QF_BV Model Checking with Property Directed Reachability

Tobias Welp[1] Andreas Kuehlmann[1,2]

[1] University of California at Berkeley, CA, USA
[2] Coverity, Inc., San Francisco, CA, USA

Abstract

In 2011, property directed reachability (PDR) was proposed as an efficient algorithm to solve hardware model checking problems. Recent experimentation suggests that it outperforms interpolation-based verification, which had been considered the best known algorithm for this purpose for almost a decade. In this work, we present a generalization of PDR to the theory of quantifier free formulae over bitvectors (QF_BV), illustrate the new algorithm with representative examples and provide experimental results obtained from experimentation with a prototype implementation.

1 Introduction

For almost a decade, interpolation-based verification [1] has been considered the best algorithm to solve hardware model checking problems. In this approach, one repetitively solves bounded model checking instances with bound k. Next, one derives interpolants from the refutation proofs, which act as overapproximations of the forward image of the initial set. After applying this procedure for several iterations, the overapproximations of the forward image often stabilizes, i.e. one finds an inductive invariant that proves the model checking problem.

Recently, a novel approach to hardware model checking has been proposed which attempts to decide a model checking problem stepwise by solving a large number of small lemmata instead of unrolling the transition relation as in interpolation-based verification which often yields large SAT instances. This algorithm, later named property directed reachability, has been originally proposed in [2] and its implementation IC3 demonstrated remarkably good performance in the hardware model checking competition (HWMCC) 2010 (3rd place). The authors of [3] have shown that a more efficient implementation of the algorithm would have won the HWMCC 2010. In particular, the new algorithm outperforms interpolation-based verification.

In addition to its excellent runtime performance on practical problems, PDR has a number of other favorable properties. For instance, the algorithm has modest memory requirements and has been shown to be parallizable, a property which has become particularly important in recent years.

Similarly as the excellent algorithmic properties of the DPLL algorithm [4] have fueled interest in generalizing the DPLL algorithm to richer logics [5], all the positive characteristics of PDR motivate research for generalizations to richer logics.

Two such generalizations have been published recently. Whereas the authors of [6] present generalizations to push-down systems and to the theory QF_LA, the authors of [7] propose to apply PDR to the Boolean skeleton of a lazy satisfiability modulo theory (SMT) solver in order to leverage it for richer logics.

As main contribution of this work, we describe a generalization of the PDR algorithm from Boolean formulae to the theory QF_BV. The generalized version incorporates analogs to all optimization techniques applied in the original version, such as hot solving, fast expansion using the minimum unsatisfiable core, simulation based expansion of proof obligations, etc.

We anticipate that an efficient QF_BV model checking algorithm will be useful in many different application domains. In particular, we envision its use in program verification.

The remainder of this paper is structured as follows: To keep the paper self-contained, we describe PDR for solving hardware model checking problems in the following section. Next, in Section 3, we present the proposed generalization from the binary PDR algorithm to an algorithm to solve problems formulated in QF_BV. As proof of concept, we have implemented the algorithm and in Section 4 we describe details of our implementation and report experimental results. Finally, we draw conclusions and describe future work in Section 5.

2 Property Directed Reachability

In this section, we start with defining the hardware model checking problem. Next, we explain the overall solving strategy of PDR and finally present the actual algorithm. Note that for space considerations, we have omitted many important details in our discussion about PDR, some of them essential for the efficiency of the algorithm. We refer the reader to [3] for a complete presentation.

2.1 Hardware Model Checking Problem

We are given a state space spanned by the domain of n Boolean variables $\mathbf{x} = x_1, x_2, \cdots, x_n$. We define two sets of states: initial states $I(\mathbf{x})$ and bad states $B(\mathbf{x})$ using Boolean formulae over the Boolean variables $\mathbf{x}$. We model our hardware design operating in the state space $\mathbf{x}$ using a transition relation $T(\mathbf{x}, \mathbf{x}')$ that is a Boolean formula over $\mathbf{x}$ and $\mathbf{x}'$, a copy of $\mathbf{x}$ which corresponds to the same variables but one time step later. The transition relation is true iff the combination $\mathbf{x}$ and $\mathbf{x}'$ represent a possible transition of the hardware design. The problem to be solved is to decide whether a state in $B(\mathbf{x})$ can be reached from a state in $I(\mathbf{x})$ using only transitions in $T(\mathbf{x}, \mathbf{x}')$. Note that for ease of notation, we will omit the dependence of I, T, and B on the variables $\mathbf{x}$ and $\mathbf{x}'$ in the remainder of this paper.

2.2 Overall Solving Strategy

The conceptual strategy of PDR to fulfill the overall proof obligation is to iteratively find small truth statements and combine all these lemmata to obtain a proof for the desired result. In contrast to many other approaches in hardware model checking, the algorithm purposely avoids unrolling the transition relation over several time steps. The motivation for this strategy is that solving a number of small problems one by one is more likely successful than attempting to solve one big problem at once.

978-3-9815370-0-0/DATE13/©2013 EDAA

More concretely, PDR constructs a trace t consisting of frames $f_0, f_1, \cdots$. Each frame contains a set of Boolean cubes $\{c_i\}$, where each cube $c_i = \bigwedge_j l_j$ is a conjunction of Boolean literals l_j where a Boolean literal l_j can either be a Boolean variable x_k or its negation $\neg x_k$. If there is a cube c in frame f_i, the semantic meaning of this is that all states contained in c cannot be reached within i steps from the initial states. In this case, we say that the states in c are *covered* in frame i and we will call all cubes in frame f_i the *cover*. The inverse of the cover in f_i is an overapproximation of the states that are reachable in i steps. In frame f_0, a state is covered if it is not reachable in 0 steps, in other words, if it is not in the initial set I.

Algorithm 1 `PDR`(I, T, B)

```
1: while true do
2:   Cube $c$ = findBadCube()
3:   int $l$ = lengthsTrace()
4:   if $c$ then
5:     if !recCoverCube($c, l$) then return "property fails"
6:   else
7:     pushNewFrame()
8:     if propagateCubes() then return "property holds"
```

Algorithm 2 `recCoverCube`(Cube c, int l)

```
1: if $l = 0$ then return false
2: while $c$ reachable from $\tilde{c}$ in one transition do
3:   if !recCoverCube($\tilde{c}, l-1$) then return false
4: expand($c, l$)
5: propagate($c, l$)
6: return true
```

Algorithm 1 shows the overall PDR algorithm. In each iteration, the algorithm searches for a cube in the last frame of the trace that is in B and not yet covered using `findBadCube()`. If such a cube c exists, the algorithm tries to recursively cover c (`recCoverCube()`, see Algorithm 2). To this end, the routine checks if c is reachable from the previous frame. Assume that this is possible and denote with $\tilde{c}$ a cube in the previous frame that is not covered and from which c can be reached in one step. Then `recCoverCube()` calls itself recursively on $\tilde{c}$. If such a sequence of recursive calls reaches back to frame f_0, the corresponding call stack effectively proves that c can be reached from I, i.e. that the property fails. Otherwise, if a cube c is proved to be unreachable from the previous frame, the algorithm expands c by iteratively attempting to remove literals from cube c. An attempt of removing a literal l_k from $c = \bigwedge_j l_j$ is successful if $\bigwedge_{j \neq k} l_j$ remains unreachable. After expansion, the algorithm attempts to propagate the cube to later frames (if existent). Continuing the discussion of Algorithm 1, if `findBadCube()` returns without successfully finding an uncovered cube in B, we know that B is covered in the last frame f_l. As we preserve the invariant that the cover in frame f_i is an underapproximation of the space not reachable within i steps, we can conclude that B is not reachable within l steps and we push a new frame to the end of the trace. Afterwards, we attempt to propagate cubes from frame l to frame $l+1$. If this is successful for all cubes, we have shown that from an overapproximation of the reachable states in frame f_l we cannot reach any state outside this overapproximation in frame f_{l+1}. In other words, we have found an inductive invariant. Moreover, the set of bad states B is disjoint of this inductive invariant. This proves that the property holds.

PDR is a sound and complete algorithm for solving hardware model checking problems. We already argued why the algorithm gives the correct response upon termination. It remains to show that the algorithm terminates. The complete proof for this result is given in [3]. Herein, we restrict ourselves in pointing out the main intuition of the proof which is based on the following idea: note that every state which is reachable within i steps is also reachable within $i+1$ steps. Hence, the cover of frame f_{i+1} implies the cover of f_i. If two succeeding frames have the same cover, the algorithm terminates. Otherwise, the cover of f_{i+1} must be strictly smaller than that of f_i. As the state space is finite, this implies that the algorithm will eventually terminate. Note, however, that the asymptotic worst case runtime is linear to the size of the state space which is exponential to the size of the problem, i.e. the algorithm has runtime $O(2^n)$ with n being the size of the problem.

The overall algorithm leverages a SAT-solver to answer atomic queries such as "Can cube c be reached from a state that is not covered in the previous frame?"

3 Generalization of PDR

Table 1 summaries the main aspects of our generalization of the binary version of PDR to a more general logic. We assume that the input format of the generalized version are QF_BV formulae, which motivated the use of a suitable SMT-solver instead of a SAT-solver.

	Binary Approach	Generalization
I, B, T	Boolean formulae	QF_BV formulae
Solver	SAT-Solver	QF_BV SMT-Solver
Atomic Reasoning Unit	Boolean Cubes	Polytopes
Expansion of Proof Obligations	Ternary Simulation	Interval Simulation

Table 1: Summary Generalization PDR

The most challenging problem for the generalization is to find a suitable representation of the atomic reasoning unit. As indicated in Table 1, we propose polytopes as the format of the atomic reasoning unit. Originally, we experimented with a simpler representation, integer cubes. For ease of exposition, in the following section, we will explain the algorithm with integer cubes and illustrate the limitations of this representation. Subsequently, we will describe polytopes and how they can be used effectively in PDR.

3.1 Formulation with Integer Cubes

We now denote with $\mathbf{x} = x_1, x_2, \cdots, x_n$ bitvector variables. We define an integer cube as a set of static intervals on the domain of these variables. The static intervals are to be interpreted in the conjunctive sense, i.e. a point is in the integer cube iff all variables are in their respective static intervals. As an example, consider the integer cube c defined by $c = (3 \leq x_1 \leq 5) \wedge (-4 \leq x_2 \leq 20)$. The point $x_1 = 4$, $x_2 = 0$ is in c whereas the point $x_1 = 4$, $x_2 = -10$ is not. Geometrically, an integer cube corresponds to an orthotope (a.k.a. hyperrectangle) in the n-dimensional space. The definition of integer cubes as atomic reasoning unit appears to be an immediate generalization of the concept of Boolean cubes, allows for an efficient implementation of the algorithm, and the expansion operation is immediate. Instead of skimming Boolean literals, we attempt to increase the intervals of the variables by decreasing lower bounds and increasing upper bounds using binary search. Also, the fact that all inductive invariants can be represented by a union of integer cubes is promising in the theoretical sense.

3.1.1 Expansion of Proof Obligations

The efficiency of the binary version of PDR as documented in [3] is partly based on its ability to generalize proof obligations using ternary simulation. We propose to use interval simulation as a generalization of this idea.

The binary version of PDR uses ternary simulation in two contexts. Firstly, to expand bad, uncovered cubes found in line 2 of Algorithm 1 and secondly to expand a cube $\tilde{c}$ from which c is reachable in line 3 of Algorithm 2. In both cases, a SAT solver is utilized to find a point in the state space which, if reachable, proves that the property does not hold. Consequently, the point represents a proof obligation. The aim of expansion is to generalize these proof obligations so that many points can be processed simultaneously. Note that for the correctness of the algorithm, expansion of proof obligations is not necessary. Hence, a conservative approximation of the expansion is sufficient where we call an approximation conservative if it underapproximates the largest possible expansion. In contrast, an overapproximation would cause the algorithm to potentially report specious counterexamples.

In our generalization of the binary PDR algorithm, the backbone for expansion of proof obligations is interval simulation on expressions. To this end, we associate an interval $\Phi_e = [l_e, u_e]$ with each expression e. We evaluate an interval Φ_e using simulation rules. A small but representative subset of these rules is given below. Note that we denote the minimally and maximally representable number for a given expression with $-\infty$ and ∞, respectively. Binary variables are represented as bitvectors with length one; the corresponding intervals are $[0,0]$, $[0,1]$, and $[1,1]$ which stand for false, false or true, and true, respectively.

$$\frac{}{\Phi_c = [c,c]}\ \text{[const]} \qquad \frac{l \le v \le u}{\Phi_v = [l,u]}\ \text{[var]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2]}{\Phi_{e_1 \wedge e_2} = [\min\{l_1,l_2\}, \max\{u_1,u_2\}]}\ \text{[and]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 + l_2 \ge -\infty \quad u_1 + u_2 \le \infty}{\Phi_{e_1 + e_2} = [l_1 + l_2, u_1 + u_2]}\ \text{[plus-r]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 + l_2 < -\infty \vee u_1 + u_2 > \infty}{\Phi_{e_1 + e_2} = [-\infty, \infty]}\ \text{[plus-o]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad u_1 < l_2}{\Phi_{e_1 < e_2} = [1,1]}\ \text{[lt-1]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad l_1 > u_2}{\Phi_{e_1 < e_2} = [0,0]}\ \text{[lt-0]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad u_1 \ge l_2 \quad l_1 \le u_2}{\Phi_{e_1 < e_2} = [0,1]}\ \text{[lt-x]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad l_1 = 1}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = \Phi_{e_2}}\ \text{[ite-t]} \qquad \frac{\Phi_{e_1} = [l_1,u_1] \quad u_1 = 0}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = \Phi_{e_3}}\ \text{[ite-e]}$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2] \quad \Phi_{e_3} = [l_3,u_3] \quad l_1 \ne u_1}{\Phi_{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} = [\min\{l_2,l_3\}, \max\{u_2,u_3\}]}\ \text{[ite-x]}$$

To illustrate the use of these rules, imagine we attempt to evaluate the expression $e = (x_1 > 2 \wedge x_2 > 2) \vee x_1 \le 2$. We represent e as a directed acyclic graph in which each vertex corresponds to a subexpression of e and each edge corresponds to a dependency between subexpressions (see Figure 1). We calculate intervals by applying the simulation rules topologically from source to sink in the expression graph. After evaluation of a subexpression, we memoize the obtained interval for future use. Denote with $\Phi_e(c)$ the simulation interval associated with expression e where c contains lower and upper bounds of referenced variables. As for our example, assume that we want to evaluate $\Phi_e(c)$ with $c = \{x_1 \ge 3, x_2 \ge 3\}$. In this case, we obtain the intervals as given in Figure 1. We have that $\Phi_e(c) = [1,1]$, i.e. the expression evaluates to one for any pair of values for x_1, x_2 in c.

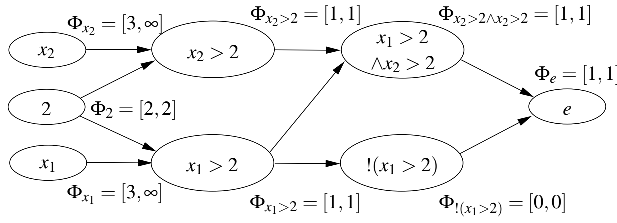


Figure 1: Example of an interval simulation.

Note that the presented interval simulation overapproximates the obtained intervals in the presence of reconvergence. To see why, imagine we removed the constraint on x_1 in c in the example above, i.e. we evaluated $\Phi_e(\tilde{c} = \{x_2 \ge 3\})$. In this case, the simulator would return $[0,1]$ meaning that depending on the actual values of x_1 and x_2 in $\tilde{c}$, expression e could evaluate to 0 or 1. However, assuming $\tilde{c}$, one can simplify e to $e = x_1 > 2 \vee x_1 \le 2 = 1$, i.e. for all values of x_1 and x_2 in $\tilde{c}$, e evaluates to 1 and the accurate result would be $[1,1]$.

For the purpose of expanding proof obligations, imagine we identified point c as a proof obligation using an SMT-solver. We use the described interval simulator as follows. For each variable x_i, we attempt to expand c by relaxing the lower and upper bounds of x_i using binary search. Denote with $\tilde{c}$ a proposed expansion during this binary search. In the context of expanding bad, uncovered cubes, we test whether a proposed expansion $\tilde{c}$ is valid by checking if

$$\Phi_{B \wedge u}(\tilde{c}) = [1,1] \tag{1}$$

where u is the expression describing the points that are uncovered. In case the equation holds, the expansion is valid.

In the context of expanding a cube c from which another proof obligation c' is reachable in one step, we check whether the intervals of the variables after one step starting from $\tilde{c}$ are included in the intervals of the variable given by c', formally

$$\forall x_i . \Phi_{\text{next}_{x_i}}(\tilde{c}) \subseteq \Phi_{x_i}(c') \tag{2}$$

where we denote with next_{x_i} the expression which captures the computation of the next value of x_i. Note that this requires that the transition relation is in the format of transition functions solvable for each variable. In many applications, this is naturally the case.

Note that the overapproximation of the intervals by the simulation procedure as illustrated above causes checks (1) and (2) to yield conservative expansion moves, as desired.

We conclude the description of the formulation of the generalized PDR algorithm with integer cubes by illustrating operation and limitations of the algorithm using integer cubes using two examples.

3.1.2 Example: Simple

Consider PDR was called with the following model checking problem in which B is unreachable.

$$\begin{aligned} I &:= (n = 1) \wedge (x = 0) \\ T &:= (n > 0) \wedge (x' = x + 1) \wedge (n' = n - 1) \\ B &:= (x \ge 3) \end{aligned}$$

Figure 2 shows how a simplified version of PDR would prove this fact. Each row corresponds to a trace at a certain point in the algorithm. Each plot in a trace corresponds to a frame where the first plot in a trace corresponds to f_0 of this trace, the second plot to f_1, and so forth.

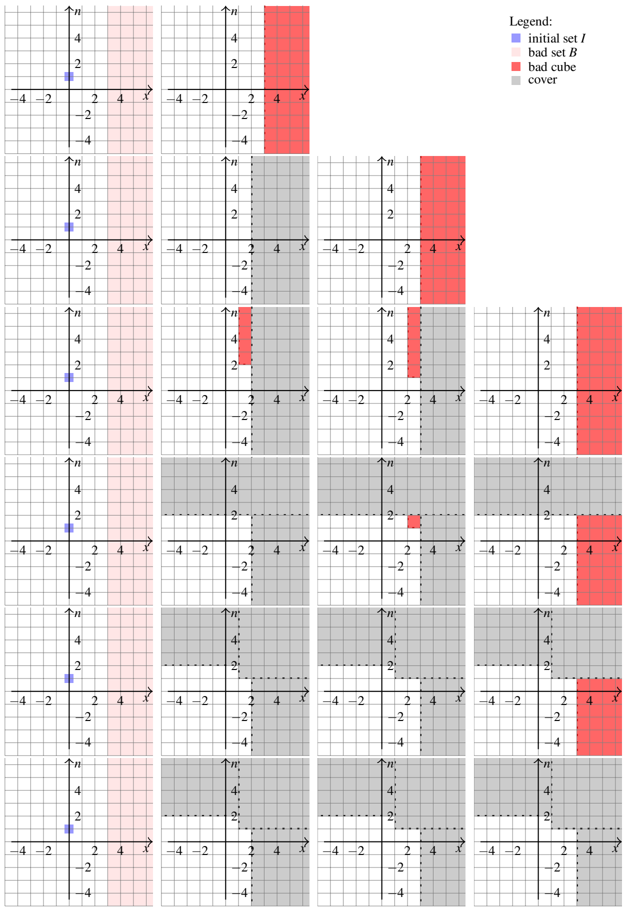


Figure 2: Iterative construction of proof for example Simple.

Initially, the trace has only one frame, f_0. As $B \wedge I =$ false, B is not reachable in zero steps and f_1 is pushed at the end of the trace. In f_1, `findBadCube()` returns integer cube $x \geq 3$ that is in B and uncovered (indicated by the red rectangle in the first row of Figure 2). Next, PDR checks whether there are points in $x \geq 3$ that are reachable from the initial set in f_0 in one transition. This is not possible, hence $x \geq 3$ can be covered. Before being added to the cover of f_1, the cube is expanded, yielding $x \geq 2$ as indicated in gray in the second row of Figure 2. After adding this integer cube to the cover, there are no longer uncovered points in B and f_2 is pushed at the back of the trace. Now, PDR attempts to propagate integer cube $x \geq 2$ to f_2. This is not possible, however, because from the overapproximation of the reachable set in f_1 (the inverse of the cover) one can reach points in $x \geq 2$ in one transition. After the propagation phase, PDR continues with finding uncovered cubes in B in f_2, yielding $x \geq 3$ for another time (see trace in the second row of Figure 2). No point in $x \geq 3$ can be reached from the reachable set in f_1 and the cube is added to the cover in f_2. Now, B is covered completely and PDR pushes f_3 in the end of the trace. In the succeeding propagation phase, the attempt to propagate $x \geq 3$ fails. The subsequent call of `findBadCube()` returns again $x \geq 3$ (see third row in Figure 2 in f_3). This time, however, points in f_3 can be reached from the uncovered region of f_2. For instance, one can reach $x = 3, n = 0$ from $x = 2, n = 1$ in f_2 in one transition. Using interval simulation, one can generalize this new proof obligation to $2 \leq x < 3 \wedge 1 \leq n$. Points in this region can also be reached from the previous frame, yielding an additional proof obligation $1 \leq x < 2 \wedge 2 \leq n$ in f_1 (see third row in Figure 2). No point in this cube can be reached from the initial set in f_0. Hence, a new cube covering the area is generated, expanded to $n \geq 2$ and added to f_1. In the sequel, PDR also attempts to propagate the new cube to the next frames and realizes that this is in fact possible. Therefore, the new cube is also added in frames f_2, and f_3 (see row four in Figure 2). As a consequence of adding cube $n \geq 2$ to f_1, all points in $2 \leq x < 3 \wedge 1 \leq n$ in f_2 cease to be reachable from f_1. After expansion, this yields an additional cube $x \geq 1 \wedge n \geq 1$ in the cover of f_2. As with cube $n \geq 2$, this cube cannot be reached in any succeeding frame either, hence it is propagated as well. Also note that if a cube cannot be reached within two steps, it can neither be reached within one step. Hence, it can also be considered covered in f_1 (see row five in Figure 2). As a consequence of adding $x \geq 1 \wedge n \geq 1$ to the cover of f_2, $x \geq 3$ in f_3 ceases to be reachable and allows to cover B completely. Note that transitions such as $x = 2, n = -2$ to $x = 3, n = -3$ are invalid by the constraint $n \geq 0$ in the transition relation. We obtain the cover in row six in Figure 2. Note that in this row, the covers in f_2 and f_3 are identical. This means that no point in the overapproximation of the reachable set in f_2 can reach any state outside this overapproximation, i.e. we have found an inductive invariant proving that B is unreachable. Technically, however, Algorithm 1 would not detect this until after pushing a new frame and successfully propagating all cubes in f_3 to f_4.

3.1.3 Example: Linear Invariant

Consider now the following model checking problem

$$\begin{aligned} I &:= (x + 2y \leq 5) \\ T &:= (x' = x + 1) \wedge (y' = y - 1) \\ B &:= (x + 2y > 5) \end{aligned}$$

Note that the initial condition is preserved by the transition relation and serves itself as an inductive invariant to prove that B is unreachable.

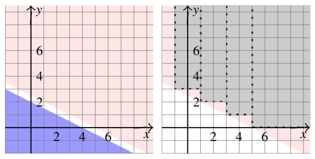


Figure 3: Attempt to decide Example Linear Invariant

For this example, the formulation of PDR using integer cubes is not able to construct an inductive invariant efficiently. The trace in Figure 3 which was recorded after a couple of iterations of the main loop illustrates this fact. For each integer on the line $x + 2y = 5$, one needs an integer cube to cover the bad area entirely. Assuming that the variables are 32-bit integers, this means that PDR needed to add 2^{31} cubes.

In general, if the inductive invariant required to decide a model checking problem contains a relation between two or more variables, it is not possible to represent the inductive invariant efficiently using integer cubes. Inductive invariants that relate variables are common in many possible applications of our model checker, which strongly suggests that integer cubes are a bad choice as atomic reasoning unit.

3.2 Formulation with Polytopes

The limitations of the algorithm with integer cubes pointed out in the previous section suggests for a more expressive formulation.

Instead of integer cubes, we propose to use polytopes as our atomic reasoning unit. Mathematically, a polytope can be represented as a system of linear inequalities $\mathbf{A}\mathbf{x} \leq \mathbf{b}$. Although frequently used atomic operations in PDR, such as checking for implication, become less efficient with polytopes than with integer cubes, the algorithm can relatively easily be extended to cope with polytopes. With respect to expansion, we now attempt to relax the individual boundaries by increasing their right-hand-sides.

Conceptually, using polytopes as the atomic reasoning unit permits us to represent any piecewise linear loop invariant efficiently. For instance, the inductive loop invariant in the second example can be represented using a single polytope.

In the current formulation of the algorithm, however, the additional expressive power of polytopes over integer cubes is not used. Initially, `findBadCube()` returns an integer cube. The defined expand operation allows to increase the polytopes by moving the limiting halfplanes parallely outward to increase the polytopes but does not allow for changing the principal shape of any polytope.

Hence, we also require a mechanism in our algorithm to use the added expressiveness of polytopes. To this end, we define a new operation, `reshape()`, which is geared towards resolving this problem. In Algorithm 2, `reshape()` is called after propagation in line 5 and is followed by an additional expansion.

Mathematically, the purpose of this reshape-operation is to increase the number of terms within the boundaries of the polytope. Initially, we only have unary boundaries, such as $a_i x_i \leq b_i$. One solution towards finding boundaries with higher arity would be to add an additional variable to obtain a boundary of the form $a_i x_i + a_j x_j \leq b_i$ and run a search algorithm to find values of a_i and a_j that are maximum with respect to a suitable optimization criterion. Note that this brute-force solution basically resembles a random search which is unlikely to be efficient.

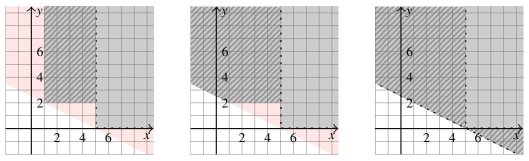
Instead, we propose a more targeted approach. The principal idea of our reshape-operation is to use information from several polytopes to make guesses for possible new boundaries. Then, one attempts to substitute these new boundaries for existing ones of lower arity.

3.2.1 Example: Linear Invariant (revisited)

Consider our second example for another time. In Figure 4, we illustrate how our reshape-operation would proceed after the second polytope has been found. Figure 4(a) displays the situation right before the call of the reshape-operation. Assume that `reshape()` is called on the striped polytope (pivot). The other polytope acts as guide. The situation suggests that the line defined by the lower-left corners of the two polytopes might be a good candidate as a new boundary. Hence, `reshape()` calculates this line and substitutes it for one of the neighbor boundaries of the corner. In Figure 4(b), the new boundary is substituted for $x \geq 1$. Next, `reshape()` checks if the polytope is still unreachable from the previous frame after the substitution. This is the case for the given example. After the reshape-operation, `expand()` is called once more which also eliminates the other neighbor boundary (see Figure 4(c)). Now, the reshaped polytope covers B completely.

3.2.2 General Reshape Algorithm

Details of the general algorithm are given in Algorithm 3. If called on a polytope, the routine iterates through all corners and finds for each promising set of guides G a corresponding hyperplane. The efficiency and effectiveness of `reshape()` depends critically on the choice of guides. With respect to effectiveness, iterating through all possible sets is clearly the optimal choice. However, the number of sets grows exponentially with the number of other polytopes that may act as guide. Consequently, a smaller choice guided by heuristics is necessary. Our experimentation showed that sets of guides that are close to p are most likely to yield a successful substitution. After a set of guides G is found, we find the hyperplane h fixed by the corners of the guides and iteratively attempt to substitute it for a neighbor boundary b of the corner in p. If such a substitution yields a polytope that is reachable from the previous frame, we reverse the substitution and continue with our attempt to reshape p by substituting h for another neighbor boundary. Otherwise, we bail out, keeping the substitution and continue with trying to reshape another corner of p.



(a) before `reshape()` (b) after `reshape()` (c) after `expand()`

Figure 4: Flow of a Reshape Operation

Algorithm 3 `reshape`(Polytope p, int l)

```
1: for all corner $c$ of $p$ do
2:   for all promising set of $n$ guides $G$ do
3:     $h$ = findHyperplane($c$, $p$, $G$)
4:     for all neighbor boundary $b$ of $p$ w.r.t. $c$ do
5:       substitute($p$, $h$, $b$)
6:       if $p$ reachable from $f_{l-1}$ then substitute($p$, $b$, $h$)
7:       else break 2
```

The reshape-operation as described in Algorithm 3 is able to substitute an n-ary boundary for a unary one where n is the number of dimensions of the state space. This comes at the expense of an asymptotic running time that grows exponentially with the number n. In problem domains which only require linear invariants that relate less than n variables with each other, Algorithm 3 should be restricted appropriately for better runtime performance.

4 Implementation and Experimentation

We implemented the proposed algorithm in C++. The skeleton of our implementation is similar to the one suggested in [3]. As backend solver, we used the QF_BV part of the SMT-solver Yices [8] instead of a SAT-solver. This choice allowed us to implement all of the suggested optimizations described in [3], in particular to consider the unsatisfiable core for fast expansion and reusing learned clauses by incremental solving with retractable assertions (hot solving). To generalize proof obligations, we use interval simulation on the transition relation instead of ternary simulation as discussed in Section 3.1.1.

For experimentation, we compiled a set of 10 representative benchmark problems from the regression test suite of InvGen [9]. To this end, we manually extracted QF_BV model checking problems suitable to infer typical loop invariants from the test programs. In all derived model checking problems, the property holds, i.e. no bad state is reachable from any initial state.

4.1 Performance

Table 2 summarizes our runtime results. We attempted to solve each model checking instance three times. For the first two attempts, we used our generalized PDR algorithm with integer cubes and polytopes, respectively, as the atomic reasoning unit. Thirdly,

Benchmark	QF_BV PDR		Binary PDR
	Integer Cubes	Polytopes	
simple	3.01	0.25	0.93
lin	timeout	0.07	7.82
max	timeout	0.40	timeout
diameter	0.22	0.25	0.15
gen_di	timeout	0.34	0.89
split	0.03	0.04	0.05
sim_inv	timeout	0.13	450.22
bound	timeout	11.27	12.02
rajam	timeout	0.62	timeout
parity	timeout	timeout	0.12

Table 2: Performance of PDR Algorithms

as a point of comparison, we attempted to solve the problems using the original binary PDR algorithm. Therefore, we formulated all problems in BTOR [10], a format for specifying word-level model checking problems, and used the tool Synthebtor, which is part of the distribution of the Boolector SMT-solver [11], to bit-blast the problems into a standard format for specifying hardware-model checking problems (AIGER [12]). Eventually, we solved the derived Boolean model checking problems with the implementation of PDR contained in the logic synthesis and verification tool ABC [13]. In Table 2, all runtime results are reported in seconds. We report timeout in case the solver did not terminate within 30 minutes.

Comparing columns two and three of the table, it is clear that polytopes are a better choice for typical model checking instances derived from programs. In all problems in which an efficient representation of the invariant is a relation of variables, the PDR algorithm with integer cubes fails to solve the problem efficiently. Interestingly, in the remaining problems, the formulation with integer cubes is faster. This is because certain operations, such as checking that one atomic reasoning unit implies another, can be solved faster with integer cubes. As an exception, note that the implementation with integer cubes solves the instance *simple* slower. Manual investigation indicates that this is because the two algorithms derive different invariants to solve the problem whereas the invariant with polytopes can be found substantially faster.

Comparing columns three and four of the table, it is visible that the generalized PDR algorithm solves the problems typically faster than the binary PDR algorithm. Most significantly, in two cases (*rajam* and *max*), the generalized algorithm solves the instances efficiently whereas the binary version times out. The binary version of PDR appears to perform similarly efficient for instances where the invariant can be represented with unary constraints (*simple*, *diameter*, *split*) which is to be expected because constraints like $x_1 \leq c$ with c constant can be represented efficiently as Boolean cubes. Note that instance *parity* could not be solved with our generalized PDR algorithm but very quickly with the original PDR algorithm. This is due to the fact that *parity* requires the invariant representing "odd number", which cannot be represented efficiently as a disjunction of polytopes. We intend to find a solution of this specific limitation as future work because we have encountered problems of this kind several times during our experimentation. In general, however, our experiments suggest that the vast majority of invariants encountered in practice can be represented as a disjunction of polytopes.

We also investigated the impact of interval simulation on the performance of the generalized PDR algorithm. Without interval simulation, the runtimes for solving the considered benchmarks increase from approximately 1.3× to over 300×. This is a similar range of performance improvements as reported in [3] with ternary simulation suggesting that interval simulation is an appropriate means for the expansion of proof obligations.

5 Conclusions and Future Work

Property directed reachability is an efficient model checking algorithm for Boolean logic. To increase the scope of possible application domains, a generalization to richer logics is desirable.

In this paper, we presented such a generalization to admit problems formulated in the theory QF_BV. We discussed two possible choices for the atomic reasoning unit, integer cubes and polytopes, and argued that polytopes are the more promising choice as they allow the representation of linear relations between variables efficiently. To use this expressiveness, we propose a heuristic approach that increases the number of terms within the constraints based on the geometric interpretation of the state space. For expansion of proof obligations, we propose the use of interval simulation.

In our implementation of the outlined algorithm, we were able to implement analogs to all optimizations known for the binary version of the algorithm. We experimented with our software prototype and reported results in which we compared the performances of the generalized algorithm formulated with integer cubes versus polytopes versus the original, binary version of the algorithm run on bit-blasted versions of the benchmark problems and found that the generalized PDR algorithm with polytopes performs best.

As future work, we intend to investigate the potential of generalizing the PDR algorithm for other logics. In particular, we anticipate that a generalization for QF_LA would be useful in many applications.

References

[1] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of Computer Aided Verification*, vol. 2742 of *Lecture Notes in Computer Science*, pp. 1–13, 2003.

[2] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Proc. of the 12th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, (Berlin, Heidelberg), pp. 70–87, Springer-Verlag, 2011.

[3] N. Eén, A. Mishchenko, and R. Brayton, "Efficient Implementation of Property Directed Reachability," in *Proc. of the Int'l Conf. on Formal Methods in Computer-Aided Design*, FMCAD '11, (Austin, TX), pp. 125–134, 2011.

[4] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *Comms. ACM*, vol. 5, pp. 394–397, July 1962.

[5] K. L. McMillan, A. Kuehlmann, and M. Sagiv, "Generalizing DPLL to Richer Logics," in *Proc. of Computer Aided Verification*, vol. 5643 of *Lecture Notes in Computer Science*, pp. 462–476, 2009.

[6] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *Proc. of the 15th Int'l Conf. Theory & Appl, Satisfiability Testing (SAT)*, (Berlin, Heidelberg), pp. 157–171, Springer-Verlag, 2012.

[7] A. Cimatti and A. Griggio, "Software Model Checking via IC3," in *Proc. of Computer Aided Verification*, vol. 7358 of *Lecture Notes in Computer Science*, pp. 277–293, 2012.

[8] B. Dutertre and L. D. Moura, "The Yices SMT Solver," tech. rep., SRI Interational, 2006.

[9] A. Gupta and A. Rybalchenko, "InvGen: An Efficient Invariant Generator," in *Proc. of Computer Aided Verification*, vol. 5643 of *Lecture Notes in Computer Science*, pp. 634–640, 2009.

[10] R. Brummayer, A. Biere, and F. Lonsing, "BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking," in *Proc. of the Joint Workshop of the 6th Int'l Workshop on SMT & 1st Int'l Workshop on Bit-Precise Reasoning*, SMT '08/BPR '08, (New York, NY, USA), pp. 33–38, ACM, 2008.

[11] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Proc. of the 15th Int'l Conf. on Tools & Algorithms for the Construction & Analysis of Systems*, TACAS '09, (Berlin, Heidelberg), pp. 174–177, Springer-Verlag, 2009.

[12] A. Biere, "The AIGER And-Inverter Graph format." http://fmv.jku.at/aiger.

[13] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," in *Proc. of Computer Aided Verification*, vol. 6174 of *Lecture Notes in Computer Science*, pp. 24–40, 2010.