To Split or to Group: From Divide-and-Conquer to Sub-Task Sharing in Verifying Multiple Properties

P. Camurati and C. Loiacono and P. Pasini and D. Patti and S. Quer
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Turin, Italy
Email: {paolo.camurati, carmelo.loiacono, paolo.pasini, denis.patti, stefano.quer}@polito.it

***Abstract*—This paper addresses the issue of property grouping, property decomposition, and property coverage in model checking problems.**

Property grouping, i.e., clustering, is a valuable solution whenever (very) large sets of properties have to be proven for a given model. As such sets often include many "easy-to-prove" and/or "similar" properties, property grouping can reduce overheads, and avoid reiteration on common sub-tasks.

On the other end of the spectrum, property decomposition can be effective whenever a given property turns-out (or it is expected) to be "hard-to-prove". Decomposition of properties into "sub-properties" follows the divide-and-conquer paradigm, and it shares with this paradigm advantages and disadvantages.

Our contribution is to present a *heuristic property manager*, running on top of a multi-engine model checking portfolio, with the specific target of optimizing productivity. We discuss, and compare, different clustering heuristics, and we exploit circuit decomposition strategies for property sub-setting. We also consider the problem of evaluating a coverage measure for properties, where the "coverage" is used to monitor the "advancement" during the (partial) verification of a given property. We finally consider estimates of the bound of a property as a measure of confidence in BMC runs.

We include preliminary experimental data indicating that the proposed approaches can provide improvements over state-of-the-art methods potentially enhancing productivity in industrial environments.

I. Introduction

Typical industrial verification frameworks are characterized by the necessity to prove a large number of properties on the same model. Nevertheless, most modern model checkers handle just single properties, verifying multiple properties one at a time in a sequential way. Little information, or no information at all, is usually retained from one verified step (property) to the following one. On the one hand, this approach does not exploit possible correlations and shared sub-problems among different properties. On the other one, it can get stuck on the hardest property without producing useful results, and with no knowledge at all on how significant the proven properties are.

Recent works on this subject [1], [2], [3] mainly study how to group, and how to sort sets of properties. Khasidashvili et. al [1] and Qin et. al [2] addressed Bounded Model Checking (BMC), and inductive Unbounded Model Checking (UMC), conjoining simple properties together, and using incremental SAT across proofs of different properties. Cabodi et al. [3] based property grouping and sorting on an affinity measure, and addressed the issue of partitioning single complex properties into sets of easier problems. In all those cases, results were limited and the issue of grouping and decomposing properties was far from being solved.

On a parallel research path, the quality and comprehensiveness of a given set of properties can be used to increase the efficacy of model checking. In this area, a first possible strategy consists in measuring the coverage of a set of properties [4], [5], [6], [7]. Coverage is usually related to the fraction of alterations to the model that would be detected by the given set of properties. A second possible approach assesses the quality of the properties detecting vacuous passes [8], [9]. Following the definition of [8] a formula φ passes vacuously in a model if it passes in the model, and there is a sub-formula $\widehat{\varphi}$ of φ that can be changed arbitrarily without affecting the outcome of model checking. The vacuous pass of a formula often signals problems in any combination of the model, its environment, and the formula itself. If a formula φ passes vacuously, it can be "vacuum cleaned", by replacing $\widehat{\varphi}$ with either `true` or `false`. In other words, vacuity detection in model checking looks for properties that hold in a model, and can be strengthened without causing it to fail. Notice that coverage and vacuity detection are complementary techniques. While coverage addresses the question of whether enough properties have been specified, vacuity detection is concerned with improving individual formulas. Nevertheless, albeit coverage is a very well known and used technique by the testing community, both coverage and vacuity are far from being broadly adopted in the formal verification area.

In this work, we focus on multiple-property detection frameworks, and we propose new heuristics to:

- Group, and order, easy-to-solve properties to avoid repeated computations.
- Decompose hard-to-solve properties into easier sub-properties.
- Gauge the progress of an entire verification process in terms of sub-properties coverage or advancement.
- Guess the SAT/UNSAT bound of a property to give some confidence on the coverage given by partial runs.

We also concentrate on how to efficiently exploit possible synergies among different verification tasks. Moreover, we oriented our effort to avoid that, while decomposing a hard-to-solve (unfeasible) property, the last property slices is (again) the hard-to-solve (unfeasible) ones. We finally present verification bound and state space estimates able to give an indication about the progress of the entire verification process.

Preliminary experimental results, on single and multi-property benchmarks, derived from the Hardware Model Checking Competition 2013, illustrate the most interesting features of our approach.

II. Background

We address systems modeled by labeled state transition structures and represented implicitly by Boolean formulas. From our standpoint, a system M is a triplet $M = (S, S_0, T)$, where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is a total transition relation. The system state space is encoded through an indexed set of Boolean variables $V = \{v_1, \dots, v_n\}$, such that a state $s \in S$ corresponds to a valuation of the variables in V and a set of states can be implicitly represented through a Boolean formula over V. With abuse of notation, hereinafter we make no distinction between a set of states and its characteristic function over V.

Given a system M, a state path of length k is a sequence of states $\pi = (s_0, \dots, s_k)$ such that $T(s_i, s_{i+1})$ is true for all $0 \leq i \leq k$. A state set R is said to be reachable if there exists a path of any length connecting a state in S_0 to another state in R. An over-approximation R^+ of a set of states R is any state set including $R : R \subseteq R^+$. Given a system M, we assume that $P = \{p_1, \dots, p_m\}$ is a set of m invariant properties to be verified over M. For each invariant $p_i \in P$, the model checking problem can be described as exploring the set of states reachable by M, while verifying whether p_i holds. If p_i is true for all such states, the invariant holds in M. On the other hand, if there exists a reachable state such that p_i is false, then the invariant does not hold for M.

For a Boolean function of n variables $\{x_1, \dots, x_n\}$, a product term in which each of the n variables appears once (in either its complemented or uncomplemented form) is called a *minterm*. Thus, a minterm is a logical expression of n variables that employs only the complement operator and the conjunction operator.

III. Property Grouping

Given a large set of properties to be verified on the same model, the straightforward approach to perform verification consists in checking them serially and individually. This strategy suffers from two potential drawbacks:

- The overhead to initialize and finalize single property checks. Though negligible in case of hard-to-check properties, the cumulative overhead can be very high with several easy-to-check properties.
- The repetition of shared sub-tasks, throughout multiple "similar" properties. This can dramatically slow down the overall process in cases where exact and/or approximate state sets are recomputed for each property and their overlap is non negligible.

In the next three subsections we will introduce some alternative possibilities to face those problems.

A. Clustering Multiple Properties

In order to overcome repeated computations and the subsequent overhead, a set P of properties can be grouped and the entire group can be checked at the same time. In [10] the authors generate and verify a "grouped-property" p (a cluster), given by the Boolean conjunction of all properties:

$$P : \; p = \bigwedge_i p_i$$

The most attractive aspect of verifying clustered properties is the opportunity to share sub-problems and learning. As the underlying circuit model M is common to all properties, this is an attractive opportunity, especially for large models with several properties that share most of their cone of influence (COI). In this scenario several verification methods, such as IC3, ITP or BDD-based verification methods, can avoid performing over and over again the same computation, such as evaluating the exact or over-estimated reachable state set. Obviously, when the grouped properties have non-overlapping COIs the COI of the grouped property can be much larger than the original one. A right balance between those two extremes can be difficult to find. Notice that when the verification of a cluster delivers a failure, the counter-example has to be examined to remove failed properties from the cluster before verifying the cluster again. The works presented in [1], [2], [3] all reap most of their benefits from this fact.

As a limit case, clustering all properties together is an option, exploiting the fact that each property p can be considered as a specification of the correct behavior for the model under verification. Obvious scalability problems are here encountered, due to extremely large properties, and subsequent excessively load on the model checker.

In [11] we have tackled the problem of optimizing multiple COI evaluation. In this paper, we study clustering strategies, and we experimentally show that it is often useful to cluster properties.

B. Single Property Verification with Learning

As the main drawback of the simultaneous multiple checks is the cumulative growth of the COI, a possible alternative is to individually verify single properties, and to inherit/propagate some learning across individual verifications. Although this idea was partially present in [1], [2], due to the shared clause database (and related learning), our target is to explore completely new techniques, specifically oriented to unbounded model checking (UMC), working at a higher level of abstraction than the SAT engine.

Our strategy is the following one:

- Given a set of properties, we classify them according to their mutual affinity or their potential ability to share sub-problems and characteristics.
- We group properties in subsets. Properties in a group are verified simultaneously. The groups are sorted by increasing expected verification effort.

- We modify UMC algorithms in order to:
  - Gain information from successful verifications, e.g., reachable states, inductive invariants, circuit transformations and simplifications.
  - Exploit assimilated data coming from the verification of other properties. We specifically deal with the problem of how to project constraints and reachable state sets from a given COI onto another one, with potentially different sets of support variables.

The overall verification of multiple properties is thus organized through a compositional-like scheme, where we sequentially prove individual properties or groups of properties, and exploit mutual propagation of learnt information.

Finally, we extend our approach beyond the case of a "given" set of properties, to be checked over the same system M, by addressing the potential decomposition of a single "monolithic" property into a set of properties. Starting from a circuit representation of an invariant property[1], it turns out that the invariant may often be expressed as the conjunction of several terms. This is true even if the property corresponds to the output of a Boolean OR gate, because it is possible to re-write those logic as a conjunction by applying simple logic rules. For example, if

$$p = z \vee (x \wedge y)$$

then we can write

$$p = (z \vee x) \wedge (z \vee y)$$

by distribution. Thus, reversing the reasoning of Section III-B the property can be viewed as a "grouped" specification, with each term of the conjunction providing a separate invariant.

Obviously, this decomposition usually gives some results for difficult model checking tasks. In other words, if the monolithic invariant is already "easy" to prove, then the overhead introduced by the decomposition may result in an overall worse performance. Further speculations of this topic are presented in Section IV.

C. Organizing Multiple Properties

As previously mentioned, the properties in P to be verified over M are initially classified according to their mutual affinity. Several criteria could be used for affinity measures, such as the amount of circuit sharing. In our current implementation, we limited ourselves to considering the sets of support variables V_i in the cone of influence of each p.

More specifically, given two invariants p_j and p_k, we compute their affinity following the Jaccard Index (or Jaccard Similarity Coefficient), which is used in statistic to compare the similarity or diversity of sample sets. Using this index, the affinity α, between p_j and p_k, is defined as the cardinality of the intersection divided by the size of the union of the sets:

$$\alpha = \frac{|V_j \cap V_k|}{|V_j \cup V_k|}$$

[1]This is a common situation, as invariants are often specified as pseudo-outputs of the model under check.

If α is larger than a given threshold, then properties p_j and p_k are grouped together and verified simultaneously, as described in Section III-B.

It should be finally observed that, before starting the model checking tasks, the grouped properties are sorted by increasing number of variables in their COI. The underlying motivation is an attempt to attack property checks by increasing complexity (complexity that is often related to the COI size and set of support variables). Although different solutions could be explored, this is a good starting point to study the simplification impact of information coming from previous verifications.

IV. Property Decomposition

In the case of hard-to-solve properties, we propose a divide-and-conquer approach, based on decomposing properties into sets of (easier-to-solve) sub-properties. In [3], the authors considered straightforward circuit decompositions, where properties were already given as conjunctions of (i.e., generated by directly AND-ing) sub-properties. However, two-level circuit rewriting and other circuit manipulations are also possible as introduced at the end of Section III-B.

Presently, we consider as hard-to-solve those properties that are unsolved after a preliminary inexpensive (in terms of wall clock time and memory) phase of model checking. However, we plan to adopt a static evaluation phase in which we take into account the COI size of the property, and an estimate of the state space it covers (see Section V). Once target properties P are selected, they are decomposed as:

$$P = \bigwedge_i p_i$$

The general goal of this sort of decomposition is obviously to produce a subset of properties individually easier to verify that their conjunct. Sub-properties are expected to have smaller COIs and/or representing sub-behaviors or be constrained to sub-spaces, possibly simpler to explore by a SAT solver. Anyhow, this sort of decomposition is not easy to obtain, as it is quite common that one or more sub-properties are as difficult to verify as the full original (monolithic) property. As a consequence, to target this problem, we also follow an orthogonal direction.

We use a SAT solver as a sub-target enumerator, and we partially relax the goal of "fully" verifying a property, i.e., we accept to (only) "partially" verify a property. More in detail, let us consider invariant properties, whose negation can be seen as a verification target (t):

$$t = \neg p$$

We consider as first sub-target a randomly chosen minterm:

$$\begin{array}{rcl} t_0 & = & SAT(t) \\ p_0 & = & \neg t_0 \end{array}$$

This property p_0 is often easy to verify, but as a sub-product of model checking (either using IC3 or interpolation), we obtain a set of over-approximated reachable states:

$$\mathcal{R}_1, \ldots, \mathcal{R}_k$$

We iteratively select as t_i sub-targets that hit the innermost reachable state ring.

The approach terminates whenever a partial target is proved reachable (and the property is thus disproved), or the set of reachable states is strong enough to prove the full p property. Even though we do not converge to full property verification, the approach is interesting as it allows the partial verification of a property, and it can provide more information than an unsuccessful verification effort ending with a time or memory overflow.

V. Coverage (Vacuity) Estimation

Given a property P and one of its signal s, Hoskote et al. [4] define coverage, of that property for the specified observed signal, as the number of reachable states in which the value of the observed signal must be checked to prove the satisfaction of the property. Nevertheless, their work left several question unanswered. Among the main ones, we recall that it was not clear how to select the observed signal, and that several properties, such as "eventually" properties, proved to have significant low coverage.

To reduce those effects, we propose a coverage metric based (again) on the size (or percentage) of reachable state set covered by the properties. Anyhow, we compute this value in an "absolute" way without any reference to observed signal. Moreover, we do not only compare the set of states covered by the property to the entire reachable state set, but we also compare the set of states covered by part of the properties or by different properties among themselves. Furthermore, in our case the state space estimation algorithm is based on a new graph-based algorithm, presented in Section V-A. Its application to the actual coverage estimate is presented in Section V-B.

A. State space estimation

Let us suppose we want to compute the number of states in which a property P is true. An exact computation is feasible only for very small and/or simple properties, for which BDDs are built in a reasonable amount of time. In most cases, however, this is not possible. We thus need a way to estimate the cardinality of the state space of a property P.

Estimating the size of a set is analogous to estimating the size of a population, a well-known problem in several domains, for example life sciences. In Computer Science many researchers are working on estimating the size of a graph, the goal being, for example, to independently compute how large the community of a social network is [12]. Kurant et al. [13] provide an extensive summary of existing techniques and suggest a new one for graph-size estimation.

Our problem is much more similar to the one found in life sciences, as we are dealing with a set of states, not with a graph where vertices (states) are connected by edges, representing a relation between them. Many population estimation techniques in life sciences rely on variants of the "capture-mark-recapture" approach: a first uniform sample of the population is taken, the individuals belonging to the sample are marked and put back in their environment, a second uniform sample is taken and the number of individuals captured twice is used to estimate the size of the population.

In our context, the uniform sampling assumption does not hold. Given n state variables, the state space may contain up to 2^n states, though in most real cases its cardinality is much less that the theoretical upper bound and the states are very scattered, i.e., the state space is very sparse. Sampling can very often result in an empty set, making it impossible to estimate the size.

Our approach to state space estimation relies on the use of a SAT solver to compute a bounded set of evaluations of the state variables that are minterms. Let this number be $\#minterms$. Notice that, to maintain the effort reasonable, we usually limit the number of SAT calls to about 2^{10}.

Starting from this bounded set, we compute new valuations at increasing Hamming distance, i.e., by randomly changing from 1 to n bits in the valuation, we produce new state variable assignments. At each step i the generated valuations are simulated and the ones that are minterms are retained. Let this number be $\#minterms_i$.

The ratio

$$\#minterms_i / \#minterms$$

measures how many valuations at distance i are still minterms with respect to the original number. Knowing the number of possible valuations at distance i, i.e., n choose i, the product of these two terms gives us an estimate of the number of minterms at Hamming distance i. Iterating from 1 to n, and adding up all the contributions, we estimate the size of the state space. However, when we have that

$$\#minterms_i < threshold$$

for example 5, as the number of minterms is statistically irrelevant, their contribution is neglected.

B. Using State Space Estimation

Given a decomposed property, we are currently studying how to use the previous estimate of the state space covered by some of the sub-properties, to evaluate a coverage/vacuity metric. The idea in this area is to have an "advancement bar", i.e., an indication of "how much" has been verified with regard to the entire property, when only a few sub-properties have been verified.

As far as estimation is concerned, we now estimate the SAT/UNSAT bound of a property by verifying the bound of one or more of its sub-properties. In this case, given a decomposition of the property we verify one (or some) of the sub-properties generated. If the verification delivers a counter-example, we are usually done. If the verification delivers a "prove" result, we then infer a "proof bound" for the entire property. This proof bound can be used to stop a subsequent BMC run on the entire property to guess (with a greater confidence) a pass result.

VI. Experimental Results

In this section we present some preliminary results on partitioning of "large" properties, and on clustering of "small" ones. Moreover, we describe some results on bound estimation. In order to analyze the full potential benefit of those techniques, we run experiments on the suite of Hardware Model Checking Competition 2013 (HWMCC'13). We consider benchmarks taken from both the single and the multi-property suites.

Our prototype software package has been implemented on top of our verification tool PdTRAV. Experiments were run on an Intel Core i7-3770, with 8 CPUs running at 3.40GHz, 16 GBytes of main memory DDR III 1333, and hosting a Ubuntu 12.04 LTS Linux distribution.

A. Clustering (Grouping) Properties

Table I reports our results on clustering heuristics. In this case we concentrate on the largest multi-property designs of the HWMCC'13. For each design, we verify the properties using a flat approach, i.e., properties are solved sequentially one at a time, and our clustering heuristic. In both cases we consider the best result in terms of wall clock time. The table reports the number of solved properties (either SAT or UNSAT) within the allowed time slot (15 minutes), of the two methods. For the sake of completeness, the table also shows the largest number of solved properties obtained during past hardware model checking competitions considering all contestants (column Best Result, HWMCC). Comparing the last two columns, advantages are evident in the majority of the designs. Clustering can then be useful to improve productivity in any verification environment where several hundreds of properties has to be verified on the same model.

Figure 1 plots the last two columns of Table I, using a logarithmic scale along both axes. It essentially shows a graphical and concise representation of the tool improvement using clustering.

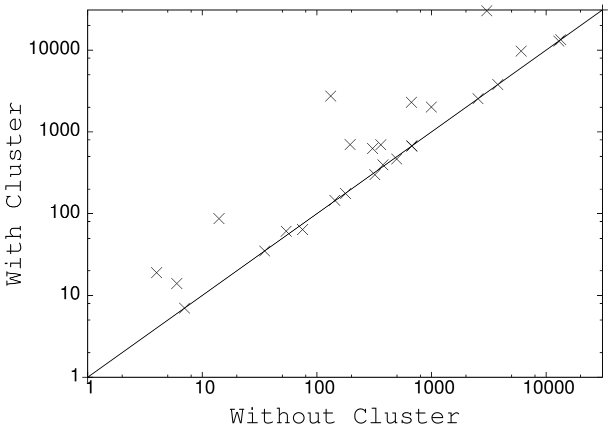

Fig. 1. Clustered approach: comparing PdTRAV results with and without clustering.

Table II compares standard PdTRAV runs, i.e., without clustering, with several other runs characterized by increasing clustering threshold. Each cluster based run is bound to a specific maximum number of clusters allowed, i.e., Max 25 means that at most 25 clusters are allowed. Threshold are selected to respect those bounds. Each column presents the maximum number of properties PdTRAV is able to complete (i.e., prove or disprove), given that specific setting, in the slotted time limit (again 15 minutes). The last column, labeled Best, represents the best result achievable through clustered approaches. Even if none of the individual configurations dominates the others, a concurrent run of the same approach with a few cluster thresholds may be really beneficial. We are currently working on strategies able to estimate the best results and based on the main benchmark characteristics.

Finally, Figure 2 compares the COI size distribution as a function of the clustering threshold. Essentially, we consider 6 representative circuits, and we apply our clustering routine with increasing threshold size. Again, for example, MAX_500 means that we use a small threshold, and we allow 500 partitions at most during clustering. For each clustering threshold, we compute the average COI size, and we express this value as a percentage of the global COI size (i.e., the total number of memory elements) for that design. Theoretically, increasing the cluster size (i.e., moving along the x axis) should imply larger COI size. Nevertheless, the graph shows that, when we reduce the number of clusters, we obtain a COI size increment in just 2 cases out of 6. Anyway, also in those two cases, the increment in the COI size is very limited. In other words, our clustering strategy is able to reduce the number of verification instances without increasing (too much) the AIG size to deal with during those instances. This again is a very useful feature in a clustering strategy, and it can improve productivity in complex multiple-property verification instances.

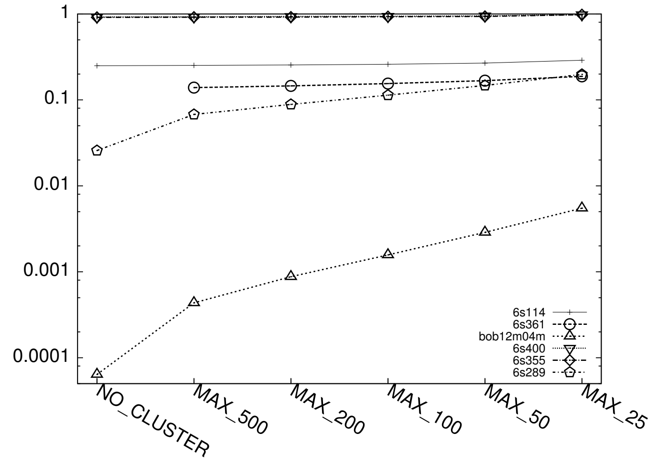

Fig. 2. COI size distribution as a function of the clustering threshold for 6 representative large HWMCC'23 benchmarks.

B. Property Decomposition and Bound Estimation

In Table III, we concentrate on deep hard-to-solve single properties on which we apply the decomposition and the bound estimation techniques introduced in Section IV and Section V-B, respectively. More specifically, for each verification

Benchmark				Best Result		
Name	#FF	#AIG Node	#Prop	HWMCC	PdTRAV without Clustering	PdTRAV with Clustering
6s361	186401	2471311	56211	5312	0	7684
6s114	101639	898079	32210	2338	997	2015
bob12m04m	439650	172107	30228	30228	3035	30228
6s400	14665	165347	13784	13555	13410	13410
6s355	15091	164299	13356	13038	12909	12907
6s289	12707	115953	10789	3070	6054	9720
6s117	23957	542556	8064	8043	132	2747
6s265	7139	155547	7300	3794	3794	3794
6s264	6360	92378	6416	3154	2549	2549
6s253	3984	101653	3893	11	306	626
6s402	13365	295376	2944	4	4	19
6s403	5468	108595	2382	2382	668	2304
6s110	807	23231	1613	1613	497	466
6s250	6185	47229	1402	1385	143	146
6s125	260713	3279044	1041	511	361	696
6s299	467369	4904114	960	265	35	35
6s176	1566	52993	952	948	321	300
6s381	12321	91582	932	150	7	7
6s380	5606	59604	897	823	17	87
6s322	80927	658407	896	144	178	176
6s405	11861	164004	874	874	195	700
6s384	14952	65415	816	793	54	61
6s275	3196	25552	673	673	673	673
6s276	3201	25549	673	673	673	673
6s277	3201	25549	673	673	673	673
6s124	6748	89589	630	619	6	14
6s413	4343	53754	597	597	377	396
6s137	32922	299551	589	1	1	0
6s301	35462	225694	569	569	75	64

TABLE I
COMPARISON OF THE BEST RESULTS ACHIEVABLE BY PdTRAV (WITH AND WITHOUT CLUSTERING) AGAINST THE OVERALL BEST RESULTS EVER OBTAINED DURING PAST HWMCC (CONSIDERING ALL CONTESTANTS).

Benchmark				Best Result						
Name	#FF	#AIG Node	#Prop	No Cluster	Max. 25	Max. 50	Max. 100	Max. 200	Max. 500	Best
6s361	186401	2471311	56211	0	4498	4500	5067	6486	7684	7684
6s114	101639	898079	32210	997	1409	1935	1938	1944	2015	2015
bob12m04m	439650	172107	30228	3035	25410	28653	30006	30228	29490	30228
6s400	14665	165347	13784	13410	13410	13410	13410	13410	13410	13410
6s355	15091	164299	13356	12909	12907	12907	12907	12907	12907	12907
6s289	12707	115953	10789	6054	9504	9720	8856	9342	7216	9720
6s117	23957	542556	8064	132	2261	2592	2349	2747	1530	2747
6s265	7139	155547	7300	3794	3794	3794	3794	3794	3794	3794
6s264	6360	92378	6416	2549	2549	2549	2549	2549	2549	2549
6s253	3984	101653	3893	306	626	548	507	380	344	626
6s402	13365	295376	2944	4	4	4	4	19	4	19
6s403	5468	108595	2382	668	2304	2208	1464	2148	2120	2304
6s110	807	23231	1613	523	497	423	448	449	466	497
6s250	6185	47229	1402	143	57	56	146	54	10	146
6s125	260713	3279044	1041	361	672	693	693	696	696	696
6s299	467369	4904114	960	35	35	35	35	35	35	35
6s176	1566	52993	952	321	273	300	242	267	294	300
6s381	12321	91582	932	7	7	7	7	7	7	7
6s380	5606	59604	897	14	61	87	14	14	13	87
6s322	80927	658407	896	178	144	162	171	170	176	176
6s405	11861	164004	874	195	700	630	513	520	356	700
6s384	14952	65415	816	54	33	51	60	54	61	61
6s275	3196	25552	673	673	673	673	673	673	673	673
6s276	3201	25549	673	673	648	673	673	673	673	673
6s277	3201	25549	673	673	648	673	673	673	673	673
6s124	6748	89589	630	6	0	13	14	4	6	14
6s413	4343	53754	597	377	312	348	366	390	396	396
6s137	32922	299551	589	1	0	0	0	0	0	0
6s301	35462	225694	569	75	3	32	40	45	64	64

TABLE II
RESULTS ON SELECTED LARGE HWMCC MULTIPLE PROPERTIES BENCHMARKS: NUMBER OF PROPERTIES COMPLETED (EITHER PROVED OR DISPROVED), WITH AND WITHOUT CLUSTERING, USING THE BEST AVAILABLE ENGINE.

Benchmark				Full Verification			Partial Verification		
Name	#PI	#FF	#AIG Node	Time	CexBound	PassBound	Time	#Part	Bound
6s207rb16	150	1747	24195	10.89	10	-	6.60	2	14
6s210b105	514	147	1716	1.49	9	-	6.20	1	8
6s215rb0	720	838	8190	3.51	9	-	8.50	3	18
6s216rb0	720	839	8220	3.71	15	-	8.16	2	18
6s218b1246	8356	2212	41858	6.12	10	-	8.00	2	45
6s289rb00529	1085	34	114	4.01	9	-	0.10	2	8
6s301rb106	24194	2521	22724	32.42	32	-	7.30	3	30
6s307rb06	1862	406	2911	81.50	15	-	9.80	3	24
6s311rb1	613	3	35554	470.94	4	-	65.15	3	22
6s318r	61	300	1621	1.44	3	-	3.50	3	32
6s335rb60	224	209	1187	2.26	6	-	0.60	2	12
6s350rb35	2281	164361	1125967	21.19	6	-	7.30	1	4
6s353rb101	7282	18544	262784	32.22	13	-	16.37	1	12
6s374b029	2102	15155	214159	286.69	11	-	85.30	1	6
6s386rb07	18612	3300	45437	19.27	14	-	13.40	2	16
6s389b11	177	3824	33149	6.40	6	-	4.20	2	28
6s401rb051	458	8328	136742	72.30	15	-	46.60	1	6
beembrdg2f1	61	56	1021	192.96	33	-	0.70	3	29
beemldelec4b1	1345	1209	28673	567.00	13	-	3.20	2	4
beempgsol5b1	541	402	8344	159.00	4	-	35.30	3	8
bob12s04	47302	21308	33823	8.58	2	-	6.34	2	2
neclaftp3002	32	2508	23171	18.11	16	-	10.60	2	12
oski1rub04	11445	1384	94374	131.51	14	-	35.30	2	15
6s131	439	812	17972	64.58	-	8	50.28	4	7
6s144	480	3337	17972	291.48	-	10	80.10	5	8
6s181	252	608	12914	43.34	-	8	20.34	4	8
6s35	77	1571	11504	588.67	-	73	301.30	1	84
6s366r	86	1998	20560	612.28	-	73	7.50	2	5
6s428rb093	410	3790	29084	746.75	-	7	244.10	1	9
6s102	72	1096	6846	488.47	-	20	97.7	5	15
6s189	479	2434	39823	214.46	-	23	180.4	5	12
6s130	439	811	25335	96.92	-	9	86.4	3	10

TABLE III
BOUND ESTIMATE FOR BOTH SAT AND UNSAT VERIFICATION INSTANCES.

instance the table reports three sections in which we summarize some design characteristics, some data regarding the verification of the original property, and some data concerning our decomposition strategy and subsequent bound estimate.

As far as the design characteristics are concerned, the table reports the benchmark name, the number of primary inputs (PI), memory elements (FF), and AIG nodes (AIG Node). For the verification of the entire property (section Full Verification), we proceed as follow. For all SAT properties, we perform verification using BMC and a time limit of 15 minutes. During those runs, we compute the bound reached to obtain a SAT result and the counter example extraction, and we report those bounds in column CexBound. For all UNSAT properties, we perform verification using our most performing engine/setting (often interpolation or IC3), and we report in column PassBound the bound at which the property passes. The wall clock times, for the entire verification process, is also reported in column Time in all cases. For our decomposition strategy (section Partial Verification), we perform decomposition, and we run an UMC engine (again, interpolation or IC3) with a time limit of 5 minutes, on one or more sub-properties. Again, we compute the bound reached to prove those sub-properties SAT or UNSAT, and we report those results in column Bound. Column Part reports the number of property partitions on which we performed verification, i.e., used to estimate the verification bound. The table shows that our bound estimates, computed in a fraction of the global verification time on a few partitions of the original property, are quite close to the exact ones. This is promising and demonstrates that it should be possible to estimate the verification bounds also for verification instances outside the verification power of modern model checking tools. Those results would give some deeper knowledge on the design, i.e., hints on how deep BMC has to be run to be confident that the property holds.

VII. CONCLUSIONS

This paper addresses the issue of property grouping, property decomposition, and property coverage (and vacuity) in model checking problems.

Our contribution is a set of heuristics targeting a productivity increment in industrial verification environments.

Preliminary experimental results are promising and demonstrate that the ideas can be really beneficial if appropriately implemented and integrated in a complete verification environment.

REFERENCES

[1] Z. Khasidashvili, A. Nadel, A. Palti, and Z. Hanna, "Simultaneous SAT-based Model Checking of Safety Properties," in *Proc. Haifa Verification Conf.*, Nov. 2005, pp. 56–75.
[2] X. Qin, M. Chen, and P. Mishra, "Synchronized Generation of Directed Tests using Satisfiability Solving," in *Proc. Int'l Conf. on VLSI Design*, Jan. 2010, pp. 351–356.

[3] G. Cabodi and S. Nocco, “Optimized Model Checking of Multiple Properties,” in *Proc. Design Automation & Test in Europe Conf.* Grenoble, France: IEEE Computer Society, Apr. 2011.

[4] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, “Coverage estimation for symbolic model checking,” in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 300–305. [Online]. Available: http://doi.acm.org/10.1145/309847.309936

[5] H. Chockler, O. Kupferman, and M. Y. Vardi, “Coverage metrics for temporal logic model checking,” in *Tools and QAlgorithms for Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 2031. Springer, 2001.

[6] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi, “A practical approach to coverage in model checking,” in *Proc. Computer Aided Verification*, ser. LNCS, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer-Verlag, Jul. 2001, pp. 66–78.

[7] N. Jayakumar, M. Purandare, and F. Somenzi, “Dos and don’ts of ctl state coverage estimation,” in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC ’03. New York, NY, USA: ACM, 2003, pp. 292–295. [Online]. Available: http://doi.acm.org/10.1145/775832.775908

[8] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, “Efficient detection of vacuity in temporal model checking,” *Form. Methods Syst. Des.*, vol. 18, no. 2, pp. 141–163, Mar. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1008779610539

[9] M. Purandare and F. Somenzi, “Vacuum Cleaning CTL Formulae,” in *Proc. Computer Aided Verification*, ser. LNCS, E. Brinksma and K. G. Larsen, Eds., vol. 2102. Copenhagen, Denmark: Springer-Verlag, Jul. 2002, pp. 485–499.

[10] R. Fraer, S. Ikram, G. Kamhi, T. Leonard, and A. Mokkedem, “Accelerated Verification of RTL Assertions based on Satisfiability Solvers,” in *Proc. High-Level Design Validation and Test Workshop*, Oct. 2002, pp. 107–110.

[11] J. Baumgartner and C. Loiacono and M. Palena and P. Pasini and D. Patti and S. Quer and S. Ricossa and D. Vendraminetto, “Fast Cone-Of-Influence Computation and Estimation in Problems with Multiple Properties,” in *Proc. Design Automation & Test in Europe Conf.* Grenoble, France: IEEE Computer Society, Mar. 2013, pp. 803–806.

[12] L. Katzir, E. Liberty, and O. Somekh, “Estimating Sizes of Social Networks via Biased Sampling,” in *Proc. World Wide Web Converence 2011*, pp. 597–605.

[13] M. Kurant, C. T. Butt, and A. Markopoulou, “Graph Size Estimate,” in *CoRR*, 2012.