Strengthening Properties using Abstraction Refinement

Mitra Purandare
Computer Systems Institute
ETH Zurich
mitra.purandare@inf.ethz.ch

Thomas Wahl
Computing Laboratory
Oxford University, U.K.
thomas.wahl@comlab.ox.ac.uk

Daniel Kroening
Computing Laboratory
Oxford University, U.K.
daniel.kroening@comlab.ox.ac.uk

***Abstract*—Model Checking is an automated formal method for verifying whether a finite-state system satisfies a user-supplied specification. The usefulness of the verification result depends on how well the specification distinguishes intended from non-intended system behavior. *Vacuity* is a notion that helps formalize this distinction in order to improve the user's understanding of why a property is satisfied. The goal of this paper is to expose vacuity in a property in a way that increases our knowledge of the design. Our approach, based on abstraction refinement, computes a maximal set of atomic subformula occurrences that can be strengthened without compromising satisfaction. The result is a shorter and stronger and thus, generally, more valuable property. We quantify the benefits of our technique on a substantial set of circuit benchmarks.**

I. Introduction

Model Checking is an automated technique to verify whether a finite-state design complies with a property given as a temporal logic formula. For certain types of properties, it is possible to compute a witness or a counterexample, attesting to their satisfaction or violation. In general, however, such evidence cannot be succinctly presented, such as in order to confirm a property that universally quantifies over all allowed executions.

This lack of evidence not only diminishes the verification engineer's confidence in the model checking result, but can also cause errors to go undetected. For example, Beatty and Bryant observed that the LTL property $G(req \rightarrow F\ ack)$ is satisfied by a model that never asserts req [1], which is likely not intended. Intuitively, a formula holds *vacuously* if it does so for "unintended reasons", as in the above case of antecedent failure. Vacuous satisfaction usually indicates a flaw in the design or the property and should be reported to the user.

Formalizing this intuition of vacuity turned out to be challenging; numerous strategies have been investigated (e.g. [2], [3], [4] and others). All these notions have in common that the satisfaction of the formula is invariant under certain modifications to the formula (*formula vacuity* [3]) or the design model (*trace and structure vacuity* [4]). Invariance under such modifications provides hints where the formula can possibly be strengthened without causing it to fail on the given design. Reporting the satisfaction of a formula stronger than the one supplied by the user substantially improves the model checking result, as it increases the certified knowledge about the design.

In this paper, we present an algorithm that, given a formula ϕ and a model M satisfying ϕ, checks whether a stronger satisfying formula can be obtained for M. The algorithm does so by replacing in ϕ a maximal number atomic subformula occurrences from some candidate set C by a constant expression that strengthens the overall formula. In the simplest case, C is just the set of all atomic subformula occurrences of ϕ. We first describe a principal strategy that explores possible replacements in order to compute a strongest formula. Success or failure in verifying candidate formulas guides the search towards the optimal solution.

We then enhance the principal strategy by exploiting counterexample information of a failed model checking run. If a candidate formula is too strong, we use a path witnessing the failure to narrow the search space before identifying new candidates. This approach can be seen as an instance of counterexample-guided abstraction refinement [5], applied to properties. The result is a maximally strengthened formula shown to hold (or, dually, a weakened formula shown to fail) on a given model. We demonstrate the efficiency of the approach using a significant set of hardware benchmarks.

Related Work

A variety of vacuity notions have been proposed in the literature. Among the earliest, [3] and [2] introduce syntactic vacuity, i.e., vacuity with respect to subformulas and to subformula occurrences, respectively. Efficient algorithms for syntactic vacuity detection for CTL are given in [6]. The semantic notions of vacuity in [4] for LTL are extended to CTL* in [7] and to RELTL in [8]. A detailed discussion on the ramifications of the various notions of vacuity can be found in [9]. A temporal logic query based approach to vacuity detection is presented in [10]. Vacuity detection in the framework of SAT-based bounded model checking is addressed in [11].

Closest to our work, Gurfinkel and Chechik present *mutual vacuity* [12]. The objective is to find a maximum set of literal occurrences that can be simultaneously replaced by false without causing the property to fail. The authors propose an exponential-time multi-valued model checking algorithm to detect mutual vacuity. We provide a solution to this problem

This research is supported by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539, by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1.

978-3-9810801-5-5/DATE09 © 2009 EDAA

for which a two-valued model checker suffices, as they are typically available in industry. Our algorithm is similar in spirit to the one proposed by Chockler and Strichman [13], [14]. They propose an iterative algorithm at the automaton level, which hides literals in the automaton and iteratively adds back a *minimum* number of literals that give rise to the counterexample. The second step requires solving an NP-complete problem (minimum hitting set). In contrast, we address the problem at the formula level and systematically exploit the relationship between candidate formulae, which allows for a much more cost-effective solution. Moreover, we use counterexample traces to eliminate many candidate formulas *without* model-checking the entire design.

A recent approach by Chockler et al. determines *vacuity values* over paths in M, in order to compute the strongest formula that satisfies M and lies in the Boolean closure of the strengthenings of the original property [15]. Although the approach finds formulas stronger than mutual vacuity does, its complexity seems impractical for large formulas.

II. Preliminaries

Model checking is a technique to verify a finite-state model of a system against a user-specified property [16]. A property is typically given in a *temporal logic*; we focus in this paper on the linear-time logic LTL [17]. LTL model checking is worst-case exponential in the size of the formula. In case of a failing property, a model checker can provide a finitely representable *counterexample* evidencing the failure. In this paper, we first transform every LTL formula into *negation normal form*, where the Boolean negation operator $\neg$ occurs only immediately in front of atomic propositions.

A *partially ordered set* ("poset") $(L, \sqsubseteq)$ is a set L together with a reflexive, transitive and anti-symmetric binary relation $\sqsubseteq$; we say the elements of L are *ordered* according to $\sqsubseteq$. Let $B \subseteq L$. An element $y \in L$ is an *upper bound* for B if for every $s \in B$, $s \sqsubseteq y$. Further, y is a *least upper bound* for L if $y \sqsubseteq y'$ for every upper bound y' of B. Lower bound and greatest lower bound are defined analogously. A *lattice* is a poset $(L, \sqsubseteq)$ such that every non-empty finite subset $B \subseteq L$ has a least upper bound and a greatest lower bound. A *complete lattice* is a poset $(L, \sqsubseteq)$ such that every subset $B \subseteq L$ has a least upper bound and a greatest lower bound. Finite lattices are complete. Fig. 1 shows a complete lattice. A *chain* (*antichain*) is a set of pairwise comparable (pairwise incomparable) elements of L. The *height* of a lattice is the cardinality of the biggest *chain*. The subsets of a finite set X, partially ordered by the subset relation $\subseteq$, form the *subset lattice* of X. Given a lattice $(L, \sqsubseteq)$ and a subset $B \subseteq L$, the *(reflexive) upward closure* of B is the set $UC(B) = \{e \in L | \exists b \in B. b \sqsubseteq e\}$; note that $B \subseteq UC(B)$. The *(reflexive) downward closure* DC is defined analogously.

Notation: For a subformula occurrence ζ of a formula ϕ, we write $\phi[\zeta \leftarrow \bot]$ to mean the result of replacing ζ in ϕ by true if ζ is immediately preceded by a negation (which implies that ζ is an atomic proposition), and by false otherwise.

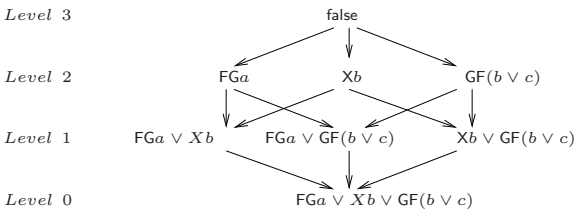

Fig. 1. Formula lattice for $\mathsf{FG}a \vee Xb \vee \mathsf{GF}(b \vee c)$

Throughout the paper, we write C for the set of atomic subformula occurrences under consideration for replacement in ϕ. As usual, 2^C denotes the power set of C. Given $Y \subseteq C$, we write ϕ_Y for the formula obtained by replacing the occurrences in Y by $\bot$. Thus, if $Y = \{y_1, \ldots, y_n\}$, then

$$\phi_Y = \phi[y_1 \leftarrow \bot, \ldots, y_n \leftarrow \bot]\,.$$

III. A Lattice of Formulae

Kupferman and Vardi present an efficient way of tightening a formula by replacing a single occurrence of a subformula by $\bot$ [2]. We first observe that the more occurrences are replaced by $\bot$ in ϕ, the stronger the resulting formula:

Theorem 1: For two formulae ϕ_Y and ϕ_X such that $Y \subseteq X \subseteq C$, $\phi_Y \rightarrow \phi_X$ is valid.

The set $\{\phi_Z : Z \subseteq C\}$ of formulae obtained by replacing certain subsets of occurrences in C by $\bot$ forms a complete lattice, with the partial order relation given by $\phi_Y \sqsubseteq \phi_X$ iff $Y \subseteq X$. Let this formula lattice be denoted by L. The top and bottom elements of L are $\phi^{stng} = \phi_C$ (all occurrences in C replaced by $\bot$) and ϕ (no occurrences in C replaced by $\bot$), respectively. Given $n := |C|$, the height of the lattice is $n + 1$; its width is the binomial coefficient $\binom{n}{\lceil n/2 \rceil}$. The upward closure of an element $\phi_Y \in L$ is $\bigcup_{X \subseteq Y} \phi_X$.

The *level* of $\phi_Y \in L$ is $|Y|$, i.e., the number of literal occurrences in ϕ that are replaced by $\bot$. The *weight imbalance* of ϕ_Y is the absolute value of the difference between the size of its upward and downward closure. A *group* is an antichain in L containing elements at the same level. The *upper (lower) weight* of a group with level l in L is the number of elements in all groups with level greater than (less than) l. The weight imbalance of a group in L is the absolute value of the difference between the upper and the lower weight of that group. A lattice element with minimum weight imbalance is said to be *balanced*. A *balanced group* is defined similarly.

As an example, the lattice for the formula $\mathsf{FG}a \vee \mathsf{X}b \vee \mathsf{GF}(b \vee c)$, with $C = \{a, b, b \vee c\}$, is given in Fig. 1. Our goal is to compute a top-most element ϕ^{top} in L that is satisfied by M. Various properties of the formula lattice can be exploited while searching for such an element.

Lemma 1: For any formula $\alpha \in L$ *satisfied* by M, ϕ^{top} belongs to $UC(\alpha)$. For any formula $\beta \in L$ *not satisfied* by M, $UC(\beta)$ does not contain any formula satisfied by M.

Given α, the lemma allows us to restrict attention to α's upward closure, which can reduce the set of candidate solutions drastically. Analogously, given β, the upward closure of β can be *eliminated*. An algorithm to compute ϕ^{top} shrinks the

solution space after every model checking run, directing the algorithm towards the solution. We now show in more detail how to search through the lattice, with the goal of verifying against M as few times as possible.

Exhaustive Search through the Lattice: Since we are interested in a top-most element in L that is satisfied by M, a reasonable strategy is to start from the top of the lattice and verify each element ϕ_Y against M (Algorithm 1). If $M \models \phi_Y$, we have found a *strongest* possible solution. If $M \not\models \phi_Y$, the algorithm eliminates $UC(\phi_Y)$ (Lemma 1). Note that $UC(\phi_Y)$ also includes ϕ_Y. We point out that the exhaustive technique returns a maximum (not maximal) set of literal occurrences that can be replaced with $\bot$.

Algorithm 1 Property Strengthening Using Exhaustive Search

EXHAUSTIVE-LATTICE-SEARCH(M, L)
Input: Model M with $M \models \phi$, formula lattice L (unmarked)
Output: a strongest lattice element satisfied by M

```
1: while true do
2:     Pick a top-most unmarked element φ_Y ∈ L
3:     if M ⊨ φ_Y then
4:         return φ_Y
5:     else
6:         mark all elements in UC(φ_Y)
7:     end if
8: end while
```

Binary Search through the Lattice: If we select a formula near the top of L, it is likely to be falsified since it is much stronger than the original formula. If we select a formula close to the bottom of L, it is likely to be satisfied, but it may not be a strongest element satisfying M. A better strategy, implemented by Algorithm 2, is therefore to select a *most balanced* group g and a most balanced element ϕ_Y within g. If M satisfies ϕ_Y, there is a solution in the upward closure of ϕ_Y (Lemma 1). In this case, the algorithm calls BINARY-LATTICE-SEARCH recursively on this closure. Once the number of unmarked elements in the current lattice is 1, the algorithm returns the unique element as a solution.

IV. GUIDED PROPERTY REFINEMENT

When $M \not\models \phi_Y$, neither of the search strategies presented so far exploits counterexample information provided by the model checker. A counterexample path π is an inexpensive guide to rule out any formula in the lattice that is not satisfied along π, since such a formula is not satisfied by the model either. We say that a formula ψ *allows* a path π if $\pi \models \psi$.

Lemma 2: If a formula $\psi \in L$ is not satisfied by M and permits a counterexample path π, the formulae in L that do not allow π can be discarded.

That is, if some formula in the lattice does not allow a valid path through the model, the upward closure of that formula can be discarded. Since a counterexample to an LTL property is a single finitely representable path, the test whether a formula allows a counterexample can be performed very efficiently using a specialized path checker, with a complexity that is worst-case linear in the size of the formula.

Algorithm 2 Property Strengthening Using Binary Search

BINARY-LATTICE-SEARCH(M, L)
Input: Model M with $M \models \phi$, formula lattice L (unmarked)
Output: a strongest lattice element satisfied by M

```
1: while true do
2:     if |L| = 1 then    // count unmarked elements
3:         return the unique element of the lattice
4:     end if
5:     Select a most balanced group g ∈ L
6:     Pick a most balanced unmarked φ_Y ∈ g
7:     if M ⊨ φ_Y then
8:         return BINARY-LATTICE-SEARCH(M, UC(φ_Y))
9:     else
10:        mark all elements in UC(φ_Y)
11:    end if
12: end while
```

The counterexample test serves as a low-cost way to reduce the set of candidate solutions in the lattice, compared to the exponential cost of model checking every candidate.

We observe that the number of elements in the upward closure of an element with level H_1 in L is less than the number of elements in the upward closure of an element with level H_2 in L where $H_1 > H_2$. That is, verification against the counterexample should be performed from bottom-to-top in order to mark as many elements as possible.

The above strategy may have very limited benefits in terms of the number of times path checking was performed compared to the number of times the formula did not allow the counterexample. The formulae at the bottom of the lattice are weaker and hence, are harder to refute. But, if refuted, they result in large reduction in the size of the lattice. Hence, we propose a strategy similar to binary lattice search to perform the counterexample tests as well. Algorithm 3 implements the strategy. The function BINARY-ELIMINATE is recursively called until the formula at the middle of the lattice allows the counterexample. Lines 6 and 10 in Algorithms 1 and 2, respectively are replaced by a call to BINARY-ELIMINATE.

Computing a most balanced group and a most balanced element: If $M \not\models \phi_Y$, the algorithm eliminates the upward closure of ϕ_Y and uses the counterexample to eliminate further elements. When an element in the lattice is marked, the weight imbalance of all the groups has to be recalculated. This requires tracking the upper weight and lower weight of each group. The weights are updated for all groups each time when an element in the lattice is marked. The weight imbalance of every individual element can be calculated on the fly using the difference between the number of incoming and outgoing edges as a rough indication of the imbalance.

We demonstrate our lattice search algorithms 1 and 2 with and without binary elimination in Algorithm 3 using an example. Consider the Kripke structure M in Fig. 2 and the property $\mathsf{FG}a \vee \mathsf{X}b \vee \mathsf{GF}(b \vee c)$. A formula lattice for the property

Algorithm 3 Eliminating lattice elements using counterexample in Binary Search manner

BINARY-ELIMINATE(ϕ_Y, L)
Input: lattice element ϕ_Y, lattice L
Effect: mark irrelevant lattice elements

```
1: let π be a counterexample witnessing M ⊭ φ_Y
2: while true do
3:     mark all elements in UC(φ_Y)
4:     Select a most balanced group g ∈ L
5:     Let φ_Y be a most balanced unmarked element in g
6:     if π ⊨ φ_Y then
7:         return;
8:     end if
9: end while
```

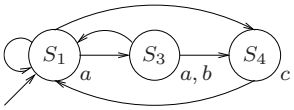

Fig. 2. Kripke structure M

with $C = \{a, b, b \vee c\}$ is given in Fig. 1.

EXHAUSTIVE-LATTICE-SEARCH: Without the counterexample test, the algorithm refutes false, $\mathsf{FG}a$, $\mathsf{X}b$, $\mathsf{GF}(b \vee c)$, and $\mathsf{FG}a \vee \mathsf{X}b$. The next candidate, i.e., $\mathsf{FG}a \vee \mathsf{GF}(b \vee c)$ is the solution. Observe that the solution does not contain the redundant subformula $\mathsf{X}b$, indicating vacuity.

When the top-most element false is refuted, let the counterexample π_1 to it be $(S_1)^w$. The counterexample test utilizes π_1 to discard more elements. Assume that the algorithm chooses the formula $\psi_1 = \mathsf{FG}a \vee \mathsf{X}b$ as a most balanced element from the balanced level 1. The element ψ_1 allows π_1 aborting the counterexample test.

Let the top-most element selected be $\mathsf{FG}a$ and the counterexample π_2 to it be $(S_1 S_4)^w$. The most balanced element ψ_1 does not allow π_2. The algorithm marks $UC(\psi_1)$. The current state of the dynamic lattice is shown in Fig. 3. The most balanced element in the balanced level 1 is $\psi_2 = \mathsf{FG}a \vee \mathsf{GF}(b \vee c)$. Since the path π_2 allows ψ_2, the counterexample test is aborted.

The algorithm now selects $\mathsf{GF}(b \vee c)$ for which the counterexample π_3 be $(S_1)^w$. The balanced level is still the same. Let the most balanced element selected be ψ_2. The element ψ_2 allows the path π_3 aborting the counterexample test. The next top-most element in the dynamic lattice is ψ_2 and is the solution. Observe that the number of model checking calls against M are reduced from six to four with the counterexample test.

BINARY-LATTICE-SEARCH: Initially, level 1 and level 2 in the lattice are balanced. The algorithm selects, say, the level 2. Assume further that the algorithm chooses the formula $\mathsf{FG}a$ as the balanced element. Since $M \not\models \mathsf{FG}a$, the upward closure of $\mathsf{FG}a$ is marked disturbing the balance the lattice. The most balanced level is shifted to level 1 in the lattice towards the solution.

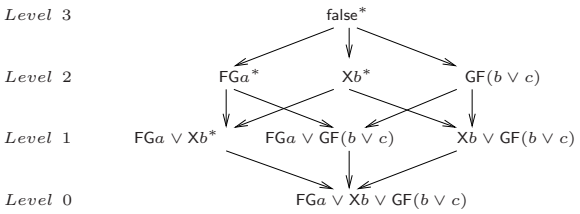

Fig. 3. Dynamic Lattice state after marking $UC(\mathsf{FG}a \vee \mathsf{X}b)$

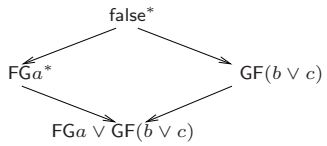

Fig. 4. State of Dynamic lattice in Fig. 3 after retaining $UC(\mathsf{FG}a \vee \mathsf{GF}(b \vee c))$

The most balanced elements in this level are $\psi_1 = \mathsf{FG}a \vee Xb$ and $\psi_2 = \mathsf{FG}a \vee \mathsf{GF}(b \vee c)$. Let the algorithm select ψ_2. Since $M \models \psi_2$, the algorithm now operates on the $UC(\psi_2)$. The algorithm selects $\mathsf{GF}(b \vee c)$, which fails in M. The lattice has only one unmarked element left, which is $\mathsf{FG}a \vee \mathsf{GF}(b \vee c)$. This formula is also the solution. The binary search requires three calls to the model checker in total.

Since $M \not\models \mathsf{FG}a$, the counterexample to it can be exploited. Let the counterexample π_1 be $(S_1 S_4)^w$. The current balanced level in the dynamic lattice is 1. Assume that the algorithm selects the element $\psi_2 = \mathsf{FG}a \vee \mathsf{X}b$. The element ψ_2 does not allow π_1. The algorithm marks $UC(\psi_2)$. Fig. 3 illustrates the state of the dynamic lattice. The balanced level continues to be 1. The algorithm selects $\psi_3 = \mathsf{FG}a \vee GF(b \vee c)$ as a most balanced element. The element ψ_3 allows the counterexample π_1 aborting the counterexample test. In the new dynamic lattice, ψ_3 is still a most balanced element. Since $M \models \psi_3$, the algorithm operates on $UC(\psi_3)$. Fig. 4 illustrates the state of the lattice. Let the balanced element selected be $\mathsf{GF}(b \vee c)$. This element is not satisfied by M. Thus, the new dynamic lattice contains only one unmarked element which is the solution.

Complexity Analysis: The number of model checking runs against M dominates the cost of our algorithms. This number is maximal, for both search techniques, when the property is not vacuous and no counterexample test eliminates any lattice element. In this case, for the exhaustive search, the number of runs against M is about equal to the size of the lattice. For the binary search, this number is about equal to $\sum_{i=1}^{\log n} \binom{n}{n/2^i}$, which can be bounded from above by $\log n \cdot \binom{n}{n/2}$. This number is only a small factor larger than the width of the lattice.

The best case for the exhaustive search technique occurs when the top-most formula in the lattice is satisfied by M, requiring only one model checking run. For the binary search algorithm, the best case occurs when none of the elements to be verified against M fail. The algorithm is recursively called on ever smaller sublattices with the height halved in each iteration. The algorithm performs about $\log n$ model checking runs against M.

Benchmark	Max. Lattice Height	Tot. Prop / Vac. Prop	Max. Strengthening Level
Chameleon	6	4/0	0
Lock	12	7/2	2
TicTacToe	10	29/1	4
Eisenberg	2	9/6	1
Heap	8	15/5	3
Coherence	6	8/3	4
Vlunc	12	2/0	0
s1269	5	9/0	0
Matrix	12	1/0	0
Needham(ns3)	9	20/8	2

TABLE I
PROPERTY CHARACTERISTICS

On average, some schedule of failing and satisfying model checking runs will occur. We can estimate in this case the number of candidate formulas in the lattice that get eliminated. In the failing case, this number is 2^k, for a formula at level k of the lattice, i.e., with k of the occurrences present: 2^k is roughly the size of the upward closure of the lattice element, which gets eliminated. In the satisfying case, the elimination count is roughly $m - 2^k$, since the upward closure is *retained*, rather than eliminated. In these estimates, m is the *current* cardinality of the (dynamic) lattice.

We emphasize that these estimates are affected by how many, and which, elements get eliminated by means of the (inexpensive) counterexample tests, whose purpose is to improve convergence of the algorithms. Their success rate is hard to predict. In the next section, we present empirical results quantify their benefit.

V. EXPERIMENTS

In this section, we present experimental results obtained with our implementation of the proposed algorithms.[1] We have implemented the proposed lattice search techniques both with and without the counterexample test, using the model checker VIS [18]. We use a significant subset of the Verilog benchmarks released with VIS. The candidate set C contains all literal occurrences in a formula. Since the counterexample path is always finitely representable, the loop-free length of the counterexample is known. We can therefore use a bounded model checker [19] from the VIS tool set to perform the counterexample test.

Table I presents information about the properties used for these benchmarks. The table lists the total number of *passing* properties that were tested, and the number of properties reported to be vacuous. The table also lists the maximum height of the formula lattice, indicating the size of the properties. The highest level in the lattice at which a solution was found is also reported, indicating how many literal occurrences were replaced by $\perp$ without sacrificing satisfaction. We found about 25% of the 104 properties checked to contain vacuity.

[1]The tool called Aardvark can be downloaded from http://www.cprover.org/aardvark/.

Tables II and III illustrate the effect of counterexample checks on the exhaustive and binary lattice search techniques.[2] Both tables are sorted by the latch count of the designs, which is a rough indicator of the hardness of the benchmarks. In both tables, #LTL is the number of lattice elements checked against the complete design, i.e., calls to the LTL Model Checker, LTL is the time taken for these calls, and Total is the total time. The time spent on the lattice operations such as finding the middle of the lattice or finding the top of the lattice is denoted by LAT. The number of times the counterexample test was performed is given in column #CE. In the same column, the number of counterexample tests that were successful, i.e., resulted in eliminating a lattice element, are listed as %SUC. The column CE lists the time spent on the counterexample checks.

Both tables indicate that the number of successful counterexample tests is considerably high. For the exhaustive and binary search technique, 62% and 81% of the counterexample tests, respectively, were successful. This causes a reduction in the number of LTL Model Checking runs, resulting in a performance improvement. Note, however, that combining the counterexample test with the exhaustive and binary search methods sometimes *degrades* their performance. As stated above, generating the counterexamples comes at a cost. In the s1296 benchmark, the time spent by VIS in generating an error trace dominates the run time. For both techniques, 55–60% of the LTL model checking time was spent on generating the counterexamples. This time is intrinsic to VIS and, hence, cannot be completely avoided. In our implementation, the counterexample generated is read by VIS each time a lattice element is checked against it. In the Lock benchmark, nearly 80% of the counterexample checking time is wasted in reading the counterexamples back into VIS. The results obtained can be further improved by implementing the presented techniques inside a model checker. The performance degradation in the Eisenberg benchmark is the combined result of the reasons mentioned above.

Adding the counterexample test to the proposed search techniques has significant benefits for the examples with larger candidate set C. The counterexample test is effective and very inexpensive for these benchmarks. The larger the set C, the larger the lattice and, hence, the larger the relative time savings due to a reduced number of expensive LTL model checking calls. On such benchmarks, we observed speedups of 30–50%.

In our benchmarks, we observe that a maximal strengthening is also a maximum strengthening. We conclude from the tables that the binary lattice search reduces the number of Model Checking runs nearly by one order of magnitude. As a result, the binary search technique is consistently faster than the exhaustive one. However, in the Lock benchmark, we observe that the time improvement is not proportional to the reduction in the number of Model Checking runs. This is because the binary method selects formulae that are harder than the ones selected by the exhaustive method. Nevertheless,

[2]The CPU times were measured on an 3 GHz Intel Xeon with 16 GB of RAM running Linux. We use the optimized BDD variable orderings that ship with the benchmarks.

Ex	Without CE test				With CE test					
	#LTL	LTL (min)	LAT (min)	Total (min)	#LTL	#CE (%SUC)	LTL (min)	CE (min)	LAT (min)	Total (min)
Chameleon	256	0.38	0.00	0.40	88	112 (42)	0.13	0.11	0.00	0.28
Lock	13543	29.38	8.40	40.40	515	2069 (80)	0.96	27.10	0.86	29.93
TicTacToe	1048	13.70	0.03	13.80	432	525 (26)	6.00	0.63	0.00	6.80
Eisenberg	24	1.73	0.00	1.73	24	15 (0)	2.05	0.31	0.00	2.38
Heap	416	1.43	0.00	1.45	158	200 (43)	0.53	0.18	0.00	0.78
Coherence	174	0.61	0.00	0.61	113	96 (19)	0.41	0.11	0.00	0.58
Vlunc	4112	5.85	2.90	9.11	1729	504 (63)	2.61	0.75	3.06	7.28
s1269	170	7.93	0.00	7.93	91	59 (52)	15.46	0.18	0.00	15.68
Matrix	4096	101.36	2.75	104.51	1590	309 (67)	39.53	0.38	1.53	42.35
Needham	2754	2637.33	0.11	2637.63	1809	500 (42)	1862.83	0.65	0.10	1864.41

TABLE II
RESULTS FOR EXHAUSTIVE SEARCH

Ex	Without CE test				With CE test					
	#LTL	LTL (min)	LAT (min)	Total (min)	#LTL	#CE (%SUC)	LTL (min)	CE (min)	LAT (min)	Total (min)
Chameleon	72	0.10	0.00	**0.11**	36	72 (50)	0.05	0.08	0.00	0.15
Lock	1833	2.76	0.78	**3.80**	119	1828 (93)	0.20	25.43	0.76	26.98
TicTacToe	270	3.60	0.01	3.63	159	263 (42)	2.20	0.31	0.01	**2.60**
Eisenberg	18	1.60	0.00	**1.60**	18	12 (0)	1.81	0.25	0.00	2.10
Heap	137	0.50	0.00	0.50	64	129 (56)	0.23	0.13	0.00	**0.40**
Coherence	56	0.20	0.00	**0.20**	46	48 (20)	0.16	0.06	0.00	0.25
Vlunc	540	0.75	0.25	**1.05**	156	454 (84)	0.23	0.46	0.25	**1.05**
s1269	62	5.50	0.00	**5.51**	35	50 (54)	12.90	0.16	0.00	13.08
Matrix	534	13.25	0.25	13.56	217	342 (92)	5.40	0.35	0.25	**6.16**
Needham	613	687.03	0.03	687.11	450	262 (62)	494.06	0.31	0.03	**494.63**

TABLE III
RESULTS FOR BINARY SEARCH (WINNING TIME PER BENCHMARK IN BOLD)

the total runtime is reduced in half.

VI. CONCLUSION

Vacuity indicates the inadequacy of a specification, or points to a design bug, and therefore needs to be addressed by the user. We provide an algorithmic way to expose the vacuity to the user in the form of a strengthened and shortened formula. In order to achieve an efficient solution, we make two contributions: 1) we define a lattice of candidate formulae and devise a binary search strategy on this lattice, and 2) we make use of refutations for failed model checking runs in order to reduce the search space. Our experimental results show that on hard benchmarks with long properties, both techniques perform best when combined.

Acknowledgments

The authors would like to thank Hana Chockler and Ofer Strichman for their suggestions.

REFERENCES

[1] D. L. Beatty and R. E. Bryant, "Formally verifying a microprocessor using a simulation methodology," in *DAC*. ACM, 1994, pp. 596–602.
[2] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," in *CHARME*. Springer-Verlag, 1999, pp. 82–96.
[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," vol. 18, no. 2. Kluwer Academic Publishers, 2001, pp. 141–163.
[4] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi, "Enhanced vacuity detection in linear temporal logic," in *CAV*. Springer-Verlag, 2003, pp. 368–380.
[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*. Springer-Verlag, 2000, pp. 154–169.
[6] M. Purandare and F. Somenzi, "Vacuum cleaning CTL formulae," in *CAV*. Springer-Verlag, 2002.
[7] A. Gurfinkel and M. Chechik, "Extending extended vacuity," in *FMCAD*. Springer-Verlag, 2004, pp. 306–321.
[8] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Y. Vardi, "Regular vacuity," in *CHARME*. Springer-Verlag, 2005, pp. 191–206.
[9] M. Samer and H. Veith, "On the notion of vacuous truth," in *LPAR*. Springer-Verlag, 2007, pp. 2–14.
[10] ——, "Parameterized vacuity," in *FMCAD*. Springer-Verlag, 2004, pp. 322–336.
[11] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, "Exploiting resolution proofs to speed up LTL vacuity detection for BMC," in *FMCAD*. IEEE Computer Society, 2007, pp. 3–12.
[12] A. Gurfinkel and M. Chechik, "How vacuous is vacuous?" in *TACAS*. Springer-Verlag, 2004, pp. 451–466.
[13] H. Chockler and O. Strichman, "Easier and more informative vacuity checks," in *MEMOCODE*. IEEE Computer Society, 2007, pp. 189–198.
[14] ——, "Before and after vacuity," *FMSD*, 2008, (to be published).
[15] H. Chockler, A. Gurfinkel, and O. Strichman, "Beyond vacuity: Towards the strongest passing formula," in *FMCAD*. IEEE Computer Society, 2008, pp. 188 – 195.
[16] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*. Springer-Verlag, 1981, pp. 52–71.
[17] A. Pnueli, "The temporal logic of programs," in *FOCS*. IEEE Computer Society, 1977, pp. 46–57.
[18] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *CAV*. Springer-Verlag, 1996, pp. 428–432.
[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*. Springer-Verlag, 1999, pp. 193–207.