Efficient Conflict-Based Learning in an RTL Circuit Constraint Solver

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

Abstract

We present new techniques for improving search in a hybrid Davis-Putnam-Logemann-Loveland based constraint solver for RTL circuits (HDPLL). In earlier work on HDPLL [7], the authors combined solvers for integer and Boolean domains using finite-domain constraint propagation with heuristic conflict-based learning. In this work, we describe a new algorithm that extends the conflict-based unique-implication point learning in Boolean SAT solvers to hybrid Boolean-Integer domains in HDPLL. We describe data-structures for efficient constraint propagation on the hybrid learned relations, similar to two-literal watching in Boolean SAT. We demonstrate that these new techniques provide considerable performance benefits when compared with other combinations of decision theories.

1. Introduction

Checking the *satisfiability* (SAT) of complex, quantifier-free first-order logic formulas, is an important problem in verifying *register-transfer level* (RTL) hardware designs. Though Boolean SAT solvers have demonstrated impressive advances in the last decade, quantifier-free arithmetic is still expensive to solve in the Boolean domain [2].

There have been several attempts to integrate decision procedures for Boolean and integer domains into a *combined decision procedure*(CDP) *e.g.*,the *stanford validity checker* (SVC) [3], the *integrated canonizer and solver* (ICS) [5], and UCLID [13]. There has been some work on an orthogonal approach where the Boolean and integer domains are considered to be domains of different sizes and values. The resulting problem is solved by a specialized solver *e.g.*,CAMA [10], or by a general finite-domain solver *e.g.*,LPSAT [15]. All these methods have performance problems on RTL circuits.

Recently, HDPLL [7], and DPLL($\mathcal{T}$) [6] independently proposed a new approach by combining decision theories under a branch-and-bound DPLL-style [11] framework. Both approaches cited the importance of constraint propagation and conflict-based learning to maintain efficiency. HDPLL used *finite-domain constraint propagation* (FDCP) to integrate Boolean SAT and integer fourier-motzkin elimination (FME) into a hybrid DPLL procedure. Heuristics for conflict-based learning were used to prune the combined search space. DPLL($\mathcal{T}$) combined the theory of Boolean logic with the theory of *uninterpreted functions with equality* (EUF). There is no mechanism for learning across theories in DPLL($\mathcal{T}$), and it can handle only comparisons with equality, which makes it currently unsuitable for RTL satisfiability.

In this paper, we address an important problem in any DPLL-based combination of decision theories – conflict-based learning. In general, it is desirable to find explanations for conflicts across theories. It is also desirable to represent these explanations in a form that can be efficiently used for constraint propagation. We present an efficient method of analyzing the combined implication graph of both sub-theories to find *unique implication points* (UIPs) [9]. We also present a *lazy implication* data-structure for efficient constraint propagation on these UIPs across theories.

The rest of this paper is organized as follows: First, we describe some basic concepts of modern DPLL and the HDPLL algorithm proposed by [7] in Section 2. We describe a systematic approach to conflict-based learning on the combined *implication graph* and *resolution graph* of HDPLL in Section 3. We then describe data-structures that allow us to learn *hybrid learned clauses* on both the integer and Boolean domains in Section 4. Finally, we compare our approach with some state-of-the-art CDPs – CVC-Lite, ICS, and UCLID for SAT on RTL circuits in Section 5.

2. Background

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

A *finite-domain* $d(v)$, is a complete mapping from a variable v to a finite set of integers. A Boolean variable has a domain of $\{0,1\}$, and a word variable of bit-width w, has a integer valued domain of $\{0,\ldots,2^w-1\}$. A *literal*, is the appearance of a variable in a *clause*. A *clause* is a disjunction of Boolean literals. A *hybrid clause* is a disjunction of Boolean and word literals. A *finite-domain* (FD) constraint is an inequality over constant coefficients a_i of arbitrary sign, word (Boolean) variables x_i with a fixed domain $d(x_i)$ ($\{0,1\}$) of the form:

$$\sum_i a_i \cdot x_i \geq r,\ a_i, r, x_i \in I$$

A literal $l_i/\neg l_i$ of a Boolean variable v_i denotes its value as 1 or 0. A word-literal of a word-variable v_j is associated with a range b_j as a pair, $l_j\langle b_j\rangle/\neg l_j\langle b_j\rangle$. A positive literal in the pair, denotes that v_j has range b_j, and a negative literal in the

1530-1591/05 $20.00 © 2005 IEEE

pair denotes that v_j has values in $d(v_j) \backslash b_j$, *i.e.*, v_j should not have any value in range b_j.

2.1. Efficient Implications

The DPLL algorithm [11] for Boolean SAT problems is based on case-splitting on variables and *Boolean constraint propagation* (BCP). Given a set of assignments, BCP finds *implications* or additional assignments, that must hold for the problem to be consistent. BCP is also used to determine when the search has moved outside the solution space *i.e.*, a *conflict*.

BCP is the most frequently used procedure in any DPLL style algorithm. Therefore, a clause should be evaluated for an implied value, only when it is guaranteed that an implication will occur. This is the basic idea of *lazy implications*. It uses the data-structure called the *two-literal* watching scheme and shown in Figure 1. On average, it yields significant improvement in performance [12, 16], on hard problems with large clauses.

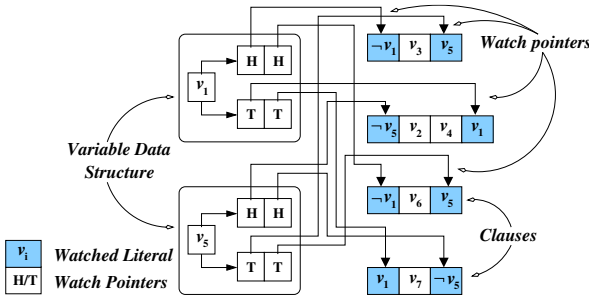


Figure 1. Data-structures for Literal Watching

Each clause initially has two pointers pointing to two distinct unassigned literals. These pointers are kept on the variable corresponding to these literals, and are called *watched* pointers. The two watched pointers on a clause move in opposite directions and schedules implications using the following rules [12]:

1. A watched pointer *moves* to the next unassigned (**unknown**) literal when its literal evaluates to **false**.
2. The pointer does not move when the literal evaluates to **true**, since the clause is satisfied *i.e.*, no further implications can be derived from it.
3. When both pointers point to the same literal on a clause, then an implication is scheduled, since all other literals except the current literal are **false**.

2.2. Conflict-based Learning from Implications

Given a sequence of value assignments in DPLL, we can construct a graph that represents the causal relationship between value assignments, called the *implication graph*. The 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. 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. A *conflict* is represented by a unique node in the IG.

A set of nodes (*cut*) in IG, covering all paths to the conflict node gives a conjunct of value assignments($\bigwedge_i l_i$), that will 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*. The process of finding and using these learned clauses is called *conflict-based learning*. A *unique implication point* (UIP) [9] is a cut in IG that is a dominator for all paths in IG to the conflict, and is smaller than any other non-UIP cut. An example will be shown in Figure 3.

2.3. Fourier-Motzkin Elimination

Fourier-motzkin elimination (FME) [4] is an efficient method to check satisfiability of a set of integer inequalities. FME iteratively eliminates a variable from a set of constraints with n variables, to find an $n-1$ dimensional projection of the constraint set. The *Omega test* by Maslow *et al.*, [14] uses fourier-motzkin with normalization and rounding for testing the satisfiability of a set of integer constraints. This corresponds exactly with the Boolean notion of resolution [8]. *Omega* is used in HDPLL as the decision procedure for the theory of integer arithmetic.

Similar to the implication graph, we can construct a directed graph called a *resolution* graph $RG(C,E)$. It represents the causal relationship between resolution steps in FME. Each node $c_i \in C$ represents a constraint. An edge $e_j \in E$, exists between two nodes, c_k and c_l, if c_k was used to form the resolvent constraint c_l. An example will be shown in Figure 4. We can analyze the resolution graph to find the set of constraints that resolve to a conflict. This yields a powerful conflict-based learning technique, which is described in Section 3.2.

2.4. Hybrid DPLL Algorithm

We now describe the main elements of the HDPLL algorithm proposed in [7], which is reproduced in Figure 2.

```
procedure hdpll()
while (true) do
  while (Decide() ≠ Done) do
    while (FDdeduce() == conflict) do
      blevel = analyze_fdcp();
      if (blevel == 0)
        return UNSATISFIABLE;
      else
        backtrack(blevel);
    end while
  end while
  if (FME() = conflict) then
    blevel = analyze_fme()
  else
    return SATISFIABLE
  if (blevel == 0) then
    return UNSATISFIABLE;
  else
    backtrack(blevel);
end while
```

Figure 2. Hybrid DPLL algorithm.

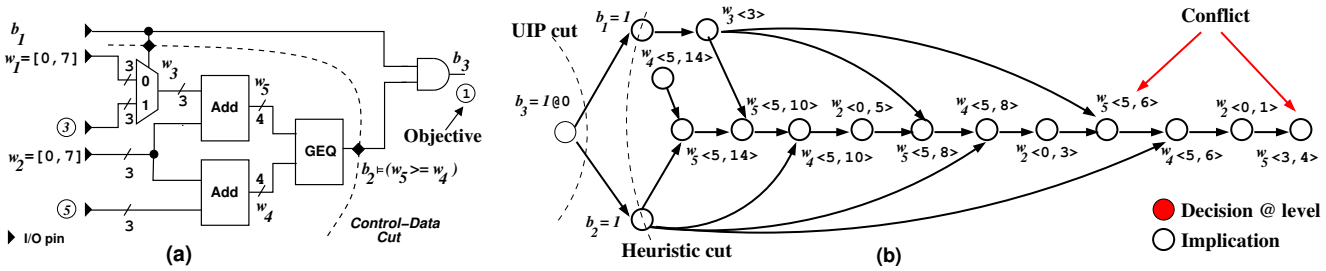


Figure 3. Example for UIP Learning. (a) Circuit with Proposition $b_3 = 1$. (b) Implications for $b_3 = 1$.

The procedure **`Decide`**() makes decisions only on Boolean variables. The procedure **`FDdeduce`**() performs hybrid consistency checking using FDCP similar to BCP in DPLL. The set of assignments to **`FDdeduce`**() can be literals appearing purely in Boolean clauses, or in integer constraints, or domain changes on word-level variables. **`analyze_fdcp`**(), which is the conflict analysis procedure, does heuristic conflict-based learning based on incomplete analysis of the implication graph. If the Boolean decision procedure does not find a conflict, the procedure **`FME`**() is called for a final consistency check on the data-path. If **`FME`**() returns UNSAT, an analysis procedure called **`analyze_fme`**() is called to find a set of conflict causes. This corresponds to a subset of assignments on a fixed cut in the implication graph [7]. This subset is found by iteratively solving the data-path relation using FME. If the learned clause conflicts with the unique implications, the problem is UNSAT. If not, HDPLL backtracks non-chronologically and continues search.

As we have noted, the conflict analysis in earlier work was based on heuristics that did incomplete analysis of the implication graph. They used conflict-avoiding clauses with only Boolean literals, by setting a fixed cut in the implication graph, since they did not have a method for efficient UIP clause learning across domains.

In this work, we concentrate on improving the procedures **`analyze_fdcp`**() and **`analyze_fme`**() to enable learning true UIP clauses [9] by a complete analysis of the implication and resolution graphs. We shall show that this is more efficient than incomplete analysis. Next, we describe conflict-based learning across theories in HDPLL.

3. Hybrid Conflict-Based Learning

In this section, we describe the details of the new conflict-based learning in the combined Boolean and integer domain with FDCP and FME. We describe this as modifications to the procedures **`analyze_fdcp`**() and **`analyze_fme`**() in Figure 2. The fundamental idea behind the new learning procedure is that the implication graph from FDCP and the resolution graph from FME are combined into a single graph for conflict analysis.

3.1. Conflict Learning in HDPLL

We use FDCP to imply values both in the data-path and the control from any decision in the Boolean control. This enables us to construct an implication graph that can be used for conflict-based learning. The graph is implicitly represented by the stack of assignments made so far, similar to modern SAT solvers. The procedure **`analyze_fdcp`**() in the HDPLL algorithm (Figure 2), performs conflict analysis by implicitly tracing the implication graph to find the UIP as in standard Boolean conflict-based learning *e.g.*, [9].

However, there are some complications introduced by FDCP. Finite-domain constraints with word-level variables can have multiple, monotonically decreasing range changes. Therefore, a variable v_i, with a range $\langle l, m\rangle$, can appear with a different range $\langle j, k\rangle$, where, either $j < l$ or $k < m$, in the current implication graph. We refer to multiple appearances of word-variables in the implication graph as *variable incarnations*. The procedure keeps track of these incarnations efficiently by keeping a stack of evolving *variable states* of the variable. The variable state contains the implying constraint for each incarnation, the index of the decision at which it was implied (or *decision level*), and the incarnations of the variables in the implying constraint.

Let us take the example shown in Figure 3(a) with the proposition, $\{b_3 = 1\}$. The proposition is UNSAT. The implication graph for the example is shown in the Figure 3(b). We find by implication of the proposition, that $b_1 = 1$ and $b_2 = 1$, which produces conflicting ranges $w_5 = \langle 5, 6\rangle$ and $w_5 = \langle 3, 4\rangle$. We can see that a value assignment in the Boolean logic resulted in a conflict on a data-path variable. Now, if we trace through the implication graph from the conflict site, we find that the UIP is $(\neg b_3)$. Since this conflicts with the proposition, we deduce that it is UNSAT. The method in [7] would find the clause $(\neg b_1 \vee \neg b_2)$, and then find the clause $(\neg b_3)$, before deducing that the proposition is UNSAT.

If no conflict is detected during regular Boolean decision making and FDCP, we must do a final consistency check using FME as shown in Figure 2. If FME returns a conflict, we use an algorithm that extracts information from FME to find a conflict-avoiding learned clause. This is described next.

3.2. Conflict Based learning in FME

In this section, we describe a method for tracing the resolution graph from FME for conflict-based learning in HDPLL. Recall that FME is called when the Boolean theory and FDCP are consistent.

We can unite the implication graph from FDCP and the resolution graph of FME (see Section 2.3) by adding edges between the shared variables of the integer and Boolean theo-

ries (*e.g.*,Boolean outputs of comparators *etc*). This results in a combined *implication-resolution graph* $HG = (IG \cup RG \cup \{GLUE\})$. $\{GLUE\}$ is a set of edges added to connect IG and RG. A directed edge $gl \in GLUE$ exists between a Boolean variable assignment b_i in IG and constraint c_j in RG, if b_i enforces c_j to be **true**, and vice-versa. Intuitively, the set of constraints and assignments on an arbitrary cut in HG that covers all paths to the conflict, are the causes for the conflict. This motivates the idea that we can extract learned information from UNSAT calls to FME in HDPLL, by analysis of the unified implication-resolution graph.

Consider the example in Figure 3(a). Assume that we do not do any constraint propagation in the data-path, which gives us a partial implication graph with assignments, $b_1 = 1$, and $b_2 = 1$, shown in the top half of Figure 4. The data-path problem is passed to FME, which performs a series of variable elimination steps. This is represented as a resolution graph as shown in the lower half of Figure 4. The problem given to the FME procedure is the set of constraints marked $1,2,3,4$ in the figure. The edges from b_1 and b_2 to constraints 2 and 1 respectively are the additional edges, $\{GLUE\}$, added to unite the two graphs. In the example, FME eliminates variable w_2 from the constraints 3 and 4 to get constraint 5. When variable w_4 is eliminated by substituting $w_4 = w_5 + 2$, in constraint 1, we get a conflict since the resolvent is the empty constraint $-2 \geq 0$.

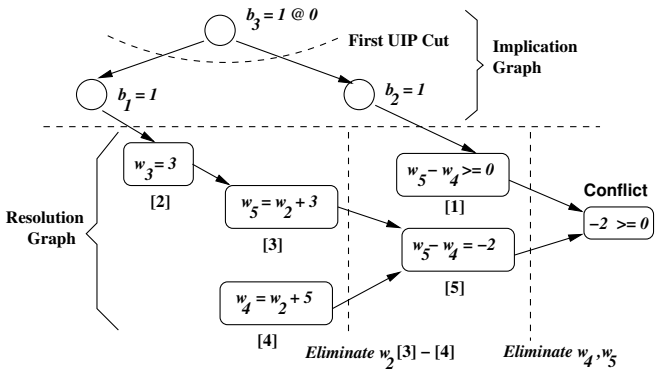


Figure 4. FME Resolution Graph on Figure 3(a).

The above method replaces the procedure **`analyze_fme`**() in Figure 2. We trace through the resolution graph from the conflict resolvent, to find the smallest set of assignments that are responsible for the generation of that conflict resolvent *i.e.,* a cut in HG. There may be other constraints on this cut (constraints with no incoming edges) that contribute to the conflict. These constraints enforce the functionality of data-path primitives. We do not need to add these constraints to the set of conflict causes, since these constraints always have to be **true** for the proposition to be **true**. If we trace through the resolution graph and continue the trace in the partial implication graph, we find the same UIP, $(\neg b_3)$.

4. Hybrid Learned Clauses

A conflict-based learning procedure can learn clauses of arbitrary length from an analysis of the implication graph. Therefore, some form of lazy implications becomes imperative. The authors of HDPLL [7] proposed an efficient method for determining when an implication is guaranteed to occur on an arithmetic inequality. However, determining the actual implied value involves applying complex rules on each variable in the inequality. This can be very expensive on large inequalities.

To our knowledge, there has been no published work on efficient two-literal watching for word-level arithmetic constraints, with the exception of CAMA [10]. CAMA's approach relies on enumeration of all values in a word-variable's domain. For example, an add operation, **a=b+c**, where **a**, **b**, and **c** are n bits each, leads to $O(3.(2^n-1)^2)$ multi-valued clauses Other arithmetic operators require in the order of $O(2^n-1)$ clauses. Our method uses an implicit representation of value sets, which makes it very efficient in practice.

4.1. Literal Watching

A conflict-avoiding constraint on word-level variables v_i, is a disjunct of word-literals with associated ranges $\bigvee_i b(v_i)$. This can be represented either as an FD constraint or as a hybrid clause (defined in Section 2). A word-literal in a hybrid clause can evaluate to **true**, **false** or **unknown**. The *containment* function, $contains(b_1, b_2)$, of a range b_1 on another range b_2 is a three-valued function defined as:

$$contains(b_1,b_2) = \begin{cases} \textbf{false} & b_1 \cap b_2 = \emptyset \\ \textbf{true} & b_1 \subseteq b_2 \\ \textbf{unknown} & (b_1 \cap b_2 \neq \emptyset) \wedge (b_1 \not\subseteq b_2) \end{cases}$$

Now it is quite straight-forward to see that we can use containment checks on the ranges of word-literals for lazy implications. Note that the basic idea is similar to that in Boolean SAT [12]. However, we use a pessimistic containment check on word-literals as compared to an exact check in CAMA.

FD constraints are more expressive than hybrid clauses, but evaluating FD constraints for new implications is very expensive as compared to hybrid clauses. For example, assume that a new value change has occurred on v_0 in the constraint $\sum_{i=0}^{n} a_i \cdot v_i \geq rhs$. We have to evaluate the relation, $a_j \cdot v_j \geq rhs - (\sum_{i=0}^{n} a_i \cdot v_i | i \neq j)$ for all variables v_j, excluding v_0. This means that the number of evaluations of the FD constraint per implication can depend both on the current ranges and number of variables. On the other hand, any implication of hybrid-clauses can be done with, at worst, $n-1$ containment checks. This motivates learning *hybrid clauses*.

An example of two-literal watching on hybrid clauses is shown in Figure 5. Assume that v_4 is a 3-bit word-level variable with initial domain $\langle 0,7 \rangle$, and v_1, v_2, v_3 are Boolean variables. Also, assume that the set of variable assignments, $\{v_4, v_1, v_2, v_3\} \mapsto \{\langle 3,4 \rangle, 1, 0, 0\}$ causes a conflict. The conflict-avoiding hybrid-clause is $(\neg v_1 \vee v_2 \vee v_3 \vee \neg v_4 \langle 3,4 \rangle)$. It is simpler to check inclusion than exclusion of ranges for the containment check. Therefore negative literals are split into a disjunct of word-literals *i.e.,* $\neg v_4 \langle 3,4 \rangle \equiv v_4 \langle 0,2 \rangle \vee v_4 \langle 5,7 \rangle$. The final learned clause, shown in Figure 5, is $(\neg v_1 \vee v_2 \vee v_3 \vee v_4 \langle 0,2 \rangle \vee v_4 \langle 5,7 \rangle)$.

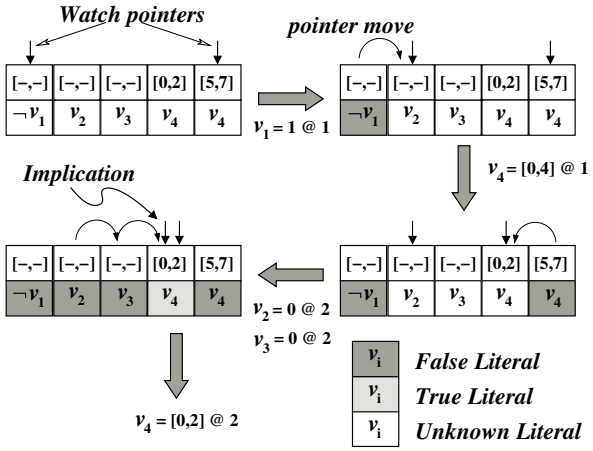


Figure 5. Hybrid clause two-literal watching

Figure 5 also shows an example of two-literal watching on a hybrid learned clause. Assume that the value changes, $v_1 = 1$ and $v_4 = \langle 0,4 \rangle$ occurred at decision level 1. The literal $\neg v_1$ becomes **false**. The word-literal $v_4 \langle 5,7 \rangle$ becomes **false** since $contains(\langle 0,4 \rangle, \langle 5,7 \rangle)$ is **false**. The pointer on each watched literal moves to the right and left respectively. Next, assume that the value changes $v_2 = 0$ and $v_3 = 0$ occurred at decision level 2. The pointer on the watched literal, v_2 moves to the right twice, since v_3 is also false. The two pointers meet at the literal $v_4 \langle 0,2 \rangle$, implying the assignment $v_4 = \langle 0,2 \rangle$.

5. Experimental Analysis

In this section, we discuss some experiments on the hybrid DPLL solver. The results of the comparison experiments are shown in the scatter plots in Figures 6 and 7. The *x* and *y* axis represent $\log_{10}$ run-times of each tool in CPU seconds. Points above (below) the diagonal correspond to examples where the tool on the *x* (*y*) axis is faster than the other. Each division above (below) the diagonal represents an order of magnitude speed-up (slow-down) for the *x* tool. Points on the right (top) grid line marked **`Tout`**, indicates the *x* (*y*) system timed-out at 1200 CPU seconds. **`Fail`**, indicates the *x* (*y*) system failed with a system error (**`seg-fault`**, *etc*).

5.1. Comparison with Heuristic Learning

In this experiment, we compare the new techniques with the heuristic conflict analysis in [7] on the same benchmarks in that paper. The results of this experiment are shown in the Figure 6. $HDPLL_{old}$ is HDPLL with the heuristic conflict-based learning in [7]. $HDPLL_{new}$ is HDPLL with the new UIP-based learning. Figure 6 shows that hybrid UIP learning is faster than the method in [7] by an order of magnitude on most cases. It completes on all cases where the old method aborts, in under 100 CPU seconds. Clearly, UIP learning in the combined implication-resolution graph is an efficient technique to prune the search space in HDPLL.

5.2. Comparison with state-of-the-art CDPs

We considered several tools that combine multiple decision theories including Boolean logic and some form of integer linear arithmetic for comparison with HDPLL:

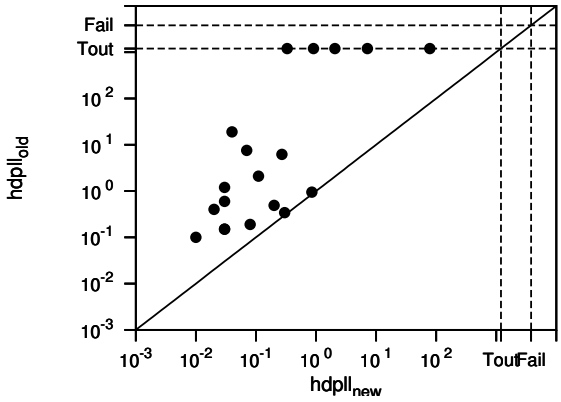


Figure 6. Hybrid UIP learning v/s Heuristic [7]

1. *CVC-Lite* [1] does not have a complete integer decision procedure. We used a real relaxation of the integer benchmarks only for CVC-Lite. Therefore, the results on SAT cases are reported as failures for CVC-Lite. We used CVC-Lite with the options – **`+sat fast`**.
2. *Integrated Canonizer and Solver*(ICS) [5] combined linear real arithmetic, bit-vectors, uninterpreted functions, and other theories. We used the default options for ICS.
3. *UCLID* [13] cannot handle general arithmetic. However, we ran it on a limited set of benchmarks with counter arithmetic. We used the options – **`sat 0 chaff`**.

All the comparison CDPS timed-out on most of the properties in the old-benchmark set. Therefore, we created new benchmarks for this experiment. Our benchmarks are 35 BMC models of safety properties on the RTL circuits, shown in Table 1. The number of properties per circuit is shown in Column 3. The maximum bound is shown in Column 4.

Table 1. Test-Case characteristics

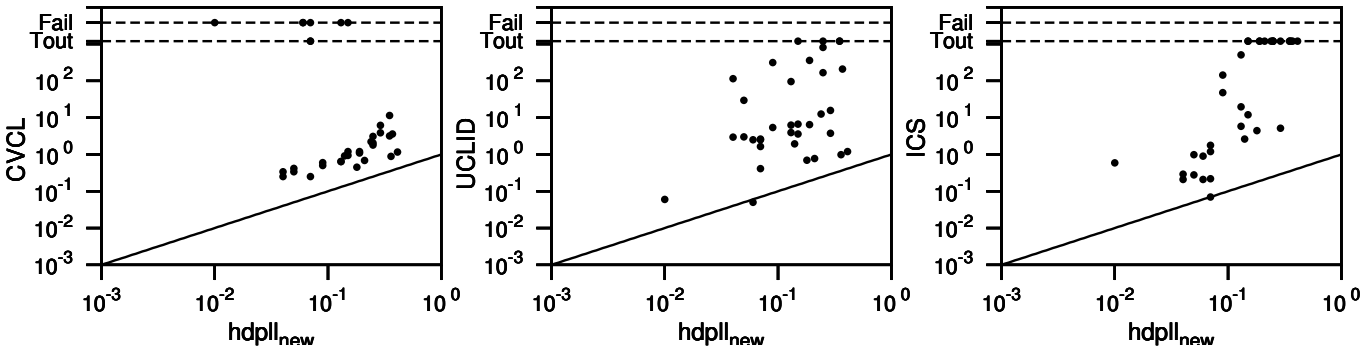
Ckt	Description of Circuit	# Props	Max. Bound
ser	compares serial data	1	20
BCD	BCD decoder	1	20
MM	min-max computation	1	20
Met	digital sensor interface	9	30

As we can see from Figures 7(a), 7(b) and 7(c), the hybrid UIP learning techniques in $HDPLL_{new}$ outperforms ICS, and UCLID on almost all cases, sometimes by several orders of magnitude. CVC-Lite performed best among the compared CDPS (Figure 7(a)), with an average of 70.12 CPU seconds per property and timed-out on only two cases. However, this is due to the real relaxation, which leads to false positives.

On comparing the CDP's with a complete integer decision procedure, we see that UCLID timed-out on 3 test-cases and took an average of 199.5 CPU seconds per property (Figure 7(b)). ICS timed-out on 16 out of 35 test-cases and took an average of 535.99 CPU seconds per property (Figure 7(c)). HDPLL finished all test-cases in less than 15 seconds.

5.3. Asymptotic Run-time comparisons

We ran a controlled experiment to compare the asymptotic run-time growth of HPDLL versus the other CDPs. We selected a safety property on the largest circuit Met, and



(a) HDPLL$_{new}$ v/s CVC-Lite (b) HDPLL$_{new}$ v/s UCLID (c) HDPLL$_{new}$ v/s ICS

Figure 7. Comparison of HDPLL$_{new}$ with CVC-Lite, ICS, and UCLID.

compared the run-time growth as the bound is increased for bounded model checking. The comparisons are accurate since the complexity of each instance increases linearly.

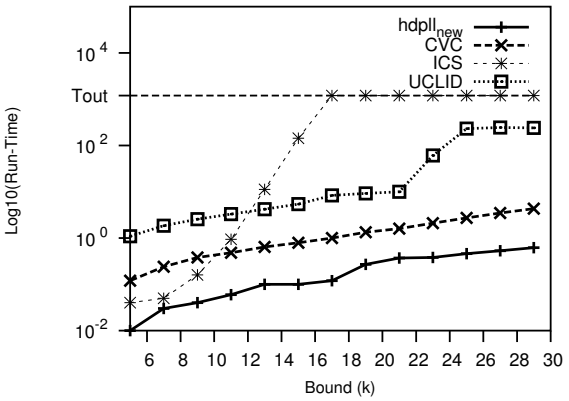


Figure 8. Asymptotic run-time comparisons.

The results of this experiment is shown in Figure 8. The x-axis shows the length of the bound and the y-axis shows the run-time in CPU seconds. As we can see the asymptotic run-time growth of HDPLL is significantly lower than that of all the other tools. CVC-Lite performed next best with a controlled growth similar to HDPLL. It is interesting that the integer decision procedures in HDPLL can compare well with a real relaxation of the problem. UCLID and ICS showed a sharp increase in run-time after bound k =21 and k =9 respectively. The results clearly show that the asymptotic run-time of HDPLL is considerably better than these tools.

6. Conclusions

We presented an efficient learning scheme for a DPLL-based constraint solver for RTL circuits. We also described compact data-structures for representing these learned relations that enable efficient constraint propagation. We demonstrated experimentally that these techniques make HDPLL more efficient than current state-of-the-art CDPs. A DPLL-style combination of decision theories seems to out-perform the other CDPs when efficient constraint propagation and conflict-based learning is used to prune the search space. In future, we shall extend the solver to other theories to handle arbitrary RTL circuits.

References

[1] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In *CAV*, pp. 500–504. July 2002.

[2] R. Brinkmann and R. Dreschler. RTL-Datapath Verification using Integer Linear Programming . In *VLSI Design*, pp. 741–746, 2001.

[3] C. Barrett, D. L. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *FMCAD*, pp. 187–201, 1996.

[4] G. Dantzig and B. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[5] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar ICS: Integrated Canonization and Solving. In *CAV* , pp. 246–249, 2001.

[6] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras and C. Tinelli.DPLL(T): Fast Decision Procedures In *CAV*, July 2004.

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

[8] J.N. Hooker. Logical Inference and Polyhedral Projection. In *Computer Scence Logic Conference*, pp. 184–200. 1992.

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

[10] C. Liu, A. Kuehlmann, and M. Moskewicz. CAMA: a multi-valued satisfiablity solver. In *ICCAD*, pp. 326 – 333, 2003.

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

[12] M. Moskiewicz, C. Madigan, *et. al*. Engineering an Efficient SAT Solver. In *38th DAC*, 2001.

[13] S. Seshia, S. Lahiri, and R. Bryant. Hybrid Sat-based Decision Procedure for Separation Logic with Uninterpreted Functions. In *DAC*, 425–430, 2003.

[14] W. Kelly, V. Maslow, *et. al*. The Omega Calculator and Library v1.1.0. Tech. report, Dept. of CS, UMCP, Nov. 1996.

[15] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A Unified Approach to RTL Satisfiability. In *DATE*, pp. 398–402, 2001.

[16] H. Zhang and M. Stickel. An Efficient Algorithm for Unit Propagation In *Int. Symp. on Artificial Intel. and Math.*, 1996.