This proved to be considerably faster than several state-of-the-art CDPs on sample benchmarks [9].

Current CDPs typically ignore the structural information in RTL circuits. This restricts the flexibility of the decision strategies that can be applied during search. Ghosh *et al.,* [6] tried to use the structure of the RTL represented as a decision diagram, using a 9-valued algebra for functional test-pattern generation. However, their approach is inefficient for satisfiability checking.

None of the current state-of-art CDPs explicitly use the structure of the problem to aid their decision strategy. It has been shown that using correlation between signals can provide significant performance gains in Boolean problems. Typically, these correlations have been found (a) explicitly through *static learning*, *e.g.,* [10, 17], (b) implicitly through a dynamic structural analysis [1] or, (c) by using probabilistic measures. It is well recognized that *predicate* analysis is one of the most important components of controlling search complexity in high-level problems. Intuitively, using structural information based on predicate analysis should improve performance in a DPLL-style search algorithm on RTL descriptions.

Paper Overview and Contributions. In this paper, we describe how structural analysis and recursive learning on predicate logic can be used to guide a DPLL style CDP that combines Boolean and integer decision procedures. In Section 2, we describe some basic concepts used in this paper including the hybrid DPLL algorithm proposed in [9, 12]. In Section 3, we describe how constraint propagation can be used to easily extend the traditional methods of static learning in Boolean circuits to RTL circuits. We demonstrate that learning correlations between predicates improves the efficiency of the constraint solving procedure. In Section 4, we describe a method to analyze the structure of the RTL description to guide decision making in HDPLL. We also describe how we do additional conflict-based learning based on inconsistencies found using this approach. In Section 5, we demonstrate that the proposed approach gives considerable performance gains on difficult examples. We present experimental results comparing the new approach with HDPLL [9], ICS [5], and UCLID [15].

2. BACKGROUND

In this section, we briefly introduce some of the main concepts relevant to this work. We begin with some definitions.

2.1 Definitions

A *domain* $D(v)$ is an integer *interval* that is a complete mapping from a variable v to a finite set of integers. A Boolean variable has a domain of $\langle 0,1\rangle$, and a word variable of bit-width w, has a integer valued domain of $\langle 0,2^w-1\rangle$. A *literal*, is the appearance of a variable in a *clause*. A literal $l_i/\overline{l_i}$ of a Boolean variable v_i denotes its value as 1 or 0. A word-literal of a word-variable v_j is associated

Permission to make digital or hard copies of all or part of this work for personal or classroom use granted without fee provided copies are not made or distributed for profit or commercial advantage and bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute, requires prior specific permission and/or a fee.
DAC 2005, June 13–17, 2005, Anaheim, California, USA.
Copyright 2005 ACM 1-59593-058-2/05/0006 ...$5.00.

with a interval b_j as a pair, $\{l_j, b_j\} / \{\overline{l_j}, b_j\}$. A positive literal in the pair, denotes that v_j has interval b_j, and a negative literal in the pair denotes that v_j has values in $d(v_j) \backslash b_j$.

A *clause* is a disjunction of Boolean literals. A *hybrid clause* is a disjunction of Boolean and word literals, with a finite interval $b_i = \langle l, m \rangle$, $l \leq m$, on each word-literal. All linear arithmetic data-path operations are represented using arithmetic constraints [12]. *comparison* operators are represented as a pair of inequalities. Let $b_1 \models w_1 \geq w_2$ and $b_2 \models w_2 \geq w_1$. Let the Boolean variable $b \models w_1 \equiv w_2$. Then the comparator is modeled by the Boolean function, $(b_1 \vee b_2) \wedge (b_1 \vee \overline{b}) \wedge (b_2 \vee \overline{b}) \wedge (\overline{b_1} \vee \overline{b_2} \vee b)$. Non-linear operations such as bit-vector concatenation, extraction, shift *etc*, are modeled as arithmetic constraints by adding auxiliary variables [2].

A *predicate constant* or *first-order predicate* is an operator or function over $\{<, >, \equiv, \leq, \geq\}$, which returns a Boolean value, **true** or **false**. All operations in RTL that return a Boolean value and interact with data-path are treated as predicates.

2.2 Interval Arithmetic

We shall limit ourselves to an informal introduction to interval arithmetic and a description of some of the aspects of interval arithmetic that we need for word-level implications in RTL data-path. The interested reader is referred to the references [8] and [14] for details.

Let $\{\langle \underline{u}, \overline{u} \rangle \, | \underline{u} \leq \overline{u}, \underline{u}, \overline{u} \in I\}$, be the set of all closed finite integer intervals. Let $\langle x \rangle = \langle \underline{x}, \overline{x} \rangle \in I$ and $\langle z \rangle = \langle \underline{z}, \overline{z} \rangle \in I$ be two intervals on variables x and z. We extend $\circ \in \{+, -, \times\}$ to an operation over intervals $\langle \circ \rangle$ by defining:

$$\begin{aligned} x \langle \circ \rangle z &= \langle \underline{x}, \overline{x} \rangle \langle \circ \rangle \langle \underline{z}, \overline{z} \rangle \\ &= \langle \min\{u \circ v\}, \max\{u \circ v\} | u \in \langle x \rangle, v \in \langle z \rangle \rangle \end{aligned} \tag{1}$$

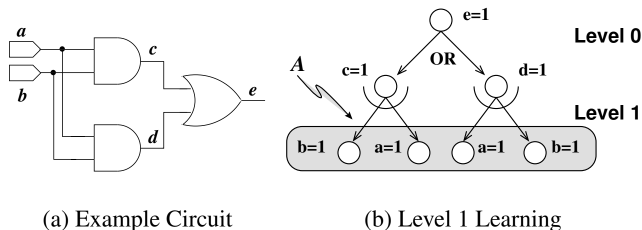
Let C be a set of constraints. We can take each constraint $c_i \in C$, choose each variable $v_j \in c_i$, and remove all values for v_j that do not participate in a solution to C. For example, given the constraint $x - z < 0 | x \in \langle 0, 15 \rangle, z \in \langle 0, 15 \rangle$, we can narrow the intervals on both x, and z to $x \in \langle 0, 14 \rangle$ and $z \in \langle 1, 15 \rangle$. More generally;

$$\text{Given} \;:\; x - z < 0; \text{ s.t. } x \in \langle \underline{x}, \overline{x} \rangle, z \in \langle \underline{z}, \overline{z} \rangle \tag{2}$$

$$\text{then} \;:\; x \in \langle \underline{x}, \min(\overline{x}, \overline{z} - 1) \rangle, z \in \langle \max(\underline{z}, \underline{x} + 1), \overline{z} \rangle \tag{3}$$

The rule in Equation 3 defines how to propagate the effect of the intervals on variables in a constraint of the type $v_i - v_j < 0$ defined in Equation 2. Similarly, we can define rules for constraints of type $x \circ z$ of arbitrary arity. This process is called interval constraint propagation or *constraint propagation* for short.

A constraint propagation procedure iteratively performs interval narrowing on an event-driven basis for all variables and all constraints in C, till no changes are possible. If no conflict is found, then each constraint $c_i \in C$ is set to be *bounds consistent*. Constraint propagation quickly removes non-solution values from the variables in C. The resulting solution-space is $\mathcal{P} = \prod_i D'(v_i) | \{v_i \in c_j, D'(v_i) \subseteq D(v_i), \forall c_j \in C\}$. P is also called a *solution box* that is guaranteed to hold all solutions to C. A non-empty solution box does not guarantee existence of a solution. Constraint propagation is monotonic since successive iterations will always reduce intervals.



(a) Example Circuit (b) Level 1 Learning

Figure 1: Recursive Learning.

2.3 Recursive Learning

Recursive learning was originally proposed for ATPG and verification on Boolean circuits [10]. The basic idea behind recursive learning is to find all possible ways W of satisfying a value assignment $val(s_i)$ for a signal s_i in a Boolean circuit. If $W \to A$, where A is a set of implications that hold for all W, then values $a_j \in A$ must hold if $val(s_i)$ holds *i.e.,* $\forall a_j \in A, val(s_i) \to a_j$.

The example in Figure 1 shows the result of recursive learning to level 1 for the signal assignment $e = 1$. $e = 1$ can be satisfied either by $c = 1$ or $d = 1$. These values are recursively satisfied at the next level (1) in isolation. The common implications are learned as $e = 1 \to a = 1$, and $e = 1 \to b = 1$. The procedure can be extended to an arbitrary recursion depth to provide a complete Boolean decision procedure. However, this can be very expensive on large circuits, and is rarely used as a stand-alone decision procedure.

2.4 Hybrid DPLL Overview

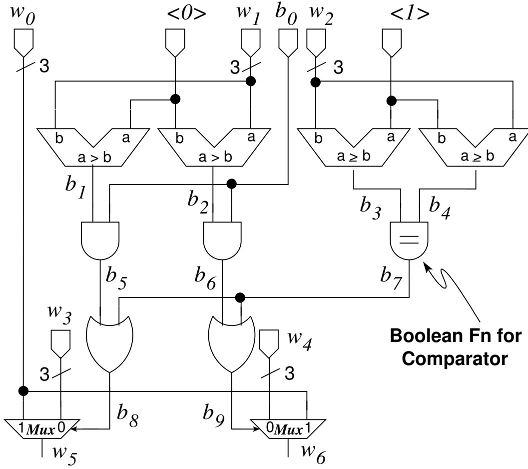
Most modern algorithms for Boolean SAT are variants of a *branch and bound* algorithm called the DPLL algorithm [4]. These algorithms use a combination of deciding value assignments on variables and *Boolean constraint propagation* (BCP), which implies additional value assignments to determine if the search has moved outside the solution space – a *conflict*. On finding a conflict, a technique called *conflict-based learning* is employed to prune the space for future search [11]. This method finds the assignments that drove the search into the unsatisfiable space. A constraint or *learned clause* is then added, to ensure that this set of assignments is never made again during search. Typically, the decision strategy is guided by these learned constraints. These techniques are the fundamental reasons for the efficiency of modern SAT solvers.

Algorithm 1: Hybrid DPLL algorithm.

```
Data: RTL satisfiability problem ℙ
1 blevel ← 0
2 while (True) do
3     while (Decide() ≠ done ) do
4         deduction ← null
5         while ((deduction ←Ddeduce() ) == conflict ) do
6             blevel ← analyze_conflicts()
7             if blevel == 0 then return UNSAT
8             else backtrack(blevel)
9         if deduction ≡ SAT then return SAT
```

The main elements of the HDPLL algorithm described in [9, 12] is shown in Algorithm 1. The procedure **Decide**() makes decisions only on Boolean variables. A decision variable is picked based on an exponentially decaying function based on its original fanout and the number of learned clauses that it appears in. The procedure **Ddeduce**() performs hybrid consistency checking using interval constraint propagation. The set of assignments to **Ddeduce**() can be Boolean values or intervals on word-variables. If constraint propagation does not find a conflict, and all decision variables are assigned, then the solution box, P, is checked for a point solution using an integer-linear solver that performs *fourier-motzkin elimination*– The Omega library [13]. If a point solution exists, the instance is certified as satisfiable. If not, then the algorithm flags a conflict.

Given a sequence of value assignments in HDPLL, we can construct a graph that represents the causal relationship between value assignments, called the *hybrid implication graph*. This implication graph is a directed graph $IG(N, E)$, where N is the set of nodes and E is the set of edges. A node $n \in N$ represents a value assignment to a variable, or a resolvent from arithmetic solving. A directed edge $e \in E$ exists from nodes n_a to n_c if n_a, is (part of) the value assignment(s) that implies the value assignment n_c or if n_a resolved to n_c.



(a) RTL Circuit Example

1) $\mathbf{b_5 = 0}$:
$\mathbf{b_1 = 0} \rightarrow \{w_1 = \langle 0 \rangle, b_2 = 0, b_6 = 0\}$
$\mathbf{b_0 = 0} \rightarrow \{b_6 = 0\}$
Learn $\overline{b_5} \rightarrow \overline{b_6} \equiv (b_5 \vee \overline{b_6})$

2) $\mathbf{b_6 = 0}$:
$\mathbf{b_2 = 0} \rightarrow \{w_1 = \langle 0 \rangle, b_1 = 0, b_5 = 0\}$
$\mathbf{b_0 = 0} \rightarrow \{b_5 = 0\}$
Learn $\overline{b_6} \rightarrow \overline{b_5} \equiv (b_6 \vee \overline{b_5})$

3) $\mathbf{b_8 = 1}$:
$\mathbf{b_5 = 1} \rightarrow \{b_6 = 1, b_9 = 1, b_2 = 1, b_0 = 1, w_1 = \langle 1, 7 \rangle\}$
$\mathbf{b_7 = 1} \rightarrow \{b_9 = 1, b_3 = 1, b_4 = 1, w_2 = \langle 1 \rangle\}$
Learn $b_8 \rightarrow b_9 \equiv (\overline{b_8} \vee b_9)$

4) $\mathbf{b_9 = 1}$:
$\mathbf{b_6 = 1} \rightarrow \{b_5 = 1, b_8 = 1, b_0 = 1, b_2 = 1, w_1 = \langle 1, 7 \rangle\}$
$\mathbf{b_7 = 1} \rightarrow \{b_8 = 1, b_3 = 1, b_4 = 1, w_2 = \langle 1 \rangle\}$
Learn $b_9 \rightarrow b_8 \equiv (\overline{b_9} \vee b_8)$

(b) Predicate Learning for example

Figure 2: Predicate-based learning in an RTL circuit

If a conflict is found during constraint propagation, we find a set of nodes (*cut*) in *IG* that covers all implication paths to the conflict. The conjunct of these nodes is a conjunct of value assignments($\bigwedge_i l_i$), that is sufficient to cause the conflict. The negation of this is a disjunct ($\bigvee_i \neg l_i$) that *must* be true for the conflict to be avoided. This is added to the problem as a *conflict-avoiding* or *learned clause*. HDPLL can learn clauses where the literals can be Boolean or word variables with associated values and intervals respectively. This yields a powerful conflict-based learning technique, which was described in [9]. If the learned clause conflicts with the proposition, (`blevel = 0` in Algorithm 1), the problem is UNSAT. If not, HDPLL backtracks and continues search.

3. PREDICATE-BASED LEARNING

The main problem in any DPLL style algorithm on RTL is the lack of information regarding correlation between predicates that control the data-path. We describe a static learning procedure that is based on recursive learning [10] extended by *interval constraint propagation* in the data-path. The procedure performs the following steps:

1. The RTL circuit is level-ordered by distance from the primary inputs. Predicate logic that controls the data-path is extracted by a cone-of-influence analysis. All Boolean inputs to arithmetic operators, such as control signals to multiplexers are classified as predicates.
2. We then do recursive learning of level 1 for the controlling value of each gate in the list of candidates, starting with the gate with the lowest level. Interval constraint propagation is used for implications across the data-path. This allows us to learn relations across control and data-path. These learned relations are stored as learned clauses, and are used in successive recursive learning steps. A threshold on the number of relations learned is used to control run-time in static learning.
3. If a conflict is found at any point, conflict analysis on the hybrid implication graph is used to determine the set of causes for the conflict.
4. All learned relations are stored as learned clauses and further used during the learning procedure.
5. The learned relations guide the decision strategy by assigning a higher weight to variables in these relations. Free decision variables are picked in order by highest weight first.

We now explain the procedure with the help of an example in Figure 2. Consider the fragment of RTL shown in Figure 2(a) (from circuit B04 in the ITC'99 benchmarks). The signals b_8 and b_9 are of interest, since they are control logic signals that define predicates for the data-path. It would be very useful to extract relations between these signals if they exist. The static learning in this example proceeds as shown in Figure 2(b).

All input assignments to set $b_5 = 0$ result in the common implication $b_6 = 0$. Therefore, we learn the clause $(b_5 \vee \overline{b_6})$, which captures both the implication and the contrapositive. Similarly, we learn the clause $(b_6 \vee \overline{b_5})$. These clauses are used in the next two variable probes $b_8 = 1$ and $b_9 = 1$, where they provide extra implications that enable us to learn $(\overline{b_8} \vee b_9)$ and $(\overline{b_9} \vee b_8)$, respectively. Note that this captures the information that $(w_5 \equiv w_3) \rightarrow (w_6 \equiv w_4)$ and $(w_5 \equiv w_0) \rightarrow (w_6 \equiv w_0)$. Therefore, some of the correlation between data-path signals $\{w_0, w_5, w_6\}$ and $\{w_3, w_4, w_5, w_6\}$ is captured by these learned relations.

3.1 Analysis

In order to evaluate the effect of predicate learning on the search, we performed the following experiment. We generated some problems based on the *bounded model checking* of safety properties on the RTL versions of a subset of the ITC'99 benchmark set. The size of the test-cases range from 186/322 to 27437/22971 word/Boolean operations with bit-widths ranging from 3 to 10.

We ran the predicate learning as a pre-processing step in order to extract relations, which are used as learned clauses to guide the search. Though the proposed learning method is limited to Boolean predicate logic in RTL, the incremental cost can be extremely high – up-to 10x the actual run-time. Therefore, we restricted the amount of learning by setting a threshold of 2500 learned relations. This allowed us to learn a limited amount of information at a relatively low cost.

We then ran HDPLL [9] without predicate learning and HDPLL with predicate learning in order to compare the efficacy of predicate learning in speeding up search. Table 1 shows the results of the experiment. Column 1 shows the name of the test-case, the property name, and the bound of the test-case. For example, b01_1(10) is a BMC problem on property 1 on b01 expanded for 10 time-frames. Column 2 denotes whether the test-case was satisfiable(S) or unsatisfiable (U). Columns 3 and 4 show the number of relations learned and the CPU time taken for learning. Columns 5 and 6 show the

run-time in CPU seconds of HDPLL without predicate learning and with predicate learning respectively. The experiments were run on a 2.4 GHz Pentium IV running linux(kernel v2.4.22).

Ckt	Type	No. Rels	Learn Time	HDPLL	HDPLL Pred. Learn
b01_1(10)	S	352	0.02	0.01	0.02
b01_1(20)	U	794	0.04	0.48	0.19
b02_1(10)	U	393	0.02	0.16	0.16
b02_1(20)	U	884	0.07	0.65	0.51
b04_1(20)	S	144	0.23	0.04	0.04
b13_5(10)	U	587	0.25	0.01	0.0
b13_1(10)	U	549	0.24	0.01	0.0
b13_5(20)	U	1169	0.91	0.09	0.13
b13_1(20)	U	1082	1.09	0.04	0.11
b13_5(30)	U	1043	1.35	0.56	0.41
b13_1(30)	U	1044	1.03	0.14	0.43
b13_5(50)	U	1036	0.79	3.86	0.22
b13_1(50)	U	2064	1.78	4.99	0.3
b13_5(100)	U	2061	1.59	111.63	11.5
b13_1(100)	U	2050	1.59	85.31	1.27
b13_5(200)	U	2063	1.85	37.69	1.96
b13_1(200)	U	2060	1.68	56.24	1.85
b13_1(300)	U	2060	1.68	587.42	21.76

Table 1: Run-Time Analysis of Predicate Learning

As we can see from Columns 5 and 6 in Table 1, the overhead of predicate learning outweighs any run-time gains on the smaller test-cases (b01_1(10)–b13_1(30)). However, as the test-cases become more complex, the benefit of learning even a limited amount of predicate correlation become evident. For the last 6 test-cases, we see 2x to 80x improvements in run-time performance.

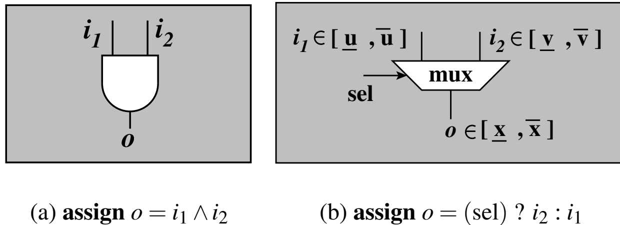
This experiment clearly illustrates the advantages and disadvantages of this kind of predicate learning. A limited amount of predicate learning is quite useful. Complete learning is infeasible due to the high-overhead. However, it is clear that taking advantage of data-path correlation is a promising method of improving search efficiency. With this in mind, in the next section, we describe a new method of using the structure of the data-path to guide the decision strategy of the hybrid search.

4. STRUCTURAL ANALYSIS

In this section, we extend the ideas behind structural decision strategies for Boolean circuits to RTL descriptions with data-path.

4.1 Justification

A Boolean gate that has a required output value that cannot be satisfied by implication (*e.g.,* $o = 0$ on the output of the AND gate in Figure 3(a)) is termed *un-justified*.



(a) **assign** $o = i_1 \wedge i_2$ (b) **assign** $o =$ (sel) ? $i_2 : i_1$

Figure 3: Justification of RTL types.

Satisfiability methods [1] that use the structure of the circuit, rely on a data-structure that holds a set of Boolean signals called the *justification frontier* or *J-frontier*. The signals in the J-frontier satisfy the property that the proposition is satisfied *if and only if* every signal in the justification frontier is satisfied. *Justification* selects one of the gates in the *J-frontier* and recursively selects an unassigned input of the gate, using a 3-valued algebra $\in \{0, 1, X\}$, in a depth-first manner until a valid decision point is reached [1]. For example, we can decide on inputs $i_1 = 0$ or $i_2 = 0$ to satisfy $o = 0$ in Figure 3(a).

Next, we describe a new decision strategy in HDPLL that includes analysis of the structure of the data-path

4.2 Justification in RTL Data-paths

The fundamental idea behind justification in RTL data-path is that *multiplexers* and other RTL operators with Boolean inputs allow a choice of data-path relations that can satisfy control point assignments. We could end up enumerating on these RTL operator inputs, without a meaningful decision strategy. We can construct one by recognizing the existence of *unjustified* arithmetic operators in the data-path. This is defined as follows:

DEFINITION 4.1 (RTL JUSTIFIABILITY). *An RTL operator is classified as* justifiable *by the following rules:*

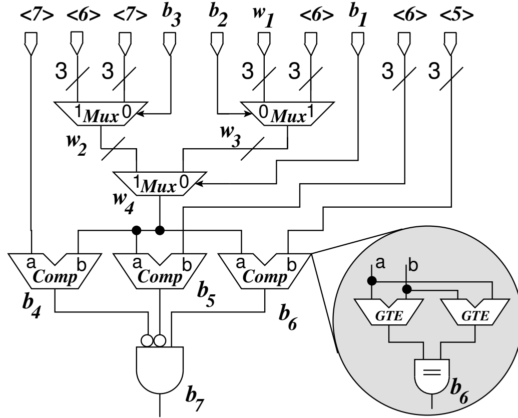
1. *Atomic Boolean Operators($\wedge, \vee$): If the output value cannot be uniquely determined by current values on inputs, similar to conditions described above and in [1].*
2. *Non-Arithmetic word-level Operators(ITE, con-cat): If the operator has a Boolean input and the output interval cannot be uniquely determined by the current input intervals/values.*

Operators such as ($\neg, +, -, \ll$ *or* $\gg$ by *k, multiplication* by *k, sign extension,* and *bit-vector extraction*) are not justifiable, *i.e.,* their input/output intervals or values are determined solely through constraint propagation. For example, consider the output interval $\langle \underline{x}, \overline{x} \rangle$ on the ITE in Figure 3(b). By the properties of interval arithmetic, the output interval can be satisfied by any of the inputs i_1 or i_2 that satisfy the condition that $\langle \underline{u}, \overline{u} \rangle \cap \langle \underline{x}, \overline{x} \rangle \neq \emptyset$ *i.e.,* (sel=**false**) or $\langle \underline{v}, \overline{v} \rangle \cap \langle \underline{x}, \overline{x} \rangle \neq \emptyset$ *i.e.,* (sel=**true**) respectively. HDPLL makes decisions only on Boolean variables. A pure arithmetic operation like $+$ cannot be justified, since it does not have any *decidable* inputs. In theory, it is possible to make decisions on word-level variables and thus all arithmetic operators can be defined as justifiable. However, in practice, we have found that it is considerably more efficient to make decisions on Boolean control than word-level data-path. Therefore, the current formulation does not consider any arithmetic operator that does not have a Boolean input as justifiable.

Algorithm 2: Algorithm for Structural Decision-making.

```
1  if ( 𝔻 ← find_free_decision_vars() == ∅) then
2      return done
3  else
4      J-frontier ← find_jfrontier(𝔻)
5  foreach ( g_j ∈ J-frontier) do
6      if ( not_justified(g_j) ) then
7          if (decision ←justify(g_j)) ≠ null then
8              make_decision(decision)
9              return True
10         else
11             blevel ← analyze_dpath_conflict (g_j)
12             backtrack(blevel)
```

The algorithm for the procedure `Decide()` in HDPLL (see Algorithm 1) is modified to perform structure-based decision making. The new algorithm is shown in Algorithm 2. The dynamic J-frontier is maintained implicitly by maintaining an ordered list of all Boolean



(a) Example circuit for RTL justification

HDPLL Setup	: $w_2 = \langle 6,7 \rangle, w_3 = \langle 0,7 \rangle, w_1 = \langle 0,7 \rangle$
Imply Proposition	: $b_7 = 1 \rightarrow \{b_4 = 0, b_5 = 0, b_6 = 1, w_4 = \langle 5 \rangle\}$
J-frontier	: $\{w_4 = \langle 5 \rangle\}$
`Decide()`	: $w_4 \cap w_2 = \emptyset$
	: $w_3 \in w_4$; **return**$(b_1 = 0)$
Imply decision	: $b_1 = 0 \rightarrow w_3 = \langle 5 \rangle$
J-frontier	: $\{w_3 = \langle 5 \rangle\}$
`Decide()`	: $\langle 6 \rangle \cap w_3 = \emptyset$
	: $w_1 \in w_3$; **return**$(b_2 = 0)$
Imply decision	: $b_2 = 0 \rightarrow w_1 = \langle 5 \rangle$
J-frontier	: $\emptyset$
`Decide()`	: **return**(`DONE`)
HDPLL	: **return**(`SATISFIABLE`)

(b) RTL justification for example

Figure 4: Example for Structural Decision making in an RTL circuit

gates. The procedure **`justify`** iteratively picks decisions in the Boolean control such that all gates in **`J-frontier`** are satisfied. If a choice of free inputs to g_j exists, the procedure uses heuristics based on fanout-count and distance from the inputs to guide it. If g_j is a justifiable RTL operator, then **`justify`**(g_j) does a breadth-first analysis to find the first Boolean decision point that *can* satisfy g_j. If all decision variables are assigned, then the Boolean control is satisfied, while the data-path is bounds consistent. At this point, we call the arithmetic solver to check for the existence of an integer solution in the *solution box* defined by interval constraint propagation.

Assume that we wish to check the satisfiability of the proposition $b_7 = 1$ in Figure 4(a). The steps of HDPLL for this example with the RTL justification is shown in Figure 4(b). Initially, we find the maximum intervals for all word-level variables, and imply the proposition. This results in the values $\{b_4 = 0, b_5 = 0, b_6 = 1, w_4 = \langle 5 \rangle\}$. The gates corresponding to b_4, b_5, b_6 are not justifiable, but w_4 is. Therefore, the J-frontier is now $\{w_4 = \langle 5 \rangle\}$.

As described before, multiplexers, such as w_4, can be satisfied by picking a Boolean value on the select signal. If no such value exists, then no value in the output interval can be satisfied, and we flag a conflict. We make no such assumptions about arithmetic operators such as addition, subtraction and comparison. Therefore, all such operators are assumed to require a complete path from the word-level PIs or a constant to all of its inputs. Any inconsistency on these variables will be detected using constraint propagation.

The first decision that is picked by **`Decide`**() is $b_1 = 0$, since $w_4 \cap w_5 = \emptyset$. This decision is made and the implications determine that w_4 is satisfied. The J-frontier is updated, and now has $w_3 = \langle 5 \rangle$. As before, the appropriate select-line value is chosen ($b_2 = 0$) as the next decision. The implications from this decision show that the J-frontier is empty. This means that the proposition is satisfiable if any solution exists in the *solution box* defined by the current intervals on the word-level variables in the data-path. This is checked by the arithmetic solver. In this case, it certifies that the solution box contains a solution, and hence the problem is satisfiable.

4.3 Structural Conflicts

RTL justification can reach a situation where all assignments to available decision points will leave the current objective unsatisfied. This is called a *J-conflict*. We can backtrack to the last decision and continue. However, this can lead to repetition of decisions which cannot produce a solution. It is possible to analyze the causes for the inconsistency and *learn* a clause that avoids these causes. This method can improve the run-time efficiency and is described below.

The causes for a J-conflict are the set of word-level variables which block the trace in the data-path and the word-level variable(s) in the J-frontier that are currently being justified. We trace through the hybrid implication graph, similar to the hybrid conflict analysis, to find the set of implying Boolean literals that led to the intervals on these word variables. This set of Boolean literals constitutes a sufficient set of causes for the J-conflict. Once we find the learned clause, we find the correct decision to *non-chronologically* backtrack to and continue search.

Let us take the same example in Figure 4(a). Assume that the Boolean variable b_2 has been set to 1, and b_1 is unassigned. This implies that the variable $w_3 = \langle 6 \rangle$. If we were to try to justify $w_4 = 5$, we would find that we cannot satisfy this objective. The causes for the conflict are the implying Boolean literals for $w_4 = 5$ and $w_3 = 6$, which are $b_6 = 1$ and $b_2 = 1$. w_2 is excluded in the conflict causes, since it is not implied. Therefore, we learn the clause $(\overline{b_6} \vee \overline{b_2})$.

4.4 Static Learning

The main weakness of the above strategy is that it has to justify all operators that lie in the cone of unjustifiable operators (such as addition). This can reduce the effectiveness of the strategy on some circuits, since it loses the correlation between predicates that have been satisfied with those that need to be satisfied in the future. We mitigate this effect by weighting the predicate logic signals with the number and type of relations learned during static learning. If we have a choice of values on a predicate signal, like a select to a mux, then we select the value that satisfies the maximum number of learned relations. This yields additional improvements, and shall be discussed in the next section.

5. EXPERIMENTAL ANALYSIS

In this section, we discuss some experiments on the new decision strategy on the hybrid DPLL solver and their results. All experiments were run on a 2.4GHz Pentium 4, with Linux (kernel v2.4.22). The test-cases are bounded-model checking cases from the RTL descriptions of the ITC'99 benchmarks supplied with the VIS distribution. The properties are safety properties on these benchmarks.

Table 2 shows the results of all the experiments in this section. Column 1 shows the test-case details, *e.g.,* b01_1(50) shows the property 1 on RTL circuit b01, with bound of 50 time-frames. The columns 3 and 4 show the number of arithmetic and Boolean operators. Columns 5–9 show the run-times of HDPLL [9], HDPLL with the structural decision strategy, HDPLL with structural deci-

sions and static learning, UCLID, and ICS respectively. All tools were given a time-out of 1200 cpu seconds. In these columns, the symbols **-to-** indicate a time-out at 1200 CPU seconds; and **-A-** indicates that the tool aborted.

Test-case	Rslt	Arith Ops	Bool Ops	HDPLL [9]	HDPLL +S	HDPLL +S+P	UCLID [15]	ICS [5]
b01_1(50)	S	1106	1922	1.75	1.46	**1.36**	2.52	4.63
b01_1(100)	U	2256	3922	7.59	10.36	**1.96**	122.07	52.09
b02_1(50)	U	2020	2115	4.31	3.51	**1.47**	23.79	37.04
b02_1(100)	U	4120	4315	7.57	3.8	**3.46**	89.88	183.26
b04_1(50)	S	1532	1117	0.64	**0.06**	0.06	**-A-**	55.26
b04_1(100)	S	3132	2267	112.78	0.34	**0.32**	**-A-**	**-to-**
b13_40(13)	S	658	676	0.04	**0.02**	0.02	3.94	19.76
b13_1(50)	U	4138	3422	5.04	0.34	**0.31**	**-to-**	**-to-**
b13_2(50)	U	3772	3313	**0.67**	1.13	0.67	**-to-**	**-to-**
b13_3(50)	U	3838	3758	0.44	**0.05**	0.05	**-to-**	**-to-**
b13_5(50)	U	4187	3471	3.74	2.19	**0.17**	4.27	**-to-**
b13_8(50)	U	3923	3468	**0.08**	0.35	0.35	**-to-**	**-to-**
b13_1(100)	U	8738	7272	86.54	0.73	**0.72**	**-to-**	**-to-**
b13_2(100)	U	8072	7063	4.41	4.29	**4.19**	**-to-**	**-to-**
b13_3(100)	U	8088	7858	**0.09**	1.94	0.09	**-to-**	**-to-**
b13_5(100)	U	8837	7371	113.67	52.96	**0.48**	5.11	**-to-**
b13_8(100)	U	8173	7218	**0.08**	0.36	0.49	**-to-**	**-to-**
b13_1(200)	U	17938	14972	56.04	4.39	**1.89**	**-to-**	**-to-**
b13_2(200)	U	16672	14563	19.1	7.47	**7.41**	**-to-**	**-to-**
b13_3(200)	U	16588	16058	0.14	4.07	**0.11**	**-to-**	**-to-**
b13_5(200)	U	18137	15171	38.07	16.34	**1.99**	10.63	**-to-**
b13_8(200)	U	16673	14718	2.58	2.69	**1.92**	**-to-**	**-to-**
b13_1(300)	U	27138	22672	576.31	245.27	**210.57**	**-to-**	**-to-**
b13_2(300)	U	25272	22063	42.82	19.15	**4.14**	**-A-**	**-to-**
b13_3(300)	U	25088	24258	**0.24**	3.33	3.27	**-to-**	**-to-**
b13_5(300)	U	27437	22971	4.6	**1.1**	1.1	16.55	**-to-**
b13_8(300)	U	25173	22218	4.6	4.1	**2.56**	**-to-**	**-to-**
b13_1(400)	U	36338	30372	8.73	6.7	**6.46**	**-to-**	**-to-**
b13_2(400)	U	33872	29563	105.67	44.83	**12.13**	**-A-**	**-to-**
b13_3(400)	U	33588	32458	**0.32**	37.55	1.32	**-to-**	**-to-**
b13_5(400)	U	36737	30771	7.85	**1.09**	1.09	21.23	**-to-**
b13_8(400)	U	33673	29718	3.85	1.21	**0.66**	**-to-**	**-to-**
-to- indicates time out at 1200 CPU secs.					+S HDPLL w/ struct. dec. strategy.			
-A- indicates abort due to tool failure.					+P HDPLL with predicate learning.			

Table 2: Run-Time Analysis of Structural Decision Strategy

5.1 RTL Justification

In this experiment, we compare the structural decision strategy with those in [9], which used a heuristic that weights Boolean decision variables based on the number of relations it appears in. Columns 5 and 6 shows that the structural decision strategy is faster by an order of magnitude on most cases. b13_3(100), b13_3(200), b13_3(300), and b13_3(400) are interesting test-cases. The randomized decision strategy without justification or learning performed better. The reason is that the property can be proved solely in the control logic for these cases – an ideal case for predicate abstraction [7]. Without the structural decision strategy, the number of implications in the data-path are considerably less (100x). This makes the RTL justification slower than randomized search on these cases.

5.2 Effect of Predicate Learning

Next, we compared the effect of partial static learning on the structural decision strategy. We set a threshold on the number of learned relations as the number of predicate logic gates or 2000, whichever is smaller. Column 6 and 7 shows the relative difference in run-times. We can see that the static learning improves run-times by a further order of magnitude on the difficult cases. On the test-cases for b13_3, the predicate learning improved the basic RTL justification, by capturing predicate relations that found conflicts without extensive implications. This demonstrates the importance of predicate logic in bounding search run-time.

5.3 Comparison with other CDPs

In the last experiment, we compared the new technique with some state-of-art CDPs. We considered several tools that combine multiple decision theories including Boolean logic and some form of integer linear arithmetic, and picked the following:

1. *Integrated Canonizer and Solver*(ICS) [5] combined linear real arithmetic, bit-vectors, uninterpreted functions, and other theories. We used the default options for ICS.
2. *UCLID* [15], which combines the logic of equality with counter arithmetic was used with the options – `sat 0 chaff`.

As we can see from Column 8 and 9, ICS performs worse than HDPLL by an order of magnitude on almost all the cases, and timed-out on the largest test-cases. UCLID performed better on some test-cases, but failed to complete on 22 out of 32 cases. In no case, did either of these CDPs do better than the current techniques.

6. CONCLUSIONS

We have described a method for structural justification for hybrid constraint solving in RTL circuits. We have also described a simple extension to recursive learning on RTL, by constraint propagation. Our experiments demonstrate that the new techniques provide significant performance improvements over state-of-art CDPs.

Predicate learning can be used to improve predicate abstraction methods by capturing relations between predicates. This has the potential to reduce the occurrence of false negatives during abstraction, which is its chief current drawback. Currently, predicate learning does not take into account arithmetic sub-functions that are created by splitting Boolean variables. This can yield a considerably more powerful learning technique, since it can remove much of the overhead of constraint propagation. This is applicable to data-path that has considerable duplication such as in an RTL-RTL equivalence checking environment. We shall explore this in the future.

Acknowledgments. We would like to gratefully acknowledge Dr. Dhiraj Pradhan, Chair, Department of Computer Science, University of Bristol, for his assistance with recursive learning.

7. REFERENCES

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. CS Press, 1st edition, 1990.

[2] R. Brinkmann and R. Dreschler. RTL-Datapath Verification using Integer Linear Programming . In *Proc. of* 15th *VLSI Design Conf.*, pages 741–746, Jan. 2001.

[3] G. Dantzig and B. Eaves. Fourier-Motzkin Elimination and its Dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[4] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Comm. of the ACM*, 5(7):394–297, 1962.

[5] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. *Computer-Aided Verification, CAV '2001*, pages 246–249, 2001.

[6] I. Ghosh and M. Fujita. Automatic Test Pattern Generation for Functional Register-Transfer Level Circuits using Assignment Decision Diagrams. *IEEE Transactions on CAD.*, 20(3):402–415, march 2001.

[7] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS, *Computer-Aided Verification, CAV '97*, pages 72–83, 1997.

[8] T. Hickey, Q. Ju, and M. H. V. Emden. Interval Arithmetic: Principles to Implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[9] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. Efficient Conflict Based Learning in a Constraint Solver for RTL Circuits. *in Proceedings of DATE'2004).* pages 666–671, Mar 2005.

[10] W. Kunz and D. Pradhan. Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems. *IEEE Trans. on CAD*, 13:1143–1158, Sept. 1994.

[11] J.P Marques-Silva and K.A. Sakallah. GRASP - A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers*, 48(5):506–521, 1999.

[12] G. Parthasarathy, M.K. Iyer, K.-T. Cheng, and Li.C. Wang. An Efficient Finite-domain Constraint Solver for RTL Circuits. In *41st DAC*, June 2004.

[13] W. Kelly, *et al.,* The Omega Calculator and Library v1.1.0. Technical report, Dept. of CS, UMCP, November 1996.

[14] R.E. Moore. *Interval Analysis*. Prentice-Hall, NJ, 1966.

[15] S. Seshia, S. Lahiri, and R. Bryant. A Hybrid SAT-based Decision Procedure for Separation Logic with Uninterpreted Functions. In *40th DAC*, pages 425–430, June 2003.

[16] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a Cooperating Validity Checker. In *Computer-Aided Verification, CAV'2002*, pages 500–504, July 2002.

[17] J.-K. Zhao, E. Rudnick, and J. Patel. Static Logic Implication with Application to Redundancy Identification. In *Proc. of the* 15th *VLSI Test Symp.*, pages 288–293, April 1997.