Property-Driven Partitioning for Abstraction Refinement

Roberto Sebastiani[1]⋆, Stefano Tonetta[2], and Moshe Y. Vardi[3]⋆⋆

[1] DIT, Università di Trento, Italy. rseba@dit.unitn.it
[2] University of Lugano, Switzerland. tonettas@lu.unisi.ch
[3] Dept. of Computer Science, Rice University, USA. vardi@cs.rice.edu

Abstract. Partitioning and abstraction have been studied extensively both in hardware and in software verification. The abstraction is typically partitioned according to the system design in the case of hardware or the control graph in the case of software. In this work we build on previous work on Property-Driven Partitioning (PDP), a hybrid Symbolic Model-Checking (SMC) technique for ω-regular properties in which the state space is partitioned according to the states of the property automaton. We investigate a new paradigm for abstraction refinement in SMC, which combines abstraction and PDP: each PDP partition may contain a different abstraction, so that it can be refined independently from the others; in case of a spurious counterexample π, the system is refined only in those partitions that are necessary to rule out π. We performed a preliminary experimental evaluation comparing standard Counterexample-Guided Abstraction Refinement (CEGAR) with its partitioned counterpart, which confirmed that the partitioned technique always allows for using coarser abstractions. While earlier work has shown that PDP almost always improves the performance of SMC, our experiments here show that this is not always the case for partitioned abstraction refinement, as in some cases the overhead due to the localization of the abstraction is too high.

1 Introduction

Verifying properties of finite-state machines is a fundamental problem in formal design verification. *Symbolic Model Checking* (SMC) [2] has been successful at verifying temporal specifications. In particular, LTL model checking is solved by building an automaton A with the complementary language of the property, computing the product of A with the system M and checking for emptiness [17]. Typically, the product is represented either explicitly, by enumerating its elements, or symbolically, by means of propositional formulas. The main obstacle to model checking is the *state-space explosion*, that is, the state-space is often too large to be handled.

One of the most efficient technique to reduce the state-space in SMC is *abstraction* [6]. An abstract system is simply an approximation of the original one. A *conservative*

⋆ Supported in part by ORCHID, a project sponsored by Provincia Autonoma di Trento, and by a grant from Intel Corporation.

⋆⋆ Supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, and ANI-0216467, by BSF grant 9800096, and by Texas ATP grant 003604-0058-2003.

abstraction preserves all the behaviors of the concrete system. This way, if a universal property is satisfied by an abstract model, then the property is correct also in the concrete model. On the contrary, false negatives may be produced. An abstract counterexample that does not correspond to any concrete behavior is called *spurious*. When such a counterexample is found, the abstraction is *refined*, i.e., we add enough information to the system in order to rule out the spurious counterexample. This process is repeated until the property is proved correct or a concrete counterexample is found. Such a loop is known as *Counterexample-Guided Abstraction Refinement* (CEGAR) [5].

Partitioning and abstraction have been extensively studied both in hardware and in software verification. For hardware, the abstraction is typically partitioned according to the design that usually is composed of submodules (see, e.g., [3, 14]). For software, the partitioning is guided by the control-flow graph of the program. In particular, *Lazy Abstraction* (LA) [11] produces different levels of precision in different parts of the program. The abstraction can be refined locally to some control paths [11] or to some control locations [10].

In [15, 16], motivated by previous work on *Generalized Symbolic Trajectory Evaluation* (GSTE) [20], *Property-Driven Partitioning* (PDP) has been proposed as a new hybrid approach to LTL model checking. In this approach, the property automaton A is constructed explicitly, but its product with the system is represented in a partitioned fashion. If the state space of the system is $\mathcal{S}$ and that of the property automaton is $\mathcal{B}$, then PDP maintains a subset $Q \subseteq \mathcal{S} \times \mathcal{B}$ of the product space as a collection $\{Q_b : b \in \mathcal{B}\}$ of sets, where each $Q_b = \{s \in \mathcal{S} \mid (s,b) \in Q\}$ is represented symbolically. Thus, PDP maintains an array of BDDs instead of a single BDD to represent a subset of the product space. Based on extensive experimentation, it has been shown that the hybrid approach of PDP is superior to pure symbolic model checking.

In GSTE, this partitioning technique is embedded into a quaternary-abstraction framework. Each partition stores a quaternary assignment (or a set of quaternary assignments, if we use *symbolic indexing* [19]). Quaternary abstraction, also known as Cartesian abstraction, is a powerful mechanism that merges a set of states into an abstract state: the value "unknown" is used for variables that have different assignments in the set. The use of this "unknown" value gives different levels of approximation. By combining PDP with quaternary abstraction, GSTE has a different abstraction precision for each partition.

Building on the ideas of LA and GSTE, we explore in this paper the combination of PDP with predicate abstraction. We investigate a localized-abstraction framework, called *partitioned abstraction*, where each partition of PDP may contain a different abstraction. This way, we allow a flexible abstraction where every partition can be refined independently from the others. Intuitively, the state of the automaton corresponds to a particular phase of the operation of the circuit, and we expect that different partitions contain the appropriate abstractions for different phases.

We adapted the CEGAR loop to this framework. In PDP, a counterexample is a sequence of state-partition pairs. When the counterexample is spurious, the failure of the concretization must happen in some partition. Thus, we tuned the refinement in order to refine the system only in those partitions that are necessary to rule out the spurious counterexample.

Although *partitioned abstraction* is independent of any specific abstraction technique, we implemented it in NuSMV [4] in combination with localization reduction and the score-based refinement heuristic of [18]. We performed an experimental evaluation providing a comparison between the standard non-partitioned refinement loop with its partitioned counterpart. We evaluated the trade-off between the disadvantages of dealing with several abstractions at the same moment and the advantages of using coarser abstractions when possible. Our testing shows that this technique can be very efficient and can solve model-checking problems where the non-partitioned version fails. Unlike SMC, however, where partitioning always improved the performance of verification, here, partitioning the abstraction may involve high overhead. These experiments, for the first time[4] , compare an abstraction-localization technique with a uniform one. The results show that abstraction partitioning is successful at allowing verification with coarser abstractions. In some cases this may lead to an order-of-magnitude improvement in running time. Nevertheless, in many cases the improvement in the coarseness of the abstractions does not compensate for the overhead of having to compute many abstractions. Overall, abstraction partitioning, in spite of its initial promise, does not lead to a consistent improvement in running time.

2 Background

2.1 Symbolic Model Checking

Given a system M and a property φ, the model checking problem consists of checking if M is a model of φ. The semantics of M and φ are based on a shared set *PROP* of atomic propositions. Both the system and the property define languages over finite or infinite words. The alphabet of such languages is $\Sigma = 2^{PROP}$.

The system M is described with a set $\mathcal{V}$ of state variables. We use $\mathcal{V}'$ to denote the set of next state variables $\{v'\}_{v\in\mathcal{V}}$, where v' represents the next value of v. We represent the system M as a symbolic finite-state machine (FSM), i.e., a tuple $\langle \mathcal{V}, I, T, \mathbb{L}\rangle$, where $\mathcal{V}$ is a set of state variables, $I(\mathcal{V})$ is a formula that represents the initial condition, $T(\mathcal{V},\mathcal{V}')$ is a formula that represents the transition relation, $\mathbb{L} = \{L_a(\mathcal{V})\}_{a\in PROP}$ is a set of formulas that associate the atomic propositions with states.

The set $\mathcal{S}_{\mathcal{V}}$ of states is given by all truth assignments to the variables $\mathcal{V}$. Given a state s, we use s' to denote the corresponding truth assignment to the next state variables, i.e. $s' = s[\mathcal{V}'/\mathcal{V}]$. A state s is labeled with the atomic proposition a iff $s \models L_a(\mathcal{V})$. A state s is initial iff $s \models I(\mathcal{V})$. Given two states s_1 and s_2, there exists a transition between s_1 and s_2 iff $s_1, s_2' \models T(\mathcal{V},\mathcal{V}')$.

A finite (resp., infinite) path of the FSM is a finite (resp. infinite) sequence of states $\pi = s_1,\ldots,s_k$ (resp. $\pi = s_1, s_2,\ldots$) such that s_1 is an initial state and, for $1 \leq i < k$ (resp. $i \geq 1$), there exists a transition between s_i and s_{i+1}. A lasso-shape path is a finite path

[4] Henzinger et al. [10] compared BLAST with and without the technique of predicate localization and showed that it led to a running-time improvement in one case out of six. Since BLAST itself uses lazy abstraction, which can be viewed as a weak form of abstraction localization, it is hard to draw conclusions from that work about the advantage of abstraction localization.

π with a point l, $1 \leq l < |\pi|$ (called loop-back point), such that $s_l = s_{|\pi|}$. Thus, a lasso-shape path represents an infinite path (though, not all infinite paths are lasso shaped).

An infinite word $w \in \Sigma^\omega$ is accepted by the FSM M if there exists a path π of M such that $\pi(i) \models L_{w(i)}(\mathcal{V})$, for all $i \geq 1$. The language $\mathcal{L}_M$ of M is defined as the set of all infinite words accepted by M. Given the language $\mathcal{L}_\varphi$ of the property φ, we say that the system M satisfies the property φ ($M \models \varphi$) iff $\mathcal{L}_M \subseteq \mathcal{L}_\varphi$. In other words, all paths of the system must correspond to a word accepted by the property.

LTL model checking A Büchi automaton (BA) is a tuple $\langle \mathcal{B}, b_0, \Sigma, \delta, F \rangle$, where $\mathcal{B}$ is a set of states, $b_0 \in \mathcal{B}$ is the initial state, $\delta \in \mathcal{B} \times \Sigma \times \mathcal{B}$ is the transition relation, and $F \subseteq \mathcal{B}$ is the fairness condition. A fair run of a BA A over the infinite word $w = l_1, l_2, \ldots \in \Sigma^\omega$ is an infinite sequence of states $\pi = b_1, b_2, \ldots$ such that $b_1 = b_0$ and, for $i \geq 1$, $(b_i, l_i, b_{i+1}) \in \delta$, and $\pi(i) \in F$ for infinitely many i. The word w belongs to the language $\mathcal{L}_A$ of A iff there exists a fair run over w.

If φ is an LTL property, the model checking problem is usually solved by building a BA $A_{\neg\varphi}$ with the complementary language of the property, computing its product with the system, and then checking for emptiness [17]. Extant model checkers use either a pure explicit-state approach, e.g., in SPIN [12], or a pure symbolic approach, e.g., in NuSMV [4]. To check language containment, a symbolic model checker implements a fixpoint algorithm [2]. Sets of states are manipulated by using basic set operations such as intersection, union, complementation, and the preimage and postimage operations. Since sets are represented by predicates on Boolean variables, intersection, union and complementation are translated into $\wedge$, $\vee$ and $\neg$ respectively. The preimage and postimage operations are translated into the following formulas:

$$\begin{aligned} preimage(Q) &= \exists \mathcal{V}'((Q[\mathcal{V}'/\mathcal{V}])(\mathcal{V}') \wedge T(\mathcal{V}, \mathcal{V}')), \\ postimage(Q) &= (\exists \mathcal{V}(Q(\mathcal{V}) \wedge T(\mathcal{V}, \mathcal{V}')))[\mathcal{V}/\mathcal{V}']. \end{aligned}$$

2.2 Abstraction

Given two systems $M = \langle \mathcal{V}, I, T, \{L_a\}_{a \in PROP} \rangle$ and $\hat{M} = \langle \hat{\mathcal{V}}, \hat{I}, \hat{T}, \{\hat{L}_a\}_{a \in PROP} \rangle$, a relation $H(\mathcal{V}, \hat{\mathcal{V}})$ is a *simulation relation* iff the following conditions hold:

- every initial state of M corresponds to an initial state of $\hat{M}$; namely, if $s \models I(\mathcal{V})$, then there exists a state $\hat{s}$ of $\hat{M}$ such that $\hat{s} \models \hat{I}(\hat{\mathcal{V}})$ and $s, \hat{s} \models H(\mathcal{V}, \hat{\mathcal{V}})$;
- every transition of M corresponds to a transition of $\hat{M}$; namely, if $s_1, \hat{s}_1 \models H(\mathcal{V}, \hat{\mathcal{V}})$, and $s_1, s_2' \models T(\mathcal{V}, \mathcal{V}')$, then there exists a state $\hat{s}_2$ of $\hat{M}$ such that $s_2, \hat{s}_2 \models H(\mathcal{V}, \hat{\mathcal{V}})$ and $\hat{s}_1, \hat{s}_2' \models \hat{T}(\mathcal{V}, \mathcal{V}')$;
- if two states are related by H, they must have the same labels; namely, if $s, \hat{s} \models H(\mathcal{V}, \hat{\mathcal{V}})$ then $s \models L_a(\mathcal{V})$ iff $\hat{s} \models \hat{L}_a(\hat{\mathcal{V}})$, for all $a \in PROP$.

If such relation exists, we say that $\hat{M}$ is an *abstraction* of M, or M *refines* $\hat{M}$ ($M \preceq_H \hat{M}$). Intuitively, every path π of M is "simulated" by some path $\hat{\pi}$ of $\hat{M}$ that accepts the same infinite word. In terms of languages, if $M \preceq_H \hat{M}$, then $\mathcal{L}_M \subseteq \mathcal{L}_{\hat{M}}$. Thus, simulation preserves properties, so that if $M \preceq_H \hat{M}$ and $\hat{M} \models \varphi$ then $M \models \varphi$ (though, in general, the reverse does not hold).

Given the relation H, we define the *abstraction function* $\alpha_H : \mathcal{S}_{\mathcal{V}} \rightarrow 2^{\mathcal{S}_{\hat{\mathcal{V}}}}$ and the *concretization function* $\gamma_H : \mathcal{S}_{\hat{\mathcal{V}}} \rightarrow 2^{\mathcal{S}_{\mathcal{V}}}$ as follows:

- $\alpha_H(s) = \{\hat{s} \in \mathcal{S}_{\hat{\mathcal{V}}} \mid s, \hat{s} \models H(s,\hat{s})\}$,
- $\gamma_H(\hat{s}) = \{s \in \mathcal{S}_{\mathcal{V}} \mid s, \hat{s} \models H(s,\hat{s})\}$.

We extend these functions to paths: if $\hat{\pi} = \hat{s_1}, \hat{s_2}, \ldots$ is a path of $\hat{M}$ then $\gamma_H(\hat{\pi})$ is a set of paths of M such that $\pi \in \gamma_H(\hat{\pi})$ iff $\pi = s_1, s_2, \ldots$ is a path of M and $s_i \in \gamma_H(\hat{s_i})$ for $1 \leq i \leq |\hat{\pi}|$ (similarly for α_H).

In *predicate abstraction* [9], the abstract state-space is described with a set of predicates; each predicate is represented by an abstract variable. Given an FSM M, we select a set $\mathbb{P}$ of predicates, such that each predicate P is a formula over the variables $\mathcal{V}$ that characterizes relevant facts of the system. For every $P \in \mathbb{P}$, we introduce a new abstract variable v_P. H is defined as follows: $H(\mathcal{V}, \mathcal{V}_{\mathbb{P}}) = \bigwedge_{P \in \mathbb{P}} v_P \leftrightarrow P(\mathcal{V})$, i.e., for every concrete state s, the corresponding abstract state $\hat{s}$ assigns v_P to true iff $s \models P(\mathcal{V})$. In the *minimal* (or *most accurate*) abstraction (wrt. $\mathbb{P}$), every abstract transition corresponds to at least one concrete transition. This involves a quantification of the concrete variables that may be computationally expensive. The usual technique to get an approximation of the minimal abstraction is *early quantification* [6]: the existential quantifications are pushed inside conjunctions; in this way, we simplify the expressions by losing precision.

Localization reduction A particular case of abstraction uses the *localization reduction* technique [13]: suppose the system is defined functionally, i.e., $M = \langle \mathcal{V}, I, T, \mathbb{L} \rangle$, where

- $I(\mathcal{V}) = \exists \mathcal{W}(\bigwedge_{v \in \mathcal{V}} v = f_v^I(\mathcal{W}))$ for some functions f_v^I and input variables $\mathcal{W}$,
- $T(\mathcal{V}, \mathcal{V}') = \exists \mathcal{W}(\bigwedge_{v \in \mathcal{V}} v' = f_v^T(\mathcal{V}, \mathcal{W}))$ for some functions f_v^T and input variables $\mathcal{W}$.

Localization reduction consists in choosing a set of state variables, removing the corresponding constraints, and considering them as input variables. Let $\hat{\mathcal{V}} \subseteq \mathcal{V}$, and $\breve{\mathcal{W}} = \mathcal{V} \setminus \hat{\mathcal{V}}$. We can build the abstraction $\hat{M} = \langle \hat{\mathcal{V}}, \hat{I}, \hat{T}, \hat{\mathbb{L}} \rangle$ where:

- $\hat{I}(\hat{\mathcal{V}}) = \exists \mathcal{W}(\bigwedge_{v \in \hat{\mathcal{V}}} v = f_v^I(\mathcal{W}))$,
- $\hat{T}(\hat{\mathcal{V}}, \hat{\mathcal{V}}') = \exists \breve{\mathcal{W}} \exists \mathcal{W}(\bigwedge_{v \in \hat{\mathcal{V}}} v' = f_v^T(\hat{\mathcal{V}}, \breve{\mathcal{W}}, \mathcal{W}))$,
- $\hat{\mathbb{L}} = \{\hat{L}^a\}_{a \in PROP}$, where $\hat{L}^a(\hat{\mathcal{V}}) = \exists \breve{\mathcal{W}}(L_a(\hat{\mathcal{V}}, \breve{\mathcal{W}}))$.

We call $\breve{\mathcal{W}}$ *invisible variables* (also known as *abstracted* or *freed* variables).

Remark 1. Note that localization reduction is a particular case of predicate abstraction with early quantification, where the set of predicates is given by the visible variables themselves, namely $\mathbb{P} = \{v\}_{v \in \hat{\mathcal{V}}}$.

2.3 Counterexample-Guided Abstraction Refinement

Counterexample-guided Abstraction Refinement (CEGAR) is an iterative abstraction refinement methodology that starts from a very coarse abstraction and incrementally refine it until a result is yielded [5]. Every iteration corresponds to a false negative

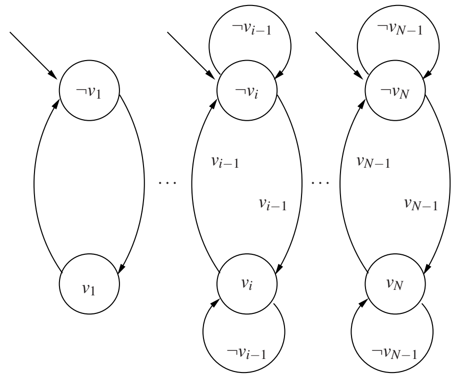

Fig. 1. N-bit counter.

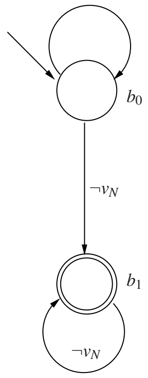

Fig. 2. BA for $!GFv_N$

(spurious counterexample), which is removed from the abstract system by refining the abstraction. Given an FSM M and a property φ, the CEGAR method is based on the following steps:

1. **[Initial Abstraction]** build an initial abstraction $\hat{M}$ ($M \preceq_H \hat{M}$);
2. **[Model Checking]** check if $\hat{M} \models \varphi$: if it does, conclude that $M \models \varphi$; otherwise, produce a counterexample π;
3. **[Concretization]** check if the counterexample is concretizable (namely, $\gamma_H(\pi) \neq \emptyset$): if it does, conclude that $M \not\models \varphi$; otherwise, call it *spurious*;
4. **[Refinement]** build a new abstraction $\tilde{M}$ that refines $\hat{M}$ (namely, $M \preceq_{\tilde{H}} \tilde{M} \preceq_{\check{H}} \hat{M}$) and rules out π ($\gamma_{\tilde{H}}(\pi) = \emptyset$); turn $\hat{M}$ into $\tilde{M}$ and go to step 2

Numerous techniques have been conceived for the refinement of the abstract FSM. Typically, they analyze the reason why a spurious counterexample cannot be concretized, and they find a set of variables that may be used to eliminate it.

The purpose of the following example is twofold: on the one hand, we show how standard refinement works; on the other hand, we present a major limitation of this approach.

Example 1. Consider a system with N symbolic variables, which represents a N-bit counter (Fig. 1). The system can be thought of as the composition of N modules, one for every variable. (The labels on the arcs represent values triggering the transition.) A property of the system is that the most significant bit v_N becomes true infinitely often. The BA A, corresponding to the negation of the property, is shown in Fig. 2.

Since only the most significant bit v_N occurs in the property, the initial abstract system contains only one module (the rightmost module of Fig. 1); all the other variables are considered as inputs. The model checking procedure will produce a counterexample in which the abstract system eventually loops on the abstract state $\neg v_N$. This counterexample is spurious because it does not correspond to any concrete path. In order to refine the abstraction and kill the spurious counterexample, we need to add v_{N-1}, because v_N becomes true only when v_{N-1} is true. Similarly we have to add v_{N-2} and so on. This

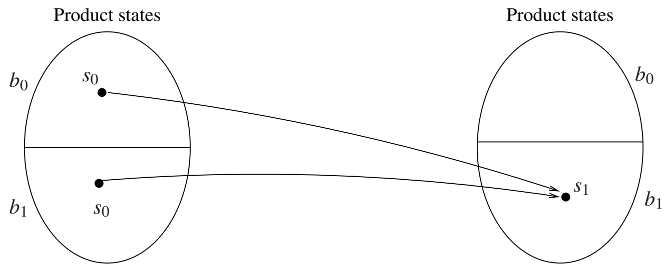

Fig. 3. Preimage in PDP.

way, standard refinement techniques end up into the concrete system. More generally, notice that, in the context of localization reduction, it is not possible to prove the property, if even one of the variable is made invisible.

3 Property-driven partitioning for abstraction refinement

Property-driven partitioning In *property-driven partitioning* (PDP), the property automaton, whose state space is often quite manageable, is represented explicitly, while the system, whose state space is typically exceedingly large, is represented symbolically. PDP partitions the symbolic state space according to an explicitly compiled property automaton.

Let $M = \langle \mathcal{V}, I, T, \{L_a\}_{a \in PROP} \rangle$ be the FSM to be verified and $A = \langle \mathcal{B}, b_0, \Sigma, \delta, F \rangle$ the BA for the complemented property. The product between M and a A is the BA $\langle \mathcal{P}, \mathcal{P}_0, \Sigma, \delta^{\mathcal{P}}, F^{\mathcal{P}} \rangle$, where

- $\mathcal{P} = \mathcal{S}_{\mathcal{V}} \times \mathcal{B}$,
- $p \in \mathcal{P}_0$ iff $p = (s, b_0)$ and $s \models I(\mathcal{V})$,
- $(p_1, l, p_2) \in \delta^{\mathcal{P}}$ iff $p_1 = (s_1, b_1)$, $p_2 = (s_2, b_2)$, $s_1, s_2' \models T(\mathcal{V}, \mathcal{V}')$, $(b_1, a, b_2) \in \delta$, and $s_1 \models L_a(\mathcal{V})$ for every $a \in l$,
- $F^{\mathcal{P}} = \{\mathcal{S}_{\mathcal{V}} \times F\}$.

We consider the partitioning of the product state space $\mathcal{P}$: $\{\mathcal{P}_b\}_{b \in \mathcal{B}}$, where $\mathcal{P}_b = \{p \in \mathcal{P} : p = (s, b)\}$. Thus, a subset Q of $\mathcal{P}$ can be represented by the following set of states of M: $\{Q_b\}_{b \in \mathcal{B}}$, where $Q_b = \{s : (s, b) \in Q\}$. If $Q^1 = \{Q_b^1\}_{b \in \mathcal{B}}$ and $Q^2 = \{Q_b^2\}_{b \in \mathcal{B}}$, we translate the set operations used in symbolic algorithms into:

$$\begin{aligned}
Q^1 \wedge Q^2 &:= \{Q_b^1 \wedge Q_b^2\}_{b \in \mathcal{B}}, \\
Q^1 \vee Q^2 &:= \{Q_b^1 \vee Q_b^2\}_{b \in \mathcal{B}}, \\
\neg Q &:= \{\neg Q_b\}_{b \in \mathcal{B}}, \\
preimage(Q) &:= \{\textstyle\bigvee_{(b,a,b') \in \delta} preimage(Q_{b'}) \wedge a\}_{b \in \mathcal{B}}, \\
postimage(Q) &:= \{\textstyle\bigvee_{(b',a,b) \in \delta} postimage(Q_{b'} \wedge a)\}_{b \in \mathcal{B}}.
\end{aligned}$$

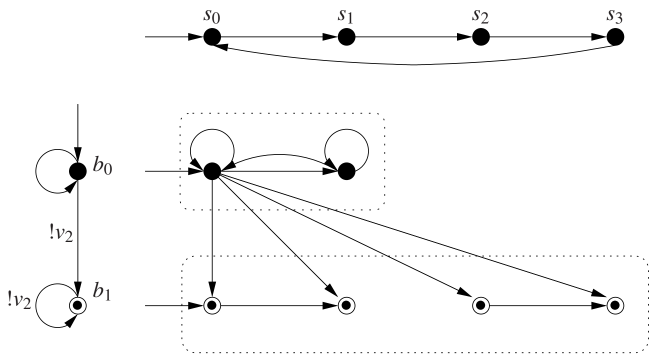

Fig. 4. Example of partitioned abstraction

Example 2. Consider the model checking problem of Example 1. The property automaton A has two states, which yield two partitions in PDP. We use s_i to denote the state of M where the counter has value i. Consider the product of state s_1 with the state b_1 of A. Suppose we want to compute the preimage of such a state. First, we compute the preimage of s_1 in M, which yields $\{s_0\}$. Then, we propagate this state to the other partitions. The result is shown in Fig. 3.

All symbolic model-checking algorithms that operate on the product can be partitioned according to the property automaton, operating on a BDD array rather than on a single BDD (see [15, 16]).

3.1 Combining PDP with abstraction

Given an FSM $M = \langle \mathcal{V}, I, T, \mathbb{L} \rangle$ and a BA $A = \langle \mathcal{B}, b_0, \Sigma, \delta, F \rangle$, we define the *abstract partitioned product* $\mathcal{P}$ between M and A as a collection of FSMs $\{M_b\}_{b \in \mathcal{B}}$, one for every state b of the automaton A, with $M \preceq_{H_b} M_b$. We use $\langle \mathcal{V}_b, I_b, T_b, \{L_b^a\}_{a \in PROP} \rangle$ to denote the abstraction M_b. A state of $\mathcal{P}$ is a pair (s, b) where the state s of M is in the partition corresponding to state b of A (namely, $s \in \mathcal{S}_{\mathcal{V}_b}$). A state (s, b) is initial if $b = b_0$ and $s \models I_b(\mathcal{V}_b)$. There exists a transition between two states (s_1, b_1) and (s_2, b_2) iff
1) there exists $l \in \Sigma$ such that $(b_1, l, b_2) \in \delta$, and $s \models L^a_{b_1}(\mathcal{V}_{b_1})$ for all $a \in l$, and
2) there exists $s_3 \in S_{\mathcal{V}_{b_1}}$ such that $s_1, s_3' \models T_{b_1}(\mathcal{V}_{b_1}, \mathcal{V}'_{b_1})$ and $s_2 \in \alpha_{H_{b_2}}(\gamma_{H_{b_1}}(s_3))$.
Intuitively, given two partitions b_1 and b_2, there is a mapping $(\alpha_{H_{b_2}} \circ \gamma_{H_{b_1}})$ between the abstract states of the partition b_1 and the abstract states of the partition b_2: the transitions from states of the partition b_1 to states of the partition b_2 are given by the composition of T_{b_1} with the mapping $\alpha_{H_{b_2}} \circ \gamma_{H_{b_1}}$. Finally, a state (s, b) is fair iff b is fair.

Note that the combination of partitioning with abstraction is independent of the particular abstraction technique used. In fact, in principle, you may use different abstraction techniques in different partitions.

Example 3. Consider the model checking problem of Example 1 with $N = 2$. We drew a possible abstract partitioned product in Fig. 4: we used the abstract system described

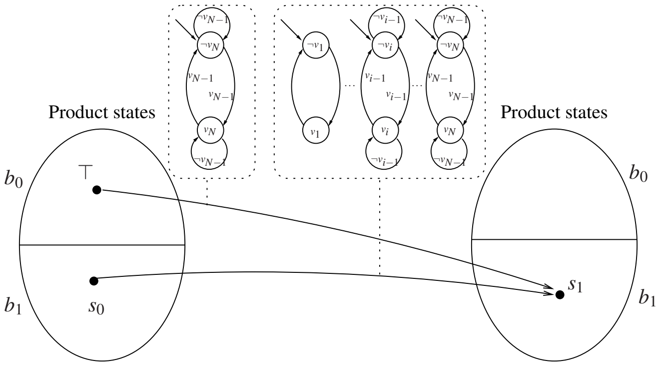

Fig. 5. Preimage in partitioned abstraction

in the Example 1 in the partition corresponding to b_0, while the partition b_1 contains the concrete system.

Theorem 1. *If $\mathcal{L}(\mathcal{P}) = \emptyset$, then $\mathcal{L}(M) \cap \mathcal{L}(A) = \emptyset$.*

Proof. Suppose $\mathcal{L}(M) \cap \mathcal{L}(A) \neq \emptyset$ and $w \in \mathcal{L}(M) \cap \mathcal{L}(A)$. Then, there exists a path $\pi_M = s_1, s_2, \ldots$ of M and a fair path $\pi_A = b_1, b_2, \ldots$ of A over some word $w = l_1, l_2, \ldots$. For every state b of A, $M \preceq M_b$ so that there exists a path $\pi_M^b \in \alpha_{H_b}(\pi_M)$ and π_M^b accepts w. Let $\pi_M^b = s_1^b, s_2^b, \ldots$ for every state b of A. Consider the sequence $\pi_{\mathcal{P}}$ of states of $\mathcal{P}$ where $\pi_{\mathcal{P}}(i) = (s_i^{b_i}, b_i)$. It is sufficient to prove that $\pi_{\mathcal{P}}$ is a fair path of $\mathcal{P}$. First, $\pi_{\mathcal{P}}(1)$ is initial since $b_1 = b_0$ and $s_1^{b_1} \models I_{b_1}(\mathcal{V}_{b_1})$. Second, for all $i \geq 1$, $(\pi_{\mathcal{P}}(i), \pi_{\mathcal{P}}(i+1))$ is a transition. In fact, 1) $(b_i, l_i, b_{i+1}) \in \delta$, and $s_i^{b_i} \models L_{b_i}^a(\mathcal{V}_{b_i})$, for all $a \in l_i$, and 2) $s_i^{b_i}, {s_{i+1}^{b_i}}' \models T_{b_i}(\mathcal{V}_{b_i}, \mathcal{V}'_{b_i})$, and $s_{i+1}^{b_{i+1}} \in \alpha_{H_{b_{i+1}}}(\gamma_{H_{b_i}}(s_{i+1}^{b_i}))$. Finally, $\pi_{\mathcal{P}}$ is fair since π_A is fair.

Notice that, if $M_b = M$ for every $b \in \mathcal{B}$, $\mathcal{P}$ is the hybrid structure we use in PDP. In this case, indeed, the opposite direction of Theorem 1 holds too.

We generalize the preimage and postimage computation defined for PDP as follows:

$$\begin{aligned} &preimage(Q) := \{\textstyle\bigvee_{(b,a,b') \in \delta} preimage_{T_b}(\alpha_{H_b}(\gamma_{H_{b'}}(Q_{b'}))) \wedge a\}_{b \in \mathcal{B}} \\ &postimage(Q) := \{\textstyle\bigvee_{(b',a,b) \in \delta} postimage_{T_b}(\alpha_{H_b}(\gamma_{H_{b'}}(Q_{b'} \wedge a)))\}_{b \in \mathcal{B}}. \end{aligned}$$

Example 4. Consider the partitioned preimage computation of Example 2. Now, we recomputed the same preimage with a partitioned abstraction product: we will use the abstract system described in the Example 1 in the partition corresponding to b_0, while the partition b_1 will contain the concrete system. In this case, before computing the component b_0 of the preimage, we have to map the state s_1 of partition b_1 to the abstraction of partition b_0. s_1 corresponds to the abstract state $\neg v_N$, whose preimage (in the abstract system) yields the whole abstract state-space (denoted with $\top$). The result is shown in Fig. 5.

$Check(M, \varphi)$
1: $A := LTL2BA(\neg\varphi)$;
2: $\hat{M} := InitialAbsctraction(M, \varphi)$
3: $\mathcal{P} := Product(\hat{M}, A)$;
4: **loop**
5: $\quad result := EmptyLanguage(\mathcal{P})$;
6: $\quad$ **if** $result = TRUE$ **then return** $M \models \varphi$
7: $\quad$ **end if**
8: $\quad \pi := GenerateCounterexample(\mathcal{P})$;
9: $\quad result := Concretizable(\pi, M)$;
10: $\quad$ **if** $result = TRUE$ **then return** $M \not\models \varphi$
11: $\quad$ **end if**
12: $\quad RefineAbstraction(\mathcal{P}, \pi)$;
13: **end loop**

Fig. 6. The procedure for checking if $M \models \varphi$.

3.2 Property-driven partitioned refinement

We investigated a new technique that combines refinement with PDP. We call such technique Property-driven Partitioned Refinement (PDPR). The idea is that, instead of refining the abstraction globally, we localize the refinement to some partition. This way, at every iteration of the CEGAR loop, we refine the abstraction trying to keep it coarser than standard refinement.

First, we adapted the standard refinement loop to LTL model checking (see Fig. 6). The LTL formula $\neg\varphi$ (the negation of the property) is translated into a *BA* A (line 1). The system is abstracted so that $\hat{M}$ contains all the information related to the variables that occur in the property. The product $\mathcal{P}$ between $\hat{M}$ and A is built (line 3). The emptiness of $\mathcal{P}$ is checked with the Emerson-Lei algorithm [8] (line 5). If a counterexample is found, we check if it corresponds to a concrete path (line 9) with Bounded Model Checking (BMC) [1], as proposed by [7]. Since the counterexample may be infinite, we consider lasso-shape paths. This way, if the counterexample is given by the abstract states $\hat{s}_1, \ldots, \hat{s}_k$ with $\hat{s}_l = \hat{s}_k$, we then generate the BMC formula

$$I(\mathcal{V}_0) \wedge \bigwedge_{1 \le i < k} T(\mathcal{V}_i, \mathcal{V}_{i+1}) \wedge \bigwedge_{1 \le i \le k} S_i(\mathcal{V}_i) \wedge \bigwedge_{v \in \mathcal{V}} v_l = v_k$$

where the predicates S_i identify the abstract states $\hat{s}_i$. If the formula is not satisfiable, we refine the abstraction (line 12). Note that this concretization step is sound but not complete: we are fixing a specific lasso shape, so it may happen that the above formula is not satisfiable, but the counterexample is concretizable. The completeness of the whole algorithm is guaranteed by the fact that the refinement procedure is monotone, since at every iteration we add some information; eventually, the concrete counterexample will be found.

PDPR follows the same schema, though the product is handled differently. The main difference between the standard framework and PDPR consists of the refinement procedure, which refines the abstract system only in some partitions. The refinement

$RefineAbstraction(\mathcal{P}, \pi)$
// $\mathcal{P}$ is the product of $\hat{M}$ and A
1: **repeat**
2: $\quad \tilde{\mathcal{V}} := \emptyset$;
3: $\quad$ **while** $|\tilde{\mathcal{V}}| < threshold$ **do**
4: $\quad\quad$ **for all** $v \in \check{\mathcal{V}}$ **do**
5: $\quad\quad\quad ComputeScore(v, \hat{M}, \pi)$;
6: $\quad\quad$ **end for**
7: $\quad\quad v := BestVar$;
8: $\quad\quad \tilde{\mathcal{V}} := \tilde{\mathcal{V}} \cup \{v\}$;
9: $\quad$ **end while**
10: $\quad \hat{M} := Abstract(\hat{M}, \tilde{\mathcal{V}})$;
11: $\quad \mathcal{P} := Product(\hat{M}, A)$;
12: $\quad result := Simulate(\mathcal{P}, \pi)$;
13: **until** $result = FALSE$

Fig. 7. Refinement function

$RefineAbstraction(\mathcal{P}, \pi)$
// $\mathcal{P}$ is the partitioned product
of $\{\hat{M}_b\}_{b \in \mathcal{B}}$ and A
1: **repeat**
2: $\quad$ **for all** $b \in \mathcal{B}$ **do**
3: $\quad\quad \tilde{\mathcal{V}}_b := \emptyset$;
4: $\quad$ **end for**
5: $\quad$ **while** $|\cup_{b \in \mathcal{B}} \tilde{\mathcal{V}}_b| < threshold$ **do**
6: $\quad\quad$ **for all** $b \in \mathcal{B}$ **do**
7: $\quad\quad\quad$ **for all** $v \in \check{\mathcal{V}}_b$ **do**
8: $\quad\quad\quad\quad ComputeScore(v, b, \hat{M}, \pi)$;
9: $\quad\quad\quad$ **end for**
10: $\quad\quad$ **end for**
11: $\quad\quad v := BestVar$;
12: $\quad\quad b := BestPartition$;
13: $\quad\quad \tilde{\mathcal{V}}_b := \tilde{\mathcal{V}}_b \cup \{v\}$;
14: $\quad$ **end while**
15: $\quad$ **for all** $b \in \mathcal{B}$ **do**
16: $\quad\quad \hat{M}_b := Abstract(\hat{M}_b, \tilde{\mathcal{V}}_b)$;
17: $\quad$ **end for**
18: $\quad \mathcal{P} := PartitionedProduct(\{\hat{M}_b\}_{b \in \mathcal{B}}, A)$;
19: $\quad result := Simulate(\mathcal{P}, \pi)$;
20: **until** $result = FALSE$

Fig. 8. Partitioned version of the refinement function

depends on the type of abstraction we are considering. In the following, we adopt the framework of localization reduction.

The refinement (Fig. 7) consists in computing the score of each invisible variable (line 5). We pick the best scored variable (line 7) and we add it to the system (line 8). We repeat the procedure until the refined abstraction rules out the spurious counterexample. This is checked with a symbolic fix-point simulation (line 12). In the case of PDPR, the refinement is similar (Fig. 8). The only difference is that we score pairs of variables and partitions (line 8): if (v, b) is the best scored pair, we add v to the system of partition b (line 13).

Example 5. Consider the model checking problem of Example 1. PDPR starts with two identical systems, which is the system containing only the constraints of v_N. Thus, PDPR will produce the same spurious counterexample described in the Example 1, which was obtained with the standard abstraction. The refinement heuristic finds that we have to add v_{N-1} in the partition b_1, in order to force v_N to become true. Similarly, we have to add v_{N-2} and all the other variables to partition b_1. PDPR ends the refinement loop with the concrete system in partition b_1. Though, the abstract system of partition b_0 is never touched. In the subsequent emptiness checking, we use the partitioned abstraction described in Example 4. This abstraction is coarser than the concrete system (necessary for a standard refinement), but it is sufficiently fine to prove the property.

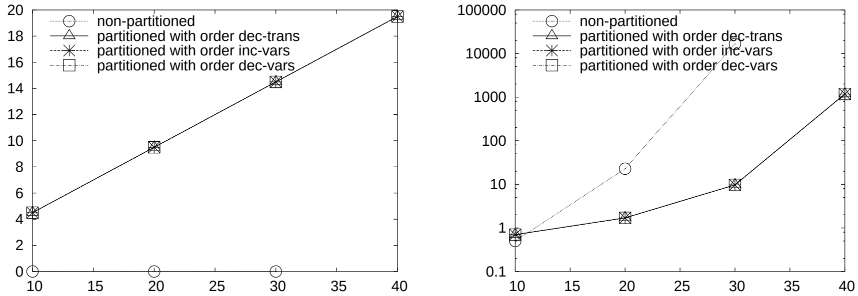

Fig. 9. Experiments on the N-bit-counter example. X axis: N. Y axis: Left plot: average number of invisible variables; Right plot: CPU time (secs).

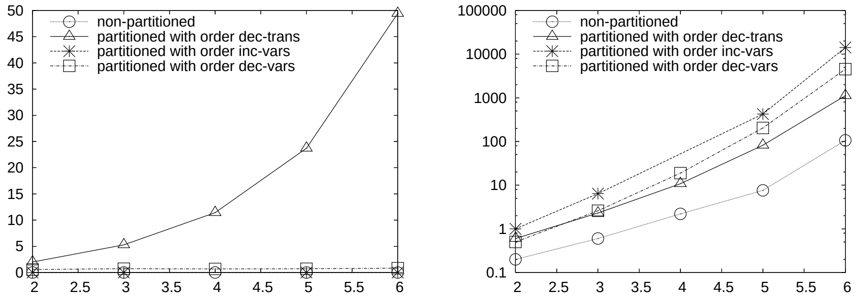

Fig. 10. Experiments on the memory example. X axis: number of bits for memory addresses. Y axis: Left plot: average number of invisible variables; Right plot: CPU time (secs).

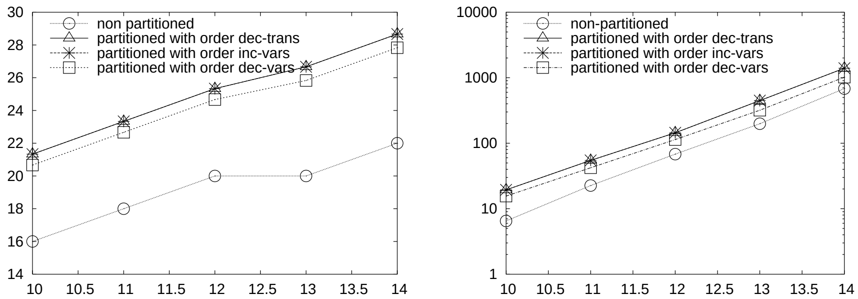

Fig. 11. Experiments on the stack example. X axis: stack's capacity. Y axis: Left plot: average number of invisible variables; Right plot: CPU time (secs).

4 Experimental results

We now evaluate the efficiency of PDPR in practice by testing it over some scalable verification models. The purpose of this experimental evaluation is to provide a compari-

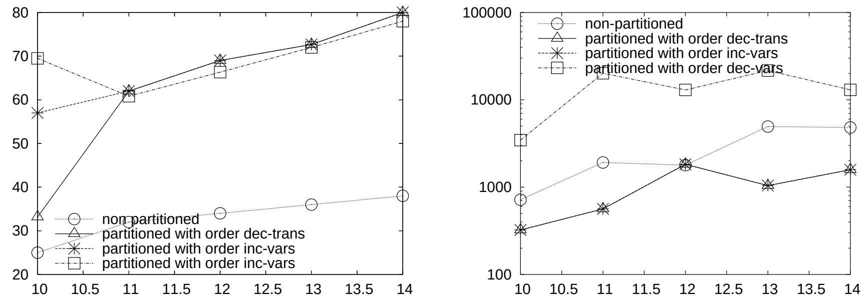

Fig. 12. Experiments on the gas-station example. X axis: number of customers. Y axis: Left plot: average number of invisible variables; Right plot: CPU time (secs).

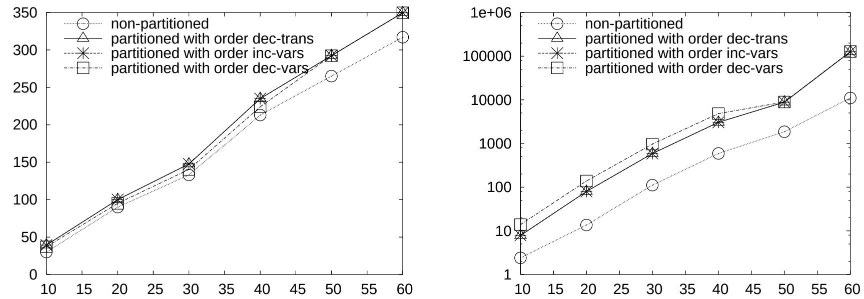

Fig. 13. Experiments on the mutual-exclusion example. X axis: number of processes. Y axis: Left plot: average number of invisible variables; Right plot: CPU time (secs).

son between the standard non-partitioned refinement approach and PDPR. Thus, we fix an abstraction refinement setting, in which we use PDP as representation of the product, and we compare the partitioned versus the non-partitioned refinement loop. Our tests do not provide a direct comparison with the numerous state-of-the-art refinement techniques because the partitioned abstraction is orthogonal to them. Rather, we try to evaluate the trade-off between the disadvantages of dealing with several abstractions at the same moment and the advantages of using coarser abstractions when possible.

NuSMVPDP is an extension of NuSMV [4] that uses a PDP representation to perform the emptiness checking of the product. We have enhanced NuSMVPDP with an abstraction refinement technique that implements localization abstraction with the refinement heuristic of [18][5]. In these settings, at every refinement iteration, a new abstract system is computed and composed with the property automaton; then, a PDP-based search checks the emptiness of the product.

We tested this standard abstraction refinement technique against its partitioned counterpart. In the partitioned refinement, we fixed an ordering of the partitions. This gives a

[5] Unlike [18], we use one counterexample per refinement iteration.

priority to the partition to be refined when two invisible variables of different partitions get the same score. We selected three relevant orderings:

- **[inc-vars]** an increasing ordering according to the number of invisible variables in each partitions; this way, we prefer refining the abstraction in those partitions that had already been refined (thus, previously guilty of other spurious counterexample);
- **[dec-vars]** a decreasing ordering according to the number of invisible variables in each partitions; this way, we prefer refining the coarsest abstraction;
- **[dec-trans]** a decreasing ordering according to the number of outgoing transitions in the property automaton; this tries to alleviate the cost of the mapping between abstractions: every time we have to traverse backwardly a transition that starts from b, we have to quantify over the invisible variables of the partition b.

We performed our tests on the Rice Terascale Cluster (RTC)[6], a TeraFLOP Linux cluster based on Intel Itanium 2 Processors. We run NuSMV with a static variable ordering so that, when the abstraction is the same, the non-partitioned and the partitioned version perform equally and the efficiency is not biased by a dynamic variable reordering. We first run both the non-partitioned and the partitioned version of abstraction refinement on the N-bit-counter system of Example 1. Then, we tested the algorithm over other scalable systems: namely, memory, stack, mutual-exclusion example, and gas station. In each system we scale up a parameter N so that the number of symbolic variables used to describe the system is linear in N (apart from the memory example where the number of variables is exponential in N).

- The memory example implements standard read and write functionalities. In order to be able to scale up the verification, we use just one bit for the data so that we can write and read only two values. We scale up the number N of bits used for the memory addresses. We verify that if we write a value on a particular address, next time we read from it we obtain the same value, unless another write operation was performed in the meanwhile. Inspired by the work on GSTE of [20], we express the property by enumerating each address case so that the property automaton contains different subgraph for each case. The partitioned abstraction should take benefit from this, by keeping in each subgraph only the information regarding that particular memory address.
- In the stack example, we have the standard pop and push functions. In this case, scalability is given by the maximum size N of the stack. The property says that whenever we push an element on the stack, and later the stack results empty, there must be a pop call in the middle
- In the mutual-exclusion example, N processes non-deterministically try to access the critical session. The access is controlled by the main module, which guarantees that a process does not wait forever. The property says that, if a process is the only one that is waiting, then it accesses the critical session in one step.
- In the gas-station example, there are N customers who want to use one pump. They have to prepay an operator who then activates the pump. When the pump has charged, the operator give the change to the customer. The property says the if a customer pays before another one, then she must be server first.

[6] `http://rcsg.rice.edu/rtc/`

We verified many other properties, but the initial abstraction was fine enough to prove the property; in one case the final abstraction coincided (with both techniques) with the concrete system. We reported only the other cases, where the refinement had been effectively useful to prove the property.[7]. The results are shown in Figures 9, 11, 12, and 13. In every plot, we scale up the size of the system. The plots on the left of each figure present the number of invisible variables (the average in the case of partitioned abstraction) of the final abstraction. The plots on the right show the time in seconds plotted in log scale.

These results show that partitioned abstraction is successful at allowing verification with significantly coarser abstractions. The coarsest abstraction is obtained when the states of the property automaton correspond to particular phases of the circuit, like in the case of the memory example (Fig. 10). In some cases (Figs. 9 and 12), this leads to an order-of-magnitude improvement in running time. Nevertheless, in some cases (Figs. 10, 11 and 13), the coarser abstractions do not compensate for the overhead of having to refine several abstractions. It is interesting to note that no single refinement order dominates in our experiments; the best refinement order seems model and property specific. For lack of space we did not show the plots with space performances but they present the same pattern of time plots. The only exception is the memory example, which is particularly interesting because the number of invisible variables grows exponentially (hiding an exponential amount of design with an order of magnitude improvement in space), but the verification time degrades by an order of magnitude.

5 Conclusions

We investigated a new approach for abstraction refinement, in which the abstraction is not uniform; rather, like in [10], the abstraction is localized and presents different precisions in different parts of the state space. We combined this idea with property-driven partitioning, which partitions the product state space for efficient LTL model checking. We presented a new refinement algorithm that exploits this approach by refining the abstraction only in those partitions where it is necessary. This way, as our tests confirmed, we can prove the correctness of a system with a higher level of abstraction, leading, in some cases, to dramatic performance improvement. Unlike, however, in SMC, where the partitioning almost always speeds up the verification time, here the use of partitioning is less compelling, as the overhead of having to refine multiple abstractions is in some cases too high, leading to performance degradation. This conclusion is consistent with the results in [10], where predicate localization improves performance only in a small number of test cases.

We note that most of time is spent in the refinment step, meaning that, the coarser abstractions do not compensate for the overhead of having to refine several abstractions. Since the partitioned abstractions are much coarser, we conjecture that we could afford to use less accurate but faster refinement techniques in order to speed up the overall verification time. This is left for future research.

[7] All data available at `www.inf.unisi.ch/postdoc/tonetta/TACAS07-PDP`.

References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS'99*, pages 193–207, 1999.
2. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
3. H. Cho, G.D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(12):1451–1464, 1996.
4. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Proceedings of CAV'99*, pages 495 - 499, 1999.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of CAV'00*, pages 154 - 169, 2000.
6. E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
7. E.M. Clarke, A. Gupta, J.H. Kukula, and O. Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In *Proceedings of CAV'02*, pages 265–279, 2002.
8. E.A. Emerson and C.L. Lei. Efficient Model Checking in Fragments of the Propositional μ-Calculus. In *Proceedings of the LICS'86*, pages 267 - 278, 1986.
9. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97*, pages 72 - 83, 1997.
10. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Proceedings of POPL'04*, pages 232 - 244, 2004.
11. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL'02*, pages 58 - 70, 2002.
12. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2003.
13. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
14. M.D. Nguyen, D. Stoffel, M. Wedler, and W. Kunz. Transition-by-Transition FSM Traversal for Reachability Analysis in Bounded Model Checking. In *Proceedings of ICCAD'05*, 2005.
15. R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi. GSTE is Partitioned Model Checking. In *Proceedings of CAV'04*, pages 229 - 241, 2004.
16. R. Sebastiani, S. Tonetta, and M.Y. Vardi. Symbolic Systems, Explicit Properties: On Hybrid Approaches for LTL Symbolic Model Checking. In *Proceedings of CAV'05*, pages 350 - 363, 2005.
17. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of LICS'86*, pages 332 - 344. IEEE Computer Society, 1986.
18. C. Wang, B. Li, H.S. Jin, G.D. Hachtel, and F. Somenzi. Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement. In *Proceedings of ICCAD'03*, pages 408 - 415, 2003.
19. J. Yang and C.-J. H. Seger. Introduction to Generalized Symbolic Trajectory Evaluation. *IEEE Transactions on Very Large Scale Integration Systems*, 11(3), 2003.
20. J. Yang and C.-J.H. Seger. Generalized Symbolic Trajectory Evaluation - Abstraction in Action. In *Proceedings of FMCAD'02*, pages 70 - 87, 2002.