SMT($\mathcal{CLU}$): A Step toward Scalability in System Verification*

Hossein M. Sheini
Electrical Engineering and Computer Science Dept.
University of Michigan, Ann Arbor, MI 48109
Email: hsheini@umich.edu

Karem A. Sakallah
Electrical Engineering and Computer Science Dept.
University of Michigan, Ann Arbor, MI 48109
Email: karem@umich.edu

ABSTRACT

We describe a SAT-based decision method for the underlying logic in many formal verification problems; i.e. the counter arithmetic logic with lambda expressions and uninterpreted functions ($\mathcal{CLU}$). This logic is well suited for equivalence checking of two versions of a hardware design or the input and output of a compiler and has been recently utilized in several model checkers. Our method follows the general Satisfiability Modulo Theories or SMT($\mathcal{T}$) framework and combines a DPLL-style SAT solver with two theory solvers; one specific to equality and the other to separation inequality atoms within $\mathcal{CLU}$. By adopting a combined implication scheme, we coordinate the efforts among theory solvers, and by efficiently processing uninterpreted functions involved in conflicts, we considerably improve the effectiveness of SAT learning and backtracking routines. Finally, we empirically demonstrate the effectiveness of our SMT($\mathcal{CLU}$) procedure and compare its performance to recent solvers on a wide range of hardware verification benchmarks.

I. INTRODUCTION

The problem of scalability or state-explosion has been regarded as the most fundamental challenge in formal verification and specifically in model checking approaches. Since the introduction of temporal logic model-checking into verification of finite state systems [1], several improvements have been achieved in addressing the scalability problem. Recent advances in this field include techniques to exploit predicate abstraction, parameterized designs, BDDs, propositional satisfiability, and counter-example-guided abstraction refinement (CEGAR). For instance in CEGAR [2], an upper approximation (abstraction) of the system model is obtained and initially solved. When an error (counter-example) is found, CEGAR checks whether the error is the result of the approximation (the condition for the error to be *spurious*) or is actually present in the original model. In the case of a spurious error, the abstracted model is refined in order to eliminate the erroneous counter-example. This process continues until either the specification is proved to be true in the abstract model or a genuine error is detected. The scalability of the CEGAR framework directly depends on 1) the efficiency of solving the abstracted model and 2) the effectiveness of refinement. In the latter case, recently developed techniques such as elimination of redundant predicates [3], or word-level refinement [4] have had some success in improving the scalability of the overall abstraction/refinement approach. On the solving front, there also has been considerable progress in both introduction of new logics for efficient modeling of verification systems as well as advances in solving procedures embedded in model checkers and adaptable to those logics.

The first-order logic of counter arithmetic with lambda expressions and uninterpreted functions ($\mathcal{CLU}$) [5] has been widely adopted to model several infinite state systems in applications such as microprocessor verification [6] or API-level security (format-string) exploit detection [7]. In the meantime, the recent improvements in the strength and versatility of DPLL-style satisfiability procedures for propositional logic (SAT) has extended the applications of SAT solvers to the decision methods for more expressive logics. These methods, known as Satisfiability Modulo Theories or SMT($\mathcal{T}$) [8], are applied to $\mathcal{CLU}$ logic by either i) transforming the $\mathcal{CLU}$ formula into an equi-satisfiable propositional formula, known as the *eager* approach [9], or ii) integrating a theory solver for systems of $\mathcal{CLU}$ atoms within a propositional SAT solver. Two common integration strategies in the latter case are the *lazy approach* [10] and the DPLL($\mathcal{T}$) *approach* [11]. In the lazy approach, a propositional abstraction of the formula is initially solved by a SAT solver and the theory consistency of each SAT solution is checked by a theory solver. This procedure terminates if the theory consistency of a SAT solution is established or none of the solutions is proved consistent. In the DPLL($\mathcal{T}$) approach, on the other hand, a combined DPLL reasoning and learning procedure is applied across theory atoms. In other words, in this approach, the consistency of the theory atoms is checked as soon as they are added to the model by the SAT solver. The theory solver also has the capability to guide the SAT search by deducing theory atoms as the search proceeds. This extends the role

of the involved theory solver from just a consistency checker as in the lazy approach to a propagation engine for theory relations, similar to SAT solver's Boolean clause propagation for logical relations. The DPLL($\mathcal{T}$) approach has been applied to the logic of equalities with uninterpreted functions in [11], the integer Unit-Two-Variable-Per-Inequality (UTVPI) logic in [12] and the difference logic in [13].

In the present paper, we extend the DPLL($\mathcal{T}$) approach to $\mathcal{CLU}$ logic and introduce novel methods to accommodate the characteristics of $\mathcal{CLU}$ atomic formulas and facilitate their integration into the SAT deduction and learning/backtracking schemes. In our approach, we combine a DPLL-style SAT solver [14] with two incremental algorithms (referred to as *checkers*), one for equality and the other for separation-inequality atoms. This is in contrast to the method of [11] and [13] where only one theory solver is integrated within SAT. We construct a hybrid implication graph that takes into account both logical implications (those due to unit-clause-propagation) and implications detected within each checker.

We also propose a combined learning scheme that detects implications due to uninterpreted functions/predicates that are involved in the conflicts in order to learn their representative consistency constraints. This method is specifically beneficial to solve $\mathcal{CLU}$ formulas derived from system verification problems where the uninterpreted functions/predicates are vastly used to abstract word-level values of data and implementation details of functional blocks. Our approach considerably strengthens the conflict-induced learning procedure and results in a more efficient non-chronological backtracking of the underlying SAT solver. Finally, we present a complete experimental evaluation on the effectiveness of each of these approaches on various benchmarks in hardware verification and thoroughly analyze different aspects of our algorithm. We also compare our solving algorithm to recent methods including the eager approach of UCLID [9], [5] and several SMT solving approaches [10], [13].

The remainder of the paper is organized as follows. In Section II, the $\mathcal{CLU}$ logic is defined and several decision methods applicable to $\mathcal{CLU}$ are described. In Section III, the organization of our overall solver and our hybrid implication scheme are explained. Our efficient learning scheme tuned toward uninterpreted functions is described in Section IV. The performance of our methods is analyzed in Section V and we conclude in Section VI.

II. Preliminaries

A. Counter Arithmetic Logic

We consider the $\mathcal{CLU}$ formula after expanding lambda applications [5] and after recursively applying *successor* and *predecessor* functions both to be replaced by simple addition with integer constants. In this context, two types of variables are defined, Boolean variables, denoted by *bool-var*, and integer variables, denoted by *int-var*. We denote integer constants by c. A *term*, denoted by t, is inductively defined as follows:

$$t ::= c \mid \textit{int-var} \mid t + c \mid f(t, \cdots, t) \tag{1}$$

where f represents an *uninterpreted*[1] *function*. Consequently, an *atom* in this logic is either

1) A Boolean variable, *bool-var*, or an uninterpreted predicate over terms, $P(t_1, \cdots, t_n)$; or
2) An equality of the form $(t_i - t_j = c)$, a disequality of the form $(t_i - t_j \neq c)$ or a *separation* inequality of the form $(t_i - t_j \leq c)$ over terms, t_i and t_j.

The former type is denoted as *Boolean* and the latter as *integer* atom. Interchangeably, we refer to atoms in the form of $(t_i - t_j = c)$, $(t_i - t_j \neq c)$ and $(t_i - t_j \leq c)$ as an *equality*, a *disequality* and an *s-inequality* respectively[2]. A *literal* is an atom or its negation. Thus, a quantifier-free $\mathcal{CLU}$ formula is constructed by combining $\mathcal{CLU}$ literals using logical connectives $(\vee, \wedge, \neg)$. Note that each uninterpreted function (predicate) necessitates that a *functional consistency* constraint of the following form is satisfied in any model of the formula (where f is an uninterpreted function and t_i's are terms):

$$\begin{aligned} &\forall t_{11}, \ldots, t_{1k}, t_{21}, \ldots, t_{2k} : \\ &\quad (t_{11} = t_{21} \wedge \ldots \wedge t_{1k} = t_{2k}) \\ &\qquad \rightarrow f(t_{11}, \cdots, t_{1k}) = f(t_{21}, \cdots, t_{2k}) \end{aligned} \tag{2}$$

As a running example in this paper, we use the following CNF formula as part of a larger problem that represents a common structure encountered in verification applications.

$$\begin{aligned} \varphi = &[f(x, z_1) \neq f(y, z_2)] \wedge [z_1 = z_2] \wedge \\ &[P_1 \rightarrow (x = x_1)] \wedge [P_2 \rightarrow (x = x_2)] \wedge \\ &[P_3 \rightarrow (x = x_3)] \wedge [P_4 \rightarrow (x = x_4)] \wedge \\ &[(y = x_1) \vee (y = x_2) \vee (y = x_3) \vee (y = x_4)] \wedge \\ &[(x_4 - z_1 \leq 1) \vee (x_1 \neq x_2) \vee (y = x_2 + 1)] \end{aligned} \tag{3}$$

In this example, x_i's, y and z are *int-vars* and P_i's are *bool-vars* and f is an uninterpreted function.

B. SAT-based Decision Methods for $\mathcal{CLU}$

With the recent advances in SAT solvers, several SMT approaches to solve verification problems involving integer atoms and uninterpreted functions such as $\mathcal{CLU}$ formulas have been developed. Below we describe the related SMT methods, applicable to $\mathcal{CLU}$, developed in the recent years:

a) Eager Approach: [9], [15], where the $\mathcal{CLU}$ formula is translated to an equi-satisfiable propositional formula following either *small domain instantiation* or *per-constraint encoding* approach [15]. The former method computes the bounds on the *int-vars* and converts each to a bit-vector of *bool-vars*. The latter method replaces all integer atoms with fresh *bool-vars* and augments the resulted Boolean formula with all possible transitivity constraints on the values of those Boolean variables. This approach, aka *EIJ method*, requires pre-solving the entire non-Boolean portion of the problem and encoding it within the Boolean formula.

[1] An uninterpreted function (and similarly an uninterpreted predicate) is a function that we only know is *consistent*; two applications of the same function on the same arguments yield the same value.

[2] Note that an equality (disequality) can be converted into a conjunction (disjunction) of two s-inequalities.

b) Lazy (layered) Approach: [10], where the $\mathcal{CLU}$ formula is initially abstracted into a Boolean formula by replacing all its integer atoms with fresh Boolean variables. The Boolean formula is then solved by a SAT solver and if satisfiable, at the first layer the consistency of the conjunction of only equalities/disequalities are checked. At the next (second) layer, the s-inequalities are also added to this conjunction and the consistency of the updated set of integer atoms is checked. This process terminates if a satisfiable solution at the highest layer is found or none of the solutions to the Boolean formula are proved consistent. If a conflict is detected in a layer, this conflict is used to prune the search in the abstract Boolean formula and the process reiterates.

c) DPLL($\mathcal{T}$) Approach: [13], where all equalities and disequalities within the $\mathcal{CLU}$ formula are initially converted to s-inequalities and a checker for deciding the satisfiability of a conjunction of s-inequalities is utilized within the SAT solver. If the SAT solver adds an s-inequality atom to the solution, that atom is passed to the checker to establish the satisfiability of the updated set of s-inequalities in the solution. Consequently, all the integer atoms implied by the updated set of s-inequalities are assigned within the SAT solver accordingly. The advantages of this approach over the layered and eager approaches can be summarized as follows:

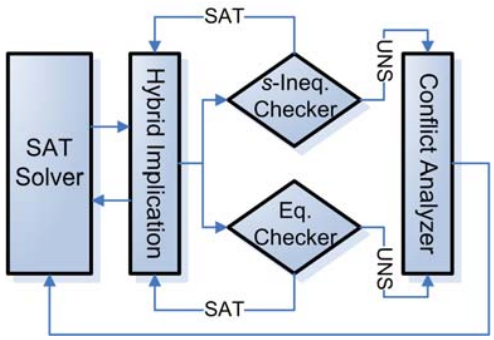
- Compared to the eager approach, the DPLL($\mathcal{T}$) method only checks the consistency of integer atoms when they are absolutely required to establish the satisfiability of the formula. This avoids "pre-solving" all integer atoms that is both time and memory-consuming.
- Compared to the layered approach, by processing each integer atom "on-demand" and propagating all its theory implications, the DPLL($\mathcal{T}$) approach detects conflicts among those atoms as they occur. This advantage is emboldened when the majority of conflicts arise due to combining integer atoms and not because of the Boolean atoms or the logical structure of the problem. To detect such conflicts, the layered approach requires solving the abstract Boolean formula at each iteration.

Note that the uninterpreted functions are either eliminated (i.e. reduced to a combination of equalities following Ackermann [16] or "ITE" [5] techniques), as is required in the eager approach or are solved by a specialized algorithm, i.e. based on congruence-closure, in a separate layer in the layered approach or on-demand in the DPLL($\mathcal{T}$) approach.

III. Characterizing $\mathcal{CLU}$ atoms in DPLL($\mathcal{T}$)

We propose a novel method for tight adaptive integration of efficient checkers for different $\mathcal{CLU}$ atoms within the DPLL($\mathcal{T}$) framework. Our approach is specifically different from previous DPLL($\mathcal{T}$)-based methods [11], [12], [13] which use a s-inequality checker when the formula includes s-inequalities, and use an equality checker only when no s-inequalities appear in the formula. For instance, in BarceLogicTools [17], if s-inequalities exist in the problem, all function symbols are first removed using the Ackermann technique [16] and then a single s-inequality checker is used.

Fig. 1. The organization of algorithms in our SMT($\mathcal{CLU}$) system



A. Organization of the Solving Procedures

In our SMT($\mathcal{CLU}$) framework, a generic CNF SAT solver "orchestrates" the search for a satisfiable set of Boolean and integer atoms within the $\mathcal{CLU}$ formula. Integer *checkers* also need to be employed to guarantee that the *conjunction* of integer atoms within the satisfiable assignment set has a solution in the integer domain at all times and at the same time to propagate integer atoms in that domain. These checkers report a conflict back to the SAT solver if an inconsistent integer atom is activated.

In many cases the most efficient approach to solve a conjunction of integer atoms all in a certain form is not necessarily applicable to integer atoms in a more expressive form. More specifically, while it is possible to solve systems of equalities and s-inequalities using a Bellman-Ford algorithm in $O(m \cdot n)$ time (where m is the number of constraints and n is the number of variables), the most efficient method to solve systems of only equalities/disequalities with successors and uninterpreted functions is the *congruence-closure* algorithm similar to that of [18] that runs in $O(n \cdot \log(n))$ time.

Based on this, we propose a hybrid architecture that *fragmentizes* the $\mathcal{CLU}$ integer atoms into two types, equalities/disequalities in the form of $(x_i \sim x_j + c)$ where x_i and x_j are *int-vars* and $\sim \in \{=, \neq\}$; and s-inequalities in the form of $(x_i - x_j \leq c)$. Consequently, we apply a specific checker for each type. As demonstrated in Figure 1, we additionally utilize a hybrid implication system to guarantee that the infeasible and feasible regions in the solution spaces of the two checkers do not overlap at any time. Note that this approach is specifically beneficial in cases that few variables are shared among the two types of integer atoms, minimizing the communication between the two checkers. Nonetheless, adopting our two congruence-closure and transitive-closure algorithms for respectively solving equalities and s-inequalities, makes this communication very cheap.

We utilize a *congruence-closure* procedure to check the consistency of a set of equalities and disequalities. In this procedure, for each *int-var* both an equality and a disequality class is maintained. Upon activating an equality between two *int-vars*, the equality and disequality classes associated with both variables are merged. Upon activating a disequality

between two *int-vars*, the variables in the equality class of each *int-var* are added to the disequality class of the other. Note that an offset value, c, is associated with each *int-var* in these classes. In both cases, the checker returns a conflict if the equality and disequality classes overlap. All occurrences of uninterpreted functions are replaced with a fresh *int-var*, referred to as a *representative variable* and their consistency is maintained adopting the currifying approach of [18]. In this method each representative variable is linked to the arguments in its specific function occurrence through a *look-up table*. In case all the equalities between the arguments of two occurrences of an uninterpreted function are established, the equality between the associated representative variables is implied. Finally, the consistency of the set of s-inequalities is checked adopting an algorithm that maintains a transitively-closed set of s-inequalities at all times. The checker in this case will return a conflict if this set contains an s-inequality in the form of $(0 \leq c)$ where $c < 0$. Note that both these algorithms can produce an explanation for any conflict within their domain and can imply any unassigned integer atom that is the consequence of their set of activated integer atoms.

B. Hybrid Implication Scheme

Employment of two incremental checkers for equalities/disequalities and s-inequalities within our SMT($\mathcal{CLU}$) framework requires a hybrid interaction system between those checkers. Specifically, in cases that the s-inequality checker implies an equality, that equality might have to be propagated within the equality checker. The equalities and disequalities generated within the equality checker due to the mergers of its internal equality classes also might have to be propagated in the s-inequality checker. In order to apply only "necessary" cross-checker implications, the overall solver maintains two sets of *int-vars*, one representing those variables present in the equalities/disequalities and the other those in s-inequalities in the formula. An equality/disequality (and similarly an s-inequality) is propagated in the s-inequality (equality) checker only if each of its variables is shared with at least one s-inequality (equality/disequality) in the formula.

To facilitate this process, in this paper, we propose to construct a hybrid implication graph to be managed outside the checkers and utilized in the overall procedure. The nodes in this graph could be either original or generated Boolean/integer atoms; An edge (and subsequently a node) is added to the graph whenever either the SAT solver's unit-clause-propagation implies an atom or the checkers imply an integer atom (possibly not in the original formula). Additionally, implying an equality atom of the form $(x = y + c)$ could trigger adding two s-inequality atoms of the form $(x - y \leq c)$ and $(y - x \leq -c)$ into the hybrid implication graph and vice versa, i.e. implying the same two s-inequalities itself could imply an equality atom $(x = y + c)$. In case of adding a dis-equality atom of the form $(x \neq y + c)$ to the implication graph, the s-inequality checker needs to decide to add either $(x - y < c)$ or $(x - y > c)$, each denoted as a *dis-equality disjunct*. If both disjuncts are inconsistent, the system reports a conflict, otherwise a consistent one is selected and added to the implication graph. In the case that both disjuncts are consistent, the procedure arbitrarily selects one, say $(x - y < c)$, and marks all the implications originating from that node in the hybrid implication graph. If further in the search, a conflict involving a marked node is detected, the consistency of the other disjunct, $(y - x < -c)$, is checked and if consistent, the marked nodes are removed and the implication graph is updated as if the other disjunct was selected.

Fig. 2. Partial hybrid implication graph for example formula of (3) with marked (shaded) nodes (atoms)

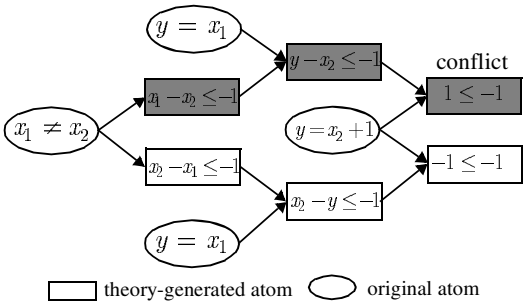


Figure 2 demonstrates a part of the implication graph under an arbitrary assignment in our running example of (3). In this example, assume that all *int-vars* are present in all types of integer atoms and the original atoms $(y = x_1)$, $(x_1 \neq x_2)$ and $(y = x_2 + 1)$ are assigned to true in consecutive decision levels. Upon assigning $(x_1 \neq x_2)$, it should be converted to either $(x_1 - x_2 \leq -1)$ or $(x_2 - x_1 \leq -1)$ (dis-equality disjuncts) to be added to the hybrid implication graph. As it is shown in Figure 2, since both disjuncts are consistent, selecting $(x_1 - x_2 \leq -1)$ requires marking all the nodes it implies. Activating $(y = x_2 + 1)$ triggers a conflict. Since at this point the other disjunct $(x_2 - x_1 \leq -1)$ is still consistent, the procedure does not report a conflict and simply removes marked nodes and updates the hybrid implication graph with $(x_2 - x_1 \leq -1)$ and generates its subsequent implications accordingly.

IV. Learning Functional Consistency Constraints

In the general DPLL($\mathcal{T}$) approach [11], [13], [12] as well as different versions of the lazy approach [19], [10], conflict-induced learning is carried out by the CNF SAT solver following its generic implication graph analysis [20]. In such cases, the SAT solver has no information on the nature of the atoms and only analyzes the implications ending in the conflict. In the end it generates a learned clause at the unique implication point (UIP) of the implication graph. In this context the implications due to the functional consistency constraint of (2) are simply viewed as another form of deduction in the hybrid implication graph besides those due to unit-clause-propagation and theory checker procedures.

It is also important to note that the most efficient approach to process the uninterpreted functions in congruence-closure solvers [18] is to utilize a look-up table which checks the

functional consistencies "on-demand", i.e. when all the pairwise equalities between the arguments of two occurrences of an uninterpreted function are established and not vice versa. In other words, if for instance in our running example of (3), we activate $(f(x,z_1) \neq f(y,z_2))$, we would not imply $(x \neq y) \vee (z_1 \neq z_2)$. In such cases, the system only checks the consistency of f when it has established both $(x = y)$ and $(z_1 = z_2)$. This approach considerably affects the efficiency of the overall solver since in the majority of cases not all the equalities between the arguments establish at the same time.

Figure 3a illustrates an assignment sequence to equality atoms following the generic DPLL($\mathcal{T}$) approach. At conflicts 1, 2 and 3 the procedure learns the following clause for i's equal to 4, 3 and 2 respectively:

$$\neg[(x = x_i) \wedge (y = x_i) \wedge (f(x,z_1) \neq f(y,z_2)) \wedge (z_1 = z_2)] \quad (4)$$

Note that as discussed earlier even though the atoms $(f(x,z_1) \neq f(y,z_2))$ and $(z_1 = z_2)$ are added to the model at the root decision level, the "on-demand" process does not imply $(x \neq y)$ accordingly, at the same decision level.

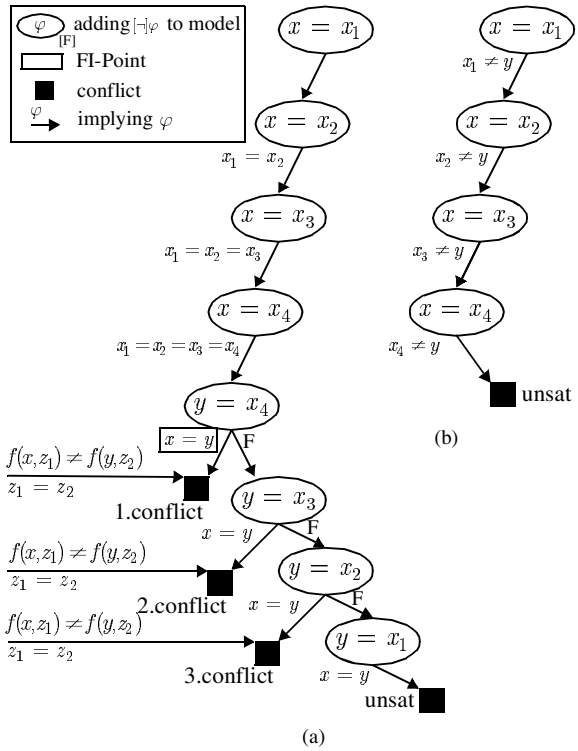
Building on this observation, in this paper, we propose a learning and backtracking scheme specifically designed to accommodate systems involving uninterpreted functions and predicates, i.e. systems of $\mathcal{CLU}$ formulas (common structures in abstract models). In our procedure, we locate all nodes in the hybrid implication graph that have triggered the functional consistency constraint of (2). We refer to these nodes as "Functional Implication Points" or FIP's. At each FIP, we learn a new constraint representing that implication, i.e. the implication due to the functional consistency constraint associated with the pair of occurrences of the uninterpreted function. If we denote the equality atoms associated with each pair of arguments at index i of two occurrences of an n-ary uninterpreted function with E_i and the equality atom between representative variables associated with those function occurrences with F, the FIP learned clause would be:

$$\neg[E_1 \wedge E_2 \wedge \cdots \wedge E_n] \vee F \quad (5)$$

The original learning process continues until it reaches the first UIP to learn a *unit* clause. Note that it is possible that several FIP's are detected between the conflict point and the UIP in the implication graph analysis.

Following on the same example of Figure 3a, upon detecting the *first* conflict, our algorithm locates the FIP, which in this case is the $(x = y)$ node since it triggered a functional consistency constraint resulting in the conflict. The original implication graph analysis continues by replacing the atom at the FIP, $(x = y)$, with its implicants, $(y = x_4)$ and $(x = x_4)$. However, a new clause is also learned representing the exact implication involved at the FIP. In this case, that implication is represented by the clause $\neg[(x = y) \wedge (z_1 = z_2)] \vee (f(x,z_1) = f(y,z_2))$ which is also added to the formula. Following on the implication graph up to its UIP yields learning $\neg[(x_4 = y) \wedge (x = x_4) \wedge (f(x,z_1) \neq f(y,z_2)) \wedge (z_1 = z_2)]$ as the original learned clause as well.

Fig. 3. Assignment/implication graph for FIP and generic learning methods



Consequently in our example, in order to satisfy $\neg[(x = y) \wedge (z_1 = z_2)] \vee (f(x,z_1) = f(y,z_2))$, the solver needs to backtrack to the root decision level where $(f(x,z_1) \neq f(y,z_2))$ and $(z_1 = z_2)$ were added to the model and therefore to imply $(x \neq y)$ at that decision level. Continuing on the overall solving process, the solver concludes the unsatisfiability of the underlying assignments with no more conflicts, as shown in Figure 3b. This saves the process, in this case, two conflict detections and backtrackings. Note that avoiding conflicts in such cases directly correlates with the efficiency of the solver especially when a high number of implications/propagations are involved at each decision level.

As was shown in this example, by learning a new clause associated with the specific instance of the functional consistency constraint at the FIP and consequently including the literal at the FIP, $(x = y)$, in the learned clause, the solver concludes the inequality $(x \neq y)$ as soon as it is established. Unlike the generic method where all combinations of literals yielding $(x = y)$ were checked and rejected, each resulting in a new learned clause in the form of (4), our method as shown in Figure 3b, concludes the dis-equalities due to the same functional consistency constraint on the fly resulting in fewer conflicts. Note that we still adopt the efficient look-up table procedure to apply the functional consistency constraints.

Theorem 1: At least one *unit* clause is learned if clauses at the UIP and FIP's are learned.

Proof: We assume wlog that only one FIP was encountered in the learning process and thus two clauses were learned

as follows:

$$C_U : \quad \neg[A_1 \wedge A_2 \wedge \cdots \wedge A_k] \tag{6}$$

$$C_F : \quad \neg[E_1 \wedge E_2 \wedge \cdots \wedge E_n] \vee F \tag{7}$$

where $A_1, A_2, \cdots, A_k$ represent the atoms at the UIP, $S_F = \{E_1, E_2, \cdots, E_n, F\}$ represents the set of atoms at the FIP associated with the arguments of two occurrences of an uninterpreted function and the equality atom (F) over representative variables associated with those occurrences. We consider that $E_1, E_2, \cdots, E_n$ literals are assigned respectively at decision levels $l_1, l_2, \cdots, l_n$ and we denote the conflict level by l_c and the backtrack level due to (6) by l_b ($< l_c$). Considering that the FIP is identified to be between the conflict point and the UIP in the implication graph, we know at least one of l_i's ($1 \leq i \leq n$) is equal to l_c. The set of those atoms in S_F that are assigned at l_c is denoted by $S_F^c = \{\hat{E}_1, \hat{E}_2, \cdots, \hat{E}_m\}$ ($m \geq 1$). The highest decision level associated with atoms in $(S_F - S_F^c)$ is denoted by l_f. The following cases could occur:

1) If $m = 1$, then both C_U and C_F are unit at l_b and l_f respectively. Therefore at level $min(l_b, l_f)$ at least one unit clause exists.
2) If $m > 1$, then C_F is not unit at any level below l_c and the only unit clause is C_U at decision level l_b.

Therefore in all cases at least one unit clause is learned. ■

Backtracking: Noting that each one of the learned clauses in our procedure could be unit at a different decision level, as shown in Theorem 1, in order to maintain the conformity between the decision levels and the implications, our system backtracks to the *lowest* decision level at which at least one learned clause is unit. The completeness of our overall SMT($\mathcal{CLU}$) is not affected by its FIP learning and is essentially a corollary to the completeness of the generic learning approach of SAT solvers [20] extended to DPLL($\mathcal{T}$).

Corollary 2: Our SMT($\mathcal{CLU}$) approach with learning a clauses representing the UIP cut and clauses of the form of (5) at the FIP's followed by backtracking to the lowest decision level where at least one clause is unit is *complete*.

Proof: Since we also follow the original SAT learning procedure, the same clause is learned in our approach as well. Therefore the completeness of our FIP learning technique is deduced from the completeness of the learning procedure of DPLL($\mathcal{T}$). If the backtrack level is lower in our case compared to the generic case, the completeness of the procedure is not affected since each learning step eliminates a *new* portion of the search space and provides an explanation for why a solution cannot be found in that portion. ■

It is important to note that the advantage of our techniques over the learning schemes of generic DPLL($\mathcal{T}$) methods is the ability to treat uninterpreted functions differently which in many system verification applications using abstraction methods is a major factor in making the process scalable.

V. Experimental Results

We implemented our SMT($\mathcal{CLU}$) decision procedure within our Ario SMT Solver. Ario utilizes MiniSAT [14] as its propositional SAT solver. We adopted the congruence-closure equality checker of [18] and the transitive-closure s-inequality checker of [21], augmented with the deduction and learning techniques of this paper. All experiments were conducted on an AMD Opteron 2.2GHz (8GB RAM) machine.

To evaluate our hybrid implication scheme, we used the Averest benchmark suite[3]. We compared two different techniques to deal with integer atoms within the problem: one that converts all equalities to conjunctions of two s-inequalities and solves the problem using a single s-inequality checker, and the combined approach of this paper that takes advantage of two (equality and s-inequality) checkers. The results of these experiments are presented in Table I. This table shows the clear advantage of adopting an equality checker to solve the equality constraints and combine it with a s-inequality checker when both types of constraints are present in the problem. Note that in these benchmarks, on average, 77.7% of *int-vars* were shared among the two types of integer atoms.

TABLE I
COMPARING THE COMBINED EQUALITY/S-INEQUALITY CHECKING TO SINGLE S-INEQUALITY CHECKING FOR ALL INTEGER ATOMS

benchmark suite	# of instances	avg. # of equalities	avg. # of s-inequalities	speed-up
Binary Search	12	55	48	80.19%
Bubble Sort	52	627	976	32.32%
Fast Max	7	28	71	62.21%
Insertion Sort	34	486	651	44.60%
Linear Search	11	134	0	646.33%
Min Max	12	129	189	63.30%
Partition	19	126	144	47.56%
Selection Sort	47	537	832	38.01%
Sorting Network	49	531	220	54.87%

To evaluate our FIP learning scheme, we used two sets of benchmarks from hardware verification domain each containing a sizable number of uninterpreted functions in their models. The description of these benchmark suites follows.

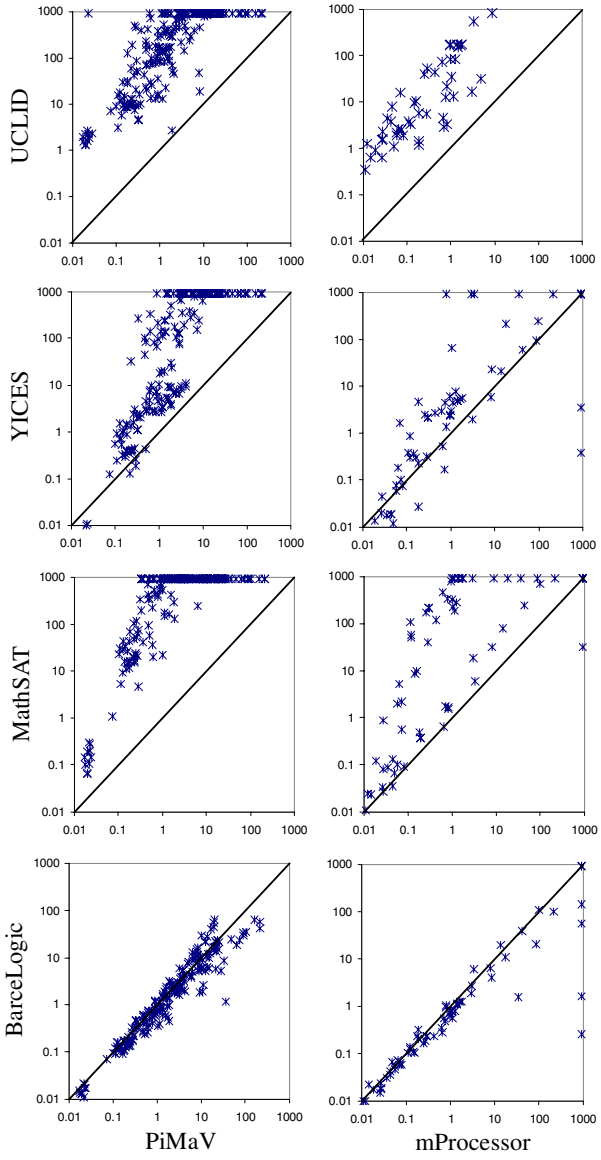
1) Pipelined-Machine-Verification Problems: (PiMaV) [22]. These benchmarks model Well-Founded Equivalence Bisimulation (WEB) refinement to show pipelined machines and their instruction set architecture have the same safety and liveness properties up to stuttering.

2) Micro-processor Verification: [6], [9]. These benchmarks collectively dubbed as "mProcessor", include instances of verifying DLX processors, directory-based cache-coherence protocols with unbounded number of clients, memory units of Elf pipelines against ISA models and out-of-order processors with arithmetic instructions and unbounded resources.

We compared Ario against recent state-of-the-art solvers including *BarceLogicTools* [17] based on DPLL($\mathcal{T}$) method and adopting a congruence-closure algorithm for solving equality formulas with uninterpreted functions [18], *MathSAT v3.3.1*

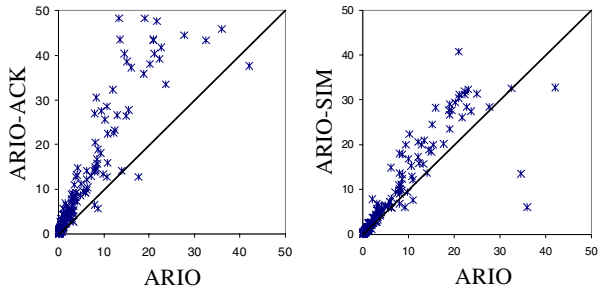
[3]Averest is a framework for the specification, verification, and implementation of reactive systems. More information can be obtained at `http://averest.org/` and the benchmarks are available at the SMT Library (`http://combination.cs.uiowa.edu/smtlib/`).

Fig. 4. Comparing (on logarithmic scale) Ario (X-axis) against UCLID, MathSAT, Yices and BarceLogic solvers (Y-Axis) on Pipelined-Machine Verification (PiMAV) and Micro-processor Verification benchmarks. (time-out = 900 s). A dot above the diagonal line represents an instance that Ario performed better and vice versa.



[10] following the layered approach and *Yices* v0.1 [23]. We also compared our method against UCLID [15] that follows the eager approach and adopts a hybrid encoding technique based on a combination of small-domain (finite instantiation) and per-constraint encodings. The results of these comparisons are demonstrated in Figure 4. Note that our SMT($\mathcal{CLU}$) algorithm performed better than the methods of UCLID, Yices and MathSAT in the majority of the benchmarks in these two suites. In case of BarceLogic, it seems that both solvers performed similar. Note that in these benchmarks, only the FIP learning is being evaluated since the problems do not include any s-inequalities. Comparing BarceLogic against

Fig. 5. Comparing (on linear scale) the FIP learning technique of Ario (X-axis) against Ario with Ackermann reduction (ARIO-ACK) and simple Ario with its FIP technique disabled (ARIO-SIM) (Y-Axis) on all benchmarks collectively. A dot above the diagonal line represents an instance that Ario performed better and vice versa.



Ario "without" FIP learning technique (of this paper) reveals a 24.71% on average better run-times for BarceLogic on these benchmarks. Therefore considering that both Ario and BarceLogic adopt the same congruence-closure algorithm within the DPLL($\mathcal{T}$) framework (fundamentally similar approaches for these problems), it can be concluded that in the majority of cases, the performance difference is mainly due to their implementations which is offset by adopting our FIP learning technique, proving the effectiveness of this method. Note that our FIP learning technique is less effective in mProcessor benchmarks since in these instances, the majority of conflicts originate from Boolean structure of the problems and not the inconsistencies of the uninterpreted functions. This also explains Ario time-outs when compared to Yices, MathSAT and BarceLogic and can be seen in the last 9 rows of Table II that correspond to this benchmark suite.

In order to further analyze the effectiveness of our FIP learning technique, we disabled this algorithm within Ario, yielding a new version of the solver, called Ario-SIM. We also implemented the Ackermann reduction method [16] within Ario that removes all occurrences of uninterpreted functions in the formulas in advance (in the pre-processing phase), yielding a simple formula in equality logic. We refer to the latter version as Ario-ACK. Figure 5 illustrates the comparison of Ario against these two versions on those benchmarks of PiMaV and mProcessor suites that all methods could solve within 50 seconds (in most of the harder instances, only Ario could solve the problem). As it is shown in this figure, in the majority of benchmarks, FIP learning considerably improved the performance of the solver. Note that all these benchmarks include a high number of uninterpreted functions and predicates which makes them favorable to our FIP learning technique.

Table II demonstrates the effectiveness of FIP learning in the learning process on a representative set of our benchmarks. As it is shown the number of additional clauses learned due to FIP's is very low compared to total number of learned clauses. This further illustrates the fact that in many of these instances few of the functional consistency constraints due to the un-

TABLE II

DATA ON THE PERFORMANCE OF FIP LEARNING

benchmark	# of detected FIP's	increase in # of learned clauses	speed-up
g10idw	58	0.46%	54.97%
g10	23	0.87%	72.61%
g8bidw	29	0.64%	15.10%
g9idw	34	0.66%	55.38%
c10bid_i	232	1.11%	32.38%
c7nidw_i	168	1.66%	18.70%
c8b	81	4.14%	27.20%
c9b_i	59	2.83%	36.17%
f10i	280	1.65%	52.31%
f8idw	327	1.18%	48.64%
8stage-flush	120	3.08%	7.03%
9stage-flush	201	2.56%	29.41%
cxs-bp-ex-inp	64	3.29%	35.25%
cxs-bp-ex-safety	45	2.12%	39.98%
cxs-bp-ex	76	3.77%	39.55%
cxs-bp	68	4.73%	22.86%
cxs-safety	32	4.17%	10.17%
fxs-bp-ex-inp-safety	103	5.75%	8.67%
fxs-bp-ex-inp	70	3.78%	13.74%
fxs	62	5.05%	14.45%
cache.inv16	0	0.00%	-2.72%
ooo.rf11	178	0.12%	25.60%
dlx1c.rwmem	22	3.69%	-5.77%
dlx1c	19	4.18%	18.99%
elf.rf7	3	4.29%	6.90%
elf.rf8	7	3.68%	17.78%
ooo.rf10	96	0.46%	35.25%
ooo.tag10	0	0.00%	-1.08%
ooo.tag12	0	0.00%	-0.50%

interpreted functions/predicates actually get involved in the conflicts (also the rationale for not propagating dis-equalities among representative variables to imply dis-equalities among their arguments). This is also the reason behind the poor performance of Ario-ACK which requires pre-processing all occurrences of uninterpreted functions/predicates. Additionally, this table demonstrates the low overhead of FIP learning since very few number of extra clauses are learned.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented a hybrid SAT-based decision method for $\mathcal{CLU}$ logic. In our approach, we introduced a tight adaptive integration of both an equality and a separation-inequality checker within the SAT solver. This was made possible by adoption of our hybrid implication scheme. We also proposed an efficient learning method specific to conflicts involving uninterpreted functions. We exhaustively evaluated the performance of these techniques on a wide range of benchmarks in hardware verification.

Our experiments clearly demonstrated the effectiveness of application-based techniques (such as our approach applied to $\mathcal{CLU}$ formulas) to provide promising frameworks to solve the scalability problem in system verification. Obviously, such algorithms if utilized in larger settings such as CEGAR and when combined with advanced refinement techniques could bring us closer to verifying systems in industrial scales. Our contributions in this paper should be regarded as a modest contribution along this road.

ACKNOWLEDGEMENT

This work was funded in part by the National Science Foundation under ITR grant No. 0205288.

REFERENCES

[1] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. London, UK: Springer-Verlag, 1982, pp. 52–71.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Conference on Computer Aided Verification*, 2000, pp. 154–169.

[3] E. M. Clarke, O. Grumberg, M. Talupur, and D. Wang, "Making predicate abstraction efficient: How to eliminate redundant predicates." in *Conference on Computer Aided Verification*, 2003, pp. 126–140.

[4] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Refinement strategies for verification methods based on datapath abstraction," in *ASP-DAC '06: Asia South Pacific design automation*, 2006, pp. 19–24.

[5] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Proceedings of the 14th International Conference on Computer Aided Verification*, 2002, pp. 78–92.

[6] S. K. Lahiri, S. A. Seshia, and R. E. Bryant, "Modeling and verification of out-of-order microprocessors in UCLID," in *FMCAD*, 2002, pp. 142–159.

[7] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant, "Automatic discovery of API-level exploits," in *ICSE*, May 2005, pp. 312–321.

[8] C. Tinelli, "A DPLL-based calculus for ground satisfiability modulo theories," in *Proceedings of the 8th European Conference on Logics in Artificial Intelligence*, 2002, pp. 308–319.

[9] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings," in *Proc. Intl. Workshop on Constraints in Formal Verification*, 2002.

[10] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani, "An incremental and layered procedure for the satisfiability of linear arithmetic logic." in *TACAS*, 2005, pp. 317–333.

[11] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast decision procedures," in *CAV*, 2004, pp. 175–188.

[12] H. M. Sheini and K. A. Sakallah, "A SAT-based decision procedure for mixed logical/integer linear problems." in *CPAIOR*, 2005, pp. 320–335.

[13] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with exhaustive theory propagation and its application to difference logic." in *Conference on Computer Aided Verification*, 2005, pp. 321–334.

[14] N. Eén and N. Sörensson, "An extensible SAT-solver." in *SAT*, 2003, pp. 502–518.

[15] S. A. Seshia, S. K. Lahiri, and R. E. Bryant, "A hybrid SAT-based decision procedure for separation logic with uninterpreted functions," in *DAC*. New York, NY, USA: ACM Press, 2003, pp. 425–430.

[16] W. Ackermann, "Solvable cases of the decision problem," in *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1954.

[17] R. Nieuwenhuis and A. Oliveras, "Decision procedures for SAT, SAT modulo theories and beyond. the BarcelogicTools." in *Conf. Logic for Programming, Artificial Intelligecne and Reasoning*, 2005, pp. 23–46.

[18] ——, "Proof-Producing Congruence Closure," in *Proceedings of the 16th Int'l Conf. on Term Rewriting and Applications*, 2005, pp. 453–468.

[19] C. W. Barrett and S. Berezin, "CVCLite: A new implementation of the cooperating validity checker category b." in *CAV*, 2004, pp. 515–518.

[20] P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, 1999.

[21] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap, "Beyond finite domains," in *Workshop on Principles and Practice of Constraint Programming*, 1994, pp. 86–94.

[22] P. Manolios and S. K. Srinivasan, "A parameterized benchmark suite of hard pipelined-machine-verification problems." in *CHARME*, 2005, pp. 363–366.

[23] L. de Moura, "Yices," 2005. [Online]. Available: http://fm.csl.sri.com/yices/