Tighter Integration of BDDs and SMT for Predicate Abstraction

A. Cimatti*, A. Franzen*, A. Griggio†, K. Kalyanasundaram*, M. Roveri*
* FBK-irst, Trento, Italy {cimatti,franzen,krishnamani,roveri}@fbk.eu
† University of Trento DISI, Trento, Italy griggio@disi.unitn.it

***Abstract*—We address the problem of computing the exact abstraction of a program with respect to a given set of predicates, a key computation step in Counter-Example Guided Abstraction Refinement. We build on a recently proposed approach that integrates BDD-based quantification techniques with SMT-based constraint solving to compute the abstraction. We extend the previous work in three main directions. First, we propose a much tighter integration of the BDD-based and SMT-based reasoning where the two solvers strongly collaborate to guide the search. Second, we propose a technique to reduce redundancy in the search by blocking already visited models. Third, we present an algorithm exploiting a conjunctively partitioned representation of the formula to quantify. This algorithm provides a general framework where all the presented optimizations integrate in a natural way. Moreover, it allows to overcome the limitations of the original approach that used a monolithic BDD representation of the formula to quantify. We experimentally evaluate the merits of the proposed optimizations, and show how they allow to significantly improve over previous approaches.**

I. Introduction

Computing the abstraction of a program with respect to a given set of predicates is a crucial step in CEGAR based verification [1]. In fact, the problem has been given a substantial amount of interest in recent years, as discussed in [2], [3], [4], [5], [6], [7], [8].

In this paper, we propose a new approach, where we push the integration between BDD-based and SMT-based reasoning to new limits. We address some of the limitations of [8], we extend it in three main directions, and we perform an experimental evaluations to understand the merits of the proposed techniques. First, we present a much tighter integration between the SMT solver and the BDD-based quantification. Second, we propose a mechanism similar to blocking clauses to avoid re-visiting portions of the abstract transition relation. Third, we replace a monolithic BDD-based representation with an implicit conjunctively partitioned representation.

The paper is structured as follows. In Section II we outline some background and we outline the algorithm of [8]. In Section III we discuss the blocking visited models optimization. In Section IV we discuss partitioned BDD representation, while in Section V we discuss the cooperation schema. In Section VI we describe related work, and in Section VII we experimentally evaluate the proposed techniques. Finally, in Section VIII we draw some conclusions and outline future work.

II. Technical Preliminaries

Our setting is standard first order logic. We consider a signature to be composed by individual constants and variables, function symbols, Boolean variables, and predicate symbols. A term is either a constant, a variable, or the application of a function symbol of arity n to n terms. A theory constraint (also called a theory atom) is the application of a predicate symbol of arity n to n terms. We use $\vec{x}, \vec{y}, \vec{v}, \ldots$ for vectors of individual variables. Terms and formulae by $e, e', e_1, e_2, \ldots,$ $c, c_1, c_2, \ldots$ for theory constraints, $c(x)$ to stress the dependence of c on x, and $c(\vec{x})$ to stress the dependence on $\vec{x}$, $P, P', P_1, P_2, \ldots, Q, Q', Q_1, Q_2, \ldots$ for Boolean variables, and $\vec{P}, \vec{Q}, \ldots$ for vectors of Boolean variables; we write $\vec{P_i}$ for the i-th variable in $\vec{P}$. We write $\Phi(Q)$ to highlight the fact that Q occurs in Φ. $\phi[e/e']$ denotes the *substitution* of every occurrence of e in ϕ with e'. If $\vec{e}$ and $\vec{e}'$ are vectors of (either individual or Boolean) expressions of the same size, we write $\phi[\vec{e}/\vec{e}']$ for the parallel substitution of every occurrence of $\vec{e}_i$ (the i-th element of $\vec{e}$) in ϕ with $\vec{e}'_i$ (the i-th element of $\vec{e}'$). We use Boolean quantification (i.e. quantification over Boolean variables) $\exists Q.(\Phi(Q))$ as a shortcut for $\Phi[Q/\top] \vee \Phi[Q/\bot]$.

We use the standard semantic notion of interpretation and satisfiability. We call truth assignment for a set (vector) of atoms $\vec{Q}$ a total function $\mu : \vec{Q} \rightarrow \{\top, \bot\}$. SAT problem is well-known and has been an area of research for more than two decades. The more general problem of SMT (satisfiability modulo a background theory) corresponds to deciding whether there exists an interpretation in the theory that satisfies the formula. [9] makes a thorough discussion on SMT solvers. For ease of explanations we follow the notations of [8] in this paper.

Predicate Abstraction [10] is one of the most widely used abstraction techniques for Model Checking within the automated Counter-Example Guided Abstraction Refinement (CEGAR) [1] framework. The abstract program is constructed over a given set of predicates $\gamma_i(\vec{x})$ which are relations over variables $\vec{x}$ of the concrete program. Each predicate is represented by a Boolean variable V_i in the abstract program. Intuitively, each variable V_i partitions the concrete state space into two sets: the set of states satisfying γ_i, ($V_i = \mathit{true}$), and those that do not ($V_i = \mathit{false}$). The abstract program tracks the behaviour of the predicates. Thus, an abstract state is associated with the set of concrete states corresponding to the intersection of the sets corresponding to each abstract literal. The transition relation of the abstract program is such that two abstract states are related iff there exist two concrete states in the respective concretizations that are in the concrete transition relation.

Predicate abstractions aims at computing a propositional formula equivalent to $\exists \vec{x}.(\Phi(\vec{x}) \wedge \bigwedge_i V_i \leftrightarrow \gamma_i(\vec{x}))$ where $\vec{x}$

are the variables we want to abstract away using predicates $V_i \leftrightarrow \gamma_i(\vec{x})$, and $\vec{V}$ is the set of Boolean variables to be retained, also called *important variables*. In the following, we denote with $\vec{c}$ the vector of theory constraints occurring either in Φ or in some γ_i.

In [8], a naive algorithm to compute predicate abstractions for a first order logic formula Φ over a set of Boolean predicates V_i, is discussed. The algorithm builds on the following steps. First, for each theory constraint $c_i(\vec{x})$ in $\vec{c}$ a fresh Boolean (BDD) variable Q_i, called its Boolean abstraction, is created. All the Q_i are grouped in $\vec{Q}$, and the abstraction bi-jection $\mathcal{A}$ is defined as $\{\langle Q_i \ . \ c_i(\vec{x})\rangle\}$. Second, the Boolean formula $(\Phi(\vec{x}) \wedge \bigwedge_i V_i \leftrightarrow \gamma_i(\vec{x}))[\vec{c}/\vec{Q}]$ in the $\vec{V}$ and $\vec{Q}$ is constructed by replacing each theory atom $c_i(\vec{x})$ with the corresponding Boolean abstraction Q_i from the matrix of the above formula. Third, standard techniques are used to construct the corresponding BDD representation, also denoted as $\Phi(\vec{V}, \vec{Q})$. Finally, the algorithm depicted in Fig. 1 is used to compute the abstraction.

```
 1: function BddThAbstract($b$, $C$, $V$)
 2:     if $(b = \top) \vee (b = \bot)$ then return $b$
 3:     $v$ := TopVar($b$);
 4:     if BooleanAtom($v$) then
 5:         $tt$ := BddThAbstract(BddThen($b$), $C$, $V$)
 6:         if $v \in V$ then
 7:             $ee$ := BddThAbstract(BddElse($b$), $C$, $V$)
 8:             return BddITE($v, tt, ee$)
 9:         else
10:             $ee$ := BddThAbstract(BddElse($b$), $C$, $V$)
11:             return BddOr($tt, ee$)
12:     else
13:         $c_v$ := VarToConstraint($v$);
14:         if BddThen($b$)= $\bot$ or ThInconsistent($C$, $c_v$) then
15:             $tt$ := $\bot$
16:         else
17:             $tt$ := BddThAbstract(BddThen($b$), $C \cup \{c_v\}$, $V$)
18:         if BddElse($b$)= $\bot$ or ThInconsistent($C$, $\neg c_v$) then
19:             $ee$ := $\bot$
20:         else
21:             $ee$ := BddThAbstract(BddElse($b$), $C \cup \{\neg c_v\}$, $V$)
22:         return BddOr($tt, ee$)
23: end function
```

Fig. 1. BDD-based quantification modulo theories..

`BddThAbstract` interleaves Boolean quantification and pruning modulo theory, and is recursively defined over the structure of the BDD b representing the formula to be quantified. The second argument C is a set of constraints, called the *context of simplification*, while the third one is the set of important variables. The $\mathcal{A}$ mapping between theory constraints and their Boolean variables is assumed to be globally available. The algorithm is best explained as an extension of the existential quantification in the purely Boolean case, i.e. when the $\mathcal{A}$ mapping is empty, so that the `BooleanAtom` test (line 4) always returns true. The lines from 12 to 22 are never executed, and the algorithm boils down to standard existential quantification for BDDs [11]. In the base case (line 2), if b is either $\top$ or $\bot$, then it is simply returned. Otherwise, recursive calls are applied both to the then (line 5) and the else (lines 7, and 10) co-factors w.r.t. variable $v = \texttt{TopVar}(b)$ of BDD b. If the top level variable $v = \texttt{TopVar}(b)$ is important, then it must not be quantified, and the results of the recursive calls are combined into an if-then-else node, otherwise disjunction is applied to the two results of the recursive calls (lines 9-11). In the more interesting case, where some variables are indeed theory constraints, a form of pruning is applied. This pruning attempts to ensure theory-consistency of the traversed paths. The key idea is the simplification context, i.e. the set of theory constraints that "get activated" when descending from the root to the node b. If the current variable is the abstraction of a theory atom, then the current context can be extended with either a positive or negative constraint, depending on the branch we expand first. However, in order for either expansion to lead to a model, it has to be theory consistent: if either extension is inconsistent with the current context, then the evaluation can be safely pruned. If the context extended with the positive constraint is not satisfiable (line 14), then $\bot$ is assigned for the right branch tt. In fact, there is no way the path can be extended to a theory-consistent assignment. In case of consistency, the context is extended $C \cup \{c_v\}$ and recursion on the then co-factor is started. The else branch is dealt with similarly. The results can be recombined with disjunction, since the Boolean abstractions of the constraints have been quantified out.

III. Blocking Visited Models

The algorithm in Fig. 1 is such that the same Boolean model may be repeatedly generated: while computing the model of important variable below b considering variable v being false we may end-up in models that are entailed by any model in tt. Suppose we are exploring $A \vee B$; once A has been computed, it is safe to explore $B \wedge \neg A$. This reduces to conjoining on the fly with the negation of what has been accumulated in the first branch that has been explored. This is exploited in the algorithm as follows.

Let $tt = \texttt{BddThAbstract}(\texttt{BddThen}(b), V)$ be the set of models of important variables below b considering variable v being true (lines 5, 17). While computing the model of important variable below b considering variable v being false we want to consider only those models that are not entailed by any model in tt. This is achieved by recurring on the BDD resulting from the conjunction of the co-factor of BDD b where variable v is considered false and the negation of BDD tt. (The negation of BDD tt represents all the models that have not yet been explored in the co-factor of BDD b when variable v is considered true.) This is justified by the fact that if v is a non important variable then:

$$\begin{aligned}\exists v.\Phi(\vec{x}) \quad &\doteq (\Phi(\vec{x})[v/\top] \vee \Phi(\vec{x})[v/\bot]) = \\ &(\Phi(\vec{x})[v/\top] \vee (\Phi(\vec{x})[v/\bot] \wedge \neg\Phi(\vec{x})[v/\top])).\end{aligned}$$

This optimization (referred in the following as *D'Agostino optimization*) is of Boolean nature, and naturally generalizes to the SMT case. The use of the negated BDD while recurring on the else branch mimics the effect of adding blocking-clauses, typical of SAT based approaches. The resulting algorithm is obtained by replacing in Fig. 1 the statement $\texttt{BddElse}(b)$ at lines 10 and 21 with $\texttt{BddAnd}(\texttt{BddElse}(b), \texttt{BddNot}(tt))$. In practice, the implementation of this optimization requires some care. At each recursive step, we conjoin the negation of the

```
1: function BddListThAbstract($\vec{b}$, $C$, $V$)
2:     if BddListIs$\bot$($\vec{b}$) then return $\bot$
3:     if BddListIs$\top$($\vec{b}$) then return $\top$
4:     $v$ := TopVar($\vec{b}$);
5:     $\vec{t}$ := BddThenAtLevel($\vec{b}$, $v$);
6:     $\vec{e}$ := BddElseAtLevel($\vec{b}$, $v$);
7:     if BooleanAtom($v$) then
8:         $tt$ := BddListThAbstract($\vec{t}$, $C$, $V$)
9:         if $v \in V$ then
10:            $ee$ := BddListThAbstract($\vec{e}$, $C$, $V$)
11:            return BddITE($v, tt, ee$)
12:        else
13:            if ($DAgostino$) then
14:                $ee$ := BddListThAbstract($\vec{e}$ :: BddNot($tt$), $C$, $V$)
15:            else
16:                $ee$ := else BddListThAbstract($\vec{e}$, $C$, $V$)
17:            return BddOr($tt, ee$)
18:    else
19:        $c_v$ := VarToConstraint($v$);
20:        if BddListIs$\bot$($\vec{t}$) or ThInconsistent($C$, $c_v$) then
21:            $tt$ := $\bot$
22:        else
23:            $tt$ := BddListThAbstract($\vec{t}$, $C \cup \{c_v\}$, $V$)
24:        if BddListIs$\bot$($\vec{e}$) or ThInconsistent($C$, $\neg c_v$) then
25:            $ee$ := $\bot$
26:        else
27:            if ($DAgostino$) then
28:                $ee$ := BddListThAbstract($\vec{e}$ :: BddNot($tt$), $C \cup \{\neg c_v\}$, $V$)
29:            else
30:                $ee$ := BddListThAbstract($\vec{e}$, $C \cup \{\neg c_v\}$, $V$)
31:        return BddOr($tt, ee$)
32: end function
```

Fig. 2. The partitioned quantification with D'Agostino optimization.

result of the then co-factor recursion with the else co-factor and then recur on the BDD resulting from this operation. The use of this standard BDD operation results in a considerable overhead in terms of unique table and cache look-ups.

IV. Using Partitioned BDDs

One of the main bottlenecks in the algorithm in [8] (see also Fig. 1) is that the BDD being traversed is monolithic, and thus subject to space explosion [12]. In this section we generalize this algorithm to the case of quantification of a list of (implicitly conjoined) BDDs. TopVar returns the top most variable over the list of BDDs. BddThenAtLevel($\vec{b}, v$) builds a new list of BDDs where BddThen is applied to all the BDDs whose top level variable is equal to the one given as argument (v), while those that do not satisfy this condition are simply copied in the new list. BddElseAtLevel($\vec{b}, v$) is similar, but it applies BddElse. BddListIs$\bot$ returns true if the list of BDDs contains at least one instance of the $\bot$ BDD. Finally, BddListIs$\top$ returns true of the list of BDDs is composed only of instances of the BDD $\top$.

The algorithm is described in Fig. 2. It is very similar to the monolithic one: each BDD in the list is processed according to the highest top variable. Then, we recur on the respective left and right branches. For the BDDs that have top variable below the highest one, we simply defer their evaluation. The D'Agostino optimization is described at lines 14, 28 as a conjunction from a logical view point. In practice it is implemented as a primitive that adds a new element to a list of BDDs, applying a series of simplifications (e.g. combining with BddAnd() the new element with all the other ones, or applying care-set simplifications [13]). See [13] for a survey of such simplification techniques. This approach treats the care set simplification as a conjunct. Whether the list is extended with one more element, or conjuncted (i.e. BddAnd() with every element), or subjected to care-set simplification, is left to the underlying implementation. The important characteristic is that at each level of the recursion we only consider BDDs whose variables are guaranteed to be at higher levels than the level of the currently considered variable.

The disadvantage is that at each recursion on the else co-factor we potentially enlarge the list of BDDs by one element to keep track of the partial result on the then co-factor. However, while recurring it is also the case the list of BDDs will simplify because of reaching constants (e.g. we can remove any element of the list which is $\top$).

V. Hybrid Abstraction Engine

We now present a novel schema for integrating BDD-based traversal with SMT techniques. In [8], SMT solver is restricted to deal only with the theory, completely unaware of the boolean part of the problem, and subordinate to the traversal carried out by the BDD-driven traversal. Here we radically change the schema, by turning it into a hybrid cooperation engine. The SMT solver is now given the whole problem in input (not shown in the algorithm), and as in [8], the splitting is driven by the BDD traversal. We retain all the features of the SMT solver, in particular unit propagation and conflict analysis, and we discard the variable selection heuristics. This upgrade is dictated by several considerations.

First, the partitioned representation may hide important inferences that would be obvious with a monolithic representation. The most obvious case is when the BDD list represents an inconsistent formula without any of the BDDs being false; similar considerations hold for the truth values of literals. The SMT solver is also dealing with a partitioned representation, but it is not bound to a fixed variable ordering; unit propagation can be seen as a way to "look ahead" paying a low cost, and overtake the BDD-based traversal order. In particular, we use the unit propagation in the SMT solver (in addition to the theory consistency checking) to detect inconsistencies, and to identify literals that are entailed by the current partial assignment. Inconsistencies can be used to stop the search of inconsistent partitioned BDD configurations, while deductions, to co-factor each of the BDDs, thus possibly enabling further reductions.

Second, the traversal in [8] is essentially limited to chronological backtracking, and the conflict analysis carried out by the SMT solver can be exploited to implement back-jumping.

Finally, D'Agostino optimization prevents repeated search on the models of the abstract space, without paying the price of a possibly large representation in form of blocking clauses. This point has also been discussed in [5].

The status of the SMT is a set of clauses (either original or learned), an implication graph, and a stack of currently assigned literals. After initialization and unit propagation, the clauses are the problem clauses, the implication graph contains the implications at level 0, and the stack contains the implied literals.

The hybrid predicate abstraction algorithm is depicted in Fig. 3.

The BDD enumerator and the SMT solver interact according to the following interface:

- TCC.IsAssigned(v) and BDD.IsAssigned$(\vec{b}, v)$ return either true (1), false (0), if the variable has an enforced value in the SMT solver and in the list of BDDs $\vec{b}$ respectively. Otherwise they return *undef* (X).
- TCC.Assume(v, s) adds a literal on the stack positive ($s = +$) or negative ($s = -$). It requires TCC.IsAssigned$(v) = X$). This information is unit propagated and in turn may return newly inferred literals.
- TCC.UndoAssumption(v, s) removes from the stack a previously assumed literal (either positively $s = +$, or negatively $s = -$).
- TCC.IsConsistentComplete() returns true if stack is consistent, otherwise false and level to back-jump.
- TCC.IsConsistentApprox() uses an approximate consistency check: it returns true or false and level to back-jump if the approximate consistency check was able to conclude, otherwise it returns unknown.
- TCC.IsSatisfiable() returns true if the stack and the rest of the SMT formula are satisfiable, otherwise false and level to back-jump if not satisfiable.
- TCC.GetUndoLevel() return the level at which the literal causing the conflict has been assumed.

The SMT status, corresponding to the active (assumed and implied) literals, is managed in a stack-based manner. The BDD-based enumerator can ask the TCC to extend its status by assuming a literal, or to undo the last assumption(s). The enumerator can ask the SMT whether its status is consistent, and ask for the current value of a variable. Depending on the theory, consistency checking can be carried out with an incomplete (possibly cheaper) procedure; in such case it is important to carry out a complete check when a complete model is found.

The SMT solver may detect reasons for inconsistencies (conflict sets), and turn them into conflict clauses. Then, by carrying out conflict analysis it can tell the enumerator the point of backtracking necessary to undo the inconsistency. Theory solvers can carry out theory deductions, and combine them with boolean constraint propagation over the active clauses. It is thus possible that a variable (be it boolean or theory) that is unassigned in the current BDD path must in fact have a value. The role of BCP is to propagate consequences of the current assumptions, e.g. based on previously discovered theory inconsistencies. Notice that the SMT solver and the BDD-based solver are no longer in sync - for instance, the D'Agostino optimization may force a certain literal in the BDD solver, while its negation could be implied in the SMT solver by theory reasoning.

This novel approach opens to several further optimizations. The theory solvers can deduce theory lemmas that can be used to co-factor the input problem. If the SMT detects implied literals (either by theory or boolean reasoning), the choice that we currently implement is to delay taking them into account

```
1: function BddListThAbstract($\vec{b}$, $C$. $V$)
2:     if BddListIs⊥($\vec{b}$) then return ($\bot$, -1)
3:     if BddListIs⊤($\vec{b}$) then
4:         if TCC.IsConsistentComplete() then return ($\top$, -1)
5:         else return ($\bot$, TCC.GetUndoLevel())
6:     if not TCC.IsConsistentApprox() then return ($\bot$, TCC.GetUndoLevel())
7:     if NoMoreImportantVariables($\vec{b}, V$) then
8:         if TCC.IsSatisfiable() then return ($\top, -1$)
9:         else return ($\bot$, TCC.GetUndoLevel())
10:    $v$ := TopVar($\vec{b}$);
11:    $bv$ := BDD.IsAssigned($\vec{b}, v$);
12:    $sv$ := TCC.IsAssigned($v$);
13:    $tt := ee := (\bot, -2)$
14:    $c_v$ := VarToConstraint($v$);
15:    $\vec{t}$ := BddThenAtLevel($\vec{b}, v$);
16:    $\vec{e}$ := BddElseAtLevel($\vec{b}, v$);
17:    if $(((bv = 1) \wedge (sv = 1)) \vee ((bv = X) \wedge (sv = 1)))$ then
18:        $tt$ := BddListThAbstract($\vec{t}, C, V$))
19:    else if $(((bv = 0) \wedge (sv = 0)) \vee ((bv = X) \wedge (sv = 0)))$ then
20:        $ee$ := BddListThAbstract($\vec{e}, C, V$))
21:    else if $((bv = 1) \wedge (sv = X))$ then
22:        TCC.Assume($v, +$)
23:        $tt$ := BddListThAbstract($\vec{t}, C \cup \{c_v\}, V$))
24:        if HaveToUndo($tt, v$)) then TCC.UndoAssumption($v, +$)
25:    else if $((bv = 0) \wedge (sv = X))$ then
26:        TCC.Assume($v, -$)
27:        $ee$ := BddListThAbstract($\vec{e}, C \cup \{\neg c_v\}, V$))
28:        if HaveToUndo($ee, v$) then TCC.UndoAssumption($v, -$)
29:    else if $((((bv = 1) \wedge (sv = 0)) \vee ((bv = 0) \wedge (sv = 1)))$ then
30:        $tt := ee := (\bot, -1)$
31:    else if $((bv = X) \wedge (sv = X))$ then
32:        TCC.Assume($v, +$)
33:        $tt$ := BddListThAbstract($\vec{t}, C \cup \{c_v\}, V$))
34:        TCC.UndoAssumption($v, +$)
35:        TCC.Assume($v, -$)
36:        if $(DAgostino \wedge v \notin V)$ then
37:            $ee$ := BddListThAbstract($\vec{e}$ :: BddNot($tt.bdd$), $C \cup \{\neg c_v\}, V$))
38:        else
39:            $ee$ := BddListThAbstract($\vec{e}, C \cup \{\neg c_v\}, V$))
40:        if HaveToUndo($ee, v$) then TCC.UndoAssumption($v, -$)
41:    return Recombine($tt, ee, v, V$)
42: end function
```

```
1: function Recombine($tt, ee, v, V$)
2:     if $v \in V$ then $r.bdd$ := BddITE($v, tt.bdd, ee.bdd$)
3:     else$r.bdd$ := BddOr($tt.bdd, ee.bdd$)
4:     if $((tt.bt = -1) \vee (ee.bt = -1))$ then $r.bt := -1$
5:     else if CompareBT($tt.bt, ee.bt$) then $r.bt = ee.bt$
6:     else$r.bt = tt.bt$
7:     return $r$
8: end function
```

Fig. 3. The hybrid abstraction algorithm.

until the corresponding level is reached in the enumeration, and to backjump accordingly. Another possibility would be to simplify the remaining BDD according to the corresponding value before continuing with enumeration.

The search also exploits the levels of the BDD. In particular, if we see that a certain node is below the level of the last important variable (line 7), then the result corresponds to simply checking the satisfiability of the sub-tree according to the current stack (i.e. to call TCC.IsSatisfiable()).

Most of the tricks in the BDD package (e.g. constant time negation based on pointer complementation) can be retained without changes. However, this is not the case for *memoization*. Similar to the approach in [8] we disregard memoization on the grounds that some memoization is carried out in the recursive calls to disjunction on unimportant variables.

VI. Related Work

The work closest to our work is [8]. The approach presented in this paper achieves a much tighter integration between

BDD algorithms and SMT reasoning, that employs the best of BDD algorithms and SMT optimization routines. We propose a mechanism similar to blocking clauses to avoid re-visiting portions of the abstract transition relation. We also replace a monolithic BDD-based representation with an implicit conjunctively partitioned representation, which helps in scalability.

BDDs are used as model enumerators in the first versions of the Harvey [14] decision procedure. BDD simplification with respect to a background theory are presented in [15]; the work is limited to abstract data types, and does not deal with quantification. Decision procedures for computing abstractions have been explored in [2], [3], [4]. [5] improves over them by lifting DPLL-based quantification to the case of SMT, but inherits the model explosion problem.

VII. Experiments

We tested the proposed algorithms within the NuSMT system and framework presented in [8]. The system provides an implementation of the CEGAR loop integrating SMT techniques (specifically, the MathSAT SMT solver [16]) and the NuSMV [17] model checker. NuSMT uses the NuSMV language extended to deal with real-valued and integer-valued variables. This allows us to represent the concrete program with formulae characterizing the set of initial states, the invariants, and the transition relation. The abstract program is a finite state program. NuSMT implements a full-blown CEGAR loop, with counterexample refinement and predicate discovery. In this paper we focus only on the predicate abstraction part.

We compared the algorithms presented in this paper against the one presented in [8]. We conducted experiments on networks of hybrid automata of different sizes (HAN$n-s-t-v$), with constraints in linear arithmetic over the reals. A test case family with name "HANn" consists of a composition of n hybrid automata, each of them having s locations and t transitions. We experimented with $n \in \{2, 3, 4\}$ and with $(s, t) \in \{(5, 10), (5, 30), (10, 10), (10, 30)\}$. Each state of the automata is associated with an invariant, while each transition has both a precondition and an effect; all of them are formulae in linear arithmetic over v variables with $v \in \{5, 10\}$. The experiments were run on a 3GHz Intel(TM) Xeon(TM) Dual Processor running Linux equipped with 4GB of RAM. We fixed a memory limit of 2 GB and a CPU time limit of 1 hour. The plots shown have the number of instances in our benchmark set along the x-axis and the cumulative abstraction time (in logarithmic scale) along the y-axis. When we mention partitioned approach (referred as Bddarray in the following), unless otherwise specified we use a partition size (also referred to as threshold) of 3000 BDD nodes for the elements of the Bddarray. In Figures 4(b), (e) and (f), we use a relatively 'good' variable order, described in the following.

We restrict ourselves to the comparison of the most important parameters of the algorithm. The relevant files of experiments are available at http://es.fbk.eu/people/roveri/date2010/.

1) Bddarray Approach: The plots in Fig. 4(a) and (b) show the comparison of Bddarray approach against [8]. Fig. 4(b) corresponds to using a reasonably good BDD ordering. We see that Bddarray approach performs consistently better.

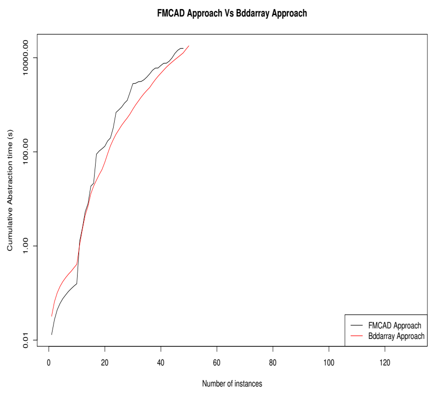
2) Impact of D'Agostino Optimization: The plot in Fig. 4(c) and (d) show the impact of D'Agostino simplification (blocking clause optimization) on the performance of abstraction algorithm using Bddarray approach (with monolithic and partitioned approach with threshold set to 3000 respectively), and the impact of this optimization is apparent.

3) Effect of Partitioning: A 'good' partition size ideally depends on the structure of the problem. To illustrate that there is an effect of threshold limit on performance, we experimented with different threshold limits to show the effect. We notice from Fig 4(e) that, a threshold of 300 BDD nodes appears to be a bad choice as it does not complete as many instances as the other thresholds within the specified time limit.

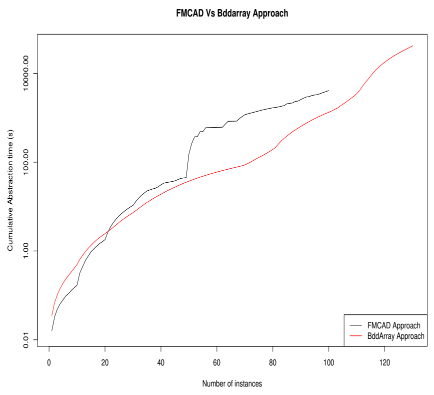
4) Impact of Variable Order: As with any technique involving BDD traversals, the performance of our approach crucially depends on variable order. However, when we are dealing with predicates (important variables) and theory variables, an ordering heuristic would be to choose the predicates to be at the top of the variable order and the remaining ones below. This is because of the following reasons. As the BDD traversal happens, we are actually accumulating models. This traversal happens according the variable order specified in an order file. When an important variable is encountered and if its value is deduced to be true (false), we are ruling out a portion of the state space corresponding to the value being false (true) respectively. In the set of benchmark instances we work with, when we encounter non-important variables at the beginning of the traversal, when it has a deduced value, we are basically ruling out only a very small portion (a small set of points that do not satisfy the constraints corresponding to the non important variables) of the search space. In Fig. 4(f), we compare the effect of different variable orderings on performance. The experiments were run with a threshold limit of 3000 BDD nodes and with D'Agostino optimization turned on. In 4(f), we refer to as 'good variable order', the experiments with important variables on the top of the variable order, and 'bad variable order' the ones which have the important variables at the bottom of order. With 'bad variable order', all except smallest benchmark instances time out. The third category, has important variables interspersed with non-important variables.

VIII. Conclusions and Future Work

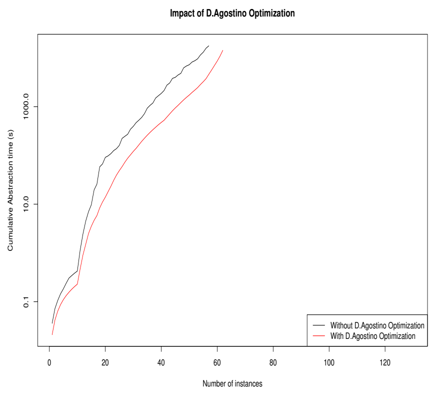
In this paper, we have addressed the problem of efficiently computing precise abstractions. We have presented a novel combination schema where BDD-based and SMT-based quantification engines exchange complementary information to aggressively limit the search space. Within this schema, additional optimizations are presented, including partitioned BDDs, a schema for blocking already visited models, and use of early calls to the SMT solver. Experimental evaluations demonstrate additional advantages over the original framework [8]. The approach embeds reasoning with respect



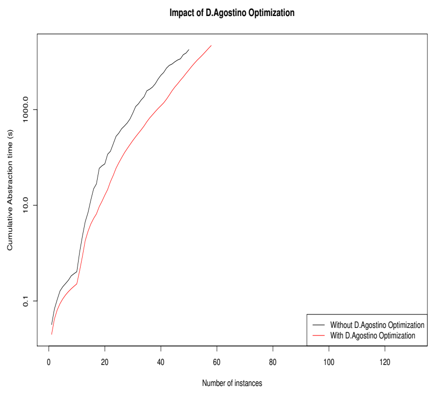
(a)



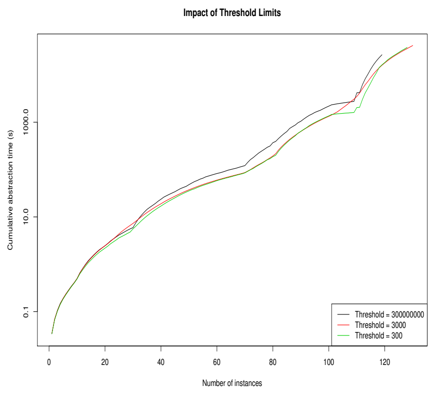
(b)



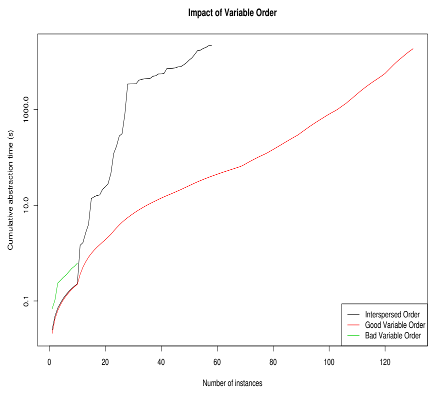
(c)



(d)



(e)



(f)

Fig. 4. Experimental Analysis

to the background theory within a BDD-based quantification algorithm, and is able to outperform previous approaches.

We plan to extend this work along different dimensions. We plan to investigate dedicated ordering heuristics, to take into account the individual variables occurring in the constrains, heuristics to guess the best threshold to perform partitioning. We will investigate techniques that will try to exploit the structure of the original problem to achieve better performance and the impact of incrementality of our approach in the setting of a CEGAR. Finally, we will apply the approach to the verification of timed and hybrid systems, as well as Verilog models.

References

[1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[2] S. K. Lahiri, R. E. Bryant, and B. Cook, “A Symbolic Approach to Predicate Abstraction,” in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 141–153.

[3] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang, “Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement,” in *CAV*, ser. LNCS, vol. 3114. Springer, 2004.

[4] S. K. Lahiri, T. Ball, and B. Cook, “Predicate abstraction via symbolic decision procedures.” in *CAV*, ser. LNCS, vol. 3576, 2005, pp. 24–38.

[5] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, “SMT Techniques for Fast Predicate Abstraction,” in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 424–437.

[6] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, “Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog,” in *Design Automation Conference (DAC)*, June 2005.

[7] D. Kroening and N. Sharygina, “Image Computation and Predicate Refinement for RTL Verilog using Word Level Proofs,” in *DATE 2007*, 2007, pp. 1325–1330.

[8] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, “Computing Predicate Abstractions by Integrating BDDs and SMT Solvers,” in *FMCAD*. IEEE CS, 2007, pp. 69–76.

[9] A. Cimatti and R. Sebastiani, “Building Efficient Decision Procedures on top of SAT Solvers,” in *Formal Methods for Hardware Verification*, ser. LNCS. Springer, 2006, no. 3965.

[10] S. Graf and H. Saidi, “Construction of Abstract State Graphs with PVS,” in *CAV*, ser. LNCS, vol. 1254. Springer, 1997, pp. 72–83.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, “Symbolic model checking: 10^{20} states and beyond,” *Inf. Comput.*, vol. 98, no. 2, pp. 142–170, 1992.

[12] R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[13] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, “Safe BDD Minimization Using Don’t Cares,” in *DAC*, 1997, pp. 208–213.

[14] D. Déharbe and S. Ranise, “Light-Weight Theorem Proving for Debugging and Verifying Units of Code,” in *SEFM*. IEEE, 2003, pp. 220–228.

[15] M. P. Schuijers, “Integrating a BDD Prover and a DPLL SAT Solver for Abstract Data Types,” Tech. Univ. Eindhoven, Tech. Rep., Jan. 2006.

[16] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, “The MathSAT 4SMT Solver,” in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 299–303.

[17] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: A New Symbolic Model Checker,” *STTT*, vol. 2, no. 4, pp. 410–425, 2000.