Efficient Implementation of Property Directed Reachability*

Niklas Een, Alan Mishchenko, Robert Brayton
{een,alanmi,brayton}@eecs.berkeley.edu

Berkeley Verification and Synthesis Research Center
EECS Department
University of California, Berkeley, USA.

Abstract. Last spring, in March 2010, Aaron Bradley published the first truly new bit-level symbolic model checking algorithm since Ken McMillan's interpolation based model checking procedure introduced in 2003. Our experience with the algorithm suggests that it is stronger than interpolation on industrial problems, and that it is an important algorithm to study further. In this paper, we present a simplified and faster implementation of Bradley's procedure, and discuss our successful and unsuccessful attempts to improve it.

1 Introduction

Sequential verification is hard, both model checking and equivalence checking. Difficult instances are typically solved using several simplification steps followed by multiple verification engines scheduled in sequence or running in parallel. Despite all the available tools, numerous practical instances remain unsolved. Therefore, research in formal verification is always on the lookout for methods that can handle difficult cases.

In 2003, a new method known as interpolation based model checking [7], was proposed by Ken McMillan to address hard UNSAT instances. Over time it was perfected and is currently considered one of the most valuable formal verification methods.

More recently, another novel method was pioneered by Aaron Bradley [1, 2]. He named his implementation IC3, but gave no name to the method itself. We choose to call it *property directed reachability* (PDR) to connect it to Bradley's earlier work on property directed invariant generation.

It came as a surprise that IC3 won the third place in the hardware model checking competition (HWMCC) at CAV 2010. It was marginally outperformed by two mature integrated verification systems, both carefully tuned to balance several different engines. As such, the new method appears to be the most important contribution to bit-level formal verification in almost a decade.

Although PDR has been generally known for less than a year, while interpolation has been around long enough for numerous improvements and extensions to be proposed, for example [4, 3], an up-to-date implementation of PDR can solve more instances from HWMCC than interpolation can. This is also true for the benchmarks our group has received from our industrial collaborators. Another remarkable property of PDR is its capability of finding deep counterexamples. Although on average BMC does better than PDR, there are many benchmarks where PDR can find counterexamples that elude both BMC and BDD reachability. Finally, PDR lends itself naturally to parallel implementation, as was explained in Bradley's original work.

In this paper, we explore PDR and try to understand the reason for its effectiveness. We propose a number of changes to the algorithm to improve its performance and to simplify its implementation. In particular:

- We achieve a significant speedup by using three-valued simulation to reduce the burden on the SAT-solver.
- We eliminate a tedious and error-prone special-case handling of counterexamples of length 0 or 1.
- We show experimentally that two elements of the original algorithm give no speedup: (i) variable activity and (ii) cube generalization beyond non-inductive regions.
- We separate the main algorithm from the handling of SAT queries through a clean interface. This separation reduces the overall complexity.
- We refute some potential improvements experimentally.
- We present detailed pseudo-code to fully document our implementation.

2 Preliminaries

This paper considers the verification of systems modeled using finite state machines (FSMs). Each state of the FSM is identified with a boolean assignment to a set of state variables. The FSM further defines a set of *initial states* and a set of *property states*. The algorithm to be presented verifies that there exists no sequence of transitions from an initial state to a non-property state ("bad" state).

In the presentation, a state variable or its negation is referred to as a *literal*, a conjunction of literals as a *cube*, and the negation of a cube (a disjunction of literals) as a *clause*. If a cube contains all the state variables, it is called a *minterm*. It is assumed that the FSM is represented symbolically in a way that can be translated into propositional logic for a SAT-solver.

*This updated version contains two fixes to Figure 6.

3 Overview of PDR

The PDR algorithm can be understood on several levels. This section addresses:

(1) How it works.
(2) Why it is complete.
(3) What makes it effective.

In particular the first two points can be understood in terms of approximate reachability analysis. For the third point one must also consider the inductive flavor of the algorithm.

3.1 Notation

Let $\mathbf{I}$ and $\mathbf{P}$ be predicates over the FSM's state variables, denoting the initial states and the property states respectively. Also let $\mathbf{T}$ denote the transition relation over the current and next state variables. Given a cube s, a call to the underlying SAT solver will be expressed as:

$$SAT?\big[\mathbf{P} \wedge \neg s \wedge \mathbf{T} \wedge s'\big]$$

using primes to denote next states. This query asks "starting in a state where the property holds, but outside the cube s, can you get inside the cube s in one transition". If the answer is UNSAT, then it has been proved that $(\mathbf{P} \wedge \neg s \wedge \mathbf{T}) \rightarrow \neg s'$, and $\neg s$ is said to be inductive *relative* to $\mathbf{P}$.

In the algorithm, some cubes will be proved unreachable from the initial states in k steps or less. Such cubes will be referred to as *blocked cubes* of frame k.

3.2 Mechanics

PDR maintains a list of facts which we will call the *trace*: $[\mathbf{R}_0, \mathbf{R}_1, \ldots \mathbf{R}_N]$. The first element $\mathbf{R}_0$ is special; it is simply identified with the initial states. For $k > 0$, $\mathbf{R}_k$ is a set of clauses that AND-ed together represent an over-approximation of the states reachable from the initial states in k steps or less. The trace is maintained in such a way that $\mathbf{R}_i$ is contained in $\mathbf{R}_{i+1}$. In fact, this relation is syntactic: the clauses of $\mathbf{R}_{i+1}$ are a subset of the clauses of $\mathbf{R}_i$, except for $i = 0$ ($\mathbf{R}_0$ has no clauses).

Together with the trace, the PDR algorithm maintains a dynamic set of *proof-obligations*. A proof-obligation consists of a frame number k and a cube s, where s is either a set of bad states or a set of states that can all reach a bad state in one or more transitions. The frame number k indicates a position in the trace where s must be proved unreachable, or else the property fails.

By manipulating the trace and the set of proof-obligations according to a scheme detailed below, PDR derives new facts and adds them to the trace until it either (i) produced an inductive invariant proving the property, or (ii) added a proof-obligation at frame 0 with a cube that intersects with the initial states. Such a cube cannot be blocked, and entails the existence of a counterexample.

3.2.1 Proof-obligations

The core of PDR lies in how proof-obligations are handled, and how new facts are derived from them. All reasoning in PDR take place on one transition relation; there is no unrolling of the FSM like in BMC or interpolation. Given the proof-obligation (s, k), consider the query:

$$SAT?\big[\mathbf{R}_{k-1} \wedge \mathbf{T} \wedge s'\big] \qquad (\mathrm{Q}_1)$$

If it is UNSAT, then the facts of $\mathbf{R}_{k-1}$ are strong enough to block s at frame k, and we can add the clause $\neg s$ to $\mathbf{R}_k$. However, the syntactic containment relation of the trace requires us also to add the same clause to all preceding $\mathbf{R}_i$, $i < k$. Is it sound to do this? Consider replacing $\mathbf{R}_{k-1}$ with $\mathbf{R}_{k-2}$ in the query. Containment states that $\mathbf{R}_{k-2}$ is stronger than $\mathbf{R}_{k-1}$, so the query remains UNSAT. Likewise for $\mathbf{R}_{k-3}$ and so on, all the way back to the initial states. The only thing left to check is whether s intersects the initial states or not. If s is not blocked by $\mathbf{R}_0$, then we cannot strengthen the trace by $\neg s$. In the algorithm, this query will not be used if s overlaps with the initial states.

Using this approach, the quality of the learned clause depends on the size of the cube in the proof-obligation. In practice, these cubes often have many literals, and the negation $\neg s$ becomes a very weak fact. It turns out to be crucial for the performance of PDR to try to learn stronger facts, i.e. cubes with fewer literals. To achieve this, the above learning scheme is improved in two ways:[1]

Improvement 1 – "Generalize s". Many modern SAT-solvers do not simply return UNSAT, but also give a reason for the unsatisfiability; either through an UNSAT-core or through a final conflict-clause. Both these mechanisms can be used to extract precise information about which clauses were actually used in the proof. Since s is a conjunction inside the query, it translates into a set of unit clauses. Not all of those clauses may actually be needed when proving UNSAT. Any literal of s corresponding to an unused clause can be removed without affecting the UNSAT status. This provides a virtually free mechanism of removing literals that just happen not to be used by the SAT-solver.

A more directed, but also more expensive, approach is to *explicitly* try to remove the literals one by one. If the query remains UNSAT in the absence of a literal, good riddance. If not, put the literal back. Although the order in which literals are probed affects the outcome, the procedure is monotone in the sense that removing a literal from a query that is SAT cannot make it UNSAT. Note also that we cannot remove a literal if it makes s intersect with the initial states, even if the query is UNSAT.

Improvement 2 – "Add $\neg s$ to the query". A key insight of Bradley was to realize that the query could be extended by the term $\neg s$:

$$SAT?\big[\mathbf{R}_{k-1} \wedge \neg s \wedge \mathbf{T} \wedge s'\big] \qquad (\mathrm{Q}_2)$$

Adding an extra conjunct means the query is more likely to be UNSAT, which improves chances of removing a literal, or

[1] These improvements are also present in Bradeley's IC3, [1]

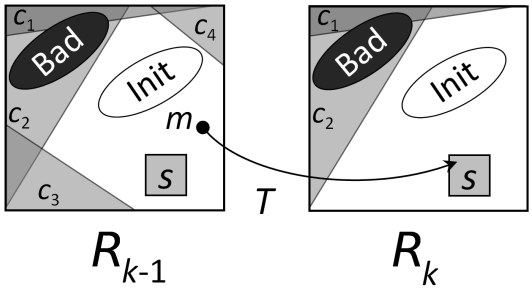

Figure 1. *Is s inductive relative to $\mathbf{R}_{k-1}$?* In the SAT query, we try to find a minterm m in the white region of the first frame, that in one transition can reach a point inside the cube s. The white region satisfies $\mathbf{R}_{k-1} \wedge \neg s$, illustrated by the four blocked cubes c_1 through c_4 and the cube s. If the query is UNSAT, it has been proved that a point outside s stays outside s for the first k transitions from the initial states. When generalizing s, we must make sure that the cube does not grow to intersect the initial states. This property, together with UNSAT, proves s to be unreachable in the first k frames. Note that the figure also illustrates how $\mathbf{R}_k$ contains a subset of the cubes of $\mathbf{R}_{k-1}$.

indeed learning a clause at all. This extended query is depicted in *Figure 1*. Having s on both sides of the transition breaks monotonicity: as s gets weaker, $\neg s$ gets stronger. A query that is SAT may become UNSAT if more literals are removed—which makes the task of finding a minimal cube much harder (exponential in the size of s). Heuristics for minimizing s are discussed in [2].

But why is it sound to add $\neg s$ to the query? It can be viewed as a bounded inductive reasoning: The base case $\mathbf{R}_0 \rightarrow \neg s$ holds by construction (s does not intersect the initial states). We have proved that $(\mathbf{R}_{k-1} \wedge \neg s \wedge \mathbf{T}) \rightarrow \neg s'$, but because $\mathbf{R}_i$ is stronger than $\mathbf{R}_{k-1}$ for $i < k-1$, we have also proved that $\neg s$ is preserved by every transition up to frame k.

3.2.2 Satisfiable queries

We now turn to the case where the query (original or extended) is SAT. This means $\mathbf{R}_{k-1}$ was not strong enough to block s at frame k, and something new must be learned at frame $k-1$. From the satisfying assignment, we can extract a minterm m in the pre-image of s, which gives us a new proof-obligation $(m,\ k-1)$.

The above learning scheme can now be applied to this proof-obligation, drawing from $\mathbf{R}_{k-2}$ to learn clauses in $\mathbf{R}_{k-1}$. If $\mathbf{R}_{k-2}$ is not strong enough, the procedure may recursively go further back into the trace and learn a whole cascade of facts over many time-frames. Eventually the procedure returns to the original proof-obligation $(s,\ k)$ and may either succeed in blocking it this time, or generate a new minterm in the pre-image of s.

As noted in the previous section, learning short clauses is crucial for PDR to work. Indeed, most of the runtime is spent on generalizing cubes by removing literals. Because a minterm is maximally long, it is a particularly undesirable starting point for this process. To alleviate this situation, we propose to shrink the proof-obligations by using three-valued (ternary) simulation.[2] It requires the FSM to be in circuit form, but in practice this is often the case.

Reducing proof-obligations by ternary simulation. For a satisfiable query, extract the minterm m from the satisfying assignment, giving values to the flop outputs as well as the primary inputs. Simulate this assignment through one time-frame. Now, probe each flop by changing its value to X and propagate the effect of this using ternary simulation. If an X does not appear at any flop input among the flops in s, then the probed flop (state variable) can be safely removed from the proof-obligation. If the X do reach a flop in s, undo the propagation and the probing, and move on to the next flop.

The resulting cube has the property that all the states it represents can reach s in one transition, and hence the entire cube must be blocked.

3.3 The Algorithm

For clarity, we state the precise properties of the trace:

(1) $\mathbf{R}_0 = \mathbf{I}$.
(2) All $\mathbf{R}_i$ except $\mathbf{R}_0$ are sets of clauses.
(3a) $\mathbf{R}_i \rightarrow \mathbf{R}_{i+1}$.
(3b) The clauses $\mathbf{R}_{i+1}$ is a subset of $\mathbf{R}_i$ for $i > 0$.
(4) $\mathbf{R}_{i+1}$ is an over-approximation of the image of $\mathbf{R}_i$.
(5) $\mathbf{R}_i \rightarrow \mathbf{P}$, except for the last element $\mathbf{R}_N$ of the trace.

We note that (5) is different from Bradley's original presentation, which also required the property to hold for $\mathbf{R}_N$. The change eliminates the need for the special BMC check of length 0 and 1, performed in Bradley's implementation of PDR.

At the start of the algorithm the trace has just one element $\mathbf{R}_0$. It then runs the following main loop:

while $SAT?\big[\mathbf{R}_N \wedge \neg \mathbf{P}\big]$ **do**
(a) extract a bad state m from the SAT model
(b) generalize m to a cube s using ternary simulation
(c) recursively block the proof-obligation $(s,\ k)$

When the loop terminates, the property holds for $\mathbf{R}_N$, and a new (empty) frame is added to the trace. The algorithm will be repeated for the new frame, but first a *propagation phase* is executed, where learned clauses are pushed forward in the trace:

for $k \in [1, N-1]$ and $c \in \mathbf{R}_k$ **do**
if c holds in frame $k+1$, add it to $\mathbf{R}_{k+1}$

During the propagation phase it is important to do syntactic subsumption. If a clause c was moved forward from frame k to $k+1$, and frame $k+1$ has a weaker clause $d \supseteq c$, then d should be removed. Subsumed clauses accumulate quickly, but serves no purpose except to slow down the SAT-solver.

[2] Ternary logic has three values: 0, 1, and X. The binary semantics is extended by: $(X \wedge 0 = 0)$, $(X \wedge 1 = X)$, $(X \wedge X = X)$, $(\neg X = X)$.

3.3.1 Queue of proof-obligations

Section 3.2.1 suggests a recursive clause-learning scheme. However, PDR can be improved by reusing proof-obligations of one time-frame in all future time-frames. After all, if a cube is bad, it should be blocked everywhere. This requires a queue, as the algorithm now can have many outstanding proof-obligations in each frame. The elements should be dequeued from the smallest time-frame first. This change has the added benefit of making PDR capable of finding counterexamples longer than the trace.

3.3.2 When does the algorithm terminate?

PDR can terminate in one of two ways: either (i) a proof-obligation at frame 0 intersects with the initial states, which implies that the property fails (in this case, a counterexample can be extracted with some additional bookkeeping); or (ii) the clause sets of two adjacent frames become syntactically identical: $\mathbf{R}_i \equiv \mathbf{R}_{i+1}$. Since $\mathbf{R}_i \rightarrow \mathbf{P}$ by (5); $\mathbf{R}_i \wedge \mathbf{T} \rightarrow \mathbf{R}'_{i+1}$ by (4); $\mathbf{I} \rightarrow \mathbf{R}_i$ by (1) and (3a); then $\mathbf{R}_i$ is an inductive invariant that proves the property.

3.4 Convergence

Must the main loop terminate for some finite trace length? When generated, each proof-obligation (s, k) contains at least one state that is not previously blocked. If the proof-obligation is immediately handled, or if the generalization procedure first checks that this still holds when the proof-obligation is dequeued, then every clause created by the learning algorithm must block at least one more state of frame k. Because there is a finite number of frames, and a finite number of states in the FSM, the main loop is guaranteed to terminate.

Can the length of the trace grow indefinitely? If the syntactic termination check ($\mathbf{R}_i \equiv \mathbf{R}_{i+1}$) were done semantically instead ($\mathbf{R}_i = \mathbf{R}_{i+1}$), then clearly this cannot happen. $\mathbf{R}_{i+1}$ would have to block at least one state less than $\mathbf{R}_i$. Suppose therefore $\mathbf{R}_i = \mathbf{R}_{i+1}$ but $\mathbf{R}_i \not\equiv \mathbf{R}_{i+1}$. During the propagation phase, all clauses of $\mathbf{R}_i$ will be moved into $\mathbf{R}_{i+1}$, making them syntactically identical and the algorithm terminates.

We note that the bound ($2^{|S|}$ frames with at most $2^{|S|}$ clauses in each) implied by the above argument is very large, and does in no way explain why the algorithm performs well in practice.

3.5 What makes PDR so effective?

The experimental analysis of Section 6 shows that PDR represents a major performance improvement over interpolation based model checking (IMC) [7], hitherto regarded as the strongest bit-level engine. Why is this?

Consider the BMC unrolling depicted in *Figure 2*. Assume for simplicity that the design can non-deterministically return to the initial states at any time.[3] This guarantees that the set of reachable states grows monotonically with the frame number k.

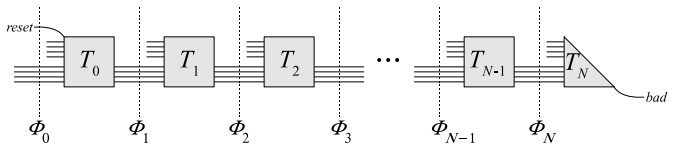

Figure 2. *BMC unrolling of length N.* The design is reset in the first frame and *Bad* is asserted in the last frame. The last frame is drawn partially because the next-state logic for the flops is not needed.

The first version of IMC, never published,[4] considered such an unrolling, and from an UNSAT proof computed interpolants Φ_i between every adjacent time-frames. This sequence of interpolants has the property:

(1) $\mathbf{I} = \Phi_0$
(2) $\Phi_k \wedge \mathbf{T} \rightarrow \Phi'_{k+1}$
(3) $\Phi_N \rightarrow \mathbf{P}$
(4) symbols(Φ_i) $\subseteq$ *state-variables*

If N is chosen large enough, one of the interpolants Φ_k must be an inductive invariant proving the property: if the suffix after Φ_k is longer than the backward diameter of the system, it cannot contain any state that can reach *Bad*; the prefix before Φ_k grows monotonically and for a finite system must eventually repeat itself.

This method is not as effective as the published version of IMC. So what is wrong with it? One can argue that the important feature of interpolation is its *generalizing* capability.[5] For instance, the interpolant Φ_1 can be viewed as an *abstraction* of the first time-frame, containing just the facts needed for the suffix to be unsatisfiable (this interpretation is particularly in accord with McMillan's asymmetric interpolant computation).

Even though logically (2) implies that each interpolant can be derived from its predecessor, this is not how the SAT-solver constructs them. During its search, the solver is free to roam all over the unrolling. We argue that this may deteriorate the generalizing capability of interpolation.

In the published algorithm, McMillan used the insight that (a) interpolants are smaller and more general toward the ends of the unrolling, and (b) repeatedly applying interpolation on its own output will improve the generalizing capability. In his algorithm, the interpolant Φ_1 is therefore repeatedly used to replace the initial states constraint, resulting in interpolants that are less and less dependent on the initial states and in an increasingly more general way imply the unsatisfiability of the suffix.

In a way, the procedure can be viewed as *committing* to the abstraction that was computed. It disallows the SAT-solver from going back to the real initial states and learning

[3] This behavior can be achieved by rewiring the flops, or, alternatively, be made part of the verification algorithm.

[4] Private conversation with Ken McMillan.

[5] Indeed, interpolation based model checking is probably better understood as a method for "guessing" an inductive invariant rather than, as often done, an approximate reachability analysis.

more facts. For this to work, the suffix must be long enough to prevent any state that can reach the bad states from entering into the interpolants. If it fails to prevent this, the algorithm has to start over from scratch, typically with a longer unrolling (although randomizing the SAT-solver and restarting with the same length works sometimes).

We now compare this to how PDR works. First note that at the end of each major cycle, just before pushing clauses forward, the $\mathbf{R}_i$ are in fact interpolants; all the facts in frame k and future frames are derived from $\mathbf{R}_k$.

During the computations, PDR completely commits to its current abstractions $\mathbf{R}_i$. The localized reasoning prevents it from learning new facts from earlier time-frames *unless* it has been proved that new facts *must* be learned. In a way, the whole procedure can be viewed as one big SAT-solving process, where the solver is carefully controlled to make sure it does *not* roam all over the unrolling. Further, when new facts are brought in from previous frames, a lot of effort is spent on simplifying those facts (the literal removing consumes ∼80% of the runtime). There is no similar mechanism in IMC, it must use whatever proof the SAT-solver happened to give it. Also, PDR constantly removes subsumed clauses, especially during the forward-propagation phase.

To summarize, PDR sticks to the facts it has learned as long as possible, similar to the way IMC commits to its interpolants. If what PDR has learned at a frame is too weak, it can repair the situation by learning new clauses rather than scrapping all the work done so far and starting over, as IMC does. PDR has a very targeted approach to producing small facts by its literal removing scheme, and it constantly weeds out redundant clauses by subsumption checking and forward propagation.

A possible drawback of PDR, however, is the strong *inductive bias* of its learning: it can only learn clauses in terms of state variables. But this bias is also the very reason it can efficiently do generalization. It might be that future improvements to the algorithm will allow it to work efficiently on a different domain.

4 Implementation

This section details our implementation of PDR. In the pseudo-code, only cubes are used and not clauses. In particular we represent the trace as sets of blocked cubes rather than learned clauses. Furthermore, we only store a cube in the *last* time-frame where it holds (to avoid duplication). We call this delta-encoded trace $\mathbf{F}$, and it relates to $\mathbf{R}$ through:

$$\mathbf{R}_k = \bigwedge_{i \geq k} \neg \mathbf{F}_i$$

We also extend $\mathbf{F}$ by a special element $\mathbf{F}_\infty$ which will hold cubes that have been proved unreachable from the initial states by any number of transitions. In the code, the following data-types are used:

– **Vec**. A dynamic vector with methods:

```
uint ***size***()          – returns size of the vector
T&   ***op***[](uint i)    – returns the i^th element
void ***push***(T elem)    – pushes an element at the end
T    ***pop***()           – pops and returns last element
```

– **Cube**. A fixed-size vector of literals (no push/pop).

– **TCube**. A pair (cube ∈ Cube, frame ∈ uint) referred to as a *timed cube*. Two special constants are defined for the frame component:

```
FRAME_NULL – cube has no time component
FRAME_INF  – cube belongs in F_∞
```

Function ***next***(TCube s) returns s with the frame number incremented by one.

```
Program State:
    Netlist          N;    – Netlist with property
    Vec⟨Vec⟨Cube⟩⟩   F;    – Blocked cubes of each frame
    PdrSat           Z;    – Supporting SAT solver(s)

Main Function:
    bool  ***pdrMain***();

Recursive Cube Generation:
    bool  ***recBlockCube***(TCube s0);
    bool  ***isBlocked***(TCube s);
    TCube ***generalize***(TCube s0);

Cube Forward Propagation:
    bool  ***propagateBlockedCubes***();

Small Helpers:
    uint  ***depth***();
    void  ***newFrame***();
    bool  ***condAssign***(TCube& s, TCube t);
    void  ***addBlockedCube***(TCube s);
```

Figure 3. *Overview of PDR algorithm.* "***pdrMain***()" will use "***recBlockCube***()" to recursively block bad states of the final time frame until the property holds, then call "***propagateBlockedCubes***()" to push blocked cubes from all frames in the trace forward to the latest frame where they hold.

An overview of the functions implementing PDR, and the program state they work on is given in *Figure 3*. The FSM is assumed to be given in circuit form, containing one safety property to be proved. The special frame $\mathbf{F}_\infty$ is stored as the last element of the vector F.

An outline of the execution: Function ***pdrMain***() gets a bad state in the last frame and calls ***recBlockCube***() to block it, using the helper function ***isBlocked***() (which checks if a proof-obligation has already been solved) and ***generalize***() (which shortens a cube). When the property has been proved for the last frame, ***propagateBlockedCubes***() pushes cubes of all time-frames forward while doing subsumption, handled by ***addBlockedCube***().

```
interface PdrSat {
    Cube   getBadCube();
    bool   isBlocked(TCube s);
    bool   isInitial(Cube c);
    TCube solveRelative(TCube s, uint params = 0);

    void   blockCubeInSolver(TCube s);
};
```

Figure 4. *Abstract interface for the SAT queries of PDR.* These methods can be implemented using either a monolithic SAT-solver, or one SAT-solver per time-frame. The roles of "Init" and "Bad" can be exchanged within this SAT abstraction to obtain the dual PDR procedure based on backward induction (although ternary simulation cannot be used backwards). The first four functions corresponds to actual SAT queries (although for some common restriction on initial states, "***isInitial***()" can be implemented by a syntactic analysis). The fifth function, "***blockCubeInSolver***()", merely informs the SAT implementation that a new cube has been added to the vector "*F*".

4.1 Separation of concerns

Our PDR implementation abstracts the handling of SAT calls through the interface in *Figure 4*. The semantics of the interface is defined as follows:

Method ***getBadCube***() returns a bad cube not yet blocked in the last frame. Method ***isBlocked***(s) returns TRUE if the cube *s.cube* is blocked at *s.frame*. Method ***isInitial***(c) returns TRUE if the cube c intersects with the initial states. Method ***blockCubeInSolver***(s) reports to *PdrSat* that a cube has been added to the vector *F*.

Finally, method ***solveRelative***(s) tests if *s.cube* can be blocked at frame *s.frame* using the extended query (Q2) of Section 3.2.1. If the answer is UNSAT, then the implementation returns a new cube z where:

$$\begin{aligned} z.cube &\subseteq s.cube \\ z.frame &\geq s.frame \end{aligned}$$

The method guarantees that not only is *s.cube* blocked at frame *s.frame*, but that actually the subset *z.cube* is blocked at a later frame. The SAT solver may learn these more general facts by inspecting the final conflict-clause of the solver (or the UNSAT core), and taking this "free" information into account.

If instead the query is satisfiable, then the implementation returns a generalization, using ternary simulation, of a minterm in the pre-image of *s.cube*. All states of the returned cube *z.cube* can reach *s.cube* in one transition. The time component *z.frame* is set to FRAME_NULL.

The behavior of ***solveRelative***() can be altered by the *params* argument. Default value **"0"** means: do not extract a model if the query satisfiable, just return (CUBE_NULL, FRAME_NULL). Parameter **"EXTRACTMODEL"** means: work as described above. Parameter **"NOIND"** means: use the original query (Q1) instead of (Q2).

```
bool pdrMain() {
    F.push();                                   – push "F_∞"
    newFrame();                                 – create "F[0]"

    Z = createPdrSat(N, F);

    forever{
        Cube c = Z.getBadCube();
        if (c != CUBE_NULL){
            if (!recBlockCube(TCube(c, depth())))
                – failed to block 'c' ⇒ CEX found
                return FALSE;
        }else{
            newFrame();
            if (propagateBlockedCubes())
                – invariant found, may store it here
                return TRUE;
        }
    }
}
```

Figure 5. *Main procedure.* The last element of **F** (referred to as "$\mathbf{F}_\infty$") contains all the cubes that have been proved to be unreachable for all k. Their negation constitutes a proper inductive invariant. Function "***newFrame***()" inserts a new frame into **F** just before $\mathbf{F}_\infty$.

5 SAT Solving

In this section, we discuss the details of implementing ***solveRelative***() of the *PdrSat* interface using MINISAT and a single SAT instance. The other methods of the *PdrSat* interface can be implemented in a similar way.

There are two features that are particularly important: (i) MINISAT allows incremental SAT through *assumption literals*; a set of unit clauses that are temporarily assumed during one SAT call. After the call, the assumptions are undone and new regular clauses can be added before the next call. (ii) For UNSAT calls, MINISAT returns the subset of assumptions that were used in the proof.

The netlist is transformed to CNF using the standard Tseitin transformation [9] plus variable elimination [6]. Logic cones are added to the solver on demand, starting with just the transitive fanin of *Bad*. Whenever a new frame is added to the trace, a new activation literal act_i is reserved. All clauses learned in that frame will be extended by $\neg act_i$ in ***blockCubeInSolver***().

Given a cube $s = (s_1 \wedge s_2 \wedge \ldots \wedge s_n)$, procedure ***solveRelative***() does the following:

(1) Reserve a new activation literal a and add the clause $\{\neg a, \neg s_1, \neg s_2, \ldots, \neg s_n\}$ (unless NOIND is given).

(2) Call the solve method with the following assumptions: $[a,\ act_k,\ act_{k+1},\ ...,\ act_{N+1},\ s'_1,\ s'_2,\ \ldots,\ s'_n]$, where s'_i denotes a flop input.

(2u) If UNSAT:

- Remove all literals of s whose corresponding assumption s'_i was not used, unless doing so makes the new cube overlap with the initial states.

```
bool recBlockCube(TCube s0) {
  PrioQ⟨TCube⟩ Q; – orders cubes from low to high frames
  Q.add(s0);

  while (Q.size() > 0){
    TCube s = Q.popMin();

    if (s.frame == 0)
      – Found counterexample, may extract it here
      return FALSE;

    if (!isBlocked(s)){
      assert(!Z.isInitial(s.cube));
      TCube z = Z.solveRelative(s, EXTRACTMODEL);

      if (z.frame != FRAME_NULL){
        – Cube 's' was blocked by image of predecessor:
        z = generalize(z);
        while (z.frame < depth()
          && condAssign(z, Z.solveRelative(next(z))));

        addBlockedCube(z);
        if (s.frame < depth() && z.frame != FRAME_INF)
          Q.add(TCube(z.frame, s.cube));

      }else{
        – Cube 's' was not blocked by image of predecessor:
        z.frame = s.frame − 1;
        Q.add(z);
        Q.add(s);
      }
    }
  }
  return TRUE;
}
```

Figure 6. *Recursively block a cube.* The priority queue "Q" stores all pending proof-obligations: a cube and a time frame where it should be blocked. In a practical implementation, it may also store the proof-obligation from which the element was generated (this facilitates extraction of counterexamples). We noticed (or think we noticed) a small performance gain by giving "PrioQ" a stack-like behavior for proof-obligations of the same frame. We left one of our program assertions in the pseudo code because this invariant is important and non-obvious. Finally, note the line "Q.**add**(TCube(...))" line (just above the "**else**"). Adding the current proof-obligation in the next frame is not necessary, but it improves performance for UNSAT problems *and* allows PDR to find counterexamples longer than the length of the trace—sometimes much longer.

```
bool isBlocked(TCube s) {
      – Check syntactic subsumption (faster than SAT):
      for (uint d = s.frame; d < F.size(); d++)
            for (uint i = 0; i < F[d].size(); i++)
                  if (subsumes(F[d][i], s.cube))
                        return TRUE;

      – Semantic subsumption thru SAT:
      return Z.isBlocked(s);
}

TCube generalize(TCube s) {
      for all literals p ∈ s {
            TCube t = "s minus literal p"
            if (!Z.isInitial(t.cube))
                  condAssign(s, Z.solveRelative(t));
      }
      return s;
}
```

Figure 7. *Helper functions for recursive cube blocking.* Function "**isBlocked**()" semantically checks if s is already blocked, which could have happened after the proof-obligation was enqueued. For efficiency reasons, it first does a syntactic check. This check is so effective that we did not notice any performance loss by disabling the semantic SAT check at the end (but we kept it to ensure convergence, as argued in Section 3.4). In fact, deriving a new cube from s, even if s is blocked, may be a good idea, as the new cube can subsume several old cubes. We note that function "**generalize**()" iterates over s while s is being modified, which the implementation must handle.

– Find the lowest act_i that was used. Return the timed cube $(s_{new}, i+1)$.

(2s) If SAT and EXTRACTMODEL is specified:

– Extract a minterm m from the satisfying assignment.
– Shorten m to cube by ternary simulation. Return $(m_{new}, \text{FRAME\_NULL})$.

(2s') else if SAT, return (CUBE_NULL, FRAME_NULL).

– (3) Add unit clause $\{\neg a\}$ permanently.

The last step (3) forever disables the temporary clause added in (1). The periodic cleanup of MINISAT will reclaim the memory. However, the variable index reserved for the activation literal cannot be reused. For that reason we recycle the solver when more then 50% of the variables currently in use are disabled activation literals. This has the added benefit of cleaning up cones of logic that may no longer be in use. We note that the previous activation literal can be reused if s is a subset of the cube of the previous call, which happens quite frequently.

6 Experimental Analysis

A number of experiments have been performed to evaluate our PDR implementation, both on public benchmarks from the Hardware Model Checking Competition of 2010 (HWMCC10) and on industrial benchmarks.[6] This section summarizes the most interesting results we have found.

6.1 Comparison of IC3 and PDR

This experiment was performed using 274 hard problems from our industrial collaborators. We simplified the designs by running the ABC command "dprove"(see Figure 4.1 of

[6] Although we cannot distribute the industrial benchmarks, we will make our implementation of PDR available at http://bvsrc.org

```
uint depth() { return F.size() − 2; }

void newFrame() {
    – Add frame to 'F' while moving 'F∞' forward:
    uint n = F.size();
    F.push();
    F[n−1].moveTo(F[n]);
}

bool condAssign(TCube& s, TCube t) {
    if (t.frame != FRAME_NULL){
        s = t;
        return TRUE;
    }else
        return FALSE;
}

void addBlockedCube(TCube s) {
    uint k = min(s.frame, depth() + 1);

    – Remove subsumed clauses:
    for (uint d = 1; d ≤ k; d++){
        for (uint i = 0; i < F[d].size();){
            if (subsumes(s.cube, F[d][i])){
                F[d][i] = F[d].last();
                F[d].pop();
            }else
                i++;
        }
    }

    – Store clause:
    F[k].push(s.cube);
    Z.blockCubeInSolver(s);
}
```

Figure 8. *Small helper functions.* Function "***addBlockedCube***()" will add a cube both to $\mathbf{F}_a$ nd the PdrSat object. It will also remove any subsumed cube in $\mathbf{F}$. Subsumed cubes in the SAT-solver will be removed through periodical recycling.

```
bool propagateBlockedCubes() {
  for (uint k = 1; k < depth(); k++){
    for all cubes c ∈ F[k] {
      TCube s = Z.solveRelative(TCube(c, k+1), NOIND);
      if (s.frame != FRAME_NULL)
        addBlockedCube(s);
    }
    if (F[k].size() == 0) return TRUE;  – Invariant found
  }
  return FALSE;
}
```

Figure 9. *Propagating blocked cubes forward.* All cubes in $\mathbf{F}$ are revisited to see if they now hold at a later time-frame. If so, they are inserted into that frame. The subsumption of "***addBlockedCube***()" will remove the cube from its current frame (and possible other cubes in the later frame). Note that in a practical implementation, the iteration over cubes in $\mathbf{F}_k$ must be aware of these updates. Because c is already present in frame k, we can use (Q1) instead of (Q2) in the call to ***solveRelative***().

Benchmark	IC3	PDR	NoSim	StGen	IMC
design01_prop1	–	–	–	–	249.5
design01_prop2	4.1	**0.3**	102.5	0.4	0.2
design01_prop3	–	**81.2**	–	126.9	–
design01_prop4	–	**70.0**	–	191.0	–
design01_prop5	–	**91.6**	–	166.5	–
design01_prop6	–	**100.7**	–	176.7	–
design01_prop7	–	–	–	–	168.8
design01_prop8	160.1	**6.1**	–	11.1	21.9
design01_prop9	130.1	**5.9**	–	10.7	42.8
design01_prop10	71.9	**7.1**	–	12.3	44.2
design02_prop1	594.0	**30.2**	–	144.0	–
design02_prop2	–	**489.2**	–	–	–
design02_prop3	–	**68.0**	–	–	–
design03_prop1	–	**466.4**	–	129.8	–
design03_prop2	–	**483.3**	–	130.8	–
design04	**84.5**	–	–	–	–
design05_prop1	–	**172.5**	–	152.5	–
design05_prop2	–	**182.1**	–	172.0	–
design06_prop1	2.7	**0.8**	1.8	1.0	–
design06_prop2	**3.1**	**3.1**	5.6	0.8	–
design07	94.4	**6.0**	88.6	13.8	–
design08	298.3	**83.6**	–	133.1	–
design09	–	**77.8**	–	151.2	–
design10_prop1	2.0	**1.0**	2.3	1.4	–
design10_prop2	2.6	**1.0**	2.7	2.7	–
design11_prop1	324.4	**28.1**	474.0	27.4	–
design11_prop2	7.7	**2.1**	8.9	3.3	–
design12	–	–	–	–	62.6
design13_prop1	–	**126.1**	–	85.0	–
design13_prop2	–	**47.2**	–	57.2	–
design13_prop3	–	**26.0**	–	22.2	–
design13_prop4	–	**17.6**	–	22.1	–
design13_prop5	–	**18.1**	–	26.4	–
design14_prop1	**41.7**	–	–	–	–
design14_prop2	**61.5**	–	–	–	–
design15_prop1	–	**5.3**	–	20.7	4.7
design15_prop2	–	**32.8**	–	10.9	595.8
design16	2.2	**0.9**	2.6	2.2	286.6
design17	–	–	–	185.8	
design18	10.8	**0.7**	4.9	1.4	409.7
design19	501.7	**13.4**	–	23.3	–
design20	17.1	**10.0**	138.0	20.4	–
design21	**169.9**	–	–	225.4	–
design22	–	**154.1**	–	185.0	–
design23_prop1	–	**438.7**	–	–	–
design23_prop2	–	**320.5**	–	133.0	–
Total solved	21	38	11	37	11

Table 1. *Comparison of IC3 and PDR on industrial problems.* Two modifications to PDR are also evaluated (disabling ternary simulation "NoSim", and stronger cube generalization "StGen"). Interpolation (IMC) is also included for comparison. All benchmarks are UNSAT except for *design02* (3 properties) and *design14* (2 properties). Boldfaced figures indicates winner between IC3 and PDR only.

[8]). With a timeout of 10 minutes, 42 problems were solved by either IC3 or PDR; included in *Table 1*. From the table we see that our implementation solves almost twice as many instances as the original IC3 (38 vs. 21), but there are also 4 instances where IC3 solves them and our PDR does not. The last column shows for comparison the results of of interpolation based model checking (IMC).

Figure 10 shows the behavior of the implementations for increasing timeout limits. For space reasons we included two more PDR runs discussed in the next section.

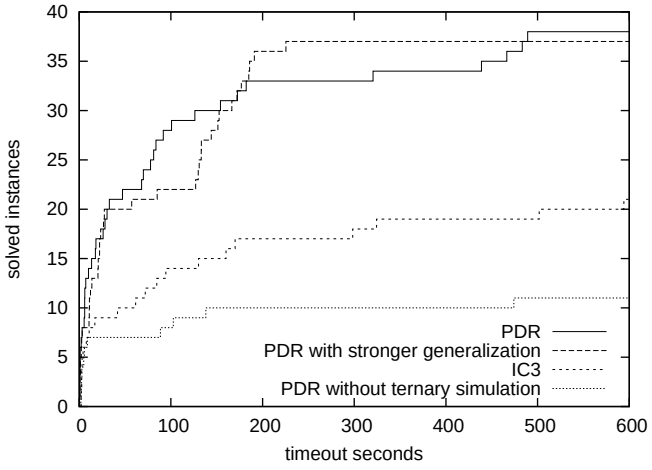

Figure 10. *Comparison of IC3 and PDR on industrial problems.* Two modifications to PDR are also evaluated.

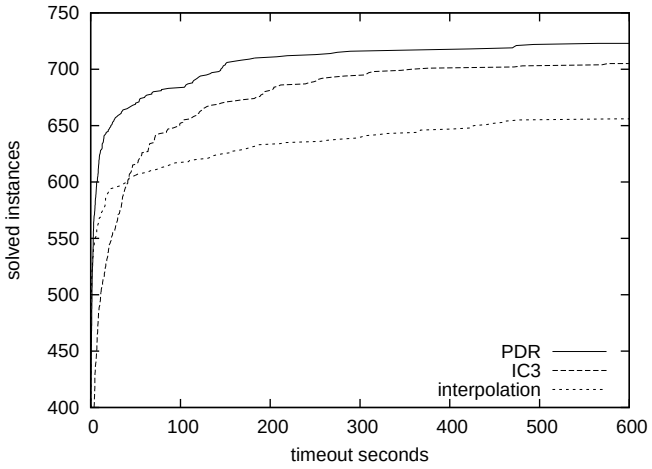

Figure 11. *Comparison of IC3 and PDR on HWMCC10 problems.*

Figure 11 shows the performance of PDR, IC3 and IMC on the HWMCC10 benchmarks. Looking closer at PDR vs. IMC reveals that the difference is mostly on UNSAT problems, where PDR solves 420 vs. 362 for IMC (14% difference). On satisfiable instance the numbers are 303 vs. 294 (only 3% difference).

6.2 Ternary simulation and Generalization

The third column of *Table 1*, and the corresponding curve in *Figure 10*, show the performance of PDR without ternary simulation. It is clear that ternary simulation has a big impact. Without it, our implementation drops way below IC3. One reason for this may be that IC3 never had ternary simulation, and Bradley implemented some other tricks that compensates for this loss, notably removing multiple literals per SAT call in the cube generalization.

The fourth column shows the effect of stronger cube generalization, as proposed in the paper on IC3. The modified procedure will try to remove a literal even if the SAT query

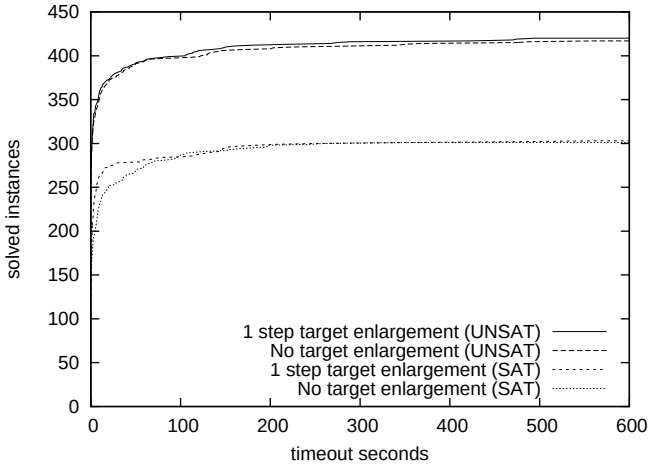

Figure 12. *Effect of target enlargement.*

is satisfiable by exploiting the non-monotonicity. As in IC3, this is done for three random literals. Our conclusion from looking at the results is that this technique was *not* helpful. Although we do not have room to present the data here, the result holds for the HWMCC10 benchmarks as well.

6.3 Effect of changing the semantics of $\mathbf{R}_N$

As pointed out in Section 3.3, we diverge from IC3 by not requiring the last frame of the trace to fulfill the property. The approach of IC3 has two effects compared to ours:

(1) When a new frame is opened, the property is known to hold, so $\mathbf{P}$ can be added to the relative induction SAT query. This means that the final invariant will be of the form $\mathbf{R}_i \wedge \mathbf{P}$ rather than just $\mathbf{R}_i$, and that the clauses in $\mathbf{R}_\infty$ may depend on $\mathbf{P}$.

(2) Seeding the recursive cube-blocking with minterms of the pre-image of $\mathbf{P}$ rather than with minterms of $\mathbf{P}$ corresponds to a one-step target-enlargement.

The second difference, target-enlargement, can be implemented by preprocessing the design (just unroll the property cone for one time-frame and combine the new and the old property outputs). *Figure 12* shows the effect it has on the HWMCC10 benchmarks. Note that it improves the performance for simple satisfiable problems solved in less than 100 seconds. The difference is substantial enough to motivate the use of target-enlargement.

We also investigated if the first difference above had any effect by running our previous PDR implementation which had the same behavior as IC3 in this respect (but includes ternary simulation and other improvements). For space reason we do not include the graph here, but the curves of the new implementation (with target enlargement) and the previous implementation match exactly on both SAT and UNSAT problems. We also tried target enlargement of 2 steps, but there was no additional benefit.

We conclude that there is no performance loss due to our modification of the original algorithm. It makes the imple-

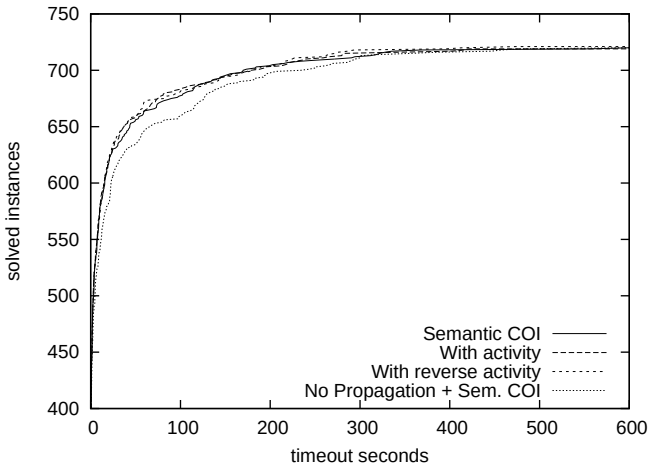

Figure 13. *Refuting activity / Semantic cone-of-influence.*

mentation simpler, and it has the extra benefit that $\mathbf{R}_\infty$ is a proper invariant, which can be used to strengthen other proof-engines running in parallel, or be useful for synthesis.

6.4 Runtime breakdown

In order to identify directions for future improvements, we ran an instrumented version of our PDR on a handful of examples. Our findings suggests that about 20% of the runtime is spent in ***propagateBlockedCubes***() and 80% in ***recBlockCube***()—most of which is in ***generalize***(), but a substantial portion also in the first call to ***solveRelative***(). Satisfiable calls to the SAT-solver are about twice as common as unsatisfiable ones, and 5x more expensive.

6.5 Other things we tried

— We evaluated the effect of the extended query (Q2) vs. the original (Q1) (Section 3.2.1). Although the (Q2) gave a clear performance boost, PDR works remarkably well even without it (it solved 704 instead of 723 problems; more than interpolation, which solved 656 problems).

— We evaluated the proposed activity scheme of IC3, which controls the order in which literals are tried for removal. We ran it against itself with the activity reversed ("worst" order) and could see no difference (*Figure 13*), and no difference to a static order either (not in the graph).

— We implemented a technique we call *semantic cone-of-influence*. At the end of each major round, all cubes in the trace that is not needed to prove the property of the final frame are removed. This analysis can be done through a series of SAT calls of roughly the same cost as forward-propagation. The method removes many cubes. However, running PDR with this turned on did not give any noticeable speedup, but it also did not degrade performance (thus the cost of doing semantic COI was amortized by the improvement). But a really interesting result is that running semantic COI, while turning off forward-propagation, works almost as well as the standard version of PDR (*Figure 13*). In contrast, turning off forward-propagation *without* semantic COI is a disaster! This shows that an important feature of the forward-propagation is the cleansing effect it has through the subsumption mechanism.

— Because most of the time is spent in satisfiable SAT calls, and this partly is a result of MiniSat always returning complete models, we made a modified version of MiniSat that only does BCP in the cone-of-influence of the flops in the query. With this version, a few more benchmarks (728 instead of 723) were solved. However, we think a justification based variable order should do even better. We are currently working on a circuit based SAT-solver with this feature.

We have also implemented a non-monolithic version of PDR (one solver instance per time-frame) that helps to localizing the SAT solving better, especially together with frequent solver recycling. For large benchmarks, where the relevant logic is small compared to the size of the design, this version does very well. It is worth noting that most of the work in PDR takes place in the last couple of time-frames where the COI is the smallest. In a monolithic PDR, early time-frames may pollute these calls.

— We made a version that finds an inductive subset of $\mathbf{R}_N$ after propagating the cubes forward. This will find true inductive invariants that can be put into $\mathbf{F}_\infty$. Although the cost of this procedure did not quite amortize over the gains, having more clauses in $\mathbf{F}_\infty$ can be useful if those facts are exported to other engines.

— We made an extension that allows PDR to develop and use an abstraction, where some flops are considered as primary inputs. This is relatively straight-forward to implement. The only tricky part in using localization abstraction is when it is combined with proof-based abstraction [5], which can shrink the current abstraction in the middle of PDR's operations. The reason is that the assertion in *Figure 6* will not hold if we apply a smaller abstraction to the initial states. The way to address this is to introduce a reset signal that gives the correct value at the flop outputs of frame 0, and then let all flops be uninitialized.

References

[1] A. R. Bradley. **SAT-based model checking without unrolling**. In *Proc. VMCAI*, 2011.

[2] A. R. Bradley and Z. Manna. **Checking safety by inductive generalization of counterexamples to induction**. In *Proc. FMCAD*, 2007.

[3] G. Cabodi, L. A. Garcia, M. Murciano, S. Nocco, and S. Quer. **Partitioning interpolant-based verification for effective unbounded model checking**. In *IEEE TCAD*, 2010.

[4] G. Cabodi, M. Murciano, S. Nocco, and S. Quer. **Boosting interpolation with dynamic localized abstraction and redundancy removal**. In *ACM TODAES*, 2008.

[5] N. Een, A. Mishchenko, and N. Amla. **A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction**. In *FMCAD*, 2010.

[6] Niklas Een and Armin Biere. **Effective Preprocessing in SAT through Variable and Clause Elimination**. In *SAT*, 2005.

[7] K. L. McMillan. **Interpolation and SAT-based Model Checking**. In *CAV*, 2003.

[8] A. Mishchenko, M. L. Case, R. K Brayton, and S. Jang. **Scalable and scalably-verifiable sequential synthesis**. In *Proc. ICCAD*, 2008.

[9] G. Tseitin. **On the complexity of derivation in propositional calculus**. *Studies in Constr. Math. and Math. Logic*, 1968.