Property Directed Reachability for QF_BV with Mixed Type Atomic Reasoning Units

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

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

Abstract

The generalization of Property Directed Reachability (PDR) for the theory QF_BV presented in [1] outperforms the original formulation if the required inductive invariant can be represented efficiently as a set of polytopes. However, many QF_BV model checking instances do not belong in this class and can be solved quickly with the original PDR algorithm. In this paper, we present a hybrid approach which uses both polytopes and Boolean cubes as atomic reasoning units combining the advantages of either homogeneous approach. We discuss theoretic properties of the presented algorithm and report experimental results demonstrating its effectiveness.

1 Introduction

Property Directed Reachability (PDR, also known as IC^3) was initially proposed in [2] and demonstrated remarkable good performance (3$^{\text{rd}}$ place) in the Hardware Model Checking Competition (HWMCC) 2011. The authors of [3] have shown that a more efficient implementation of the algorithm would have won the HWMCC 2011.

In addition to the excellent runtime performance, PDR has additional favorable properties. For instance, it excels in finding counterexamples and in proving that none exists alike, it has modest memory requirements, it is parallelizable, and it is incremental.

These convincing algorithmic properties of PDR have instigated interest in generalizations of the binary formulation to richer logics. For instance, the authors of [4] present generalizations for pushdown systems and to the theory of quantifier-free formulae over linear arithmetic (QF_LA), the authors of [5] an extension for the verification of timed systems, and the authors of [6] propose the use of PDR for verifying safety properties in well-structured transition systems.

In [1], a generalization of PDR to the theory of quantifier-free formulae over bitvectors (QF_BV) was proposed. In contrast to the binary formulation of the algorithm that uses Boolean cubes as atomic reasoning units (ARUs), the presented approach uses polytopes as ARUs. The main incentive of polytopes is that they can capture linear relations between bitvectors that are interpreted as integers efficiently. As a consequence, the PDR algorithm with polytopes outperforms the original algorithm on instances which require inductive invariants composed of linear relations for their solution.

Although QF_BV model checking instances with relevant piecewise linear relations as inductive invariants are frequent in application domains such as program verification, many instances also require inference of logic invariants that cannot be efficiently represented as piecewise linear invariants. For instances of this type, the Boolean PDR algorithm [2] often outperforms its generalization with polytopes. Whereas one could use the original version for these instances, none of the homogeneous approaches is successful in cases where invariants are required which combine logic parts with piecewise linear parts.

In this paper, we present a novel hybrid approach that uses both Boolean cubes and polytopes as ARUs. As such, the presented approach overcomes the limitations of exclusive use of polytopes or Boolean cubes and experiments with a set of benchmarks derived from program verification suggest that the proposed algorithm with mixed type ARUs outperforms either homogeneous approach.

The remainder of this paper is structured as follows: To keep the paper self-contained, we provide a cursory description of a generic PDR algorithm in the following section. Next, in Section 3, we present details of the proposed hybrid approach and discuss some of its characteristics followed by a description of our implementation and experimental results in Section 4. Finally, we draw conclusions and describe future work in Section 5.

2 Property Directed Reachability for QF_BV

In this section, we present aspects of the QF_BV PDR algorithm relevant to this paper. For conciseness, many details, some of them fundamental for the performance of the algorithm, are omitted. We refer the reader to [3] for a detailed discussion of the algorithmic skeleton and to [1] for a detailed discussion of the specifics associated to polytopes.

Given a state space spanned by the domain of n bitvector variables $\mathbf{x} = x_1, x_2, \ldots, x_n$, the input of the PDR algorithm is a model checking instance $M = (I, T, B)$ where the QF_BV formula I defines a set of initial states in $\mathbf{x}$, B a set of bad states in $\mathbf{x}$ and T a transition relation over $\mathbf{x}$ and $\mathbf{x}'$, where $\mathbf{x}'$ is a copy of $\mathbf{x}$ which corresponds to the same variables one time step later. T is true iff $\mathbf{x} \rightarrow \mathbf{x}'$ represents a valid transition of the system. The model checking problem is to decide whether a state in B can be reached from a state in I by using only valid transitions in T.

The conceptual strategy of PDR to solve a given model checking problem is to iteratively find small truth statements and combine all these lemmata to obtain the desired proof for the given problem. The motivation for this strategy is that solving a large number of small problems may be more tractable than attempting to solve one big problem at once.

More concretely, PDR constructs a trace t consisting of frames $f_0, f_1, \cdots$. Each frame contains a set of ARUs $\{a_i\}$. For the original Boolean algorithm, each ARU a is a Boolean cube and for the formulation with polytopes, each ARU a is a set of linear inequalities $\sum_i c_i x_i \leq b$. If there is an ARU a in frame f_i, the semantic meaning of this is that all states contained in a cannot be reached within i steps from the initial states. In this case, we say that the states in

978-1-4799-2816-3/14/$31.00 ©2014 IEEE

a are *covered* in frame i and we will call all ARUs in frame f_i the *cover*. The inverse of the cover in f_i represents an overapproximation of the states that are reachable in i steps. In frame f_0, a state is covered if it is not in the initial set I.

Algorithm 1 PDR(I,T,B)

```
1: while true do
2:   ARU a = FINDBADARU()
3:   int l = LENGTHSTRACE()
4:   if a then
5:     if !RECCOVERARU(a,l) then return "property fails"
6:   else
7:     PUSHNEWFRAME()
8:     if PROPAGATEARUS() then return "property holds"
```

Algorithm 2 RECCOVERARU(ARU a, int l)

```
1: if l = 0 then return false
2: while a reachable from ã in one transition do
3:   if !RECCOVERARU(ã, l − 1) then return false
4: EXPAND(a,l) ; PROPAGATE(a,l)
5: add a to F_l
6: return true
```

Algorithm 1 shows the overall PDR algorithm. In each iteration, the algorithm searches for an ARU in the last frame of the trace that is in B and not yet covered using FINDBADARU(). If such an ARU a exists, the algorithm tries to recursively cover the proof obligation a (RECCOVERARU(), see Algorithm 2). To this end, the routine checks if a is reachable from the previous frame. Assume for now that this is the case and denote with $\tilde{a}$ an ARU in the previous frame that is not covered and from which a can be reached in one step. Then RECCOVERARU() calls itself recursively on $\tilde{a}$. If such a sequence of recursive calls reaches back to frame f_0, the corresponding call stack effectively proves that a can be reached from I, i.e. that the property fails. Otherwise, if an ARU a is proved to be unreachable from the previous frame, it can be added to the cover of the current frame. For efficiency, however, the algorithm attempts to expand and to propagate the ARU to later frames before doing so. Continuing the discussion of Algorithm 1, if FINDBADARU() returns without successfully finding an uncovered ARU in B, we know that B is covered in the last frame f_l. As we maintain 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 ARUs from frame l to frame $l+1$. If this is successful for all ARUs, 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.

3 Hybrid Approach with Mixed Type ARUs

We start this section by discussing a motivating example for the approach with mixed type ARUs which illustrates the shortcomings of using either Boolean cubes or polytopes exclusively. Next, we will define the hybrid approach and discuss some of its properties. Finally, in Subsection 3.3, we discuss an extension of simulation based expansion of proof obligations suitable for the hybrid approach.

3.1 Example: Hybrid Invariant

Consider the following model checking problem

$$\begin{aligned} I &:= (2\times x_1 \equiv x_2) \wedge (x_2 + x_1 \leq 3) \\ T &:= (x_1' \equiv x_1 + 1) \wedge (x_2' \equiv x_2 - 2) \wedge (x_1' > x_1) \wedge (x_2' < x_2) \\ B &:= (x_2 + x_1 \geq 4) \vee (x_2 \bmod 2 \equiv 1) \end{aligned}$$

The snapshot of the PDR trace in Figure 1 illustrates the issue the PDR algorithm with polytopes encounters when attempting to solve this model checking instance.

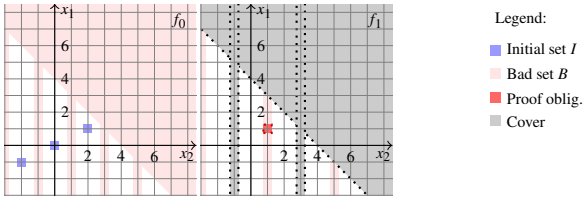


Figure 1: Attempt to Solve Example Hybrid Invariant

The inductive invariant required to solve this instance is $(x_2 + x_1 \leq 3) \wedge (x_2 \text{ even})$. The corresponding cover in the PDR trace includes two parts, '$x_2 + x_1 \geq 4$' and 'x_2 odd'. The first part can be covered using a polytope. However, the other aspect can not be efficiently represented using polytopes. Instead, the algorithm would add a polytope for each odd number representable by x_2. Assuming that x_2 is a 32-bit bitvector, this would require 2^{31} polytopes, causing the overall algorithm to be very inefficient.

In contrast, the Boolean PDR algorithm would efficiently find this second part of the inductive invariant as it can be represented by a single Boolean cube but fail in finding the first part efficiently.

3.2 Probabilistic Specialization

The shortcomings of using either exclusively Boolean cubes or polytopes motivate a hybrid approach. To this end, we define the ARU to be either a Boolean cube or a polytope. Each ARU is created either in FINDBADARU() in line 2 of Algorithm 1 as a point in bad that is not yet covered or in line 2 of Algorithm 2 as point from which another proof obligation is reachable. In both cases, the point is found by a theory solver and can be interpreted as a polytope or a Boolean cube. A specialization to a specific kind of ARU becomes necessary before the proof obligation is expanded because the expansion sequence is different for polytopes (relaxing individual inequalities) and Boolean cubes (chopping of literals). We propose to specialize a point probabilistically. Therefore, we define probability c and specialize to a Boolean cube with probability c and to a polytope with probability $1-c$.

This probabilistic choice guarantees that a favorable specialization can be expected to be chosen in a constant number of attempts. To see why, consider the example in Figure 1 for another time. With probability c, the marked proof obligation is specialized to a Boolean cube. In this case, the desired part 'x_2 even' of the inductive invariant would be found immediately. Otherwise, the proof obligation would be covered by the polytope $x_2 = 1$. In this case, after the next call to FINDBADARU(), a new proof obligation with '**x** odd' is found followed by another probabilistic decision for specialization.

The probability that a choice for a Boolean cube is made in the i^{th} iteration is $c(1-c)^{i-1}$. Hence, the expected number of trials

until the favorable specialization is found can be calculated to be

$$E\{\text{trials until Boolean cube specialization}\} = c\sum_{i=1}^{\infty} i(1-c)^{i-1} = \frac{1}{c}$$

Analogously, one calculates

$$E\{\text{trials until polytope specialization}\} = (1-c)\sum_{i=1}^{\infty} ic^{i-1} = \frac{1}{1-c}$$

Consequently, as long as $c = (0,1)$, one can expect a favorable decision within a constant number of times. The optimal choice for c depends on the concrete model checking instance. We will investigate the impact of c in the experimental section of this paper.

3.3 Expansion of Proof Obligations

The excellent performance of the binary version of PDR as documented in [3] can be partly attributed to its ability to efficiently and effectively expand proof obligations using ternary simulation. In [1], this idea was generalized to interval simulation for the QF_BV PDR model checker with polytopes. In this section, we describe a suitable generalization for the presented QF_BV PDR model checker with mixed type ARUs.

Proof obligations are expanded after their specialization. At this time, a proof obligation a consists of a single point in the state space. The aim of the expansion is to add additional points to a so that the resulting set can be processed as a union in the sequel, stimulating more abstract reasoning.

Note that expansion of proof obligations, albeit practically critical for performance, is not required for the correctness of the algorithm. The goal is to find a close underapproximation of the maximum possible expansion that can be calculated efficiently.

The expansion is achieved using an expansion sequence suitable for the type of the proof obligation. At each step of the sequence, an expansion $\hat{a}$ is proposed. The proposal will be accepted if we can reach a point in B from any point in $\hat{a}$.

The backbone for this test is simulation of expressions. In the following, we will denote with $\Phi_e(\hat{a})$ an overapproximation of the values expression e can take under the constraint $\hat{a}$ on the variables.

Expansion of proof obligations is used in two contexts. First, we use it for expanding ARUs in B that are not yet covered (line 2 in Algorithm 1). To this end, we test whether a proposed expansion $\hat{a}$ is valid by checking if

$$\Phi_{b\wedge u}(\hat{a}) \equiv \text{true} \tag{1}$$

where b is an expression describing the bad set B of the model checking instance and u is the expression describing the points that are not yet covered. If the equation holds, the expansion is valid.

Second, in the context of expanding an ARU $\tilde{a}$ from which another proof obligation a is reachable in one step (line 2 in Algorithm 2), we check whether the possible valuations of the variables after one step starting from $\hat{\tilde{a}}$ are included in the possible valuations of the corresponding variables under a, formally

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

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.

Under the assumption that the simulation values $\Phi_e(\hat{a})$ represent overapproximations, the checks (1) and (2) are conservative in the sense that an expansion to $\hat{a}$ or $\hat{\tilde{a}}$, respectively, is accepted only if a bad state is reachable from all points in the proposed expansion.

3.3.1 Ternary Simulation

The original PDR algorithm uses ternary simulation to calculate simulation values for expressions. Representing binary variables as intervals $[1,1]$, $[0,1]$, and $[0,0]$ which stand for true, true or false (X), and false, respectively, one can use the rules in Figure 2 to obtain simulation values for any expression composed of AND- and INV-operations. As one can decompose any bitvector expression

$$\frac{}{\Phi_c = [c,c]}\ [\text{const}] \qquad \frac{}{\Phi_X = [0,1]}\ [\text{x}] \qquad \frac{l \le v \le u}{\Phi_v = [l,u]}\ [\text{var}]$$

$$\frac{\Phi_{e_1} = [l_1,u_1] \quad \Phi_{e_2} = [l_2,u_2]}{\Phi_{e_1\wedge e_2} = [\min\{l_1,l_2\},\max\{u_1,u_2\}]}\ [\text{and}] \qquad \frac{\Phi_e = [l,u]}{\Phi_{\neg e} = [1-u,1-l]}\ [\text{inv}]$$

Figure 2: Simulation Rules for Ternary Simulation

into equivalent AND-Inverter expressions using their circuit representations, this allows to calculate the simulation values required in checks (1) and (2).

It remains to discuss how the range of variables is restricted given an ARU $\hat{a}$. If the ARU is a Boolean cube, this is straightforward. Otherwise, if the ARU is a polytope, the situation is slightly more complicated: It is instructive to make two observations. First, note that at the beginning of the expansion move, $\hat{a}$ represents a point. Second, any proposed expansion for a polytope is derived by relaxing individual inequalities of the polytope. Combining these two observations, we can conclude that at any point of the expansion sequence, the polytope encodes a static integer interval for each variable. For a bitvector variable x of m bits, let $[l,u]$ be the interval x can take while staying in $\hat{a}$. With i being the index of the bit we are interested in, we can use $\text{SELECT}(l,u,i)$ in Algorithm 3 to obtain the corresponding range of values the bit can take. Note that in Algorithm 3, we denote with $l[i:m-1]$ the vector of the $m-i$ most significant bits of l if l was stored in a bitvector of length m. The definition of $u[i:m-1]$ is analog.

Algorithm 3 SELECT(l,u,i)

1: **if** $l[i:m-1] \equiv u[i:m-1]$ **then return** $[l[i],l[i]]$
2: **else return** [0,1]

We can conclude that ternary simulation can be used for the expansion of proof obligations for the QF_BV generalization of the PDR algorithm. However, expansion of proof obligations using ternary simulation cannot be expected to perform effectively if used for a model checking instance where the transition function captures high-level constraints where bitvectors are interpreted as integers. In particular, using ternary simulation, it is not possible to represent any intervals of integers that contain both positive and negative values in the two's complement representation causing gross overapproximation of simulation values in practice.

3.3.2 Interval Simulation

These limitations of ternary simulation suggests a simulation method that captures arithmetic operations more naturally. One way to achieve this is interval simulation as proposed in [1]. Consider the representative choice of simulation rules in Figure 3 where we denote the minimally and maximally representable numbers for a given expression with $-\infty$ and ∞, respectively.

In the case of interval simulation, restricting the range of variables given an ARU $\hat{a}$ is immediate if $\hat{a}$ is a polytope. In the case $\hat{a}$

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

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

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

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

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

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

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

Figure 3: Choice of Simulation Rules for Interval Simulation

is a Boolean cube constraining the m bits of x such as $x[i]\subseteq[l_i,u_i]$, we can use procedure $\text{CONJOIN}(l_0,u_0,\ldots,l_{m-1},u_{m-1})$ in Algorithm 4 to calculate the interval for bitvector x.

Algorithm 4 $\text{CONJOIN}(l_0,u_0,\ldots,l_{m-1},u_{m-1})$

1: l,u = bitvector with length m
2: **for** $i=0$ to $m-1$ **do**
3: **if** $l_i\equiv u_i$ **then** $u[i]=l[i]=l_i$
4: **else**
5: $l[i]=\begin{cases}1 & \text{if } i\equiv m-1\\ 0 & \text{otherwise}\end{cases}$; $u[i]=1-l[i]$
6: **return** [INTEGERVALUE(l),INTEGERVALUE(u)]

Though these rules capture the meaning of arithmetic operations well, analogous bitvector rules for logic operations such as AND are difficult to formulate and representing their results as integer intervals can lose a lot of precision. Consider, e.g. we wanted to calculate $\Phi_{e_1\wedge e_2}$ given $\Phi_{e_1}=[-1,0]$ and $\Phi_{e_2}=[-8,-7]$ where all expressions are 4-bit integers. Using the two's complement representation of integers, this corresponds to the following calculation

$$\begin{array}{r}[-1,\ 0]\\ \wedge\ [-8,-7]\\ \hline [-8,\ 1]\end{array}\ \begin{pmatrix}\text{interval}\\ \text{simulation}\end{pmatrix}\ \leftrightarrow\ \begin{array}{r}\text{- - - -}\\ \wedge\ 1\,0\,0\,\text{-}\\ \hline \text{- }0\,0\,\text{-}\end{array}\ \begin{pmatrix}\text{ternary}\\ \text{simulation}\end{pmatrix}$$

Even the optimal (tightest overapproximation) result representable as an integer interval $[-8,1]$ contains more points than the exact result on the right which does e.g. not include -2.

3.3.3 Hybrid Simulation

The strengths and weaknesses of each homogeneous simulation strategy suggests a hybrid approach. To this end, we define two additional simulation rules [select] and [conjoin] as in Figure 4 that use Algorithms 3 and 4, respectively. The rule [select] is for extracting individual bits from a bitvector and the rule [conjoin] serves for combining bits to a bitvector. Combining these rules with the rules from ternary simulation in Figure 2 and the rules of interval simulation as in Figure 3 allows for a simulation where ternary simulation is used for the logic operations and interval simulation for the arithmetic operations. At the interfaces between the

$$\frac{\Phi_e=[l,u]}{\Phi_{e[i]}=\text{SELECT}(l,u,i)}\ \text{[select]}$$

$$\frac{\Phi_{e[0]}=[l_0,u_0] \quad \ldots \quad \Phi_{e[m-1]}=[l_{m-1},u_{m-1}]}{\Phi_e=\text{CONJOIN}(l_0,u_0,\ldots,l_{m-1},u_{m-1})}\ \text{[conjoin]}$$

Figure 4: Conjoin and Select Simulation Rules

two kinds of operations act the transformation rules [select] and [conjoin].

As such, the choice between interval simulation and ternary simulation is driven by the specific expressions and an operation is treated by the kind of simulation which captures its conceptional idea the best. The rules [conjoin] and [select] which are susceptible to losing precision are only applied when necessary at the interface between operations of different kinds.

3.3.4 Example: Simulation

We conclude the discussion of the simulation based expansion of proof obligations with an example. Assume we attempt to simulate $\Phi_e(\hat{a})$ with

$$\begin{array}{llllll} e & := & (e_1<2)\wedge(0\leq e_1) & e_1 & := & e_2\wedge e_3\\ e_2 & := & x_1-x_2+2 & e_3 & := & y_1\vee y_2\end{array}$$

and $\hat{a}:=(1\leq x_1\leq 5)\wedge(0\leq x_2\leq 3)\wedge(y_1\in\text{-00-})\wedge(y_2\in 100\text{-})$

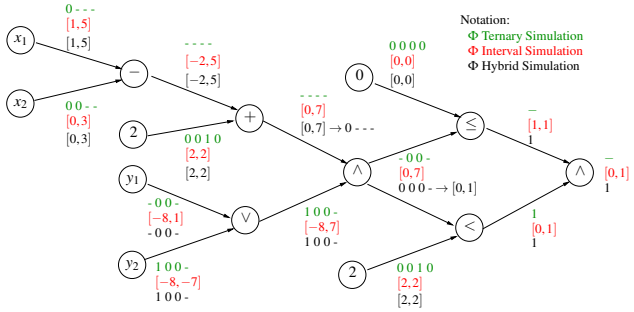
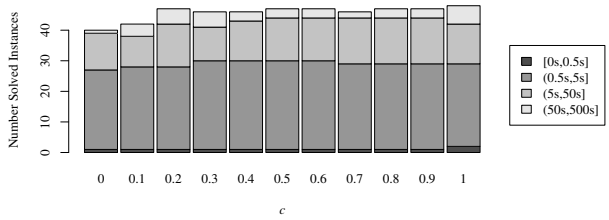


Figure 5: Example of a Hybrid Simulation

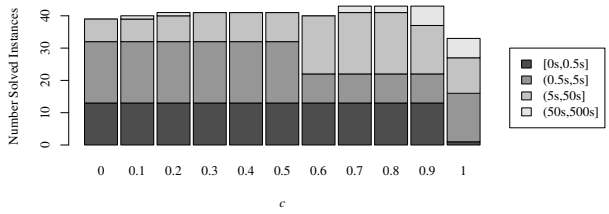
Figure 5 shows the expression DAG of e and the simulation results for all (sub)expressions using ternary, interval, and hybrid simulation. On the one hand, ternary simulation works well for the simulation of subexpression e_3 sustaining the information that the second and third bits of the expression are low whereas interval simulation looses this information entirely. On the other hand, interval simulation performs well for the simulation of subexpression e_2 pertaining the information that the value must be positive while ternary simulation looses all information for this expression. However, only hybrid simulation is able to maintain both pieces of information and to combine them to $\Phi_{e_1}(\hat{a})=[0,1]$. Hence, hybrid simulation yields the correct result that $\Phi_e(\hat{a})=\text{true}$ while both ternary and interval simulation fail due to their individual shortcomings. If e corresponded to $B\wedge u$ in check (1), the expansion to $\hat{a}$ would only succeed with hybrid simulation.

4 Implementation and Experimentation

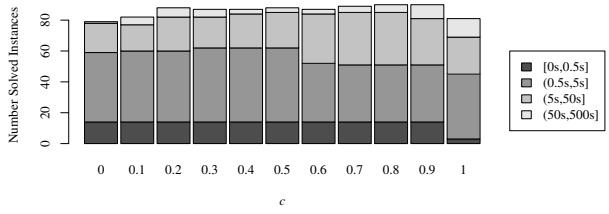
We implemented the proposed hybrid approach. The skeleton of our implementation is similar to the one suggested in [3]. As backend solver, we use the SMT-solver Z3 [9] instead of a SAT-solver. This choice allowed us to implement all of the suggested optimization techniques described in [3], in particular to consider the unsatisfiable core for fast expansion and reusing learned clauses by



(a) Instances extracted from the SV-COMP [7] bitvector benchmark set.



(b) Instances extracted from the INVGEN [8] benchmark set.



(c) All benchmark instances.

Figure 6: Impact of c on Number of Solved Benchmark Instances

incremental solving with retractable assertions (hot solving). To generalize proof obligations, we use hybrid simulation as discussed in Section 3.3.

For experimentation, we compiled a set of roughly 100 benchmark problems derived from the regression test suite of INVGEN [8] and the bitvector set of the SV-COMP software competition benchmarks [7]. In all model checking problems, the property holds, i.e. no bad states are reachable from the initial states.

4.1 Impact of Parameter c on Quality of Results

As discussed in Section 3, the choice of a specific kind of ARU is important for the success of the QF_BV PDR algorithm. In the proposed hybrid algorithm with polytopes and Boolean cubes, parameter c controls the probability that a point in the state space is specialized to a Boolean cube. The point is specialized to a polytope with probability $1-c$.

Figure 6 documents the impact of parameter c on the efficacy of the overall algorithm. The extreme cases $c = 1$ and $c = 0$ are the configurations of the algorithm with exclusive use of Boolean cubes (corresponding to the approach presented in [3]) and polytopes (corresponding to the approach presented in [1]), respectively. Figure 6a shows the results for a subset of the entire set of instances derived from the bitvector set of the SV-COMP [7] benchmarks. On this subset, the algorithm with Boolean cubes only performs the best. Figure 6b displays the same plot with instances derived from the INVGEN [8] benchmark set. On this subset, the configuration with Boolean cubes only performs the worst. The varied result can be explained by the characteristics of the benchmark sets. On the one hand, the benchmarks derived from the SV-COMP benchmarks focus on bit-level characteristics. For instance, the inductive invariants required to solve these problems often specify a specific bit having a specific value or two bits having the same value. On the other hand, in the INVGEN benchmarks, bitvectors are usually interpreted as integers and the relevant inductive invariants often represent linear relations between different variables using this interpretation. In this setting, the polytope interpretation is particularly suitable. In Figure 6c, we illustrate the impact of c for the entire set of model checking instances. Here, the algorithm performs similarly for all configurations but the two pure versions (only Boolean cubes or only polytopes) for which the algorithm perform worse. This validates the theoretic discussion in 3 in so far as that the specific value for c does not matter much in practice unless c is chosen at the extreme values.

4.2 Impact of Simulation Type on Quality of Results

In our second experiment, we investigated the impact of the simulation type on the performance of the algorithm. To this end, we attempt to solve the set of model checking instances with ternary simulation, interval simulation, and hybrid simulation.

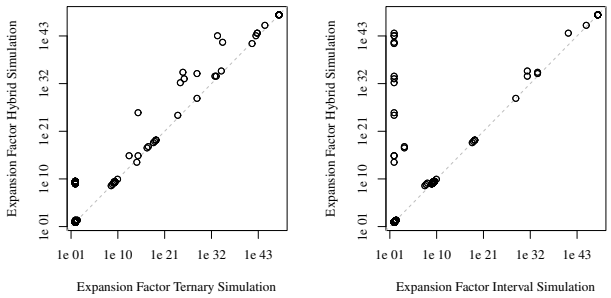


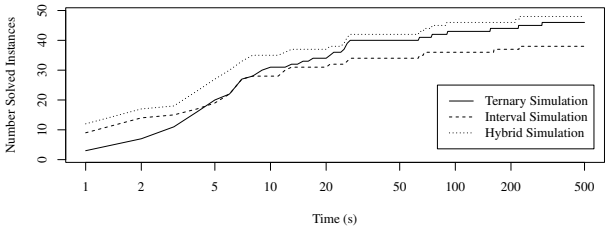
Figure 7: Impact of Simulation Type on Expansion Factor

Figure 7 shows the *expansion factor*, the quotient of the volumes of ARUs after and before the expansion move, with ternary simulation versus hybrid simulation and with interval simulation versus hybrid simulation, respectively, accumulated for several individual expansion attempts. As can be seen, hybrid simulation outperforms the pure simulation types in many cases, often by several orders of magnitude.

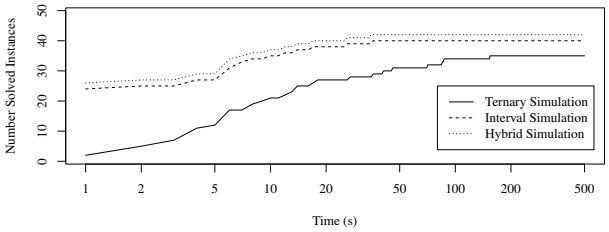
Figure 8 illustrates how this increase in simulation efficacy impacts the overall performance of the model checker when run with $c = 0.5$. As in Subsection 4.1, we show the result both for the complete set of benchmarks and for the two subsets separately. All plots suggest that hybrid simulation performs best. Interestingly, for the model checking instances derived from the SV-COMP benchmarks, ternary simulation outperforms interval simulation. For the benchmarks originating from the INVGEN benchmarks, one would draw the converse conclusion. Similar as discussed in Subsection 4.1, the different observations can be explained by the focus on logic operations in the first subset of benchmarks rather than on arithmetic operations which dominate in the second subset.

4.3 Performance Comparison vs ABC PDR

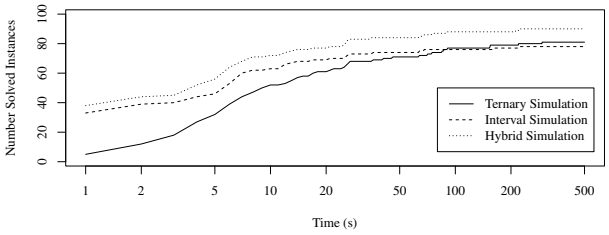
In the last experiment, we compare the performance of our generalized PDR algorithm with the original Boolean version contained in the logic synthesis and verification tool ABC [10]. To this end, we translated the benchmark model checking instances into BTOR [11], a format for specifying word-level model checking problems, and used the tool SYNTHEBTOR, which is part of the distribution of the BOOLECTOR SMT-solver [12], to bit-blast



(a) Instances extracted from the SV-COMP [7] bitvector benchmark set.



(b) Instances extracted from the INVGEN [8] benchmark set.



(c) All benchmark instances.

Figure 8: Impact of Simulation Type on Number of Solved Benchmark Instances

the problems into AIGER [13], a standard format for specifying hardware-model checking problems supported by ABC.

Figure 9 shows a comparison of running times to solve the given benchmark model checking instances with the presented QF_BV generalization of PDR versus the Boolean PDR model checker in ABC. For roughly 95% of the benchmarks, the presented QF_BV generalization outperforms the Boolean PDR algorithm when run on the bit-blasted versions of the benchmarks, often by more than an order of magnitude. This demonstrates that the QF_BV algorithm is able to use the additional information encoded in the word-level formulation of the model checking instances to speed up the solving. In the other roughly 5% of the benchmarks, the inductive invariant required to solve the model checking instance can simply be represented as a set of Boolean cubes. In this case, the generalized algorithm has no conceptional advantage over the Boolean algorithm and attempting to solve the model checking instance using polytopes is not purposeful, hence one cannot expect the generalized algorithm to outperform the original one.

5 Conclusions and Future Work

PDR originated as an efficient and effective algorithm for solving hardware model checking problems and its excellent algorithmic properties motivate research to find generalizations for richer logics. In this paper, we presented an extension to the generalization of PDR to the theory QF_BV with polytopes presented in [1]. Instead of only using polytopes, we propose a hybrid approach with Boolean cubes and polytopes as ARUs that overcomes some of the limitations associated with either homogeneous approach. At the

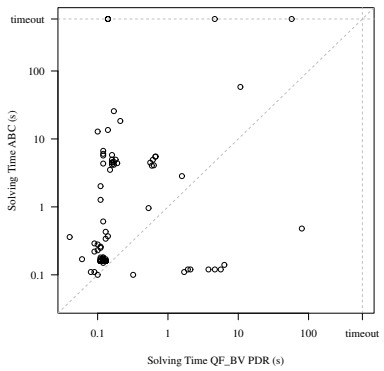


Figure 9: Comparison of QOR of QF_BV PDR vs ABC PDR.

core of the hybrid approach rest (1) a probabilistic specialization of proof obligations after being found by the theory solver and (2) hybrid simulation for the expansion of proof obligations. We implemented the novel algorithm and reported experimental results that show that the PDR algorithm with hybrid ARUs is dominating both homogeneous alternatives on a set of benchmarks extracted from program verification benchmarks.

As future work, it would be interesting to investigate the potential of adding other types of ARUs to the algorithm.

References

[1] T. Welp and A. Kuehlmann, "QF_BV Model Checking with Property Directed Reachability," in *Proc. of the 16th Int'l Conf. on Design, Automation and Test in Europe*, DATE '13, (Washington, DC, USA), IEEE Computer Society, 2013.

[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] 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.

[5] R. Kindermann, T. Junttila, and I. Niemelä, "SMT-based Induction Methods for Timed Systems," in *Proc. of the 10th Int'l Conf. on Formal Modeling and Analysis of Timed Systems*, FORMATS'12, (Berlin, Heidelberg), pp. 171–187, Springer-Verlag, 2012.

[6] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac, "Incremental, Inductive Coverability," in *Proc. of Computer Aided Verification*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 158–173, Springer-Verlag, 2013.

[7] D. Beyer, "Competition on Software Verification," in *Proc. of the 18th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, (Berlin, Heidelberg), pp. 504–524, Springer-Verlag, 2012.

[8] 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.

[9] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[10] 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, Springer-Verlag, 2010.

[11] 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.

[12] 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.

[13] A. Biere, "The AIGER And-Inverter Graph format." http://fmv.jku.at/aiger, 2006.