DPLL(T): Fast Decision Procedures

Harald Ganzinger[1], George Hagen[2], Robert Nieuwenhuis[3], Albert Oliveras[3], and Cesare Tinelli[2]

[1] MPI für Informatik, Saarbrücken, Germany
www.mpi-sb.mpg.de/~hg
[2] Dept. of Computer Science, University of Iowa*
www.cs.uiowa.edu/~{ghagen,tinelli}
[3] Tech. Univ. of Catalonia, Barcelona**
www.lsi.upc.es/~{roberto,oliveras}

Abstract. The logic of equality with uninterpreted functions (EUF) and its extensions have been widely applied to processor verification, by means of a large variety of progressively more sophisticated (*lazy* or *eager*) translations into propositional SAT. Here we propose a new approach, namely a general DPLL(X) engine, whose parameter X can be instantiated with a specialized solver $Solver_T$ for a given theory T, thus producing a system DPLL(T). We describe this DPLL(T) scheme, the interface between DPLL(X) and $Solver_T$, the architecture of DPLL(X), and our solver for EUF, which includes incremental and backtrackable congruence closure algorithms for dealing with the built-in equality and the integer successor and predecessor symbols. Experiments with a first implementation indicate that our technique already outperforms the previous methods on most benchmarks, and scales up very well.

1 Introduction

The logic of equality with uninterpreted functions (EUF) [BD94] and its extensions has been widely used for processor verification (see, e.g., [BD94,BGV01] [BLS02b,VB03]).

For deciding validity – or, dually, unsatisfiability – of formulas in this kind of logics, during the last five years many successively more sophisticated techniques have been developed, most of which can be classified as being *eager* or *lazy*. In the eager approaches the input formula is translated, in a single satisfiability-preserving step, into a propositional CNF, which is checked by a SAT solver for satisfiability. The lazy approaches [ACG00,dMR02,ABC+02,BDS02,FJOS03] instead abstract each atom of the input formula by a distinct propositional variable, use a SAT solver to find a propositional model of the formula, and then check that model against the theory. Models that are incompatible with

* Work partially supported by NSF Grant No. 237422.
** Work partially supported by the Spanish CICYT project Maverish ref. TIC2001-2476 and by a FPU grant, ref. AP2002-3533, from the Spanish Secretaría de Estado de Educación y Universidades.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 175–188, 2004.

the theory are discarded from later consideration by adding a proper *lemma* to the original formula. This process is repeated until a model compatible with the theory is found or all possible propositional models have been explored. Also less lazy variants exist (e.g., in CVC, [BDS02]), in which (partial) propositional models are checked incrementally against the theory while they are built by the SAT solver; however, from partial models that are consistent with the theory no information is derived, contrary to what is proposed here. The main advantage of such lazy approaches is their flexibility, since they can relatively easily combine new decision procedures for different logics with existing SAT solvers.

For the logic of EUF the eager approaches are in general faster than the lazy ones, probably because the theory information is used to *prune* the search, rather than to validate it *a posteriori*. Among the eager approaches for EUF, two different encodings into propositional logic are at present predominant. The first one, known as the EIJ (or *per-constraint*) encoding, abstracts each equality atom $t_i{=}t_j$ by a propositional variable e_{ij} and takes care of the equality by imposing additional transitivity constraints [BV02,SSB02]. A drawback of this technique is that when the number of transitivity constraints is too large, the exponential blowup in the formula may make it too hard for the SAT-solver. The second one, known as the small domain (SD) encoding [PRSS99,BLS02b], is essentially an $O(n\ log\ n)$-size encoding that gets transitivity for free, at the expense of a certain loss of structure, by translating equalities into formulas instead of into single propositional variables. In order to get the best of both encodings, two different *hybrid* approaches were presented in [BLS02a] and [SLB03], based on an analysis of the input problem that estimates properties such as the number of transitivity constraints.

In this paper we introduce the first (to our knowledge) technique that is not based on such lazy or eager translations into SAT. Building on independent previous work by some of the authors [Tin02,NO03], we propose a new approach based on a general engine for propositional solving, DPLL(X), parametrized by a solver for a theory of interest. A system DPLL(T) for deciding the satisfiability of CNF formulas in a theory T is produced by instantiating the parameter X with a module $Solver_T$ that can handle conjunctions of literals in T. For instance, in the case of the pure EUF logic, T is just the theory of equality.

The basic idea is similar to the $CLP(X)$ scheme for constraint logic programming: provide a clean and modular, but at the same time efficient, integration of specialized theory solvers within a general purpose engine, in our case one based on the Davis-Putnam-Logemann-Loveland procedure [DP60,DLL62]. In [Tin02] a DPLL(T) scheme was already given in a more high-level abstract form, as a sequent-style calculus. Although no detailed interface was defined there between the DPLL(X) engine and the theory solver, several optimization strategies were already discussed. The framework given here can be seen as a concrete realization of the calculus in [Tin02], except that, contrary to [Tin02], we do not expect $Solver_T$ to give always complete answers. Relaxing that requirement does not affect completeness, but turns out to be crucial for efficiency, at least in the EUF case. A DPLL(X) scheme was introduced and informally described in [NO03].

There, however, emphasis was placed on the development of new congruence closure algorithms to be used in the theory solver for the EUF logic.

The concrete DPLL(T) scheme and its architecture and implementation presented here combine the advantages of the eager and lazy approaches. On the one hand, experiments reveal that, as soon as the theory predicates start playing a significant role in the formula, our initial implementation of DPLL(T) already outperforms all other approaches. On the other hand, our approach is similar in flexibility to the lazy approaches: more general logics can be dealt with by simply plugging in other solvers into our general DPLL(X) engine, provided that these solvers conform to a minimal interface, described later.

This paper is structured as follows. In Section 2 we describe in detail the DPLL(T) scheme and the kind of logics it can be applied to, discussing the advantages and disadvantages of the small interface. The architecture of our current DPLL(X) implementation is given in Section 3. Section 4 describes our solver for EUF, which includes incremental congruence closure algorithms that deal with the built-in equality and the integer successor and predecessor symbols, as well as with backtracking. Finally, Section 5 gives experimental results of our preliminary implementation of DPLL(T) for EUF, and Section 6 concludes and outlines a number of promising research directions for future work, in particular for instantiating the DPLL(X) engine with solvers for other theories.

2 From DPLL to DPLL(T)

In this section we describe the main characteristics of the DPLL(T) scheme. Any DPLL(T) system consists of two parts: the global DPLL(X) module and a solver $Solver_T$ for the given theory T. The DPLL(X) part is a general DPLL engine that is independent of any particular theory T. Here we will use as examples three possible instances for the theory part T: the ones for propositional logic, pure EUF, and EUF with successors and predecessors.

2.1 The Logics under Consideration

In this paper we will consider the satisfiability problem of formulas in CNF, that is, of a given set S (a conjunction) of clauses. By a clause we mean a disjunction of literals, each literal l being of the form A or $\neg A$, where A is an atom drawn from a set $\mathcal{A}$. What $\mathcal{A}$ is depends on the theory T under consideration.

An *interpretation* I is a function $I\colon \mathcal{A} \to \{0, 1\}$. We write $I \models l$ if the literal l *is true in* I, that is, if l is a positive atom A with $I(A) = 1$ or l is a negated atom $\neg A$ with $I(A) = 0$. For each theory T, we will consider only T*-interpretations*, that is, interpretations that *agree* with the axioms of T, in a sense that will be made precise below for the theories considered here. An interpretation (resp. T-interpretation) I is a *model* (resp. T*-model*) of a clause set S if in each clause of S there is at least one literal l that is true in I. The aim of this work is to design algorithms for deciding whether a clause set S has a T-model, and exhibiting such a model whenever it exists. For all logics under consideration in this paper, this problem is easily shown to be NP-complete.

Propositional Logic. The atoms are just propositional symbols of a set $\mathcal{P}$, and the interpretations under consideration are unrestricted, i.e, any truth assignment $I\colon \mathcal{P} \to \{0,1\}$ is admitted: the theory T is empty in this case.

Pure EUF. An atom is either of the form $P(t_1,\ldots,t_n)$ where P is an n-ary symbol of a set of fixed-arity predicate symbols $\mathcal{P}$, or an equation of the form $s{=}t$, where s, t and $t_1,\ldots,t_n$ are terms built over a set of fixed-arity function symbols. All constants (0-ary symbols) are terms, and $f(s_1,\ldots,s_n)$ is a term whenever f is a non-constant n-ary symbol and $s_1,\ldots,s_n$ are terms[1]. In the following, lowercase (possibly indexed) s, t, and u always denote terms, and literals $\neg s{=}t$ are usually written as $s{\neq}t$. In pure EUF the theory T expresses that '=' is a congruence, i.e., it is reflexive (R), symmetric (S), transitive (T), and monotonic (M); hence the only admissible interpretations I are the ones satisfying the conditions below for all terms s, t, s_i, and t_i:

R: $I \models s{=}s$
S: $I \models s{=}t$ if $I \models t{=}s$
T: $I \models s{=}t$ if $I \models s{=}u$ and $I \models u{=}t$ for some term u
M1: $I \models f(s_1 \ldots s_n){=}f(t_1 \ldots t_n)$ if $I \models s_i{=}t_i$ for all i in $1..n$
M2: $I \models P(s_1 \ldots s_n)$ if $I \models P(t_1 \ldots t_n)$ and $I \models s_i{=}t_i$ for all i in $1..n$

EUF with Successors and Predecessors. This subset of the CLU logic from [BLS02b] is the extension of EUF with two distinguished unary function symbols, *Succ* and *Pred*. Besides the congruence axioms for '=', the interpretations I must also satisfy the following for all terms t:

$$I \models \mathit{Succ}(\mathit{Pred}(t)){=}t \qquad I \models \mathit{Pred}(\mathit{Succ}(t)){=}t \qquad \forall\, n{>}0,\ I \models \mathit{Succ}^n(t){\neq}t$$

where $\mathit{Succ}^n(t)$ denotes the term $\mathit{Succ}(\ldots \mathit{Succ}(t)\ldots)$ headed with n *Succ* symbols applied to t; hence the last axiom scheme denotes an infinite set of axioms. Note that $I \models \mathit{Pred}^n(t){\neq}t$ is a consequence of the above.

2.2 DPLL(X): The General DPLL Part of DPLL(T)

The DPLL(X) part of a DPLL(T) system is the one that does not depend on the concrete theory T. It can be like any DPLL procedure, with basically all its usual features such as heuristics for selecting the next decision literal, unit propagation procedures, conflict analysis and clause learning, or its policy for doing restarts. Like in the propositional case, it always considers atoms as purely syntactic objects. The only substantial difference with a standard propositional DPLL procedure is that the DPLL(X) engine relies on a *theory solver* for T, denoted here by Solver_T, for managing all information about the current interpretation I.

[1] Here we do not consider any *if-then-else* constructs, since they do not increase the expressive power of the logics and are eliminated, in linear time, in a structure-preserving preprocessing phase, which moreover preserves conjunctive normal forms. Each occurrence of a (sub)term *if-then-else*(F,s,t) is replaced by a new constant symbol v, and $(\neg F \vee v = s) \ \wedge \ (F \vee v = t)$ is added to the formula. Boolean *if-then-else*(A,B,C) constructs are simply considered as $(\neg A \vee B) \ \wedge \ (A \vee C)$.

Another difference is that it does not use certain optimization for SAT solvers, such as the *pure literal* rule, which as already pointed out in [BDS02,Tin02], among others, is in general not sound in the presence of a theory T.

2.3 The *Solver*$_T$ Part of DPLL(T)

Solver$_T$ knows what the real atoms $\mathcal{A}$ are, and knows about the theory T under consideration. It maintains a *partial* interpretation I: one that is defined only for some literals of the set $\mathcal{L}$ of all literals occurring in the problem input to the DPLL(T) prover. In the following, the literals of $\mathcal{L}$ are called $\mathcal{L}$-literals. Inside *Solver*$_T$, I is seen as a stack of literals, which is possible because I can equivalently be considered as the set of all $\mathcal{L}$-literals that are true in I. In the following, this stack will be called the I-stack. We say that a literal l is a T-*consequence* of I, denoted $I \models_T l$, if l is true in all total T-interpretations extending I. It is assumed that for every $\mathcal{L}$-literal l the solver is able to decide whether $I \models_T l$ or not. Essentially, *Solver*$_T$ is an abstract data type with the following five simple operations, which are all is needed for the interface between DPLL(X) and *Solver*$_T$:

`Initialize(`$\mathcal{L}$`: Literal set).` This procedure initializes *Solver*$_T$ with $\mathcal{L}$. It also initializes the I-stack to the empty stack.

`SetTrue(`l`: `$\mathcal{L}$`-literal): `$\mathcal{L}$`-literal set.` This function raises an "inconsistency" exception if $I \models_T \neg l$ with the current I-stack. Otherwise it pushes l onto the I-stack, and returns a set of $\mathcal{L}$-literals that have become a T-consequence of I only *after* extending I with l.
For example, in the EUF case, if $I \models a{=}b$ and $I \models d{=}c$, one of the literals returned by `SetTrue(`$d{=}b$`)` can be $f(a,a){=}f(b,c)$, if this is an $\mathcal{L}$-literal.
For efficiency reasons, the returned set can be incomplete; for example, in EUF, with `SetTrue(`$f(a) \neq f(b)$`)` it may be expensive to detect and return all consequences $a' \neq b'$ where $I \models a{=}a'$ and $I \models b{=}b'$ (see also Section 4).

`IsTrue?(`l`: `$\mathcal{L}$`-literal): Boolean.` This function returns `true` if $I \models_T l$, and `false` otherwise. Note that the latter can be either because $I \models_T \neg l$, or because neither $I \models_T l$ nor $I \models_T \neg l$.

`Backtrack(`n`: Natural).` This procedure pops n literals from the I-stack. Note that n is expected to be no bigger than the size of the I-stack.

`Explanation(`l`: `$\mathcal{L}$`-literal): `$\mathcal{L}$`-literal set.` This function returns a subset J of I such that $J \models_T l$, for a given literal l such that l was returned as a T-consequence of a `SetTrue(`l'`)` operation and no backtracking popping off this l' has taken place since then.
Intuitively, these conditions ensure that the "proof" of $I \models_T l$ that was found when l' was asserted is still valid. Note that several such J may exist; for example, in EUF, both $\{\ a \neq b,\ b{=}c\ \}$ and $\{\ a \neq d,\ d{=}c\ \}$ may be correct explanations for the literal $a \neq c$.

This operation is used by DPLL(X) for conflict analysis and clause learning. In our implementation it is used specifically to build an implication graph similar to those built by modern SAT solvers. For this application, it is also required that all literals in J are in the I-stack at heights lower than or equal to the height of the literal l' (i.e., no other "later" proof is returned; see Section 3 for an example).

We point out that the solver is independent from the characteristics of the DPLL(X) procedure that will use it. For example, it does not know about decision levels, unit propagation, or related features. In fact, a solver with this interface could be used as well in the context of non-DPLL systems, such as the lazy (or lemmas-on-demand) approaches, or even in resolution-based systems. Note that something like the `Explanation` operation is needed as well to produce the lemmas in such lazy approaches, and in general, in any deduction system that has to produce proof objects.

These five operations constitute in some sense a minimal set of operations for the exchange of information between the two main components of a DPLL(T) system. Having only these operations provides a high degree of modularity and independence between the global DPLL(X) engine and the theory solver $Solver_T$.

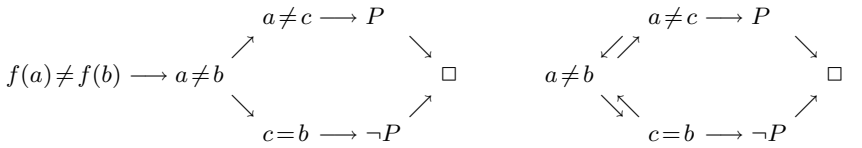
Our current DPLL(T) implementation has a DPLL(X) engine built in house to use the operations above. However, because of the simplicity and modularity of the solver interface, we believe that developers of state-of-the-art SAT solvers would need relatively little work to turn their solvers into DPLL(X) engines.

Although our implementation already performs very well for EUF, a tighter interconnection by means of more operations might enhance its performance. This could be achieved for example by having more fine-tuned theory-specific heuristics for choosing the next decision literal (see Section 6). However, possible efficiency gains would come at the expense of less modularity, and hence require more implementation work, especially when developing solvers for new theories.

3 Our Architecture for DPLL(X)

The current modular design is the result of developments and experiments with our own DPLL(X) implementation (in C). The current system has a clear Chaff-like flavor, mirroring Chaff's features as described in [MMZ+01,ZMMM01], with a 2-watched literal implementation for efficient unit propagation, and the VSIDS heuristic for selecting the next decision literal, in combination with restarts and the 1UIP learning scheme. Most of Chaff's features lift from DPLL to DPLL(X) without major modifications, so here we only point out some differences.

One difference arises in unit clause detection. In the propositional case, when a literal l becomes true, $\neg l$ is the only literal that becomes false before unit propagation is applied to the clause set, so new unit clauses can only come from clauses containing $\neg l$. In DPLL(T), on the other hand, additional literals can be set to false as a consequence of the assertions made to the theory solver. This possibility considerably increases the extent of unit propagations in DPLL(T) and, correspondingly reduces the size of the search space for the DPLL(X) engine. For example, in the EUF case, if $a \neq c \vee P$ is in the clause set and $I \models_T a{=}b$,



(a) Correct `Explanation` creates no cycles

(b) Wrong `Explanation` may create cycles

Fig. 1. Creation of undesirable cycles in the implication graph.

then setting a literal such as $b{=}c$ to true will make $a \neq c$ false and so produce P as a new unit consequence. We point out that this fact will be discovered by the DPLL(X) engine only if the set returned by the call `SetTrue`($b{=}c$) does in fact contain $a \neq c$ which, depending on the theory solver implementation, might or might not be the case. In any case, failing to return $a \neq c$ does not affect the completeness of the overall DPLL(T) system, only the extent of unit propagation.

Another source of incomplete unit clause detection is that two different literals may in fact be equivalent in the current state of the interpretation: again in the EUF case, if in $Solver_T$ we have $I \models_T a{=}b$, and there is a two-literal clause $a{=}c \vee b{=}c$, then the unit consequence $a{=}c$ (or, equivalently, $b{=}c$) will not be detected.

Concerning conflict analysis, the construction of the implication graph is more complex in DPLL(T) than in Chaff. In the latter, a node labeled with a literal l with antecedents nodes $l_1, \ldots, l_n$ is always due to a clause $\neg l_1 \vee \ldots \vee \neg l_n \vee l$ on which unit propagation has taken place. Hence, to build the implication graph it suffices to have, together with each such l, a pointer to its associated clause. In DPLL(T), an implication graph can contain, in addition to nodes like the above, also nodes l that are a T-consequence of their antecedents $l_1, \ldots, l_n$. Such nodes l are those returned by `SetTrue` calls to $Solver_T$, and can be recognized as such since they have no pointer to an associated clause. Their antecedents can be obtained from the solver itself by calling `Explanation`(l). For example, a run of the DPLL(X) algorithm on the clauses:

$$f(a) \neq f(b) \vee d \neq e \qquad a{=}b \vee a \neq c \qquad a{=}b \vee c{=}b \qquad a{=}c \vee P \qquad c \neq b \vee \neg P$$

can be as follows:

1.	`SetTrue`($f(a) \neq f(b)$).	Decision. `SetTrue` returns $a \neq b$, a T-consequence.
2.	`SetTrue`($a \neq c$).	Unit propagation of $a \neq b$ on $a{=}b \vee a \neq c$.
3.	`SetTrue`($c{=}b$).	Unit propagation of $a \neq b$ on $a{=}b \vee c{=}b$.
4.	`SetTrue`(P).	Unit propagation of $a \neq c$ on $a{=}c \vee P$.
5.	`SetTrue`($\neg P$).	Unit propagation of $c{=}b$ on $c \neq b \vee \neg P$. Conflict!

The implication graph is built backwards from P and $\neg P$. When $a \neq b$ is reached, `Explanation`($a \neq b$) is called, and indeed one possible explanation is $f(a) \neq f(b)$. This process results in the graph depicted in Figure 1(a). Note that $\{a \neq c, c{=}b\}$

also has $a \neq b$ as a T-consequence, but using that set would lead to the graph in Figure 1(b), which has cycles. To avoid cycles, it is enough for `Explanation`(l) to return explanations that are "older" than l, as precised in the definition of `Explanation`.

4 A Solver for EUF

A key ingredient for a solver for EUF is an algorithm for congruence closure. Now, the $O(n\ log\ n)$ DST congruence closure algorithm given in [DST80] (see also [NO80]) needs a relatively expensive initial transformation to directed acyclic graphs of outdegree 2. In [NO03] we proposed to replace this transformation by another one, at the formula representation level: we *Currify*, like in the implementation of functional languages; as a result, there will be only one binary "apply" function symbol (denoted here by a dot "$\cdot$") and constants. For example, Currifying $f(a, g(b), b)$ gives $\cdot(\cdot(\cdot(f, a), \cdot(g, b)), b)$. Furthermore, like in the abstract congruence closure approaches (cf. [Kap97,BT00]), we introduce new constant symbols c for giving names to non-constant subterms t; such t are then replaced everywhere by c, and the equation $t = c$ is added. Then, in combination with Currification, one can obtain the same efficiency as in more sophisticated DAG implementations by appropriately indexing the new constants like c, which play the role of the pointers to the (shared) subterms like t in the DAG approaches. For example, we flatten the equation $\cdot(\cdot(\cdot(f, a), \cdot(g, b)), b) = b$ by replacing it by the four equations $\cdot(f, a) = c$, $\cdot(g, b) = d$, $\cdot(c, d) = e$, and $\cdot(e, b) = b$.

These two (structure-preserving) transformations are done in linear time *once and for all* on the input formula given to our DPLL(T) procedure. As a consequence, since all compound terms (atoms) occuring in the EUF formula have gotten a "name", i.e., they are equal to some constant, all equality atoms in the EUF formula are in fact equations between constants. Furthermore, the congruence closure algorithm of $Solver_T$ only needs to infer consequences from a fixed, static set of equations E of the form $\cdot(a, b) = c$, where a, b and c are constants (note that the $\cdot$ symbol does not occur outside E). This makes our (DST-like) algorithm surprisingly simple and clean, and hence easier to extend. Our implementation of [NO03] also runs in $O(n\ log\ n)$ time, but of course it can be much faster than algorithms for arbitrary terms[2]. Once the closure is computed, deciding whether two constants a and b belong to the same class, i.e., all positive `IsTrue?`(a=b) operations, can be done in constant time.

The data structures for congruence closure are:

1. *Pending unions*: a list of pairs of constants yet to be merged.
2. The *Representative* table: an array indexed by constants, containing for each constant its current representative (this can also be seen as a union-find data structure with *eager path compression*).
3. The *Class lists*: for each representative, the list of all constants in its class.

[2] In fact, it is about 50 times faster than earlier implementations such as [TV01] on the benchmarks of [TV01].

4. The *Lookup table*: for each input term $\cdot(a, b)$, a call *Lookup*(*Representative*(a), *Representative*(b)) returns in constant time a constant c such that $\cdot(a, b)$ is equivalent to c, and returns $\perp$ if there is no such c.
5. The *Use lists*: for each representative a, the list of input equations $\cdot(b, c)=d$ such that a is the representative of b or c (or of both).

Each iteration of the algorithm roughly amounts to picking a pending union, and then using the lookup table and the use lists for efficiently detecting new pairs of constants to be merged. We refer to [NO03] for a detailed description and analysis of the algorithm, as well as for its extension to successor and predecessor symbols, which is also $O(n\ log\ n)$.

We now briefly describe the other data structures and algorithms for the $Solver_T$ operations, following a notation where unprimed constants a, b, c are always representatives of primed ones a', b', c'; symbols d and e can be representatives or not.

The Literal Lists and Their Use for `SetTrue`. Upon its initialization, $Solver_T$ builds, once and for all, for each constant d, a *positive literal list* containing all positive DPLL(X) literals of the form $d=e$, i.e., all positive literals containing d that have been communicated to $Solver_T$ in the `Initialize` operation. Analogously, there is also a *negative literal list* for each d.

If a positive `SetTrue`, and its subsequent congruence closure, produces a union such that a class with former representative a is now represented by a different b, then, for each a' in the class list of a, the positive literal list of a' is traversed and all $a'=b'$ in this list are returned as T-consequences. Also the negative literal list of all such a' is traversed, returning those $a' \neq c'$ such that $a \neq c$ is stored in *Diseq*, a hash table containing all currently true disequalities between representatives; analogously also the negative literal list of all b' is traversed.

After a `SetTrue` operation of a negative equation with representative form $a \neq b$, the negative literal list of all a' is traversed (or equivalently, the one of all b', if this is expected to be less work), returning all $a' \neq b'$.

Backtracking. This is dealt with by means of a mixed policy: unions between constants are stacked in order to be undone. The *Diseq* hash table and the *Lookup* data structure (another hash table) are not restored under backtracking but instead have time stamps (the I-stack height combined with a global counter).

Explanations. A newly derived positive equality is a T-consequence of a set of positive `SetTrue` operations and of the equations inside the congruence closure module. Negative equations are also T-consequences of such positive equalities, but in addition they are always caused as well by a single negative `SetTrue`.

Hence, for retrieving the explanations for DPLL(X) literals, we need to be able to proceed "backwards" in our data structures until the set of initial `SetTrue` operations is reached. At each congruence propagation deriving $d=e$ as a consequence of $\cdot(a', b')=d$ and $\cdot(a'', b'')=e$, (pointers to) these two equations are kept, allowing one to continue backwards with the explanations of $a'=a''$ and $b'=b''$. In addition, there is a method for efficiently extracting from a (slightly extended) union-find data structure the list of unions explaining a certain equality.

Finally, for explaining a negative equality $d_1 \neq e_1$, there is an additional table for retrieving the single negative $\texttt{SetTrue}(d_2 \neq e_2)$ operation that caused it; the remaining part of the explanation can then be found backwards, as the union of the explanations of $d_1{=}d_2$ and $e_1{=}e_2$.

5 Experimental Results

Experiments have been done with all 54 available benchmarks (generated from verification problems) that can be handled in EUF with successors and predecessors. Most of them were provided by the UCLID group at CMU[3]. All benchmarks are in SVC format [BDL96] (see [BLS02a] for more details on them).

The table below contains (in its second column) the translation times for the four eager approaches described in Section 1: SD and EIJ [VB03], and the two hybrid methods Hybrid1 [BLS02a] and Hybrid2 [SLB03]. All translations were done using the state-of-the-art translator provided within the UCLID 1.0 tool. Furthermore, for two major SAT solvers, zChaff [MMZ+01] (version 2003.12.04) and BerkMin [GN02] (its recent version 561) the running times on the translated formulas are given. For a fair comparison with our system, the times for the zChaff and BerkMin runs *include* the translation times as well, since those translations are not mere formats conversions but the result of sophisticated algorithms for reducing the size and the search space of the propositional formula produced. The choice of zChaff and BerkMin is motivated by the fact that our current DPLL(X) engine is modeled after zChaff and that BerkMin is presently considered one of the best SAT solvers overall. The first table has an extra column with the running times of SVC (version 1.1) as well.

Results are in seconds and are aggregated per family of benchmarks, with times greater than 100s rounded to whole numbers[4]. All experiments were run on a 2GHz 512MB Pentium-IV under Linux, with the same settings for each benchmark except for the "Two queues" benchmarks where our system had learning turned off. Each benchmark was run for 6000 seconds. An annotation of the form (n t) or (n m) in a column indicates respectively that the system timed out or ran out of memory on n benchmarks. Each timeout or memory out is counted as 6000s.

We point out that our current DPLL(X) implementation is far less tightly coded than zChaff, and is more than one order of magnitude slower than zChaff on propositional problems, even when the overhead due to the calls to $Solver_T$ is eliminated. Considering that BerkMin consistently dominates zChaff on all the benchmark families, it is remarkable that our system performs better than the UCLID+BerkMin combination on the great majority of the families. In fact, there is no unique translation-based approach that outperforms DPLL(T) on more than two benchmark families. Furthermore, DPLL(T) is faster than SVC on all benchmarks, and so also faster than the lazy approach-based systems CVC [BDS02] and Verifun [FJOS03] which, as shown in [FJOS03], are outperformed

[3] We are grateful to Shuvendu Lahiri and Sanjit Seshia for their kind assistance.

[4] Individual results for each benchmark can be found at `www.lsi.upc.es/~oliveras`, together with all the benchmarks and an executable of our system.

by SVC. We see this as strong initial evidence that DPLL(T) is qualitatively superior to existing approaches for deciding satisfiability modulo theories[5].

Benchmark family	**SD**	**BerkMin**	**Chaff**	**DPLL(T)**	**SVC**
Buggy Cache	2.3	**2.4**	3.4	6.7	(1t) 6000
Code Validation Suite	16.2	44.9	43.9	**3.7**	56.9
DLX processor	3.9	10.2	13.3	**1.2**	16.9
Elf processor	34.1	5882	(1t) 6104	**575**	(1t) 6078
Out of order proc.(rf)	27.1	(2t) 18211	(3t) 19213	**6385**	(2t) 12666
Out of order proc.(tag)	54.3	**247**	1457	1979	(4t) 28788
Load-Store processor	22.2	51.4	239	**30.3**	(3t) 18476
Cache Coherence Prot.	20.4	4151	(1t) 9634	**3601**	(4t) 26112
Two queues	5.1	407	1148	**73.6**	1872

Benchmark family	**EIJ**	**BerkMin**	**Chaff**	**DPLL(T)**
Buggy Cache	6.3	**6.4**	9.3	6.7
Code Validation Suite	40.7	41	41.8	**3.7**
DLX processor	13	13.9	14.4	**1.2**
Elf processor	(2m) 20.1	(2m) 12021	(2m) 12021	**575**
Out of order proc.(rf)	70.6	(1t) 7453	(2t) 13926	**6385**
Out of order proc.(tag)	210	**510**	837	1979
Load-Store processor	(1m) 32	(1m) 6034	(1m) 6037	**30.3**
Cache Coherence Prot.	(3m) 102	(3m) 18257	(3m) 18437	**3601**
Two queues	(3m) 19.4	(3m) 18028	(3m) 18034	**73.6**

Benchmark family	**Hybrid 1**	**BerkMin**	**Chaff**	**DPLL(T)**
Buggy Cache	6.2	**6.4**	9.2	6.7
Code Validation Suite	13.2	13.5	13.5	**3.7**
DLX processor	13.1	14.1	15.2	**1.2**
Elf processor	187	941	1646	**575**
Out of order proc.(rf)	65.3	(1t) 7524	(2t) 13009	**6385**
Out of order proc.(tag)	175	**612**	799	1979
Load-Store processor	64.1	79.6	88.4	**30.3**
Cache Coherence Prot.	(3m) 102	(3m) 18257	(3m) 18438	**3601**
Two queues	(3m) 19.5	(3m) 18019	(3m) 18028	**73.6**

Benchmark family	**Hybrid 2**	**BerkMin**	**Chaff**	**DPLL(T)**
Buggy Cache	3	**3.2**	3.7	6.7
Code Validation Suite	27.7	28	28.6	**3.7**
DLX processor	11.3	12.9	14.3	**1.2**
Elf processor	47.2	3182	5467	**575**
Out of order proc.(rf)	53	(1t) 10626	(2t) 13913	**6385**
Out of order proc.(tag)	140	6918	(2t) 12173	**1979**
Load-Store processor	40.1	45.47	47.71	**30.3**
Cache Coherence Prot.	37.3	**209**	690	3601
Two queues	5.7	793	1832	**73.6**

[5] This evidence was confirmed recently by further experiments (also reported in detail at `www.lsi.upc.es/~oliveras`) showing that DPLL(T)'s perfomance dominates that of ICS 2.0 as well on the benchmarks listed here.

As expected, translation-based methods will normally outperform DPLL(T) for problems where the theory T plays a very small role. But this is no longer the case when theory predicates start playing a significant role, as in the families Code Validation, Elf and OOO processors (rf), and Two queues. This phenomenon becomes dramatic for instance in benchmarks from group theory[6].

6 Conclusions and Future Work

We have presented a new approach for checking satisfiability in the EUF logic. This approach is based on a general framework and architecture in which a generic DPLL-style propositional solver, DPLL(X), is coupled with a specialized solver $Solver_T$ for a given theory T of interest. The architecture is highly modular, allowing any theory solver conforming to a simple, minimal interface to be plugged in into the DPLL(X) engine. The fundamental advantage with respect to previous approaches is that the theory solver is not only used to *validate* the choices made by the SAT engine, as done for instance in CVC, but also to propagate the entailed literals returned by `SetTrue`, using information from *consistent* partial models, considerably reducing the search space of the SAT engine. Initial results indicate that in the EUF case this leads to significant speed-ups in overall performance.

More work needs to be done on our implementation. Aspects such as lemma management, decision heuristics and restarting policies are still immature, More accurate theory-dependent heuristics need to be explored. Also minimal-model-preserving optimizations should be worked out; for instance, the notion of P-terms [BGV01] has its counterpart in our framework, and so could be used.

Finally, other future work of course concerns the development of new theory solvers, or the conversion of existing ones (e.g., those used in CVC), into theory solvers conforming to our interface: solvers for EUF with associativity and transitivity (AC) properties for certain symbols, EUFM (EUF+memories) [BD94], Separation Logic [SLB03], or the full *CLU* logic [BLS02b].

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE-18*, LNCS 2392, pages 195–210, 2002.

[ACG00] A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Procs. of the 5th European Conference on Planning*, LNCS 1809, pages 97–108. Springer, 2000.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Procs. 6th Int. Conf. Computer Aided Verification (CAV)*, LNCS 818, pages 68–80, 1994.

[BDL96] C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Procs. 1st Intl. Conference on Formal Methods in Computer Aided Design*, LNCS 1166, pages 187–201, 1996.

[6] See again `www.lsi.upc.es/~oliveras` for details.

[BDS02] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation into sat. In *Procs. 14th Intl. Conf. on Computer Aided Verification (CAV)*, LNCS 2404, 2002.

[BGV01] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Computational Logic*, 2(1):93–134, 2001.

[BLS02a] R. E. Bryant, S. Lahiri, and S. Seshia. Deciding CLU logic formulas via boolean and pseudo-boolean en codings. In *Procs. 1st Int. Workshop on Constraints in Formal Verification*, 2002.

[BLS02b] R. E. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Procs. of CAV'02*, LNCS 2404, 2002.

[BT00] L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Conf. Autom. Deduction, CADE*, LNAI 1831, pages 64–78, 2000.

[BV02] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. *ACM Trans. Computational Logic*, 3(4):604–627, 2002.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *CACM*, 5(7):394–397, 1962.

[dMR02] L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. In *Procs. 5th Int. Symp. on the Theory and Applications of Satisfiability Testing, SAT'02*, pages 244–251, 2002.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[DST80] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *JACM* 27(4):758–771, 1980.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explanation. In *Procs. 15th Int. Conf. on Computer Aided Verification (CAV)*, LNCS 2725, 2003.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, 2002.

[Kap97] D. Kapur. Shostak's congruence closure as completion. In *Procs. 8th Int. Conf. on Rewriting Techniques and Applications*, LNCS 1232, 1997.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC'01)*, 2001.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures bases on congruence closure. *JACM*, 27(2):356–364, 1980.

[NO03] R. Nieuwenhuis and A. Oliveras. Congruence closure with integer offsets. In *10th Int. Conf. Logic for Programming, Artif. Intell. and Reasoning (LPAR)*, LNAI 2850, pages 78–90, 2003.

[PRSS99] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domains instantiations. In *Procs. 11th Int. Conf. on Computer Aided Verification (CAV)*, LNCS 1633, pages 455–469, 1999.

[SLB03] S. Seshia, S. Lahiri, and R. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Procs. 40th Design Automation Conference (DAC)*, pages 425–430, 2003.

[SSB02] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Procs. 14th Intl. Conference on Computer Aided Verification (CAV)*, LNCS 2404, pages 209–222, 2002.

[Tin02] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Procs. 8th European Conf. on Logics in Artificial Intelligence*, LNAI 2424, pages 308–319, 2002.

[TV01] A. Tiwari and L. Vigneron. Implementation of Abstract Congruence Closure, 2001. At `www.csl.sri.com/users/tiwari`.

[VB03] M, N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Int. Conf. on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.