Turbo-Charging Lemmas on Demand with Don't Care Reasoning

Aina Niemetz, Mathias Preiner, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***Abstract*—Lemmas on demand is an abstraction/refinement technique for procedures deciding Satisfiability Modulo Theories (SMT), which iteratively refines full candidate models of the formula abstraction until convergence. In this paper, we introduce a dual propagation-based technique for optimizing lemmas on demand by extracting partial candidate models via don't care reasoning on full candidate models. Further, we compare our approach to a justification-based approach similar to techniques employed in the context of model checking. We implemented both optimizations in our SMT solver Boolector and provide an extensive experimental evaluation, which shows that by enhancing lemmas on demand with don't care reasoning, the number of lemmas generated, and consequently the solver runtime, is reduced considerably.**

I. Introduction

Procedures for deciding satisfiability of first order formulas w.r.t. first order theories, also known as Satisfiability Modulo Theories (SMT), are usually divided into so-called *eager* and *lazy* approaches. Eager SMT approaches eagerly encode an SMT formula into an equisatisfiable Boolean formula, which then serves as input for a SAT solver. Lazy approaches, on the other hand, are generally based on a tight integration of a SAT solver and one or more theory solvers. The SAT solver typically enumerates Boolean truth assignments satisfying a Boolean abstraction of the input formula, whereas the theory solver(s) not only check if those assignments are consistent w.r.t. the first order theorie(s), but guide the SAT solver through its search. The majority of state-of-the-art SMT solvers employ lazy SMT approaches, where the *lemmas on demand* procedure as introduced for the extensional theory of arrays in [7] is one extreme variant thereof [20]. The core idea of lemmas on demand is similar to the Counterexample-Guided Abstraction Refinement (CEGAR) approach introduced in [9] and goes back to [11], while at the same time, a related technique was proposed in the context of bounded model checking, where all-different constraints are lazily encoded over bit vectors (see also [5]). Recently, in [19] we introduced a generalization of the lemmas on demand decision procedure in [7] to lazily handle λ terms.

Similar to other lazy SMT approaches, lemmas on demand as in [7][19] enumerates truth assignments (so-called *candidate models*) of the bit vector abstraction of the (preprocessed) input formula and iteratively refines those assignments with lemmas until convergence. Each of these candidate models is a full truth assignment of the formula abstraction, which subsequently needs to be checked for consistency w.r.t. the theory of bit vectors with arrays. A full candidate model, however, includes parts of the formula abstraction irrelevant to its satisfiability under the current assignment and might therefore be over-determined.

In this paper we aim at exploiting *a posteriori observability don't cares*, i.e., parts of the formula abstraction irrelevant under the current assignment. We show that don't care reasoning on full candidate models to extract partial candidate models subsequently reduces the cost for consistency checking by focusing on the relevant parts of the formula, only. Motivated by *dual propagation* techniques in the context of quantified boolean formulas (QBF) [15][16], we propose an optimization of the lemmas on demand procedure in [19] and compare our approach to a technique based on *justification* heuristics in ATPG [18]. We implemented both techniques in our SMT solver Boolector and analyse the results in comparison to the version of Boolector that won the QF_AUFBV track of the SMT competition 2012.

Note that in this paper, our justification-based approach mainly serves as a basis for comparison to our dual propagation-based approach. In the context of SMT, Barrett and Donham [3] and De Moura and Bjørner [10] applied justification-based techniques to prune the search space of DPLL(T). In the context of model checking, justification-based techniques have been previously employed to identify a posteriori observability don't cares. Bingham and Hu [6], e.g., prune the search space of their simulation-based bounded model checking engine by means of a justification-based generalization mechanism (*skip cubes*) similar to learning and non-chronological backtracking of conventional SAT procedures. Eén et al. [13] employ a related approach when generalizing proof obligations by *ternary simulation* for property directed reachability (PDR), whereas Chockler et al. [8] use a variant of offline dual propagation for SAT. The verification tool Reveal [2][1], on the other hand, employs a CEGAR approach for model checking complex hardware designs and generalizes candidate counter examples by justification techniques similar to our justification-based method. Their (and our) justification-based approach, however, is only applicable on structural (non-clausal) problems. In contrast, our dual propagation-based approach generalizes full candidate models by exploiting the duality of the Boolean layer of the input formula and is not restricted to structural formula abstractions.

This work was funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

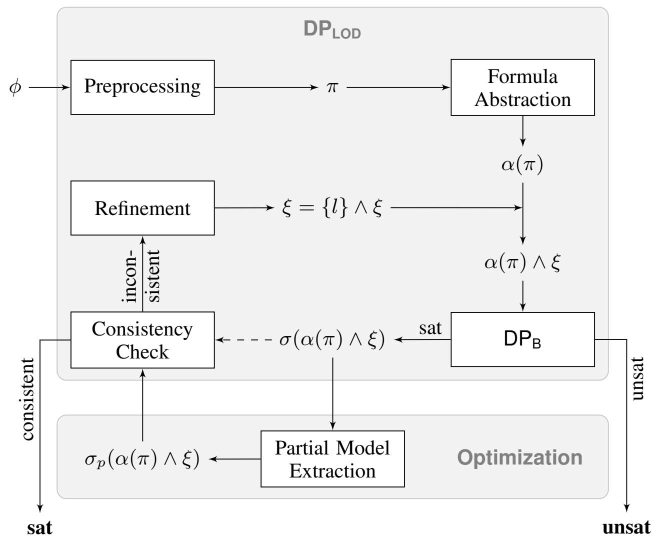


Fig. 1: The workflow of the *lemmas on demand* decision procedure $\mathsf{DP_{LODopt}}$ in Boolector. The original procedure $\mathsf{DP_{LOD}}$ (indicated by the dashed line) works on full candidate models, whereas the optimized procedure $\mathsf{DP_{LODopt}}$ extracts partial candidate models prior to consistency checking.

II. Lemmas on Demand at a Glance

The *lemmas on demand* decision procedure as implemented in Boolector is an iterative abstraction/refinement approach for the quantifier-free theory of fixed-sized bit vectors and arrays. Figure 1 gives a high-level view of the procedure and introduces both the original, unoptimized approach $\mathsf{DP_{LOD}}$ and our optimized approach $\mathsf{DP_{LODopt}}$ as follows.

Given a formula ϕ, $\mathsf{DP_{LOD}}$ uses a *bit vector* skeleton of the preprocessed formula π as formula abstraction $\alpha(\pi)$. In each iteration, an underlying decision procedure $\mathsf{DP_B}$ determines the satisfiability of the refined formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$ by encoding Γ to SAT and determining its satisfiability by means of a SAT solver. Note that initially, formula refinement ξ is $\top$. As Γ is an overapproximation of ϕ, $\mathsf{DP_{LOD}}$ immediately concludes with *unsat* if Γ is unsatisfiable. If Γ is satisfiable, the current (full) candidate model $\sigma(\alpha(\pi) \wedge \xi)$ is checked for consistency w.r.t. the preprocessed input formula π. If $\sigma(\alpha(\pi) \wedge \xi)$ is consistent, $\mathsf{DP_{LOD}}$ immediately concludes with *sat*. Otherwise, $\sigma(\alpha(\pi) \wedge \xi)$ is *spurious* and a lemma l is added to formula refinement ξ.

As indicated in Fig. 1, $\mathsf{DP_{LOD}}$ iteratively refines $\alpha(\pi)$ by consistency checking *full* candidate models, which usually include parts of the bit vector skeleton irrelevant to its satisfiability under the current assignment. In the following section, we will introduce an optimization to extract a partial candidate model $\sigma_p(\alpha(\pi) \wedge \xi)$ from the full candidate model $\sigma(\alpha(\pi) \wedge \xi)$ in order to guide the consistency check towards the relevant parts of $\alpha(\pi)$ only.

III. Partial Model Extraction

In terms of runtime, abstraction refinement usually is the most costly part of the lemmas on demand procedure $\mathsf{DP_{LOD}}$, where cost generally correlates with the number of lemmas generated. During refinement, procedure $\mathsf{DP_B}$ (and consequently the call to the underlying SAT solver) constitutes the majority of the overall runtime per iteration, which adds up when a great number of refinement iterations is needed. Hence, optimizing $\mathsf{DP_{LOD}}$ in terms of runtime directly translates to reducing the number of lemmas generated.

```
1 procedure consistent ($\Gamma$, $\sigma$)
2   $S$ ← search_initial_applies ($\Gamma$)
3   while $S \neq \emptyset$
4       $f(a_0, \ldots, a_n)$ ← pop ($S$)
5       $consistent$ ← check_consistency ($f(a_0, \ldots, a_n)$, $\sigma$)
6       if not $consistent$ return $\bot$
7       $S'$ ← search_applies_for_consistency_check ($f(a_0, \ldots, a_n)$)
8       push ($S$, $s' \in S'$)
9   return $\top$
```

Fig. 2: Procedure consistent in pseudo-code.

In contrast to other lazy SMT approaches [20], formula abstraction in $\mathsf{DP_{LOD}}$ does not produce a pure Boolean skeleton, but a bit vector skeleton, where each function application $f(a_0, \ldots, a_n)$ in the preprocessed formula π is mapped to a fresh bit vector variable. Consequently, consistency checking in $\mathsf{DP_{LOD}}$ is performed on *all* function applications in the bit vector skeleton (for details see [19]). A high level view of the consistency checking algorithm consistent in $\mathsf{DP_{LOD}}$ is given in Fig. 2 and proceeds as follows. Given the refined formula abstraction Γ and the full candidate model σ, search_initial_applies collects all function applications in Γ that need to be checked for consistency (line 2) and iteratively checks each APPLY $f(a_0, \ldots, a_n)$ w.r.t. the current assignment σ (lines 4-5). If check_consistency encounters an inconsistency, consistent immediately returns with $\bot$. Else, search_applies_for_consistency_check instantiates function f with arguments $a_0, \ldots, a_n$, which yields term t, and subsequently collects all function applications in formula abstraction $\alpha(t)$ for consistency checking (lines 7-8). If all applies in S have been checked without inconsistencies, procedure consistent concludes that current candidate model σ is consistent and returns $\top$.

Consistency checking all function applications in formula abstraction Γ corresponds to checking the *full* candidate model σ for consistency, with the order in which applies are checked as the only way to positively influence the number of refinement iterations (by coincidentally finding lemmas that shortcut the search, early on). Checking the full candidate model, however, is often not required, as only a small subset of the full candidate model is responsible for actually satisfying the formula abstraction. As a consequence, parts of the formula abstraction irrelevant to its satisfiability under the current assignment are checked, which subsequently produces lemmas that do not necessarily prune the search space and therefore mainly cost runtime.

Example 1: As a running example, consider the formula

$$\psi_1 \equiv i \neq k \wedge (f(i) = e \vee f(k) = v) \wedge v = ite(i = j, e, g(j))$$

as given in Fig. 3. Its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$

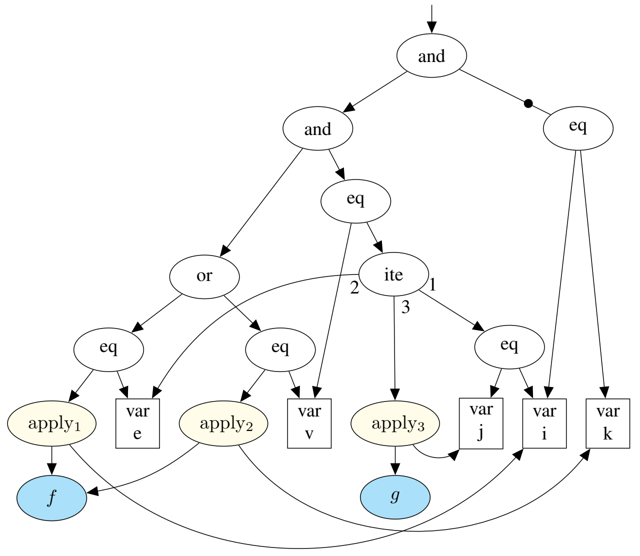


Fig. 3: DAG representation of formula ψ_1 (running example).

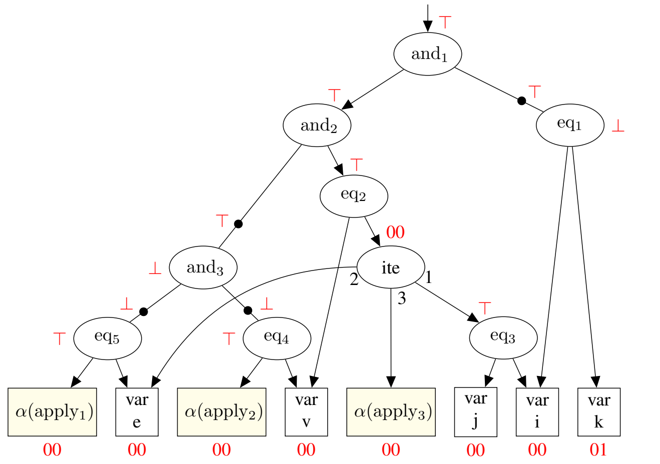


Fig. 4: Formula abstraction Γ_{ψ_1} of formula ψ_1 with candidate model $\sigma(\Gamma_{\psi_1})$ indicated in red (running example).

and a (possible) initial full candidate model $\sigma(\Gamma_{\psi_1})$ (indicated in red) is given in Fig. 4. In the following, we assume that all variables in ψ_1 are bit vector variables of size 2 and Γ_{ψ_1} is a bit vector skeleton. For the sake of simplicity, we further assume that functions f and g represent uninterpreted functions, i.e., we concentrate on consistency checking of the *full* versus a *partial* candidate model (via procedure search_initial_applies) and do not bother with details of the internals of the actual consistency check (for details, see [19]). Procedure search_initial_applies initially collects all function applications in Γ_{ψ_1} (apply$_1$, apply$_2$, apply$_3$) to be checked for consistency. During consistency checking, however, no further applies are identified as required to being checked (procedure search_applies_for_consistency_check) as both f and g do not make subsequent calls to other functions. Note that given $\sigma(\Gamma_{\psi_1})$, instead of checking all applies in ψ_1, either checking apply$_1$ or apply$_2$ would be sufficient.

In the following, we consider two techniques for identifying irrelevant parts of the formula abstraction by extracting partial candidate models, which subsequently reduces the number of refinement iterations, and therefore, the overall runtime of the lemmas on demand procedure.

A. *Justification-Based Partial Model Extraction*

In the context of ATPG [18], sets of don't care conditions are usually divided into *observability don't cares (ODC)* and *controllability don't cares (CDC)*. The former denotes lines that do not influence the primary outputs (independent from the current assignment to the primary inputs), and the latter identifies line values that can not be justified and are therefore illegal under any assignment to the primary inputs. Given a concrete assignment to the primary inputs, however, we can determine what we call *a posteriori* observability don't cares, i.e., lines that do not influence the output of a gate under its current assignment. In the context of model checking, such *a posteriori* ODC have already been exploited by Bingham and Hu [6], Eén et al. [13], and Andraus et al. [2][1].

In this section, we introduce a technique similar to [2][1] and extract partial candidate models by identifying parts of the formula abstraction Γ that are irrelevant to its satisfiability under the current assignment σ. As indicated above, this directly translates to collecting and checking function applications in relevant parts of Γ only. In the following, we assume that Γ is represented as a directed acyclic graph (DAG) with exactly one root, where all Boolean operations are expressed by means of NOT and (two-input) AND gates. In place of procedure search_initial_applies, we introduce search_initial_applies$_{\text{just}}$ (Fig. 5), which collects function applications while traversing all relevant paths in Γ as follows.

Given Γ and a full candidate model σ, starting from the root, search_initial_applies$_{\text{just}}$ iteratively traverses Γ towards its primary inputs (bit vector variables and function applications) in depth first search (DFS) order. That is, initially, root(Γ) is pushed onto stack X (line 2) and for each node $x \in X$ we determine the paths to be skipped as follows. If a node x is an AND node and its output is assigned to $\bot$, we follow (one of) its controlling input(s), i.e., one of its inputs with controlling value ($\bot$ for an AND) [18], and skip the other (lines 7-14). Similarly, if x is an IF-THEN-ELSE (ITE) node and its condition is assigned to $\top$ (resp. $\bot$), we follow both its condition and its *then* (resp. *else*) branch (lines 15-20). In any other case where x is not an APPLY node, we follow all inputs of node x (line 22). However, if x is an APPLY node, we collect x (line 6) and cut off the traversal, as function applications are treated as fresh bit vector variables in the formula abstraction.

Note that in the case that *both* inputs of an AND node are controlling, we can skip *either* one of them (lines 9-10). Hence, we choose to follow the input with minimum cost in terms of consistency checking, where the cost of a node x is defined as the minimum number of (unique) applies along a path from x to the primary inputs in the preprocessed formula π. Similar as controllability measures in ATPG [18],

```
procedure search_initial_applies_just ($\Gamma$, $\sigma$)
  $S \leftarrow \emptyset$, $X \leftarrow$ {root ($\Gamma$)}
  while $X \neq \emptyset$
      $x \leftarrow$ pop ($X$)
      if is_apply ($x$)
          push ($S$, $x$)
      elif is_and ($x$) and $\sigma(x) = \bot$
          $l \leftarrow$ left_input ($x$), $r \leftarrow$ right_input ($x$)
          if is_controlling ($l$) and is_controlling ($r$)
              push ($X$, choose ($l$, $r$))
          elif is_controlling ($l$)
              push ($X$, $l$)
          else
              push ($X$, $r$)
      elif is_ite ($x$)
          push ($x$, condition ($x$))
          if $\sigma$(condition ($x$)) = $\top$
              push ($X$, then ($x$))
          else
              push ($X$, else ($x$))
      else
          push ($X$, $i \in$ inputs ($x$))
  return $S$
```

Fig. 5: Procedure $\mathsf{search\_initial\_applies_{just}}$ in pseudo-code.

we recursively define a cost function cost (x) as follows.

$$\mathsf{cost}\,(x) = \begin{cases} 0 & \text{if is\_var}\,(x) \\ \min\{\mathsf{cost}\,(i) \mid i \in \mathsf{inputs}\,(x)\} & \text{if is\_and}\,(x) \\ \mathrm{sum}\{\mathsf{cost}\,(i) \mid i \in \mathsf{inputs}\,(x)\} + 1 & \text{if is\_apply}\,(x) \\ \mathrm{sum}\{\mathsf{cost}\,(i) \mid i \in \mathsf{inputs}\,(x)\} & \text{otherwise} \end{cases} \quad (1)$$

Given formula π, a bit vector variable is a primary input, hence its cost is defined as 0. Function applications, on the other hand, are not primary inputs but define the cost of a path from input x to the primary inputs. Hence, the cost of an APPLY is defined as the sum of the costs of its inputs increased by one. In case of an AND node, we want to choose the input with minimum cost if both inputs are controlling, hence cost is defined as the minimum cost of its inputs. In any other case, all input paths have to be followed and cost (x) is defined as the sum of the costs of all inputs of x.

Example 2: Consider formula ψ_1, formula abstraction Γ_{ψ_1}, and a full candidate model $\sigma(\Gamma_{\psi_1})$ as given in Example 1. Starting from the root (and_1), procedure search_initial_applies$_{\mathsf{just}}$ traverses Γ_{ψ_1} in DFS order while identifying (and skipping) all paths irrelevant w.r.t. assignment $\sigma(\Gamma_{\psi_1})$. Note that in Fig. 3 and 4, inverted nodes are indicated by black dots. In the following, however, we will interpret an inverted node as two distinct nodes (with resp. distinct assignments), i.e., $\neg\mathrm{and}_3$ with $\sigma(\neg\mathrm{and}_3) = \top$ in Fig. 4, for example, is treated as a NOT (assigned to $\top$) in front of an AND (assigned to $\bot$). Starting with root and_1, which is assigned to $\top$, neither of its inputs may be skipped and we first travel down towards eq_1, whose inputs are both bit vector variables. Hence, we immediately continue with and_2 (also assigned to $\top$) and follow its input eq_2, where we encounter an ite with its condition assigned to $\top$. We skip the *else* branch, no APPLY is collected, and we continue down the input path leading to and_3, which is assigned to $\bot$. Both inputs of and_3 are controlling (i.e., assigned to $\bot$), hence we choose one of them heuristically. The minimum cost for both paths, however, is 0 (as the body of function f does not contain any further applies), hence we may choose either. We decide on the path to apply_1 and conclude with $S = \{\mathrm{apply}_1\}$, which corresponds to the partial model to be subsequently checked for consistency.

B. Dual Propagation-Based Partial Model Extraction

Exploiting the duality of QBF by propagating a dual set of values through a QBF ϕ and its negation $\neg\phi$, also referred to as *dual propagation*, has successfully been employed in [15] to significantly prune, and therefore speed up the search in circuit-based QBF solvers. The core idea of *dual propagation*, however, is neither restricted to circuit-based representations [16] nor to QBF and is based on the fact that assignments satisfying an input formula ϕ (the *primal* channel), falsify its negation $\neg\phi$ (the *dual* channel) and vice versa. Given a Boolean formula $\psi_2 \equiv (a \wedge b) \vee (c \wedge d)$, for example, assignment $\{\sigma(a) = \top, \sigma(b) = \top, \sigma(c) = \top, \sigma(d) = \top\}$ satisfies ψ_2, but falsifies its negation $\neg\psi_2 \equiv (\neg a \vee \neg b) \wedge (\neg c \vee \neg d)$.

The duality of formula ψ_2, however, can be further exploited. Assume, for example, that given ψ_2 and $\sigma(\psi_2)$ as above, we fix the values of all input variables assigned in $\sigma(\psi_2)$ by making assumptions $\{a{=}\top, b{=}\top, c{=}\top, d{=}\top\}$ to a SAT solver maintaining its negation $\neg\psi_2$. All assumptions inconsistent with $\neg\psi_2$, also called *failed assumptions* [14], identify all input assignments sufficient to falsify $\neg\psi_2$, hence sufficient to satisfy ψ_2. This set of failed assumptions, for example $\{a = \top, b = \top\}$, therefore represents a *partial model* satisfying ψ_2. Note that our approach does not require a structural SAT solver—structural don't care reasoning is simulated via the dual solver, which maintains $\neg\psi_2$ in CNF. Consequently, given a CNF representation of ψ_2 (where structural information of ψ_2 is essentially lost), we extract a partial model (disregarding structural don't cares w.r.t. assignment σ) that satisfies ψ_2 but not necessarily its encoding to CNF. Consider, for example, the Tseitin encoding CNF(ψ_2) $\equiv (\neg o \vee x \vee y) \wedge (\neg x \vee o) \wedge (\neg y \vee o) \wedge (\neg x \vee a) \wedge (\neg x \vee b) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg y \vee c) \wedge (\neg y \vee d) \wedge (\neg c \vee \neg d \vee y)$. Our previous partial model $\{a = \top, b = \top\}$ satisfies ψ_2 (and therefore identifies those parts of ψ_2 relevant to its satisfiability) but does not satisfy all clauses in CNF(ψ_2). This is in contrast to other partial model extraction techniques based on iterative removal of unnecessary assignments on the CNF level (e.g. [12]), which do not enable structural don't care reasoning and therefore need to satisfy all clauses in CNF(ψ_2).

In this section, we lift the approach sketched above to the word level by means of a dual SMT solver and introduce a technique to extract partial candidate models via dual propagation-based don't care reasoning. Given a formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$, we use a single *dual* solver instance to maintain $\neg\Gamma$ over all refinement iterations in combination with the *primal* (or *main*) solver. However, since in each iteration i a new lemma l_i is added to $\xi \equiv l_1 \wedge \ldots \wedge l_{i-1}$, we set up the dual solver to maintain $\neg\Gamma \equiv \neg(\alpha(\pi) \wedge l_1 \wedge \ldots \wedge l_{i-1} \wedge l_i)$

```
1  procedure search_initial_applies_dp ($\Gamma$, $\sigma$)
2      $S \leftarrow \emptyset$,  $A \leftarrow \emptyset$
3      assume ($dual\_solver$, $\neg\Gamma$)
4      $X \leftarrow$ collect_primary_inputs ($\Gamma$)
5      for $x \in X$
6          $a \leftarrow x = \sigma(x)$,  $A \leftarrow A \cup a$
7          assume ($dual\_solver$, $a$)
8      $res \leftarrow$ DP_B ($dual\_solver$)
9      assert $res = UNSAT$
10     for $a \in A$
11         $x$, $\sigma(x) \leftarrow a$
12         if is_failed ($a$) and is_apply ($x$)
13             push ($S$, $x$)
14     return $S$
```

Fig. 6: Procedure search_initial_applies$_{\mathsf{dp}}$ in pseudo-code. Solver instance $dual\_solver$ simulates the dual channel and is maintained globally.

as assumption rather than assertion. As illustrated in Fig. 6, we introduce search_initial_applies$_{\mathsf{dp}}$ in place of procedure search_initial_applies as follows.

Given Γ and a full candidate model σ, procedure search_initial_applies$_{\mathsf{dp}}$ initializes the dual solver by assuming $\neg\Gamma$ (line 3). The value of all primary inputs in $\neg\Gamma$ is then fixed by making assumptions of the form $x = \sigma(x)$, where x is either a bit vector variable or an abstracted function application, and $\sigma(x)$ is its assignment in the current full candidate model σ (lines 4-7). Candidate model σ represents a satisfying assignment for Γ, hence decision procedure $\mathsf{DP_B}$ must conclude that assuming σ, $\neg\Gamma$ is unsatisfiable (lines 8-9). The resulting set of failed assumptions identifies all relevant parts of Γ w.r.t. assignment σ, and all function applications in the set of failed assumptions are subsequently collected for consistency checking (lines 10-13).

Example 3: Again, consider formula ψ_1, its initial formula abstraction $\Gamma_{\psi_1} \equiv \alpha(\psi_1)$, and a (possible) full candidate model $\sigma(\psi_1)$ as given in Example 1. Procedure search_initial_applies$_{\mathsf{dp}}$ initializes the dual solver by assuming $\neg\Gamma_{\psi_1} \equiv \neg(i \neq k \wedge (\alpha(\text{apply}_1) = e \vee \alpha(\text{apply}_2) = v) \wedge v = ite(i = j, e, \alpha(\text{apply}_3)))$, and subsequently collects all bit vector variables i, j, k, e, v and abstracted function applications $\alpha(\text{apply}_1)$, $\alpha(\text{apply}_2)$, $\alpha(\text{apply}_3)$ in Γ_{ψ_1} onto stack X. All primary inputs $x \in X$ are then fixed by making assumptions $\{i = 00, j = 00, k = 01, e = 00, v = 00, \alpha(\text{apply}_1) = 00, \alpha(\text{apply}_2) = 00, \alpha(\text{apply}_3) = 00\}$ to the dual SMT solver instance, which concludes that under the current set of assumptions, $\neg\Gamma_{\psi_1}$ is unsatisfiable. Assumption $\alpha(\text{apply}_1) = 00$ is identified as failed assumption and we conclude with $S = \{\text{apply}_1\}$ to be subsequently checked for consistency.

Note that in a sense, our dual propagation-based approach as discussed above simulates dual propagation as introduced in the context of QBF [15][16] rather than literally lifting it to bit vectors with arrays. Dual propagation as in [15][16] is done *eagerly* by means of one single solver instance maintaining a primal and a dual channel without additional overhead. Primary inputs are shared between both channels,

	Solver	Solved (sat/unsat)	TO	MO	Time [s]	DS [s]
SMT'12	Boolector$_{sc}$	140 (83/57)	9	0	15882	-
	Boolector$_{ba}$	141 (83/58)	8	0	19312	-
	Boolector$_{ju}$	142 (84/58)	7	0	15709	-
	Boolector$_{dp}$	142 (84/58)	7	0	20992	5045
Selected	Boolector$_{sc}$	116 (72/44)	50	7	85863	-
	Boolector$_{ba}$	121 (76/45)	45	7	76104	-
	Boolector$_{ju}$	130 (85/45)	36	7	63202	-
	Boolector$_{dp}$	130 (85/45)	36	7	66991	4705

TABLE I: Overall results on sets *SMT'12* and *Selected.*

which enables symmetric propagation between the primal and dual channel and allows to detect partial models early—even before a full assignment has been generated. In our approach, however, propagation is not interleaved, but consecutive—the primal solver generates a full assignment before the dual solver enables partial model extraction based on the primal full assignment. Further, primary inputs are not physically shared as the dual solver discretely maintains $\neg\phi$ (while mapping primary inputs back to the primal solver and vice versa). Hence we have to simulate shared inputs via fixing input values by means of assumptions to the dual solver, which simply acts as "slave" for partial model extraction to the primal solver. In order to adopt a more eager approach to enable early partial model extraction while reducing the dual solver overhead, interleaved execution between the primal and dual solver similar to "SAT modulo SAT" [4] would be required. Integrating such an interleaved decision process into an existing SMT solver has high potential, however, is rather involved to implement and left to future work.

IV. Experimental Evaluation

We implemented justification-based and dual propagation-based partial model extraction in our SMT solver Boolector and provide a comparison of the following four configurations:

- Boolector$_{sc}$: The version that won the QF_AUFBV track of the SMT competition 2012.
- Boolector$_{ba}$: Our current base version of Boolector, a slightly optimized version of [19], with partial model extraction disabled.
- Boolector$_{ju}$: Our base version of Boolector with justification-based partial model extraction enabled.
- Boolector$_{dp}$: Our base version of Boolector with dual propagation-based partial model extraction enabled.

We compiled two benchmarks sets for our experimental evaluation: (1) *SMT'12* (149 instances), which consists of all non-extensional benchmarks used for the SMT competition 2012 and (2) *Selected* (173 instances), which includes all non-extensional benchmarks from the QF_AUFBV category of SMT-LIBfor which Boolector$_{sc}$ required at least 10 seconds (CPU time) for solving (incl. timeouts and memouts). Note that we had to exclude extensional benchmarks as Boolector$_{ba}$ and its optimized versions Boolector$_{ju}$ and Boolector$_{dp}$ do not yet support extensionality on arrays. Further note that 58 instances of the benchmark set *SMT'12* are included in *Selected.* All experiments were performed on 2.83Ghz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 12.04.

	Solver	Time [s]			Sat [s]			DS overhead [s]			LOD			Array Model Size		
		Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.	Total	Avg.	Med.
SMT'12	Boolector$_{sc}$	4129	29	2	3662	26	0	-	-	-	30741	221	0	184032	2272	20
	Boolector$_{ba}$	8564	61	6	7262	52	1	-	-	-	33013	237	0	33310	411	20
	Boolector$_{ju}$	6362	45	4	5226	37	0	-	-	-	23660	170	0	19751	243	13
	Boolector$_{dp}$	10145	72	5	4700	33	0	4109	29	0	33492	240	0	27912	344	12
Selected	Boolector$_{sc}$	15037	133	35	12836	113	34	-	-	-	104646	926	175	512225	7645	1257
	Boolector$_{ba}$	10001	88	35	8330	73	22	-	-	-	31752	280	88	136681	2040	212
	Boolector$_{ju}$	8182	72	29	6639	58	19	-	-	-	28215	249	28	122763	1832	154
	Boolector$_{dp}$	10838	95	30	6164	54	15	3036	26	0	24866	220	29	130440	1946	170

TABLE II: Results for commonly solved instances on sets *SMT'12* (139 benchmarks, 82 sat, 57 unsat) and *Selected* (113 benchmarks, 70 sat, 43 unsat). Commonly solved satisfiable instances for determining array model size were 81 (out of 82) for *SMT'12* and 67 (out of 70) for *Selected*. Array model size is measured in terms of number of index/value pairs.

The memory and time limits for each solver instance were set to 7GB and 1200 seconds, respectively.

A. *Results Overview*

The overall results of all four solver configurations on both benchmark sets *SMT'12* and *Selected* are shown in Table I, which summarizes the number of solved instances (Solved), timeouts (TO), memouts (MO), total CPU time (Time), and the overhead produced by the dual solver in terms of CPU time (DS). Note that the overhead introduced by our justification-based approach is negligible. Further note that in case of a timeout or memout, a penalty of 1200 seconds was added to the total CPU time. On the *SMT'12* benchmark set, in terms of solved instances, Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$ perform slightly better than Boolector$_{sc}$. In terms of runtime, however, only Boolector$_{ju}$ shows a significant improvement (of about 20%), while Boolector$_{dp}$ appears to even perform worse than Boolector$_{ba}$, which is mainly due to the runtime overhead introduced by the dual solver. If we disregard this overhead, the overall runtime of Boolector$_{dp}$ is competitive with the runtime of Boolector$_{ju}$. It is conceivable that an eager implementation of dual propagation would perform equally well, i.e., at least as fast as Boolector$_{dp}$ without the overhead.

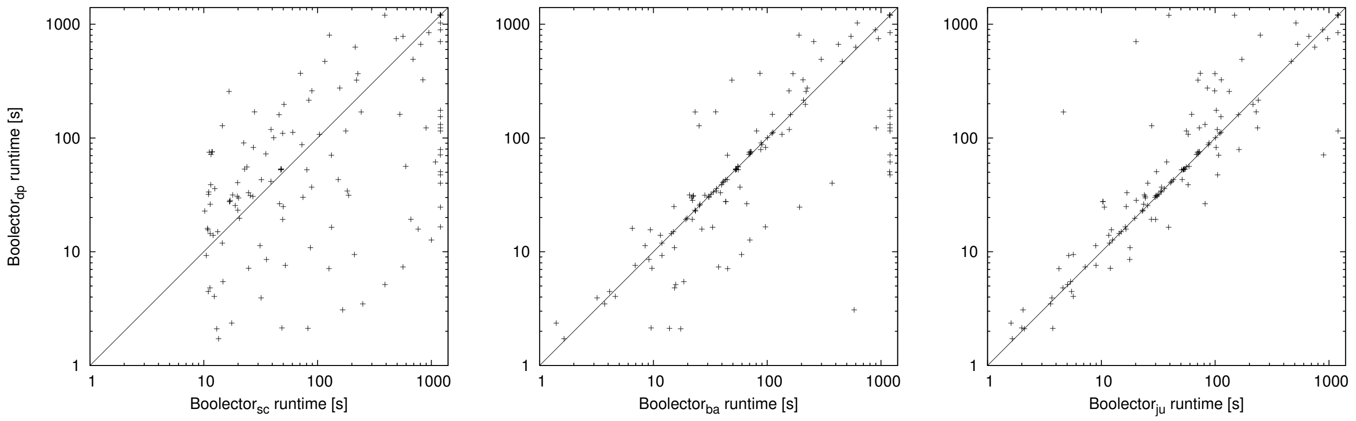
Interestingly, Boolector$_{sc}$ clearly outperforms all other three solver configurations on the benchmark family "platania strcmp" (9 instances). Boolector$_{sc}$ solved these benchmarks in about 31 seconds, whereas the other solvers required 4416 seconds (Boolector$_{ba}$), 2308 seconds (Boolector$_{ju}$), and 4527 seconds (Boolector$_{dp}$, incl. 2277 seconds dual solver overhead), respectively. The base version Boolector$_{ba}$, and consequently both Boolector$_{ju}$ and Boolector$_{dp}$, obviously struggle on these benchmarks, which needs further investigation.

Note that benchmark set *SMT'12* is not necessarily representative for lemmas on demand in Boolector, as 79 (53%) out of a total of 149 instances are immediately solved by Boolector$_{sc}$ without a single refinement iteration. Benchmark set *Selected*, on the other hand, has been compiled based on the runtime performance of the SMT competition 2012 winner Boolector$_{sc}$ (incl. timeouts and memouts) and represents a set considered to be "harder" for Boolector. As indicated in Table I, on set *Selected* both Boolector$_{ju}$ and Boolector$_{dp}$ clearly outperform their base version Boolector$_{ba}$ as well as the competition configuration Boolector$_{sc}$. More precisely, both our justification-based and dual propagation-based optimizations considerably reduce the overall runtime while solving 14 (9) additional instances compared to Boolector$_{sc}$ (Boolector$_{ba}$), where 13 (9) out of 14 (9) are satisfiable instances. Again, Boolector$_{dp}$ is slowed down by the dual solver overhead, but still manages to solve as many instances as Boolector$_{ju}$. Disregarding the dual solver overhead, Boolector$_{dp}$ even outperforms Boolector$_{ju}$ in terms of runtime. Note that the dual solver overhead in general correlates with the number of lemmas generated. This is due to the fact that in each refinement iteration a partial candidate model is extracted from the full candidate model, which requires an additional call to the dual solver. On set *Selected*, for 10 out of 130 instances, the dual solver overhead constitutes about 50-70% of the total runtime per instance, whereas for 83 instances it does not exceed 10%.

B. *Results Commonly Solved Instances*

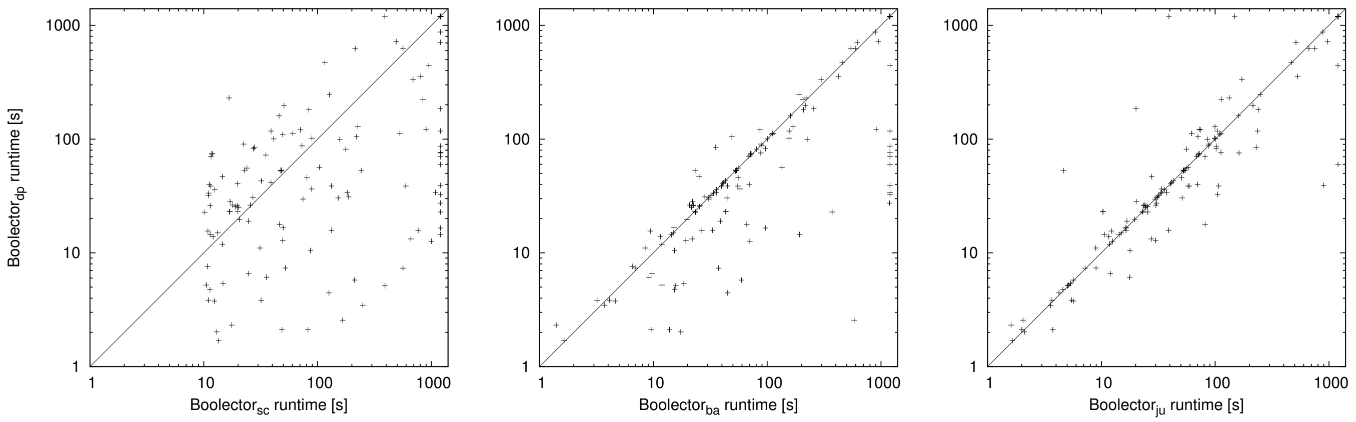
Table II summarizes all instances in each benchmark set that could be solved by all four solver configurations and gives an overview of the runtime required for solving (Time), the runtime required by the underlying SAT solver (Sat), the dual solver overhead (DS), the number of lemmas generated (LOD), and the size of the array models for satisfiable instances (Array Model Size). For all four solver configurations, we identified 139 common instances (82 sat, 57 unsat) on benchmark set *SMT'12* and 113 common instances (70 sat, 43 unsat) on benchmark set *Selected*. Array model size is measured in terms of the number of index/value pairs identified by each solver with model generation enabled. However, unlike Boolector$_{ba}$ (and consequently Boolector$_{ju}$ and Boolector$_{dp}$), Boolector$_{sc}$ requires additional overhead for model generation, which has a negative impact on the overall number of solved instances. As a consequence, Boolector$_{sc}$ effectively "lost" 1 (resp. 3) satisfiable instance(s) on set *SMT'12* (resp. *Selected*). We therefore compiled all columns except column Array Model Size with model generation disabled.

On the 139 common instances in the *SMT'12* benchmark set, Boolector$_{sc}$ is still the fastest solver, albeit only due to the "platania strcmp" benchmarks mentioned above—on those nine instances, Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$ spent 50%, 35% and 45% of the overall runtime, respectively. A similar picture emerges when comparing the number of refinement iterations required for these nine instances, which constitutes 59%, 47%, and 60% of the total number of lemmas generated by Boolector$_{ba}$, Boolector$_{ju}$, and Boolector$_{dp}$, respectively. In comparison to the base version Boolector$_{ba}$, however, Boolector$_{dp}$ shows the most notable



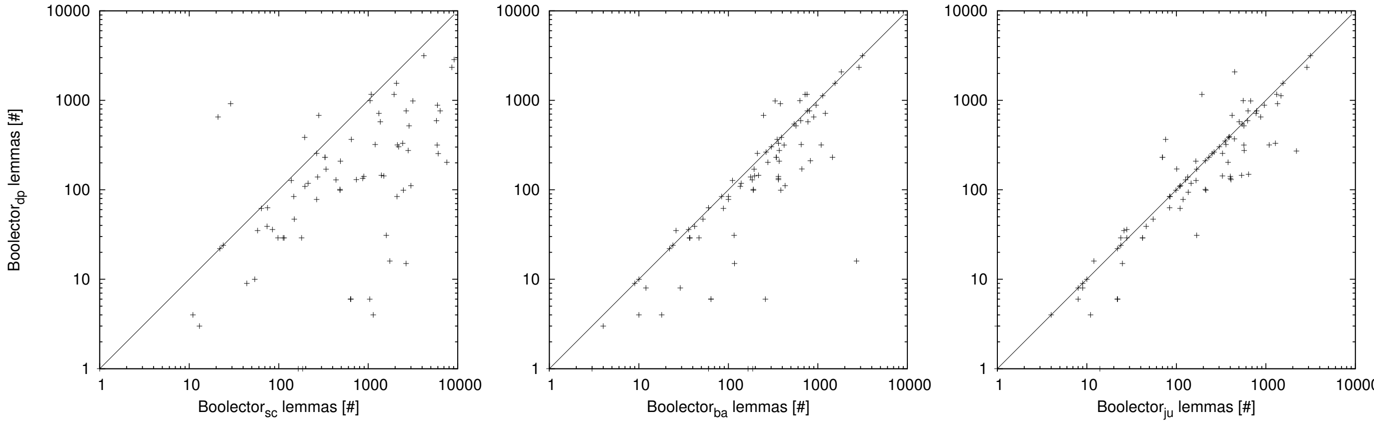
(a) (b) (c)

Fig. 7: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead included.



(a) (b) (c)

Fig. 8: Runtime comparison of Boolector$_{dp}$ vs. Boolector$_{sc}$ (8a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (8b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (8c) on benchmark set *Selected* with 1200 seconds timeout, dual solver overhead *not* included.



(a) (b) (c)

Fig. 9: Comparison of the number of lemmas generated by Boolector$_{dp}$ vs. Boolector$_{sc}$ (9a), Boolector$_{dp}$ vs. Boolector$_{ba}$ (9b), and Boolector$_{dp}$ vs. Boolector$_{ju}$ (9c) on benchmark set *Selected*.

improvement (about 26%) in terms of runtime required by the underlying SAT solver on the 139 common instances in *SMT'12*. Disregarding the dual solver overhead, Boolector$_{dp}$ even outperforms Boolector$_{ju}$ in terms of overall runtime. Interestingly, in terms of the number of lemmas generated, Boolector$_{dp}$ requires slightly more lemmas than the base version, which is in stark contrast to Boolector$_{ju}$. However, in case of Boolector$_{dp}$, this can be contributed to a relative small number of instances. On 14 instances, Boolector$_{dp}$ generates 1.5 to 2.6 times more lemmas than Boolector$_{ba}$, whereas on all other instances, Boolector$_{ba}$ requires considerably more refinement iterations than Boolector$_{dp}$. This might indicate that in some cases, Boolector$_{ba}$ coincidentally generates lemmas that shortcut the search early on. In terms of array model size, both optimized configurations Boolector$_{ju}$ and Boolector$_{dp}$ clearly show a reduction in the number of array index/value pairs compared to the base version Boolector$_{ba}$.

Note that the considerable difference in array model size between Boolector$_{sc}$ and Boolector$_{ba}$ is due to an optimization of procedure search_applies_for_consistency_check (see Section III) introduced subsequent to [19]. In essence, given a function application $f(a)$, this optimization aims at consistency checking APPLY nodes reachable while traversing in DFS order from $f(a)$ to the primary inputs, only. In contrast, prior to that optimization it was possible that function applications irrelevant to consistency checking $f(a)$ were pulled in. The effect of this optimization is even more notable on the *Selected* benchmark set, where Boolector$_{ba}$ clearly outperforms Boolector$_{sc}$ in every aspect.

On the 113 common instances in set *Selected*, Boolector$_{dp}$ clearly outperforms Boolector$_{ju}$ and Boolector$_{ba}$ not only in terms of runtime required by the underlying SAT solver, but in the number of lemmas generated. Disregarding the dual solver overhead, Boolector$_{dp}$ shows even more improvement in terms of overall runtime than Boolector$_{ju}$. Note that without the optimization of procedure search_applies_for_consistency_check mentioned above, the difference in terms of overall runtime between Boolector$_{ba}$ and both optimized versions Boolector$_{ju}$ and Boolector$_{dp}$ would be even greater, i.e., comparable to the difference between both optimized versions and Boolector$_{sc}$.

C. Results Dual Propagation-Based Optimization

A more detailed overview of the instance-based results of our dual propagation-based approach Boolector$_{dp}$ on benchmark set *Selected* is given in Fig. 7-9. Figure 7 compares the overall runtime of Boolector$_{dp}$ (incl. the overhead introduced by the dual solver) with the runtime of Boolector$_{sc}$ (7a), Boolector$_{ba}$ (7b), and Boolector$_{ju}$ (7c). Even though the dual solver overhead constitutes 31% of the total runtime of Boolector$_{dp}$, it still outperforms Boolector$_{sc}$ and Boolector$_{ba}$ on a majority of the instances and is even competitive with Boolector$_{ju}$. Disregarding the overhead of the dual solver (Fig. 8), Boolector$_{dp}$ even outperforms Boolector$_{ju}$ on a majority of the instances (Fig. 8c). In terms of the number of lemmas generated (Fig. 9), in comparison to all three solver configurations Boolector$_{sc}$, Boolector$_{ba}$, and Boolector$_{ju}$, our dual propagation-based solver Boolector$_{dp}$ clearly shows the most notable improvement.

V. Conclusion

In this paper we introduced a dual propagation-based optimization of the lemmas on demand procedure for bit vectors with arrays as implemented in Boolector. We compared our approach with a justification-based approach similar to [2][1]. We showed that don't care reasoning on full candidate models improves the performance of lemmas on demand considerably, Our current simulation of dual propagation is competitive with our justification-based optimization and clearly outperforms the winner of the SMT competition 2012, even though the dual solver introduces a considerable amount of overhead to the overall runtime. Adopting a more eager dual propagation approach promises to render the dual solver overhead negligible, while further improving the overall performance by enabling partial model extraction even before a full candidate model has been generated. However, this would require an interleaved execution between the primal and the dual solver, which is rather involved to implement and subject of future work. Further, our current version of dual propagation-based partial model extraction heavily relies on incremental SAT solving under assumptions, which can benefit from dedicated data structures [17]. The integration of such SAT solver level optimization techniques is also left to future work.

Binaries of Boolector and all log files of our experimental evaluation can be found at http://fmv.jku.at/dpjust.

References

[1] Z. S. Andraus. *Automatic Formal Verification of Control Logic in Hardware Designs*. PhD thesis, University of Michigan, 2009.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Reveal: A formal verification tool for Verilog designs. In *LPAR'08*, volume 5330 of *LNCS*. Springer, 2008.

[3] C. Barrett and J. Donham. Combining sat methods with non-clausal decision heuristics. *ENTCS*, 125(3), 2005.

[4] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *FMCAD'13*. IEEE, 2013.

[5] A. Biere and R. Brummayer. Consistency checking of all different constraints over bit-vectors within a SAT solver. In *FMCAD'08*. IEEE.

[6] J. D. Bingham and A. J. Hu. Semi-formal bounded model checking. In *CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[7] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3), 2009.

[8] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *FMCAD'11*. FMCAD Inc., 2011.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[10] L. de Moura and N. Bjørner. Relevancy propagation. Technical Report MSR-TR-2007-140, Microsoft Research, 2007.

[11] L. M. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE'02*, volume 2392 of *LNCS*. Springer, 2002.

[12] D. Déharbe, P. Fontaine, D. L. Berre, and B. Mazure. Computing prime implicants. In *FMCAD'13*. IEEE, 2013.

[13] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *FMCAD'11*. FMCAD Inc., 2011.

[14] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT'04*, volume 2919 of *LNCS*. Springer, 2004.

[15] A. Goultiaeva and F. Bacchus. Exploiting QBF duality on a circuit representation. In *AAAI'10*. AAAI Press, 2010.

[16] A. Goultiaeva, M. Seidl, and A. Biere. Bridging the gap between dual propagation and CNF-based QBF solving. In *DATE'13*. ACM, 2013.

[17] J.-M. Lagniez and A. Biere. Factoring out assumptions to speed up MUS extraction. In *SAT'13*, volume 7962 of *LNCS*. Springer, 2013.

[18] Z. Navabi. *Digital Sytem Test and Testable Design*. Springer, 2011.

[19] M. Preiner, A. Niemetz, and A. Biere. Lemmas on demand for lambdas. In *DIFTS'13*, volume 1130 of *CEUR Workshop Proceedings*, 2013.

[20] R. Sebastiani. Lazy Satisfiability Modulo Theories. *JSAT*, 3(3-4), 2007.