Predicate Learning and Selective Theory Deduction for a Difference Logic Solver

Chao Wang Aarti Gupta Malay Ganai

NEC Laboratories America
4 Independence way, suite 200, Princeton, NJ 08540
{chaowang, agupta, malay}@nec-labs.com

Abstract

Design and verification of systems at the Register-Transfer (RT) or behavioral level require the ability to reason at higher levels of abstraction. Difference logic consists of an arbitrary Boolean combination of propositional variables and difference predicates and therefore provides an appropriate abstraction. In this paper, we present several new optimization techniques for efficiently deciding difference logic formulas. We use the lazy approach by combining a DPLL Boolean SAT procedure with a dedicated graph-based theory solver, which adds transitivity constraints among difference predicates on a "need-to" basis. Our new optimization techniques include flexible theory constraint propagation, selective theory deduction, and dynamic predicate learning. We have implemented these techniques in our lazy solver. We demonstrate the effectiveness of the proposed techniques on public benchmarks through a set of controlled experiments.

Categories and Subject Descriptors
B.6.3 [**Logic design**]: Design aids—*Verification*
General Terms: Verification, Algorithms
Keywords: Difference logic, decision procedure, SMT solver, SAT

1. Introduction

Difference logic, known also as separation logic, consists of standard Boolean connectives as well as difference predicates of the form $(v_i - v_j \leq c)$ where v_i, v_j are integer variables and c is an integer constant. In contrast to Boolean level modeling where integer variables are converted into bit-vectors, difference logic models systems at a higher level of abstraction. Difference logic is a subset of quantifier-free first order logic for which efficient decision procedures exist. It has been widely used in the automated verification of pipelined micro-processors [4] and real-time systems [12]. A satisfiability solver for this logic can also be used in software verification and sequential equivalence checking [15] between system-level models and RTL. Because of these applications, SAT Modulo Theory (SMT) solvers, including solvers for difference logic, have become an important research topic.

This paper is focused on the efficient implementation of a word-level solver for difference logic. We adopt a framework based on the so-called *lazy* approach [3, 8, 1, 10, 19, 16, 18, 26], which combines a DPLL [7] Boolean SAT procedure with a dedicated theory solver. Transitivity constraints among difference predicates, or *theory lemmas*, are derived by the theory solver and added on-demand. In contrast to this is the *eager* approach [4, 24, 22, 14, 25], which translates a difference logic formula directly into an equi-satisfiable Boolean formula, on which it applies the Boolean SAT solver. As has been demonstrated by several recent papers [16, 18, 26], the lazy approach is often better for deciding this particular logic. In a more recent paper [9], the strengths of both eager small-domain encoding and lazy approach have been combined to yield a robust performance over a wide set of benchmarks.

In the lazy approach, a Boolean SAT solver works on a Boolean abstraction of the original formula by replacing difference predicates with fresh Boolean variables. The theory solver receives assignments from the Boolean solver and performs theory propagation, including consistency check of the assignments and theory-based deduction. If a conflict occurs in the consistency check, it returns a blocking clause to the Boolean solver to rule out the particular assignment; the theory solver can also deduce new implications and feed them back to the Boolean solver.

We propose three new optimization techniques to improve the performance of a lazy solver. First, we make the invocation of theory propagation adaptive. In principle, theory propagation can be invoked at any stage of the Boolean constraint propagation (BCP). Calling theory propagation earlier can prevent the Boolean solver from exploring further an ultimately unsatisfying partial assignment through the detection of a theory conflict; however, too frequent theory solver calls may also incur a significant overhead. We propose a unified theory constraint propagation framework in which the timing of invocation can be changed easily, so that we can conduct controlled experiments to find the best overall setting. Our experiments show that overall it is desirable to call theory constraint propagation at every decision level after BCP finishes.

Second, we propose an efficient algorithm for theory deduction. In [18], exhaustive theory deduction has been used to derive all possible theory related implications whenever a predicate changes its value. Such an exhaustive approach can be costly since it requires each time a complete single-source shortest path enumeration. As shown in [26], the cost of a conventional shortest-path computation is often orders-of-magnitude more expensive than an incremental one. The authors of [26] choose to avoid theory deduction in order to minimize the overhead (instead, they try to make consistency check more efficient through incremental conflict detection). We will show that incremental conflict detection can be augmented with selectively deducing implications at no additional cost, therefore combining the aforementioned advantages of both approaches.

Third, we propose a new algorithm for dynamically learning new difference predicates. We note that existing lazy solvers can only learn new lemmas, i.e. relations among existing difference predicates, but not new predicates[1]. In many cases this becomes a seri-

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

[1] we have noticed that the word "predicate-learning" is overloaded. In [19],

ous limitation. In [24, 25], the authors used the *diamonds* formula to demonstrate the worst-case exponential blow-up in lazy solvers. To the best of our knowledge, all existing lazy solvers show exponential cost in dealing with diamonds. The reason is that their theory propagation can add lemmas on existing predicates, but not new difference predicates. The solutions provided in [24, 25] were based on eager approaches (*per constraint* and *small-domain* encoding, respectively); however, eager approaches have their own limitations. In this paper, we give a heuristic algorithm to automatically learn important new predicates. Our experiments show that the algorithm is often "smart" enough to pick up the right set of predicates and as a result, it solves the diamonds example in linear time. The identification of new predicates is similar in principle to automatic predicate abstraction techniques [2, 11, 13] which discover new predicates in order to refine a given abstraction.

The rest of the paper is organized as follows. We provide technical background in Section 2. The three proposed techniques will be presented in Sections 3, 4, and 5, respectively. We give our experiments results in Section 6 and then conclude in Section 7.

2. Preliminaries

We represent all difference predicates in the form $v_i - v_j \leq c$, where v_i and v_j are integer variable and c is a constant. Although the above representation uses $\leq$ only, predicates with other equality and inequality relations can be rewritten into the above form.

2.1 Boolean Skeleton

A difference logic formula can be abstracted into a pure Boolean skeleton by replacing all the difference predicates with new Boolean variables. The Boolean skeleton is an abstraction because transitivity constraints among predicates are removed. Therefore, the Boolean skeleton may have more satisfying assignments than the original formula. A (partial) assignment of the Boolean skeleton corresponds to a conjunction set of difference constraints; some of the predicates are set to true while others are set to false. If a solution (which is a set of constant values for the integer variables that satisfy all difference constraints) exists, the assignment is called *consistent*.

The consistency of a Boolean assignment can be checked by a dedicated theory solver on the constraint subgraph. A *constraint subgraph* $G(V, E)$ is a weighted directed graph whose nodes are the integer variables and whose edges are difference predicates or their negations. For example, if the predicate $(v_i - v_j \leq c)$ is set to true, it corresponds to an edge (v_j, v_i) with weight c. Similarly, the constraint $\neg(v_i - v_j \leq c)$, which is equivalent to $(v_j - v_i \leq -c-1)$, corresponds to an edge (v_i, v_j) with weight $(-c-1)$. An assignment is consistent if and only if the corresponding constraint subgraph does not have a negative weighted cycle.

If a negative weighted cycle exists, we can derive a transitivity constraint from the cycle and add it as a blocking clause to augment the Boolean skeleton. A lazy solver typically starts a DPLL search on the Boolean skeleton and propagates assignments to a theory solver for consistency check. It keeps augmenting the Boolean skeleton with transitivity constraints until either a consistent full assignment is found, or the augmented Boolean skeleton itself becomes unsatisfiable.

2.2 Theory Conflicts

Deciding the satisfiability of a difference logic formula in general is NP-complete [20]. Consistency check on a conjunction set of difference constraints requires polynomial time. A straightforward implementation of conflict detection based on the Bellman-Ford single-source shortest path algorithm [5] requires $O(|V|\,|E|)$ run time. Since consistency check needs to be called frequently during a DPLL search, it has been made incremental by several recent solvers [16, 6, 26]. The best complexity bound of incremental negative cycle detection [21] is $O(|V|\ log\ |V| + |E|)$.

Following the notation in [5], we represent the weight of edge (u, v) by $w[u, v]$ and the cost of node v by $d[v]$. We call the edge (u, v) *stable* if the costs of the two nodes satisfy $d[v] \leq d[u] + w[u, v]$. Edges that are not stable can be *relaxed* through the modification of costs of tail nodes, by assigning $d[v] = d[u] + w[u, v]$. Every time the cost of a node v changes by relaxation, we also record in field $\pi[v]$ the node responsible for that change—that is, $\pi[v] = u$ if the change is due to stabilizing the edge (u, v).

Without the presence of negative cycles, all edges can be made stable after a finite number of relax operations. The key to incremental conflict detection is making sure that all edges remain stable before and after every call to the theory solver (such a requirement can often be met easily [26]). If adding a new edge creates a negative weighted cycle, the cycle must go through the added edge. Therefore, an incremental conflict detection algorithm goes as follows: for every newly added edge, we check whether it is stable and if not, keep relaxing it and edges affected by relaxation until all edges become stable. If during this process the newly added edge needs relaxation again, we find a negative cycle. We can retrieve the cycle by following $\pi[v]$ fields, since node responsible for the last change to $d[v]$ has been recorded in $\pi[v]$,

Depending on how edges are ordered in the sequence of relax operations, incremental cycle detection may have different run time complexity bounds. (The best one is achieved by ordering nodes in a Fibonacci heap based priority queue.) It was reported in [26] that for deciding difference logic, an incremental algorithm on-average can have orders-of-magnitude reduction in the number of relax operations compared to a non-incremental one.

3. Theory Constraint Propagation

In principle, the Boolean SAT solver can invoke the theory solver at any stage of DPLL search to perform consistency check or theory deduction. Invoking the theory solver earlier prevents the Boolean solver from exploring further a partial assignment that is ultimately unsatisfying. On the other hand, too frequent invocations may cause a significant overhead since constraint propagation in the theory solver, although it has been significantly improved lately, is still more time-consuming than BCP. Whether an invocation scheme is good or not also depends on the type of the input formula. Suppose the formula is strongly constrained in the theory part but loosely constrained in the Boolean part, calling the theory solver as early as possible is better, because there is often a large number of satisfying assignments for the Boolean skeleton that are not satisfying for the entire formula. On the other hand, if the formula is strongly constrained in the Boolean part or does not even have a satisfying Boolean assignment, postponing theory propagation may avoid unnecessary overhead.

We have designed the theory propagation procedure in such a way that it can easily incorporate many different invocation settings. The procedure is given in Figure 1, where the pseudo code of `deduce` is a slight modification of standard BCP procedure of Boolean SAT solvers (without the last five lines of the `while` loop, it becomes standard BCP).

The function `time_for_theory_propagation()` returns true whenever the appropriate time has come for calling the theory solver – this is where we incorporate different invocation settings. The function `theory_detect_conflict()` implements an incremen-

for instance, it was used as a synonym of "lemma-learning."

```
deduce()
{
   while (implications.empty()) {
      set_var_value(implications.pop());
      if (detect_conflict())
         return CONFLICT;
      add_new_implications();
      if (time_for_theory_propagation()) {
         if (theory_detect_conflict())
            return CONFLICT;
         theory_add_new_implications();
      }
   }
   return NO_CONFLICT;
}
```

Figure 1: Adaptive constraint propagation (Boolean and theory).

tal negative cycle detection algorithm as in [26], which adds a blocking clause whenever a negative cycle is found. A blocking clause consists of the negation of all edges appearing in the negative cycle, which at this point of time is guaranteed to be evaluated to false (a conflicting clause). Suppose the cycle (A, B, C) is detected then the added clause is $(\neg A \vee \neg B \vee \neg C)$. The clause is used by the Boolean SAT solver later for conflict analysis and non-chronological backtracking. Theory deduction may be implemented in `theory_add_new_implications`, which derives and adds pure Boolean level implications to the queue `implications`.

Inside the function `time_for_theory_propagation()`, one can easily implement the following invocation schemes (and many others): (1) after the assignment of each predicate; (2) after BCP finishes at each decision level; and (3) after a full Boolean assignment. Making the function always return true will implement the per predicate assignment approach, making it return true whenever the implication queue is empty will implement the per decision level approach, and making it return true when the implication queue is empty and at the same time the decision level is 0 will implement the per full assignment approach.

Since the cost of theory constraint propagation is often higher than Boolean constraint propagation, if a certain search subspace can be blocked by conflicts of both types, it is better to block it by Boolean conflicts rather than by theory conflicts. It then follows that the per predicate assignment approach is not particularly interesting, since under this setting a blocking clause added by the theory solver may not prevent the same future theory conflict. We illustrate this using an example.

Example 1 Consider the formula fragment

$$(a \vee X) \wedge (a \vee Y) \wedge (a \vee b) \wedge (\neg X \vee c) \wedge ... \wedge (\neg X \vee \neg Y) \ ,$$

where $(\neg X \vee \neg Y)$ is a blocking clause learned from a previous call to theory solver (i.e., predicates X and Y cannot be true simultaneously). A BCP procedure, such as the one in Chaff [17], may produce the following execution sequence:

set_var_value	detect_conflict	implications
$(\neg a)$	( )	$\{X, Y, b\}$
(X)	( )	$\{Y, b, c, \ldots, \neg Y\}$
(Y)	(*)	$\{b, c, \ldots, \neg Y, \neg X\}$
(b)	( )	$\{c, \ldots, \neg Y, \neg X\}$
...		
$(\neg Y)$	$(Y, \neg Y)$	

Note that Chaff reports a conflict only when `set_var_value` tries to set a previously assigned variable to a different value, and the above assignments will be prevented at the end of BCP due to $(\neg X \vee \neg Y)$. However, under the per predicate assignment setting, the theory solver is called right after setting (Y), which will discover the same conflict and add a learned clause $(\neg X \vee \neg Y)$. More importantly, this clause no longer acts as a blocking clause—this is disadvantageous because conflict clause based learning is key to the performance of modern SAT solvers.

To make sure that BCP fires first whenever a conflict can be detected by both, one needs to wait at least until BCP finishes at each decision level before calling the theory solver. We will give experimental comparison of the three settings in Section 6.

4. Selective Theory Deduction

An important part of theory propagation[2] is to deduce new implications, i.e. implied values of some unassigned predicates. Note that the Boolean counterpart in BCP uses the unit-literal rule. Theory deduction, on the other hand, relies on the creation of negative cycles in the constraint subgraph: the predicate p is implied if adding an edge for its opposite literal $\neg p$ to the graph creates a negative cycle.

Exhaustive theory deduction was used in DPLL(T) [18] to derive all possible theory implications under the current assignment. Such deduction is carried out whenever a new edge is added. For instance, after adding edge (x, y), they enumerate all the shortest paths $(y \rightarrow y_j)$ from the node y, as well as all the shortest paths $(x_i \rightarrow x)$ to the node x; they find sets $\{x_i\}$ and $\{y_j\}$ by calling the Bellman-Ford single-source shortest path algorithm twice. Then, for every un-assigned predicate $p : (y_j - x_i \leq c)$ such that $d[y_j] - d[x_i] \leq c$, they add (p) as an implication; note that setting p to false would create a negative cycle $(x_i \rightarrow x, y \rightarrow y_j, x_i)$. Exhaustive theory deduction can remove the burden on theory conflict detection since future conflicts are prevented by theory based implications. However, such an exhaustive approach may be costly.

Another extreme approach was adopted in [26], where the authors chose to make conflict detection efficient but not to do theory deduction at all. In Figure 1, their approach corresponds to the use of `theory_detect_conflict()` in full strength but not doing anything inside `theory_add_new_implications()`. Their approach has the advantage of relaxing edges incrementally and backtracking with zero cost in the theory solver, and according to [26], the number of relaxed edges in their theory solver is often orders-of-magnitude less than that of a full-blown shortest path computation.

We show in the following that the benefits of the aforementioned two approaches can be combined. In particular, incremental cycle detection can be augmented with the capability of performing selective theory deduction, and at no additional cost to the incremental algorithm in [26]. The key is to reuse existing cost values of the nodes computed during incremental negative cycle detection. Recall that in the incremental algorithm, all edges are made stable before and after the calls to theory constraint propagation, and all nodes changed by the newly added edge through relaxation are recorded in the $\pi[v]$ field. When the conflict detection routine returns no conflict, we have a set $\{y'_j\}$ of nodes that have just been relaxed in `theory_detect_conflict()` — this is because each time an edge is relaxed, we can mark its tail node as an element of this subset. Note that $\{y'_j\}$ is a subset of the set $\{y_j\}$ in [18] for exhaustive theory propagation, since the latter may also include tail nodes of some already stable edges. Similarly, we can get a subset

[2]The word "theory propagation" is also overloaded. We take it to mean a combination of consistency check and theory deduction. However, [18] used it as a synonym of "theory deduction."

$\{x'_i\}$ of the set $\{x_i\}$ in [18], by following $\pi[x]$ backward from x — this is a subset of nodes on a shortest path to x.

In contrast to the high overhead of exhaustive theory propagation, finding $\{x'_i\}$ and $\{y'_j\}$ requires no additional relax operation other than those already in the incremental cycle detection procedure. Next, we search for all unassigned predicates $p : (y'_j - x'_i \leq c)$ such that $d[y'_j] - d[x'_i] \leq c$, and add (p) to the queue `implications`.

When an implication is added to `implications`, we need to record an explanation (i.e. what implies it) so that later it can be used just like an implication produced by BCP during the standard conflict analysis [23]. The explanation is the *would-be* negative cycle, which can be recorded either inside the constraint graph or as a new clause added into the Boolean skeleton. The new clause consists of the negation of all edges appearing in the negative cycle. Adding new clauses to the Boolean skeleton allows BCP to derive the same implication in the future without running theory deduction again, which is advantageous since BCP is often much faster than graph manipulation. However, adding too many such clauses can also slow down BCP. An alternative is holding the explanation in the graph and adding it as a clause only when it is actually used during conflict analysis. Our experience shows that holding explanations outside the Boolean skeleton (until needed) is a better choice for most examples.

We have implemented two different versions of selective theory deduction:

- *Forward deduction*: finding only implications in the form of $(y'_j - x \leq c)$. In other words, let $\{x'_i\}$ be $\{x\}$ to reduce the overhead.
- *Both directions*: finding all implications in the form of $(y'_j - x'_i \leq c)$.

The *no-deduction* approach in [26] can be incorporated as well by assuming that both $\{x'_i\}$ and $\{y'_j\}$ are empty sets. The set of implications that can be derived in selective theory deduction is a subset of the implications derived in exhaustive theory deduction, but this is done with much less overhead. We will give experimental comparisons of the two approaches and that of [26] in Section 6; the results suggest that overall *forward-deduction* is the best in balancing between gain and overhead.

5. Dynamic Predicate Learning

A fundamental problem of existing lazy solvers is that they build the Boolean skeleton only with predicates in the original formula. In other words, any predicate not in the original formula does not appear in the initial Boolean skeleton and will never be added later. Inside their theory propagation procedures, transitivity lemmas and deduced implications must be expressed in terms of the original set of predicates. This restriction, however, may prevent the theory solver from returning concise lemmas. In [24, 25], the authors used the *diamonds* formula to illustrate the worst case exponential blow-up of solvers based on the lazy approach. A diamonds formula with $O(n)$ nodes in its constraint graph, as shown in Figure 2, has $O(2^n)$ negative weighted cycles. To our best knowledge, all existing solvers based on the lazy approaches, including the most recent ones [16, 18, 26], show exponential cost in dealing with diamonds.

Eager approaches such as [24, 25] work much better in this case, by using a *per constraint* or *small-domain* encoding scheme to build an equi-satisfiable Boolean formula. In [24], for instance, the total number of chordal edges needed for encoding the Boolean formula is polynomial (although the number is often large for an arbitrary difference logic formula). In this section, we propose a method to make the lazy solver perform well in such cases.

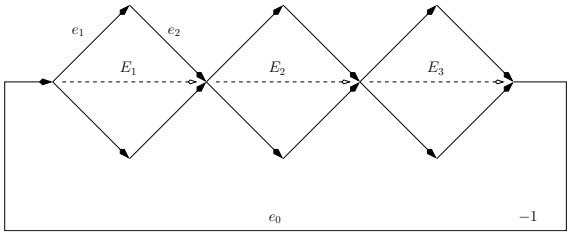

Figure 2: The diamonds example.

The key observation is that the number of transitivity constraints can be significantly reduced if we are allowed to use new predicates. For example, the dotted lines E_i in Figure 2 are not original predicates, but can be added due to transitivity constraints like $(e_1 \wedge e_2 \rightarrow E_1)$. Suppose, for instance, the conflict due to the shortest negative cycle (e_0, E_1, E_2, E_3) is detected at some point of time and the blocking clause $(\neg e_0 \vee \neg E_1 \vee \neg E_2 \vee \neg E_3)$ is added to the Boolean skeleton, then the number of additional blocking clauses needed to solve the diamond formula will become polynomial. This is because any conflict corresponding to a longer negative cycle also makes this clause false, and therefore is blocked during BCP. As a result, assignments corresponding to longer negative cycles will never be presented to the theory solver.

We propose a method to add new short-cut predicates lazily. An important feature of our method is that we do not immediately associate any transitivity constraint (e.g., $e_1 \wedge e_2 \rightarrow E_1$) with the new predicates. Conceptually, we add for each new predicate E_i a redundant clause $(E_i \vee \neg E_i)$. Since these clauses are redundant, the Boolean skeleton remains an abstraction. However, because of the connections between new and old predicates, lemmas on new predicates will be added automatically in the DPLL search—suppose assignments to some new predicates conflict with the rest of the formula, these new predicates will appear in a blocking clause whereas irrelevant new predicates will stay unused. This is in contrast to the eager encoding approach in [24], which adds transitivity lemmas over all possible short-cuts regardless of whether they will be useful to the satisfiability check. There is a clear advantage of our dynamic predicate learning approach, since the number of short-cuts in the graph is typically large and there is no easy way of knowing before-hand which one is useful.

To reduce overhead, we do not consider all short-cuts of a graph as new predicates. Our algorithm in selecting candidate short-cuts is based how many times their head and tail nodes show up in negative cycles and whether the nodes are re-convergence points in the graph—we are interested in nodes that appear frequently in the negative cycles and at the same time are re-convergence points. Our procedure goes as follows: we associate a counter to each node of the graph and increase the counter every time the node appears in a theory conflict. As time goes by, the counter values will keep increasing. If the nodes x and y appear in the current negative cycle, both have high in-degree or out-degree, and their counter values exceed an empirical threshold, we will allocate a new Boolean variable $E : x - y \leq (d[x] - d[y])$. We also add to the Boolean skeleton a redundant clause $(E \vee \neg E)$. The intuition behind our node selection algorithm is that re-convergence points of the constraint subgraph are the fundamental reason for a potentially exponential number of negative cycles, and a node frequently appearing in the negative cycle may be part of an ongoing blow-up.

A problem arises when we implement dynamic predicate learning in a DPLL based Boolean SAT solver. Most existing solvers, including Chaff [17], do not support dynamically adding new Boolean

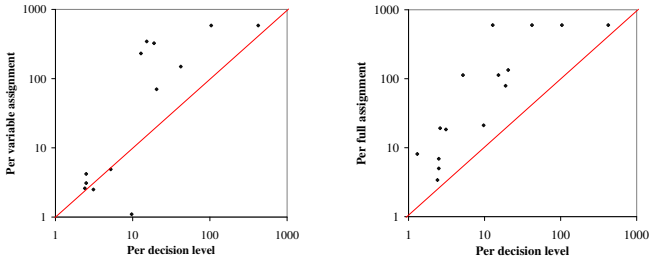

Figure 3: Different invoking settings for theory propagation: per predicate assignment, per decision level, and per full assignment.

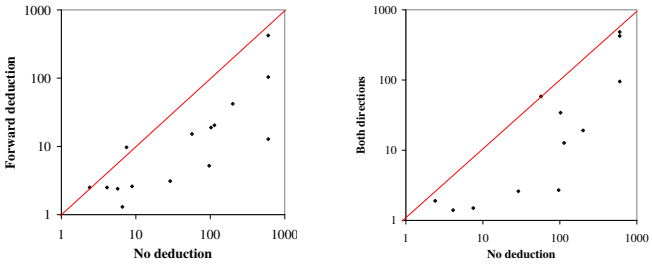

Figure 4: Comparison of different theory deduction schemes: no-deduction, forward-deduction, both-directions.

variables. We circumvent this problem by pre-allocating a fixed number of new Boolean variables but do not associate them with any new predicate (or short-cut). The binding between a pre-allocated Boolean variable and a short-cut predicate in the graph comes later during the solving process.

We also want to avoid adding too many such predicates. Since the goal is to avoid the potentially exponential number of negative cycles, we only add new predicates for nodes that appear very often in negative cycles. In our current implementation, we set the frequency threshold for short-cuts to 200. That is, we add new predicates between x and y only when both appear in the negative cycles more than 200 times, and only when both nodes have more than one incoming/outgoing edges. We have found that this empirical setting works very well in practice.

6. Experiments

We have implemented the proposed techniques on top of the SAT solver Chaff [17] and our own graph manipulation procedure. During the implementation, we have tried to optimize the performance of the solver whenever possible. Our base-line comparison [26], which has incremental cycle detection but none of the techniques proposed in this paper, already has a better runtime performance than the most recent lazy solvers [16, 18] and is considerably better than the eager solver in UCLID [22] on a large set of public benchmark formulas. We have conducted controlled experiments to evaluate the effectiveness of the three proposed techniques. All the experiments were run on a workstation with 3.0 GHz Intel Pentium 4 processor and 2 GB of RAM running Red Hat Linux 7.2. We set the memory limit to 1 GB and the time limit for each formula to 600 seconds.

6.1 Efficient Theory Constraint Propagation

First, we give experimental results on the different settings for invoking theory constraint propagation. The benchmark formulas belongs to the DTP suite [18] that are randomly generated to cover a wide range of different scenarios. The results are given in Figure 3, where we compare the CPU time for different settings. The scatter plot on the left-hand side compares the solver that invokes theory propagation per predicate assignment to the solver that invokes theory propagation per decision level after BCP finishes. The scatter plot on the right-hand side compares the solver that invokes propagation per full assignment to the solver that invokes propagation per decision level after BCP finishes. Points above the diagonal lines are winning cases for the per decision level setting.

On these benchmark formulas, the best performance is achieved by invoking theory constraint propagation after BCP finishes at each decision level. Note that in terms of the timing for invoking theory constraint propagation, it is in the middle of the three settings. It is also possible to devise a heuristic algorithm to automatically adapt the time interval between consecutive invocations of the theory solver.

6.2 Selective Theory Deduction

We now give the experimental comparison of the three theory deduction schemes, i.e. *no-deduction*, *forward-deduction*, and *both-directions*. The results are given as scatter plots in Figure 4, where we compare the CPU Time for different invocation schemes. The scatter plot on the left-hand side compares the solver with forward-deduction scheme to the one without theory deduction; the scatter plot on the right-hand side compares the solver with both-direction scheme to the one without theory deduction. Note that in both plots, points above the diagonal lines are the losing cases for the one without theory deduction.

The direct comparison of the two selective deduction schemes are not provided in the scatter plots, but can be made in the total CPU time for running all benchmarks. The time is 2438 seconds for the no-deduction scheme, 666 seconds for the forward-deduction scheme, and 1138 seconds for the both-directions scheme. Again, the best setting is the one in the middle. The results show that selective theory deduction can significantly improve the run time performance of the solver. However, the fact that *forward-deduction* is faster than *both-directions* suggests that too much effort in theory deduction may incur a significant amount of overhead.

6.3 Dynamic Predicate Learning

Finally, we give the evaluation of our dynamic predicate learning algorithm. The first test of its effectiveness was conducted on a set of parameterized diamonds examples. The results are given in Figure 5 where for each formula in the x-axis, the run time of the solver with and without predicate learning is compared. As expected, the solver without predicate learning demonstrates an exponential trend in run time, whereas the one augmented with dynamic predicate learning remains linear in run time. This indicates that our learning algorithm can automatically identify important new predicates.

We also evaluated the algorithm on the DTP examples and observed a significant performance improvement. The results are given in Figure 6 where points under the diagonal line are winning cases for the solver with dynamic predicate learning. The total CPU time for completing all benchmarks is 437 seconds with dynamic predicate learning, and 666 seconds without it. Furthermore, the performance improvement brought by dynamic predicate learning is very consistent. Note that the DTP formulas do not have any specific structure in the constraint subgraphs, indicating that this algorithm is effective for formulas of general types.

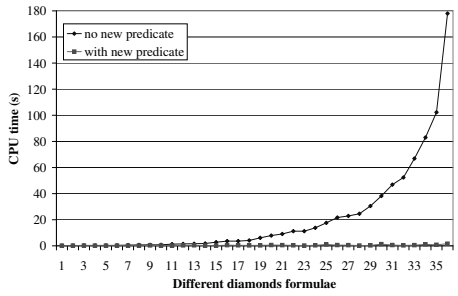

Figure 5: Dynamic predicate learning: on diamonds.

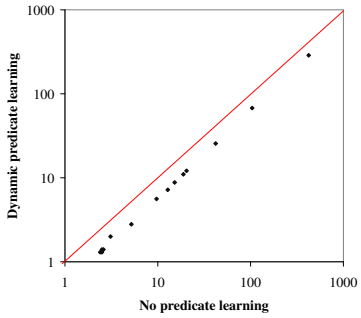

Figure 6: Dynamic predicate learning: on DTP formulas.

7. Conclusions

We have presented three new techniques for efficiently deciding difference logic formulas, which includes flexible theory constraint propagation, selective theory deduction, and dynamic predicate learning. Theses techniques can be easily implemented in the DPLL framework of many modern Boolean SAT solvers. Our experiments show that the proposed techniques can significantly improve the run time performance of the solver. We believe that the same techniques can be applied to SMT solvers for other logic as well. For future work, we want to investigate the possibility of applying high-level model information to guide the satisfiability check inside the solver, especially in the context of electronic system level design and verification of embedded systems.

References

[1] A. Armando, C. Castellini, E. Giunchiglia, and M. Maratea. A SAT-based decision procedure for the boolean combination of difference constraints. In *Theory and Applications of Satisfiability Testing (SAT'04)*, pages 166–173, Vancouver, CA, May 2004.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI'01)*, Snowbird, UT, June 2001.

[3] C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer Aided Design*. Springer, Nov. 1996. LNCS 1166.

[4] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer-Aided Verification (CAV'02)*. Springer, July 2002. LNCS 2404.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] S. Cotton. Satisfiability checking with difference constraints. Msc thesis, IMPRS Computer Science, Saarbrucken, 2005.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[8] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Computer Aided Verification (CAV'01)*, pages 246–249. Springer, 2001. LNCS 2102.

[9] M. K. Ganai, M. Talupur, and A. Gupta. SDSAT: Tight integration of small domain encoding and lazy approaches in a separation logic solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 135–150. Springer, 2006. LNCS 3920.

[10] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer-Aided Verification (CAV'04)*, pages 175–188. Springer, July 2004. LNCS 3114.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of programming languages (POPL'02)*, pages 58–70, 2002.

[12] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *IEEE International Symposium on Logic in Computer Science*, pages 394–406, 1992.

[13] H. Jain, F. Ivančić, A. Gupta, and M. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for the Construction and Abnalysis of Systems (TACAS'05)*, pages 394–409. Springer, 2005. LNCS 3440.

[14] S. Lahiri and S. Seshia. The UCLID decision procedure. In *Computer Aided Verification (CAV'04)*, pages 475–478. Springer, 2004. LNCS 3114.

[15] Y. Lu, A. Koelbl, and A. Mathur. Formal equivalence checking between system-level models and RTL. In *International Conference on Computer-Aided Design (ICCAD'05)*, 2005. Embedded tutorial.

[16] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 317–333. Springer, 2005. LNCS 3440.

[17] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[18] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Computer Aided Verification (CAV'05)*, pages 321–334. Springer, 2005. LNCS 3576.

[19] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and F. Brewer. Structural search for rtl with predicate learning. In *Proceedings of the Design Automation Conference (DAC'05)*, pages 451–456, Anaheim, CA, June 2005.

[20] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):275–293, Oct. 2002.

[21] G. Ramalingam, J. Song, L. Joscovicz, and R. Miller. Solving difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.

[22] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the Design Automation Conference*, pages 425–430, Anaheim, CA, June 2003.

[23] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[24] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Computer-Aided Verification (CAV'02)*, pages 209–222. Springer, July 2002. LNCS 2404.

[25] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *Computer-Aided Verification (CAV'04)*, pages 148–161. Springer, July 2004. LNCS 3114.

[26] C. Wang, F. Ivančić, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR'05)*, pages 322–336. Springer, 2005. LNCS 3835.