Speeding up Model Checking by Exploiting Explicit and Hidden Verification Constraints

G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, S. Quer
Dipartimento di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
Email: {gianpiero.cabodi, paolo.camurati, luz.garcia, marco.murciano, sergio.nocco, stefano.quer}@polito.it

***Abstract*—Constraints represent a key component of state-of-the-art verification tools based on compositional approaches and assume–guarantee reasoning. In recent years, most of the research efforts on verification constraints have focused on defining formats and techniques to encode, or to synthesize, constraints starting from the specification of the design.**

In this paper, we analyze the impact of constraints on the performance of model checking tools, and we discuss how to effectively exploit them. We also introduce an approach to explicitly derive verification constraints hidden in the design and/or in the property under verification. Such constraints may simply come from true design constraints, embedded within the properties, or may be generated in the general effort to reduce or partition the state space. Experimental results show that, in both cases, we can reap benefits for the overall verification process in several hard-to-solve designs, where we obtain speed-ups of more than one order of magnitude.

I. Introduction

Constraints represent a key component of state-of-the-art simulation and verification tools focusing on compositional verification. In conventional simulation-based frameworks, constraints are very popular to model the environment behavior, often called test-bench. The environment model ensures that only acceptable sequences of values are applied to the design under test. Moreover, the same model can be used to monitor the outputs of the design [1], [2]. In model checking, constraints are used to represent the environment of a block under verification, i.e., the assumptions that the environment must satisfy. Furthermore, constraints are verified as assertions when the design is connected to its real environment. This methodology, usually known as assume–guarantee [3], has gained widespread industrial acceptance [4]. Today, most industrial verification languages, such as PSL [5], CBV [6] and *e* [7], include constructs to specify constraints.

As hundreds of constraints can be necessary to model the environment of commercial designs, in recent years most of the research efforts have focused on two main paths. Along the first one, researchers concentrated on how to encode and represent design constraints [6], [7]. On the second one, research groups have focused on efficient algorithms to synthesize constraints, i.e., to derive them automatically from specifications [8], [9].

In this work, our main target is to show how to effectively exploit constraints to improve the performance of model checking algorithms. To the best of our knowledge, the number of scientific works specifically addressing this issue is very limited, although it is important to leverage constraints in order to enhance the overall verification process. For instance, Paruthi et al. [10] discuss ways to adopt constraints as don't cares when building BDDs for symbolic simulation. Mony et at. [11] show how to perform several design transformations in the presence of constraints, but they do not consider the issue of improving model checking algorithms. The reason for this is that, although deemed very important in industrial settings, constraints are often disregarded while studying and tuning model checking procedures. Furthermore, constraints are often embedded in circuit models and/or verification properties in such a way that they are not directly available for specific optimizations of model checking algorithms.

The main contribution of this paper is to propose an approach to automatically extract hidden verification constraints embedded within circuits and properties. To this extent, our strategy is based on detecting terminal sub-graph components in the state graph (represented by inductive invariants), where the property under check is implied. Furthermore, we show how to directly extract from the design circuit a set of *assumptions*, implied by the properties under check, to be exploited during the verification task. Finally, we discuss how the model checking procedures can be optimized by exploiting such a set of constraints/assumptions.

The paper is organized as follows. Section II introduces our notation and a few preliminaries. Section III presents some notions on design constraints, including a motivating example for this work. Section IV discusses how constraints can be exploited by model checking tools, and Section V shows our approach to derive constraints from designs. Finally, Section VI discusses the experiments we performed, and Section VII concludes the paper.

II. Background

We address systems modeled by labeled state transition structures, and represented implicitly by Boolean formulas. The state space and the primary inputs are defined by indexed sets of Boolean variables $V = \{v_1, \ldots, v_n\}$ and $W = \{w_1, \ldots, w_m\}$, respectively. States correspond to valuations of variables in V, whereas transition labels correspond to valuations of variables in W. We indicate next states with the primed variable set $V' = \{v'_1, \ldots, v'_n\}$. In our notation, superscripts denote the time frame. For example, whenever we explicitly need time frame variables, we use $V^i = \{v^i_1, \ldots, v^i_n\}$

and $W^i = \{w_1^i, \ldots, w_m^i\}$ for variable instances at the $i-$th time frame. We also adopt the short notation $V^{i..j}$ ($W^{i..j}$) for $V^i, V^{i+1}, \ldots, V^j$ ($W^i, W^{i+1}, \ldots, W^j$)[1].

A set of states is expressed by a state predicate $S(V)$ (or $S(V')$ for the next state space). $I(V)$ is the initial state predicate. We use $P(V)$ to denote an invariant property, and $F(V) = \neg P(V)$ for its complement (as it is often used as target for bug search). With abuse of notation, in the rest of this paper, we make no distinction between the characteristic function of a set and the set itself.

$T(V, W, V')$ is the transition relation, that we assume given by a *circuit graph*, with state variables mapped to latches. Present and next state variables correspond to latch outputs and inputs, respectively. δ indicates the next state function array, i.e., the input of the $i-$th latch is fed by a combinational circuit, described by the $\delta_i(V, W)$ Boolean function[2]. As a consequence, the transition relation can be expressed as:

$$T(V,W,V') \;=\; \bigwedge_i t_i(V,W,v_i') \;=\; \bigwedge_i \left(v_i' \Leftrightarrow \delta_i(V,W)\right)$$

A state path of length k is a sequence of states $\sigma_0, \ldots, \sigma_k$ such that $T(\sigma_i, \omega_i, \sigma_{i+1})$ is true, given some input pattern ω_i, for all $0 \leq i < k$.

A state set S' is reachable from state set S in k steps if there exists a path of length k, in the labeled state transition structure, connecting some state in S to some state in S':

$$S(V^0) \wedge \left(\bigwedge_{i=0}^{k-1} T(V^i, W^i, V^{i+1})\right) \wedge S'(V^k) \;\neq\; 0$$

The image operator $\textsc{Img}(T, From)$ computes the set of states To reachable in one step from the states in $From$:

$$\begin{aligned} To(V') &= \textsc{Img}\big(T(V,W,V'), From(V)\big) \\ &= \exists_{V,W}\big(From(V) \wedge T(V,W,V')\big) \end{aligned}$$

III. Design Constraints

In this paper, we refer to the most general case of invariant constraints as functions in the form $C(V, W, V')$, where the constraint filters out *invalid* state transitions. The C constraint can be applied to the system transition relation, thus generating a new (constrained) transition relation T_C:

$$T_C(V,W,V') \;=\; T(V,W,V') \wedge C(V,W,V') \qquad (1)$$

In most practical cases, constraints are expressed as functions of present states ($C(V)$), or inputs and present states only ($C(V, W)$).

Sequential systems are usually described by transition functions (circuits), with transition relations built just within the model checker. Thus, a possible way to generate T_C is to provide an explicit representation of C.

An alternative way is to fully hide the constraints to the model checking tool. This can be achieved by modifying the property under check in the following way [11]. Each invariant property, $P(V)$, can be transformed into a new property:

$$P_C(V, v_C) \;=\; \big(v_C \Rightarrow P(V)\big) \qquad (2)$$

[1]$V^{i..j}$ and $W^{i..j}$ are appropriately defined if $i \leq j$, otherwise we conventionally define them as empty variable sets.

[2]Notice that subscripts indicate a single component in our notation.

where v_C is a new state variable, with initial value equal to 1, introduced to detect (and remember) constraints violations:

$$v_C' \;\Leftrightarrow\; \big(v_C \wedge C(V,W)\big)$$

As far as the C constraint holds, the value of v_C is 1. When C is violated, v_C becomes 0 from the next clock cycle on.

Another possible approach consists in modifying the next state functions $\delta(V, W)$ as:

$$\delta_C(V,W) \;=\; ITE(C(V,W),\ \delta(V,W),\ \sigma) \qquad (3)$$

being σ a state reachable under legal (within the C constraint) behavior. Among the available choices for σ, the identity function (next state equal to present state) is a straightforward solution:

$$\delta_C(V,W) \;=\; ITE\big(C(V,W), \delta(V,W), V\big) \qquad (4)$$

A. An Example

The Distributed Mutual Exclusion (DME) system is composed of an arbitrary number of identical cells forming a ring, as depicted in Figure 1. Each DME cell provides a connection point for one user. All users compete for the access to a (unique) shared resource. The circuit works by passing a token around the ring, via the request and acknowledge signals *lr* and *la* on the left-hand side, and *rr* and *ra* on the right-hand side. A user of the DME gains exclusive access to the resource through the request and acknowledge signals *ur* and *ua*. The invariant specification for the circuit is that two users can never be acknowledged simultaneously:

$$P \;=\; \big(\bigwedge_{i \neq j} \neg(ua_i \wedge ua_j)\big) \qquad (5)$$

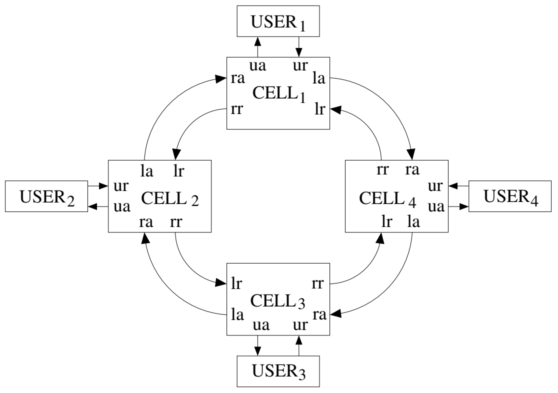

Fig. 1. A DME system with 4 cells.

Figure 2 shows an (asynchronous) implementation of the DME cells of Figure 1, due to McMillan [12]. The C device is a sequential block which stays in a memory state or propagate one of the input values to the output. The ME component behaves as an arbiter, reporting to its outputs (o_u and o_l) the value of its input signals (ur and lr) in mutual exclusion. In other words, it guarantees that o_u and o_l are never asserted at the same time.

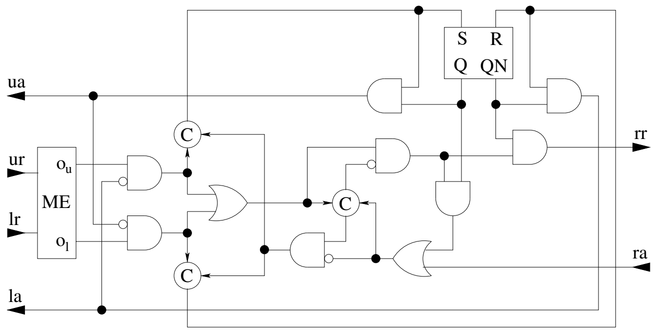

Fig. 2. A DME cell.

At the Model Checking Competition [13] each cell, implemented as in Figure 2, and the global property, expressed in Equation 5, have been modified in the following way:

1) The ME component in every cell has been replaced with two buffers, directly connecting o_u and o_l with ur and lr.
2) A new state variable v_C, with initial value equal to 1, has been added to capture and remember the cases in which the *ur* and *lr* inputs of a cell are both asserted:

$$v'_C \quad \Leftrightarrow \quad \big(v_C \wedge \bigwedge_i \neg(ur_i \wedge lr_i)\big)$$

3) The property has been constrained to the subspace in which v_C is true:

$$P_C \quad = \quad (v_C \Rightarrow P)$$

Obviously, the verification of P on the original system and of P_C on the modified system leads to the same result, even if the working spaces of the two designs are different. Nonetheless, when the number of cells is small, the original version of the DME system can be verified quite easily, whereas the modified version cannot. For example, through a forward BDD-based state space exploration, the original design with 8 cells can be verified in about 20 seconds. On the contrary, a modified DME system with just 3 cells was the smallest unsolved benchmark at both the 2007 and 2008 Model Checking Competitions [13].

IV. Exploiting Constraints in Model Checking

Common experience suggests that constraints can help reducing the visited state space in symbolic model checking, or they can generate don't care for circuit and/or state set optimizations. In this section, we describe possible ways to exploit constraints, by considering both BDD- and SAT-based verification frameworks.

A. Constrained Transition Relation

A first option to take into account a design constraint is the use of the constrained transition relation T_C, as described by Equation 1. This prevents considering illegal (with respect to C) states and/or state transitions.

However, C can also be used as a care set for circuit optimizations of the transition relation T. Furthermore, whenever state sets are represented as characteristic functions, optimizations can be performed on state sets as well. If we consider the case of state–based constraints ($C(V)$), the reachable states can be simplified using C as a *care* set:

$$To_C(V) \quad = \quad \text{SIMPLIFY}\big(To(V), C(V)\big)$$

The SIMPLIFY operator can be implemented through the constrain, restrict and compact operators with BDDs (as recently analyzed in [10]), or redundancy removal in circuit based representations.

B. Constrained Next State Circuit

In SAT-based verification (either bounded or unbounded) circuit unrollings are often preferred to explicit representation of the transition relation. Next state circuit transformations have been shown in Equations 3 and 4, but circuit optimizations, using C as a care set, are also possible. Another interesting byproduct of circuit transformations is to tighten some sets of invariants (including the inductive ones). Due to the fact that constraints limit the flow of information through the circuit, the chance to falsify a given invariant is reduced. More formally, let us consider the combinational circuit corresponding to the δ functions. Let $\{n_1, n_2, \ldots, n_M\}$ be the set of circuit nodes, and $Q(n_1, n_2, \ldots, n_M)$ an invariant. If Q holds on δ, then it holds also on δ_C, whereas the reverse implication is not true.

C. Constrained Property

Whenever the property transformation of Equation 2 is implemented, the circuit and/or the transition relation are allowed to attain illegal (outside constraints) states. In this case, circuit optimizations and inductive invariants generation have a lower impact. Furthermore, more reachable states are accepted, possibly increasing the sequential diameter of the circuit. The effect can even be amplified by approximate reachability.

V. Automated Detection of Hidden Design Constraints

In order to possibly exploit design constraints to enhance model checking (see Section IV) we seek for techniques to reveal them. Although inspired by hidden constraints, our approach is more general, as we can find constraints embedded in designs and/or properties. Our constraints are similar in their role to window functions in partitioned reachability [14]. Given a transition relation, we look for a set of constraints $\{C_i\}$ such that the overall verification task can be decomposed in sub-tasks, one for each constrained transition relation T_{C_i}, and the approach is sound and complete.

An arbitrary set of constraints is not able to guarantee completeness, due to possibly ignoring some states and/or state transitions. In partitioned reachability [14], completeness was guaranteed by exchanging messages (reached states) across partitions. For the sake of simplicity (and to support general verification approaches), we limit ourselves to constraints supporting decomposed verification under the assume–guarantee paradigm.

More specifically, we propose two simple constraining (and partitioning) schemes, sharing the common idea that verification is required just in one partition, being straightforward in the other one.

A. Inductive Constraints

Let us consider an inductive constraint $C(V)$ implying the properties, and not implied by initial states:

$$\begin{array}{rcl} C(V) \wedge T(V,W,V') & \Rightarrow & C(V') \\ C(V) & \Rightarrow & P(V) \\ \neg\big(I(V) & \Rightarrow & C(V)\big) \end{array} \tag{6}$$

As the property holds on all states in $C(V)$, and no transition exists from $C(V)$ to states in $\neg C(V)$, the verification can be constrained to states in $\neg C(V)$. Notice that, although this computation is similar to inductive verification [15], $P(V)$ is not proved inductively, because the base case, i.e., the initial state condition, is not expressed.

Whenever $C(V)$ is simply a state variable ($C(V) = v_c$), this case includes the one of Section IV-C. Once we detect such a condition, we transform it to the cases of sections IV-A and IV-B, that apply constraint–based state space restriction.

B. Implied Constraints

As an alternative and/or complement to inductive constraints, we consider constraints implied by the property under verification:

$$P(V) \quad \Rightarrow \quad C(V) \tag{7}$$

Our intuition is that it is enough to work within the subspace constrained by $C(V)$, as the latter fully includes the $P(V)$ subspace. More specifically, we show that the constraint $C(V)$ can be applied in the present state in T or δ, and not to the P property.

$$\begin{array}{rcl} T_C(V,W,V') & = & C(V) \wedge T(V,W,V') \\ \delta_C(V,W) & = & ITE\big(C(V), \delta(V,W), V\big) \end{array}$$

The transformation we adopt is legal whenever P holds on initial states, as no state transition from $P(V)$ to $\neg P(V')$ starts from states in $\neg C(V)$.

Theorem 1 If $P(V) \Rightarrow C(V)$ and $I(V) \Rightarrow P(V)$, the verification using the constrained transition relation T_C and property P is sound and complete.

Sketch of Proof. *Soundness*. The behavior of T and T_C is the same for all state transitions starting from P, thus for all state paths within P, except the last state (the next state of the last transition is possibly outside P). Given a failure state path $\sigma_0, \ldots, \sigma_k$, computed by T, such that $\neg P(\sigma_k)$, it includes a path prefix $\sigma_0, \ldots, \sigma_j (j \leq k)$, such that P holds on all states but the last one (σ_j). T_C has the same behavior of T on $\sigma_0, \ldots, \sigma_j$, hence, verification using T_C results in a failure.

Completeness. The reasoning is dual to soundness. No failure behavior is omitted while using T_C, as all failure paths have a prefix within P, except the last state.

□

C. Generating Hidden Constraints

The main challenge of the method we propose is obviously how to find really useful $C(V)$ constraints.

The $P(V)$ property itself can represent an implied constraint, as well as all terms in a conjoined decomposition of $P(V)$, $P(V) = \bigwedge_i p_i(V)$.

As our main purpose is to explore the benefits of constraining, we consider implications and equivalences, due to their ability to simplify next state functions and/or circuits. We prove invariant constraints (either inductive or implied) by SAT checks done over sets of invariant candidates. As we don't consider behaviors starting from the initial state, simulation is not a valid support for preliminary invariant pruning. To limit candidate sets and the related SAT calls, we confine our analysis to state variables, or to state variables and to close combinational nodes, depending on the size of the design. We thus look for following two sets of constraints.

- Inductive constraints:

$$C_{ind}(V) \quad = \quad \bigvee_i l_i(V)$$

  where the l_i terms are literals obtained from state variables (i.e., state variables or their complementations) that satisfy Equations 6. The set is generated by SAT-based checks over all literals of state variables. In other words the constraint is the union (disjunction) of literal constraints in such a way that its complement is a conjunction:

$$\neg C_{ind}(V) \quad = \quad \bigwedge_i \big(\neg l_i(V)\big)$$

- Implication constraints:

$$C_{impl}(V) \quad = \quad \bigwedge_{i,j} \big(l_i(V) \Rightarrow l_j(V)\big)$$

  such that each individual implication satisfy Equation 7. We perform multiple SAT calls to possibly falsify each $\big(l_i(V) \Rightarrow l_j(V)\big)$ implication. Whenever $l_i(V) \Rightarrow l_j(V)$ and $l_j(V) \Rightarrow l_i(V)$, $l_i(V)$ and $l_j(V)$ are equivalent, so we directly merge them on the circuit.

D. Hidden Constraints and Reachability

The classes of automatically detected constraints, as previously described, share similarities and common ideas with backward reachability approaches. *Inductive constraints* of Section V-A are cognate of over-approximate backward state sets, whereas our usage of *implied constraints* of Section V-B is close to target enlargement.

Inductive invariants in backward reachability are described in [16] as the dual case of forward induction. As a matter of fact, the (forward inductive) $C(V)$ constraint of section V-A can be seen as the complement of over-approximate backward reachable states. Let $BckR(V)$ represent the set of states backward reachable from $F(V)$, and let $BckR^+(V) = \neg C(V)$ represent the complement of the $C(V)$ inductive constraint. As $C(V)$ is a fixed point, $C(V)$ and $F(V)$ are mutually unreachable. Then $BckR^+(V) \supseteq BckR(V)$. Our approach thus differs from the typical inductive model checking schemes, as we seek for invariants implying (and not implied by) the property. Furthermore, the complement of our set of constraints is implied by the target (negated property). Forward inductive invariant generation typically relies on simulation filtering out sets of invariant candidates, to be individually proved by BDD- and/or SAT-based checks. Then a dual approach is poorly applicable in the backward direction (backward simulation from target states). As a compromise choice, we avoid simulation and we restrict our proofs to implications/equivalences involving state variables.

Target enlargement is the closest known approach to our search and usage of implied constraints. Any subset of backward reachable states can be seen as an enlarged substitute of the target. In a similar way, the complement of a constraint implied by $P(V)$ (or the complement of $P(V)$ itself) is a subset of backward reachable states. Moreover, our main focus is to find constraints able to simplify further steps, whereas target enlargement approaches typically rely on exact pre-image steps, and take entire state sets as new targets. Target enlargement is orthogonal to our method, as sets of constraints implied by tightened properties (complement of enlarged targets) could be considered as a tighter $C(V)$ implied constraint

VI. Experimental Results

We implemented a prototype version of our methodology, on top of the PdTrav tool, a state-of-the-art verification framework which participated to the last two Model Checking Competitions [13]. This framework allows experiments with different verification methodologies going from BDD-based forward reachability analysis [17] to interpolant-based verification [18]. In all cases, inductive invariants [15], [19] are applied on the design up-front, and all successfully verified instances are ruled-out from the subsequent analysis.

We performed experiments on 756 circuits, among which 645 are taken from the 2008 Model Checking Competition suite [13], and 111 are coming from industry. All experiments ran on a Dual Core Pentium IV 3 GHz Workstation with 3 GBytes of main memory, hosting a Debian Linux distribution. In all the experiments, we used a 900 seconds time limit and a 3 GBytes memory limit.

We proceeded in two separate phases.

In the first phase, we ran our methodology to discover the number of inductive and implied constraints (see Section V) on all circuits of our suite. Figures 3 and 4 plot the distribution of the considered designs (HWMCC'08 and industrial) as a function of the number of inductive and implied constraints discovered. Out of the $645 + 111$ instances (HWMCC plus industrials) we have $205 + 108$ designs (HWMCC plus industrials) with at least one constraint. Our methodology is then able to find inductive/implied constraints in about 30% of the HWMCC'08 designs, and 97% of the industrial circuits. As represented by the graphs, the number of constraints may be quite high in several industrial instances.

In the second phase, we compare several model checking strategies with and without the reduction techniques presented in Section IV on the circuits with at least one inductive or

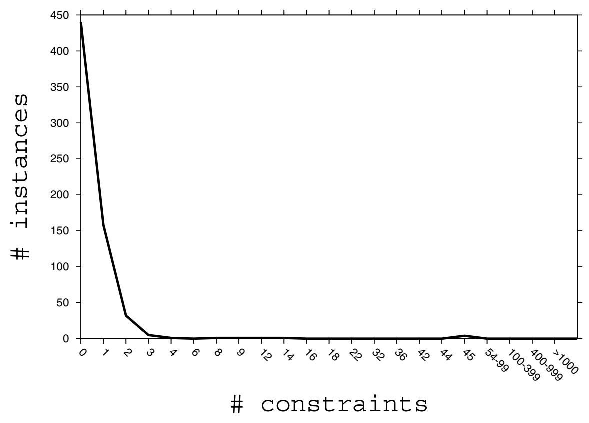

Fig. 3. HWMCC'08 benchmarks: Number of design instances as a function of inductive/implied constraints.

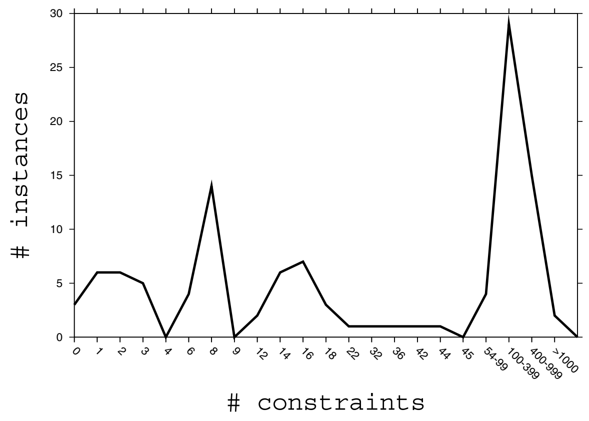

Fig. 4. Industrial benchmarks: Number of design instances as a function of inductive/implied constraints.

implied constraint. Table I reports all significant data, i.e., experiments that required more than 30 seconds of CPU time, and could be completed within the time limit of 900 seconds. The meaning of the columns is the following. Model is the instance name, # PI, # FF and Nodes represent the number of primary inputs, memory elements, and AIG nodes of the circuit, respectively. Column Method shows that most of the properties are proved with interpolant-based verification [18] (ITP), even if a few easier cases were completed faster with BDD-based forward verification [17] (BDD).

For both the Original Strategy and the New Strategy we provide the verification time in seconds. Moreover, columns # IndC and # ImpC report the number of inductive and implied constraints found for the design (their sum is the value plotted in Figures 3 and 4). The data in the table clearly show that, when applicable, the methodology strongly simplifies the verification problem, as we observed advantages up to more than one order of magnitude. Moreover we were able to complete benchmarks unsolvable with the original techniques. Among them, let us notice that the design named cmudme1 is the DME system presented in Section III-A, and it is also the smallest unsolved benchmark at both the 2007 and 2008

TABLE I

VERIFICATION RESULTS: DETAILED ANALYSIS OF SOME DESIGNS. A DASH (−) INDICATES A TIME OVERFLOW (WITH A TIME LIMIT OF 900 SECONDS).

Model	#PI	#FF	#Nodes	Method	Original Strategy	New Strategy		
					Time [s]	#IndC	#ImpC	Time [s]
cmudme1	54	55	380	BDD	–	1	0	3.90
cmudme2	56	55	549	BDD	–	1	0	4.11
intel004	82	50	380	ITP	85.41	1	1	8.99
intel006	345	182	1718	ITP	437.63	1	1	149.93
intel049	136	77	1143	ITP	273.65	1	1	17.10
intel055	222	125	2841	ITP	–	1	1	35.92
intel056	301	147	2603	ITP	–	1	1	26.06
intel059	280	141	2410	ITP	–	1	1	24.43
intel063	288	241	2305	ITP	38.21	1	1	7.09
eijkbs3271	26	261	1726	ITP	850.21	0	9	350.12
eijkbs3384	43	427	2323	ITP	514.33	0	4	211.29
eijkbs6669	78	322	3313	ITP	–	0	14	648.15
Industrial_A1	20	589	5617	ITP	67.79	4	1179	6.58
Industrial_A2	55	622	5490	ITP	67.92	9	1245	7.20
Industrial_A3	436	1298	1155	ITP	79.64	936	2597	30.98
Industrial_A4	192	859	6398	ITP	68.86	3	1719	6.78
Industrial_A5	58	299	2661	ITP	30.72	19	599	3.81
Industrial_A6	107	293	2434	ITP	34.65	241	587	3.79
Industrial_A7	108	294	2394	ITP	30.66	242	589	3.79
Industrial_A8	312	300	4494	ITP	35.06	3	7	4.22

Model Checking Competitions [13].

VII. CONCLUSIONS

Constraints represent a key component of state-of-the-art verification tools based on compositional approaches and the assume–guarantee reasoning. In this paper, we analyze the impact of constraints on the performance of model checking tools, and we discuss how to effectively exploit them. We also introduce an approach to explicitly derive verification constraints that are hidden in the design and/or in the property under verification. Experimental results show interesting performance enhancements (to more than one order of magnitude in terms of CPU time) in several hard-to-solve verification instances.

REFERENCES

[1] J. Baumgartner, H. Mony, and A. Aziz, "Optimal Constraint-Preserving Netlist Simplification," in *to appear in Proc. Formal Methods in Computer-Aided Design*, 2008.

[2] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling Design Constraints and Biasing in Simulation using BDDs," in *Proc. Int'l Conf. on Computer-Aided Design*. San Jose, California, USA: IEEE Press, 1999, pp. 584–590.

[3] R. Alur and T. A. Henzinger, "Reactive Modules," in *Formal Methods in System Design*. IEEE Computer Society Press, 1996, pp. 207–218.

[4] C. Pixley, "Integrating Model Checking into the Semiconductor Design Flow," *Electronic Systems Technology & Design*, pp. 67–74, Mar. 1999.

[5] "Accelera. PSL RML, http://www.eda.org/vfv."

[6] M. Kaufmann, A. Martin, and C. Pixley, "Design Constraints in Symbolic Model Checking," in *Proc. Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 477–487.

[7] Y. Hollander, M. Morley, and A. Noy, "The e Language: A fresh Separation of Concerns," in *Proc. Technology of Object-Oriented Languages and Systems*, 2001.

[8] M. S. Jahanpour and O. A. Mohamed, "Automatic Generation of Model Checking Properties and Constraints from Production Based Specifications," in *Midwest Symposium on Circuits and Systems*, Jul. 2004, pp. 435–438.

[9] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint Synthesis for Environment Modeling in Functional Verification," in *Proc. Design Automation Conf.* New York, NY, USA: ACM, 2003, pp. 296–299.

[10] V. Paruthi, C. Jacobi, and K. Weber, "Efficient symbolic simulation via dynamic scheduling, don't caring, and case splitting," in *Proc. Correct Hardware Design and Verification Methods*, ser. LNCS, D. Borrione and W. Paul, Eds., vol. 3275. Edimburgh, Scotland, UK: Springer, 2005, pp. 114–128.

[11] H. Mony, J. Baumgartner, and A. Aziz, "Exploiting Constraints in Transformation-based Verification," in *Proc. Correct Hardware Design and Verification Methods*, ser. LNCS, D. Borrione and W. Paul, Eds., vol. 3275. Edimburgh, Scotland, UK: Springer, 2005, pp. 269–284.

[12] K. L. McMillan, "Symbolic Model Checking," Ph.D. dissertation, Boston, Massachussetts, 1992.

[13] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc."

[14] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Reachability Analysis Using Partitioned–ROBDDs," in *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, Nov. 1997, pp. 388–393.

[15] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT Solver," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, W. A. Hunt and S. D. Johnson, Eds., vol. 1954. Austin, Texas, USA: Springer, Nov. 2000, pp. 108–125.

[16] P. Bjesse and K. Claessen, "SAT–Based Verification without State Space Traversal," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 1954. Austin, TX, USA: Springer, 2000.

[17] R. K. Brayton et al., "VIS," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, M. Srivas and A. Camilleri, Eds., vol. 1166. Palo Alto, California: Springer, Nov. 1996, pp. 248–256.

[18] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, W. A. Hunt and F. Somenzi, Eds., vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[19] C. A. J. van Eijk, "Sequential Equivalence Checking Based on Structural Similarities," *IEEE Trans. on Computer-Aided Design*, vol. 19, pp. 814–819, Jul. 2000.