Approximating Predicate Images for Bit-Vector Logic[⋆]

Daniel Kroening[1] and Natasha Sharygina[2]

[1] Computer Systems Institute, ETH Zürich
[2] University of Lugano, Switzerland

Abstract. Predicate abstraction refinement is a successful technique for verifying large ANSI-C programs. However, computing the image of the predicates with respect to the transition relation is computationally expensive. Recent results have shown that predicate images can be computed by transforming a *proof* of a formula over integers into a Boolean formula that is satisfiable if and only if the original formula is satisfiable. However, the existing algorithms compute the closure of the proof rules that are used to axiomatize the logic, and thus, rely on the fact that the set of axioms is small. They are therefore limited to logics of low complexity, such as difference logic.

We describe a proof-based algorithm that computes an over-approximation of the predicate image but in turn allows a rich set of axioms. The algorithm can be used to compute images of predicates using a combination of bit-vector logic, the theory of arrays, and pointer arithmetic. The proof-based approach can also be used to refine the image. We quantify the performance of the algorithm in comparison with a Das/Dill-like greedy incremental refinement of the image and a proof-based incremental refinement.

1 Introduction

In the hardware industry, formal verification is well established. Introduced in 1981, *Model Checking* [1, 2] is one of the most commonly used formal verification techniques in a commercial setting. However, it suffers from the state-space explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [3].

A principal method for addressing the state-space explosion problem is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

Predicate abstraction [4, 5] is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Verification of a software system with predicate abstraction consists of constructing and evaluating a finite-state system that is an abstraction of the original system with respect to a set of predicates.

[⋆] Ideas of this paper first appeared as a position paper at "Verified Software: Theories, Tools, Experiments", an international conference of Working Group 2.3 (Programming Methodology) of the International Federation for Information Processing.

The abstraction refinement process using predicate abstraction has been promoted by the success of the SLAM [6, 7, 8, 9, 10, 11, 12] project at Microsoft Research. One starts with a coarse abstraction of the program. If the property holds on the abstract model, we can conclude that the property holds on the original model as well. If the abstract model contains an error trace, the algorithm attempts to simulate this error trace on the original model. If this succeeds, the error trace is reported to the user.

If it is found that the error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found or the simulation succeeds. The actual steps of the loop follow the *abstract-verify-refine* paradigm [13] and depend on the abstraction and refinement techniques used [14].

A main task of the refinement loop is to compute an abstract model $\hat{M}$ from the concrete model M given a set of n predicates $\Pi = \{\pi_1, \ldots, \pi_n\}$. An abstract state $\hat{x}$ is a valuation of the n predicates. Most algorithms that aim at verifying safety properties compute an *existential abstraction* [15], i.e., any concrete transition in M has a corresponding abstract transition in the transition relation $\hat{R}$ of $\hat{M}$. Formally, the abstract transition relation $\hat{R}$ is the image of the current state vector $\hat{x}$ and next state vector $\hat{x}'$ under the concrete transition relation R of M.

Computationally, this corresponds to an existential quantification of the concrete state vectors x and x'. This computation, if done in a precise manner, is very expensive and typically exponential in the number of predicates n. All existing tools, with the exception of MAGIC, therefore compute over-approximations $\hat{R}' \supseteq \hat{R}$. Computing such an over-approximation can be substantially faster than computing the exact image. This is a safe and sound technique if the goal is to show safety properties, as any safety property that holds on $\hat{M}$ also holds on M.

However, the over-approximation in $\hat{M}$ may result in additional spurious counterexamples, which are costly to eliminate. There therefore exists a trade-off between the cost of computing the initial abstraction and the cost related to successive refinements. A wide range of options exists between the two extremes of a) computing the precise image and b) using $\hat{R}'(\hat{x}, \hat{x}') = \mathsf{true}$ as initial abstraction.

Most existing approaches that compute or refine predicate images are based on decision procedures for the respective logic. In contrast to that, the authors of [16] generate a generic *proof* that the transition $\hat{x}, \hat{x}'$ does not exist. They then extract a Boolean formula from the proof steps. This formula is satisfiable if and only if the transition from $\hat{x}$ to $\hat{x}'$ exists in the concrete model. This Boolean formula is then used as the abstract transition relation. For equality and difference logic, the approach is shown to be polynomial instead of exponential. Results on a more expressive logic, e.g., full linear arithmetic, are not reported.

Most program analysis tools use theories for arithmetic over unbounded integers or even the reals to reason about the program variables. As motivated in [17], these theories are a poor fit for program analysis, especially when applied to low-level software. Programs in languages such as Java, C or C++ require reasoning for bounded-width bit-vector arithmetic that takes overflow into account, and allows bit-wise operators.

We proposed the use of propositional SAT-solvers as a reasoning engine for the verification of low-level software in [18]. The astonishing progress SAT solvers made in the past few years is the enabling technology for this approach. As in Bounded Model Checking (BMC), the arithmetic operators in the formula are replaced by corresponding circuits. The resulting net-list is converted into CNF and passed to a propositional SAT solver. This allows supporting all operators as defined in the ANSI-C standard.

We report experimental results that quantify the impact of replacing ZAPATO, a decision procedure for integers, with Cogent, a decision procedure built using a SAT solver [17]: The increased precision of Cogent improves the performance of SLAM, while the support for bit-level operators resulted in the discovery of a previously unknown bug in a Windows device driver.

The disadvantage of such a bit-level representation of arithmetic operators is that the variables are split into individual bits and the word-level information is lost. For example, encoding an addition in propositional logic results in one XOR per bit, which are chained together through the carry bit. It is known that such XOR chains can result in very hard SAT instances. As a result, there are many programs (and circuits) that cannot be verified by means of a bit-level SAT solver. This is a justification for using a solver for linear arithmetic for program verification, as the reasoning is done at the word-level, and not at the bit-level.

Contribution. This paper makes two contributions.

1. We present a word-level algorithm for approximating predicate images in bit-vector logic. The algorithm is based on the approach in [16]. In contrast to [16], we compute an over-approximation instead of the precise image. This allows us to support a rich logic, as the size of the formula that is generated no longer explodes as the number of proof rules grows. The implementation reported in [16] is limited to difference predicates. In contrast to that we implement combined theories for bit-vector-, array-, and pointer-logic, including non-linear arithmetic. In contrast to [16], we also support transition relations with a non-trivial propositional structure.
2. The Boolean formulas obtained from proof trees contain fresh Boolean variables. These variables have to be quantified in order to obtain a formula over the predicates. The implementation reported in [16] is enumerating the cubes of a BDD for this task, whereas we are integrating this step into the model checker used for the abstract model.

We present experimental results on software model checking benchmarks that show that the new algorithm outperforms a predicate abstraction refinement loop that uses proof-based refinement of transitions.

Related Work. Abstract interpretation [19] is a very general framework to reason about transition systems. ASTRÉE implements static program analysis [20] using abstract interpretation and widening. It automatically refines abstractions of programs in order to prove the specification. However, if the proof fails, no simulation step is attempted, and thus, the algorithm may generate false alarms.

MAGIC [21] implements predicate abstraction and computes the exact image. The individual transitions $\hat{x}, \hat{x}'$ are enumerated and checked individually using Simplify [22]. Lahiri et al. [23] use SAT-based existential quantification taken from [18] to compute the exact image. The quantification is performed over reductions from linear arithmetic over integers to propositional logic computed using UCLID.

In the SLAM framework, the abstract model is computed by the C2BP component [7]. It enumerates Boolean combinations of a bounded number of current state predicates in order to infer constraints on the next state. C2BP has been replaced by FASTABS, which computes faster, but also more coarse abstractions. In order to address the spurious traces introduced this way, SLAM uses a component called CONSTRAIN [9]. CONSTRAIN uses the decision procedure ZAPATO [24] in order to decide if a given abstract transition is spurious or not. ZAPATO implements a fragment of linear arithmetic over integers.

A completely demand-driven way of constructing $\hat{M}$ was proposed by Das and Dill [25]: starting with no restrictions on abstract transitions, the spurious abstract transitions are removed following the counterexamples produced by the model checker. A similar approach is implemented in BLAST [26]: initially, BLAST computes an abstraction based on the Cartesian product, which is refined subsequently. This refinement is done using Craig interpolants in the current version of BLAST [27].

The first efficient proof-based reduction from integer and real valued linear arithmetic to propositional logic was introduced by Strichman [28]. The proof is generated using Fourier-Motzkin variable elimination for the reals and the Omega test for the integers. These algorithms come with various heuristics to guide the proof, a fact which promises more compact proofs.

Decision procedures for bit-vector arithmetic have been found in tools such as SVC and ICS for years. ICS uses BDDs in order to represent the arithmetic operators, whereas SVC is based on a computation of a canonizer and a solver [29]. SVC has been superseded by CVC, and then CVC-Lite [30], which uses a propositional SAT-solver to decide satisfiability of a circuit-based translation of the bit-vector formula.

The related work on bit-vector decision procedures is mostly in the hardware verification domain. Wedler et al. normalize bit-vector formulas in order to simplify the generated SAT instance in [31]. Word-level reasoning using a decision procedure such as the Omega test or the like is typically not employed. One exception is Brinkmann and Drechsler [32], who use an encoding of linear bit-vector arithmetic into ILP in order to decide properties of circuit data-paths given at the RT-level. The Omega test is used as a decision procedure for the ILP instance. However, [32] only aims at the data-paths, and thus, does not allow a Boolean part within the original formula. This is mended by [33] using a lazy encoding with a modified DPLL search.

Outline. In Section 2, we provide background information about lazy and eager encodings of decision problems. We describe how to use proof encodings as over-approximations of abstractions in Section 3. Experimental results are reported in Section 4.

2 Background

2.1 Bit-Vector Arithmetic

The subset of bit-vector arithmetic we consider is defined by the language L_B according to the following grammar:

$$\begin{aligned}
\mathit{formula} &: \mathit{formula} \vee \mathit{formula} \mid \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \\
\mathit{rel} &: = \ \mid \neq \mid \leq \mid \geq \mid < \mid > \\
\mathit{term} &: \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \ \sim \mathit{term} \mid \mathit{constant} \mid \mathit{atom} ?\, \mathit{term} : \mathit{term} \\
\mathit{op} &: \oplus \mid \ominus \mid \otimes \mid \oslash \mid << \mid >> \mid \& \mid \ | \ \mid \hat{}
\end{aligned}$$

With each expression, we associate a type. The type is the width of the expression in bits and whether it is signed (two's complement encoding) or unsigned (binary encoding). Assigning semantics to this language is straight-forward, e.g., as done in [32].

We do not consider bit-extraction and concatenation operators, as they are not offered by ANSI-C. However, adding these operators as part of the bit-wise operators is a simple extension. We use the ANSI-C symbols to denote the bit-wise operators, e.g., & denotes bit-wise AND, while ^ denotes bit-wise XOR. The trinary operator $c ? a : b$ is a case-split: the operator evaluates to a if c holds, and to b otherwise.

We use $\oplus$ to distinguish addition on bit-vectors with modular arithmetic from addition on unbounded integers. Note that the relational operators $>, <, \leq, \geq$, the multiplicative operators $\otimes, \oslash$ and the right-shift operator depend on whether an unsigned, binary encoding or a two's complement encoding is used. We assume that the type of the expression is clear from the context.

Following the notation in [32], we add an index to the operator and operands in order to denote the bit-width. As an example, $a_{[32]} \otimes_{[32]} b_{[32]}$ denotes the 32-bit multiplication of a and b. Both the result and the operands are 32 bits wide, the remaining 32 bits of the result are discarded.

Example 1. As a motivating example, the following formula obviously holds on the integers:

$$(x - y > 0) \iff (x > y) \tag{1}$$

However, if x and y are interpreted as bit-vectors, this equivalence no longer holds, due to possible overflow on the subtraction operation.

Definition 1. *Let ϕ^B denote a formula. The set of all atoms in ϕ^B that are not Boolean identifiers is denoted by $\mathcal{A}(\phi^B)$. The i-th distinct atom in ϕ^B is denoted by $\mathcal{A}_i(\phi^B)$. The* Propositional Skeleton *ϕ_{sk} of a bit-vector formula ϕ^B is obtained by replacing all atoms $a \in \mathcal{A}(\phi^B)$ by fresh Boolean identifiers $e_1, \ldots, e_\nu$, where $\nu = |\mathcal{A}(\phi^B)|$.*

As an example, the propositional skeleton of $\phi^B = (x = y) \wedge ((a \oplus b = c) \vee (x \neq y))$ is $e_1 \wedge (e_2 \vee \neg e_1)$, and $\mathcal{A}(\phi^B) = \{x = y, a \oplus b = c\}$.

We denote the vector of the variables $E = \{e_1, \ldots, e_\nu\}$ by $\overline{e}$. Furthermore, let $\psi(a, p)$ denote the atom a with polarity p:

$$\psi(a, p) := \begin{cases} a & : p \\ \neg a & : \text{otherwise} \end{cases}. \tag{2}$$

2.2 Encoding Decision Problems into Propositional Logic

Lazy vs. Eager Encodings. There are two basic ways to compute an encoding of a decision problem ϕ into propositional logic. In both cases, the propositional part ϕ_{sk}

of the formula is converted into CNF first. Linear-time algorithms for computing CNF for ϕ_{sk} are well-known [34]. The algorithms differ in how the non-propositional part is handled.

The vector of variables $\overline{e} : \mathcal{A}(\phi) \longrightarrow \{\mathsf{true}, \mathsf{false}\}$ as defined above denotes a truth assignment to the atoms in ϕ. Let $\varPsi_{\mathcal{A}(\phi)}(\overline{e})$ denote the conjunction of the atoms $\mathcal{A}(\phi)_i$, where the atom number i is in the polarity given by e_i:

$$\varPsi_{\mathcal{A}(\phi)}(\overline{e}) := \bigwedge_{i=1}^{\nu} \psi(\mathcal{A}_i(\phi), e_i) \tag{3}$$

An *Eager Encoding* considers all possible truth assignments $\overline{e}$ before invoking the SAT solver, and computes a Boolean constraint $\phi_E(\overline{e})$ such that

$$\phi_E(\overline{e}) \iff \varPsi_{\mathcal{A}(\phi)}(\overline{e}) \tag{4}$$

The number of cases considered while building ϕ_E can often be dramatically reduced by exploiting the polarity information of the atoms, i.e., whether $\mathcal{A}_i(\phi)$ appears in negated form or without negation in the negation normal form (NNF) of ϕ. After computing ϕ_E, ϕ_E is conjoined with ϕ_{sk}, and passed to a SAT solver. A prominent example of a decision procedure implemented using an eager encoding is UCLID [35].

A *Lazy Encoding* means that a series of encodings ϕ_L^1, ϕ_L^2 and so on with $\phi \implies \phi_L^i$ is built. Most tools implementing a lazy encoding start off with $\phi_L^1 = \phi_{sk}$. In each iteration, ϕ_L^i is passed to the SAT solver. If the SAT solver determines ϕ_L^i to be unsatisfiable, so is ϕ. If the SAT solver determines ϕ_L^i to be satisfiable, it also provides a satisfying assignment, and thus, an assignment $\overline{e}^i$ to $\mathcal{A}(\phi)$.

The algorithm proceeds by checking if $\varPsi_{\mathcal{A}\phi}(\overline{e}^i)$ is satisfiable. If so, ϕ is satisfiable, and the algorithm terminates. If not so, a subset of the atoms $\mathcal{A}' \subseteq \mathcal{A}(\phi)$ is determined that is already unsatisfiable under $\overline{e}^i$. The algorithm builds a *blocking clause* b, which prohibits this truth assignment to $\mathcal{A}'$. The next encoding ϕ_L^{i+1} is $\phi_L^i \wedge b$. Since the formula becomes only stronger, the algorithm can be tightly integrated into one SAT-solver run, which preserves the learning done in prior iterations.

Among many others, CVC-Lite [30] implements a lazy encoding of integer linear arithmetic. The decision problem for the conjunction $\varPsi_{\mathcal{A}\phi}(\overline{e}^i)$ is solved using the Omega test.

2.3 Encodings from Proofs

A proof is a sequence of transformations of facts. The transformations follow specific rules, i.e., proof rules, which are usually derived from an axiomatization of the logic at hand. A proof of a formula ϕ in a particular logic can be used to obtain another formula ϕ_P in propositional logic that is valid if and only if the original formula is valid, i.e., $\phi \iff \phi_P$. Let $\mathscr{F}$ denote the set of facts used in the proof.

Given a proof of ϕ, a propositional encoding of ϕ can be obtained as follows:

1. Assign a fresh propositional variable v_f to each fact $f \in \mathscr{F}$ that occurs anywhere in the proof.

2. For each proof step i, generate a constraint c_i that captures the dependencies between the facts. As an example, the derivation
$$\frac{A, B}{C}$$
with variables v_A, v_B, v_C for the facts A, B, and C generates the constraint $(v_A \wedge v_B) \longrightarrow v_C$.
3. The formula ϕ_P is obtained by conjoining the constraints:
$$\phi_P := \bigwedge_i c_i$$

However, the generation of such a proof is often difficult to begin with. In particular, it often suffers from a blowup due to case-splitting caused by the Boolean structure present in ϕ. This is addressed by a technique introduced by Strichman in [28]. His paper describes an eager encoding of linear arithmetic on both real numbers and integers into propositional logic using the Fourier-Motzkin transformation for the reals and the Omega-Test [36] for the integers. The idea of [28] is applicable to any proof-generating decision-procedure:

– All atoms $\mathcal{A}(\phi)$ are passed to the prover *completely disregarding the Boolean structure* of ϕ.
– For facts f that are also atoms assign $v_f := e_f$.
– The prover must be modified to obtain *all* possible proofs, i.e., must not terminate even if the empty clause is resolved.

Since the formula that is passed to the prover does not contain any propositional structure, obtaining a proof is considerably simplified. The formula ϕ_P obtained from the proof as described above is then conjoined with the propositional skeleton ϕ_{sk}. The conjunction of both is equi-satisfiable with ϕ. As $\phi_P \wedge \phi_{sk}$ is purely propositional, it can be solved by an efficient propositional SAT-solver.

3 Computing Predicate Images

3.1 Existential Abstraction

Let S denote the set of concrete states, and $R(x, x')$ denote the concrete transition relation. As an example, consider the basic block

```
i++;
j=i;
```

We use $x.v$ to denote the value of the variable v in state x. The transition relation corresponding to this basic block is then $x'.i = x.i + 1 \wedge x'.j = x'.i$.

Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ denote the set of predicates. The abstraction function $\alpha(x)$ maps a concrete state $x \in S$ to an abstract state $\hat{x} \in \{\mathsf{true}, \mathsf{false}\}^n$:
$$\alpha(x) := (\pi_1(x), \ldots, \pi_n(x))$$

Definition 2 (Abstract Transition Relation). *The abstract model can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from x to x' in the concrete model and x is abstracted to $\hat{x}$ and x' is abstracted to $\hat{x}'$. We denote abstract transition relation by $\hat{R}$:*

$$\hat{R} := \{(\hat{x}, \hat{x}') \,|\, \exists x, x' \in S : R(x, x') \wedge \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\}$$

$\hat{R}$ is also called the image of the predicates Π over R. In [23], $\hat{R}$ is computed following the definition above by means of SAT or BDD-based quantification. Due to the quantification over the concrete states this corresponds to an all-SAT instance. Solving such instances is usually exponential in n.

3.2 Predicate Images from Proofs

As an alternative, $\hat{R}$ can be computed using a generic *proof of validity* of the following formula:

$$R(x, x') \wedge \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'$$

Within this formula, only R contains propositional operators, as the predicates in α are assumed to be atomic. The computation of ϕ_{sk} therefore only has to take the propositional structure of R into account. In case of software, the propositional structure of R is typically trivial, as the abstraction is performed for each basic block separately. Thus, the facts (atoms) given to the prover are:

1. All the predicates evaluated over state x, i.e., $\pi_i(x)$,
2. all the predicates evaluated over state x', i.e., $\pi_i(x')$,
3. the atoms in the transition relation $R(x, x')$.

We then obtain ϕ_P as described in section 2.3. Both ϕ_P and ϕ_{sk} contain fresh propositional variables for the atoms $\mathcal{A}(R)$ in R, for the predicates Π over x and x', and for the facts $f \in \mathscr{F}$ found during the derivation. Let V_R denote the set of propositional variables corresponding to atoms in R that are not predicates, and let V_F denote the set of propositional variables corresponding to facts $f \in \mathscr{F}$ that are not predicates.

The propositional variables that do not correspond to predicates are quantified existentially to obtain the predicate image. Let $\overline{v}_R$ denote the vector of variables in V_R, let $\overline{v}_F$ denote the vector of variables in V_F, and let $\mu_R = |V_R|$ and $\mu_F = |V_F|$ denote the number of such variables.

$$\begin{aligned}\hat{R} := \{(\hat{x}, \hat{x}') \,|\, &\exists \overline{v}_R \in \{0,1\}^{\mu_R}, \overline{v}_F \in \{0,1\}^{\mu_F} : \\ &\phi_{sk}(\hat{x}, \hat{x}', v_R) \wedge \phi_P(\hat{x}, \hat{x}', v_F)\}\end{aligned} \tag{5}$$

Thus, we replace the existential quantification of concrete program variables $x, x' \in S^2$ by an existential quantification of $\mu_R + \mu_F$ Boolean variables. The authors of [23, 37] report experiments in which this quantification is actually performed by means of either BDDs or the SAT-engine of [18].

The authors of [16] use BDDs to obtain all cubes over the variables in V_F, and then enumerate these cubes. This operation is again worst-case exponential. The next two sections describe how to overcome the limitations of the proof-based predicate image computation.

3.3 Quantification as Part of the Abstract Model

Instead of performing the quantification in equation 5 upfront, we propose to perform this step inside the model checker for the abstract model. When performing the fixed-point iteration, a symbolic model checker computes an image of the given transition relation, and usually contains algorithms that are well optimized for this task. Furthermore, the image only has to be computed with respect to the set of reachable states, whereas performing the quantification upfront has to consider all possible state pairs $(\hat{x}, \hat{x}')$.

It is important to point out that most model checkers for abstract models do not require modifications for this purpose. As an example, consider the following abstract transition relation over state variables x, y, and their next-state versions x' and y':

$$\exists v_1 \in \{0,1\}.(x' \iff v_1) \wedge (v_1 \iff x \vee y) \wedge (y' \iff v_1) \tag{6}$$

This abstract transition relation can be translated into a closed form by enumerating the values of v_1, as done in [16]:

$$\begin{array}{c}(\neg x' \wedge \neg(x \vee y) \wedge \neg y') \vee \\ (x' \wedge (x \vee y) \wedge y')\end{array} \tag{7}$$

However, if we add v_1 as a state variable to the abstract model, we can use the following equivalent SMV code without having to resort to existential quantification[1]:

```
TRANS next(x)=next(v1) &
      next(v1)=(x|y) &
      next(y)=next(v1)
```

Integration in Boppo. The addition of state variables comes at an expense. Since these variables never have direct constraints that relate their current state to their next-state value, it is not actually necessary to store any representation of their values. BOPPO [38] is a symbolic model checker for Boolean programs, i.e., abstract models of C programs. It uses symbolic simulation for checking reachability. We have modified BOPPO to allow the definition of variables that can be used in `constrain` clauses, but are not part of the state vector and are therefore disregarded during the fixed-point detection. Our experiments indicate that the additional variables do not noticeably increase the run-time of BOPPO.

3.4 Predicate Images in Bit-Vector Logic

As motivated above, reasoning for integers is a bad fit for system-level software. We would therefore like a proof-based method for a bit-vector logic. The main challenge is that any axiomatization for a reasonably rich bit-vector logic permits too many ways of proving the same fact, as the procedure as described above relies on enumerating *all* proofs.

[1] We use `next(v1)` instead of `v1` in order to avoid a transition relation that is not total.

Even if great care is taken to obtain a small set of axioms, the number of proofs is still too large. Furthermore, the proofs include derivations that are based on reasoning about single bits of the vectors involved, resulting in a flattening of the formula, which resembles the circuit-based models used for encodings of bit-vector logic into propositional logic.

We therefore sacrifice precision in order to be able to reason about bit-vectors, and compute an over-approximation of $\hat{R}$. This is a commonly applied technique, e.g., used by SLAM and BLAST. If this over-approximation results in a spurious transition, it can be refined by any of the existing refinement methods, e.g., based on UNSAT cores as in [39] or based on interpolants as in [27].

The over-approximation of $\hat{R}$ is obtained as follows: Instead of aiming at a minimalistic set of axioms, we aim at the richest possible set of axioms. This permits proofs (or refutations) of facts with very few proof steps. It also allows to support a very rich logic, which is bit-vector logic including bit-wise shifts, extraction, concatenation, non-linear arithmetic, the theory of arrays, and pointer logic permitting pointer arithmetic in our case.

Definition 3. *The* derivation depth $d(f)$ *of a fact* $f \in \mathscr{F}$ *is defined recursively as follows:*

- *Axioms and the facts given as input have depth zero.*
- *Any new fact* f *derived from a set of existing facts* $f_1, \ldots, f_k$ *has depth* $d(f) = \max\{d(f_1), \ldots, d(f_k)\} + 1$.

In order to avoid that $d(f)$ depends on the order the facts are derived, we generate the facts in a BFS manner, i.e., new facts are derived preferably from existing facts with a low number of derivation steps.

Definition 4. *Given a maximum depth* δ, *a* depth-bounded *derivation tree is a set of derivations of facts* f_i *such that* $d(f_i) \leq \delta$.

Note that a depth-bounded derivation tree not necessarily constitutes a proof or refutation of any of the facts that are given as input, as the shortest proof or refutation could require more than δ steps.

Claim. Let ϕ_P^δ denote the formula corresponding to a derivation tree with maximum depth δ. The formula corresponding to the full unbounded proof tree ϕ_P implies ϕ_P^δ.

Let $\hat{R}^\delta$ denote the transition relation obtained by using ϕ_P^δ instead of ϕ_P. $\hat{R}^\delta$ is an over-approximation of $\hat{R}$, i.e., $\hat{R}(\hat{x}, \hat{x}') \longrightarrow \hat{R}^\delta(\hat{x}, \hat{x}')$, and thus, $\hat{R}^\delta$ is a conservative abstraction for reachability properties.

Example. Assume we have, among others, the following derivation rules:

$$\frac{}{(a|b)\&b == b} \quad (8) \qquad\qquad \frac{b\&c == 0}{(a|b)\&c == a\&c} \quad (9)$$

The predicates we consider are $\pi_1 \iff (x\&1 = 0)$ and $\pi_2 \iff (x\&2 = 0)$, and the statement to be executed is:

```
x|=2;
```

$$\begin{tikzcd}
x'\&2=0 \arrow[dr] & & x'=x|2 \arrow[dl] & \text{Rule 8} \arrow[d] \\
& (x|2)\&2=0 \arrow[dr] & & (x|2)\&2=2 \arrow[dl] \\
& & \perp &
\end{tikzcd}
\qquad
\begin{tikzcd}
\pi_2' \arrow[dr] & & \mathsf{T} \arrow[dl] & & \mathsf{T} \arrow[d] \\
& v_1 \arrow[drr] & & & \mathsf{T} \arrow[dl] \\
& & & \mathsf{F} &
\end{tikzcd}$$

Fig. 1. Derivation of constraints for π_2'

$$\begin{tikzcd}
x'\&1=0 \arrow[dr] & & x'=x|2 \arrow[dl] & \text{Rule 9} \arrow[d] \\
& (x|2)\&1=0 \arrow[dr] & & (x|2)\&1=x\&1 \arrow[dl] \\
& & x\&1=0 &
\end{tikzcd}
\qquad
\begin{tikzcd}
\pi_1' \arrow[dr] & & \mathsf{T} \arrow[dl] & & \mathsf{T} \arrow[d] \\
& v_2 \arrow[drr] & & & \mathsf{T} \arrow[dl] \\
& & & \pi_1 &
\end{tikzcd}$$

Fig. 2. Derivation of constraints for π_1'

The facts passed to the prover are $x\&1 = 0$, $x\&2 = 0$, $x'\&1 = 0$, $x'\&2 = 0$, and $x' = x|2$. Figure 1 shows a derivation on the left hand side and on the right hand side the same derivation tree in which the atoms are replaced by their propositional variables. The derivation results in the constraint $(\pi_2' \longrightarrow v_1) \wedge (v_1 \longrightarrow \mathsf{F})$, which is equivalent to $\neg\pi_2'$. Figure 2 shows a derivation that ends in an existing atom π_1 rather than F. The constraint generated is equivalent to $\pi_1' \longrightarrow \pi_1$.

We collected a set of over 100 (highly redundant) proof rules for bit-vector arithmetic, pointer arithmetic, and the theory of arrays; we typically limit the depth of the proofs to 3 derivation steps.

4 Experimental Results

We implemented the proof-based predicate image approximation as described above in SATABS [40]. SATABS uses an abstraction refinement loop to prove reachability properties of ANSI-C programs. We make our implementation available to other researchers[2] for experimentation. We have three different configurations:

1. The first configuration ("Greedy Ref.") follows the suggestions by Das/Dill [25]: a syntactic heuristic is used to compute the initial image. The image is subsequently refined using a greedy heuristic.

[2] `http://www.inf.ethz.ch/personal/daniekro/satabs/`

Table 1. Summary of results: the column "Max. n" shows the largest number of predicates in any program location. The columns under "Greedy Ref." show the number of iterations, the time spent during refinement, and the total run-time of SATABS using the Greedy heuristic described in [25]. The columns under "Proof Ref." show the results using a proof-based refinement. The columns under "Proof Image+Ref." show the results using the technique proposed in this paper in order to obtain the initial predicate image and to refine the image. A star denotes that the one hour timeout was exceeded.

			Greedy Ref.			Proof Ref.			Proof Image+Ref.			
Benchmark	Result	Max. n	It.	Ref.	Total	It.	Ref.	Total	It.	Abstr.	Ref.	Total
B1	T	25	85	$\star$	$\star$	180	$\star$	$\star$	2	1.3s	0.2s	2.0s
B2	F	11	51	4.8s	5.9s	46	3.5s	5.2s	8	0.1s	0.8s	1.2s
B3	T	7	14	8.4s	9.5s	15	4.8s	5.3s	6	0.1s	1.9s	2.2s
B4	F	21	102	65.2s	70.2s	78	15.1s	20.9s	20	2.1s	13.6s	18.3s
MD2	T	10	62	58.8s	67.9s	50	19.1s	24.7s	14	0.3s	9.2s	11.1s
B5'	F	81	242	$\star$	$\star$	241	$\star$	$\star$	80	123.1s	843.7s	1112.7s
AP1	F	149	31	$\star$	$\star$	154	$\star$	$\star$	201	210.8s	1532.2s	2102.8s

2. The second configuration ("Proof Ref.") replaces the greedy heuristic proposed in [25] by a proof-based refinement strategy that uses the proof of unsatisfiability to refine the transitions.
3. The third configuration ("Proof Image+Ref.") combines the proof-based refinement with the word-level proof-based initial abstraction as proposed in this paper.

To the best of our knowledge, a comparison of these image approximation heuristics on software given in C has not yet been made; Das/Dill [25] use examples from protocol verification.

For all configurations, we use the modified version of BOPPO [38] as Model Checker for the abstract model. All configurations use the following simulation phase: the path generated by model checker is transformed into SSA (static single assignment) form. The resulting bit-vector formula is translated using a circuit representation and passed to a propositional SAT-solver. We are using Booleforce[3] for our experiments. We also experimented with ZChaff 2003, but Booleforce outperformed ZChaff on all benchmarks.

The refinement phase depends on whether the spurious trace is due to predicate image over-approximation or due to a lack of sufficiently strong predicates. This is determined by the simulation phase. If a transition is found to be spurious due to image over-approximation, the incremental approach described in [25] uses a greedy heuristic to generalize the transition and refine the abstract transition relation. In contrast to that, configuration 2) and 3) use the proof of unsatisfiability to generalize the transition.

If the spurious counterexample is due to insufficient predicates, we use weakest preconditions to compute new predicates. The set of new predicates is limited to those transitions found in the UNSAT core of the SAT instance used for simulation.

The experiments have been performed on an Intel Xenon Processor with a clock frequency of 2.8 GHz running Linux. The results are summarized in table 1. The bench-

[3] A recent SAT-solver based on MiniSAT written by A. Biere.

marks are C programs that require a moderate to large number of predicates per program location (the table shows the largest number of predicates required in any program location). All benchmarks make use of bit-vector and pointer arithmetic, and arrays. The benchmark AP1 is an array-bounds property of the Apache httpd server, which makes heavy use of pointers and pointer arithmetic.

The experiments show that a small additional expense for computing an initial predicate image can reduce the number of iterations required and the time spent for refinement dramatically. The larger the number of predicates, the bigger the benefit of using an initial abstraction usually is. Due to the depth bound of the proofs, the abstraction phase never uses an excessive amount of time.

5 Conclusion

This paper shows two things:

1. When extracting predicate images for abstract models, it is not necessary to perform the existential quantification upfront. It can be performed within the model checker instead. Potentially expensive methods, such as BDD-based enumeration as in [16], can be avoided that way.
2. A rich logic, including bit-vector logic, can be supported if we sacrifice some precision and abort proofs after a small number of steps. The experiments show that we in many cases still right away obtain an abstract model that is strong enough to terminate without many refinement iterations, and thus, often as good as a model computed using the precise image.

Future Work. The algorithm presented here uses the propositional encoding ϕ_{sk} to handle a complex Boolean structure of the transition relation. The transition relation of software programs, when partitioned using a program counter construction, usually only contains very few facts (one per statement of any basic block). As future work, we plan to experiment with the algorithm using larger transition relations, e.g., those of circuits given in Verilog.

We also plan to investigate even richer logics, e.g., non-standard logics such as separation logic [41] in order to reason about dynamic data structures.

Acknowledgment

The authors would like to thank Ofer Strichman for valuable comments, and Armin Biere for his proof-generating SAT solver Booleforce.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Clarke, E.M., Emerson, E.A.: Synthesis of synchronization skeletons for branching time temporal logic. In: Logic of Programs: Workshop. Volume 131 of LNCS. Springer (1981) 52–71

3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98** (1992) 142–170
4. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification (CAV). Volume 1254 of LNCS. Springer (1997) 72–83
5. Colón, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: Computer Aided Verification (CAV). Volume 1427 of LNCS. Springer (1998) 293–304
6. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
7. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Language Design and Implementation (PLDI), ACM (2001) 203–213
8. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
9. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 2988 of LNCS. Springer (2004)
10. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN. Volume 1885 of LNCS. Springer (2000) 113–130
11. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN. Volume 1885 of LNCS. Springer (2000) 113–130
12. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM (2001) 97–103
13. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
14. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification (CAV). Volume 1855 of LNCS. Springer (2000) 154–169
15. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. In: Principles of Programming Languages (POPL), ACM (1992) 343–354
16. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In: Computer Aided Verification (CAV). Volume 3576 of LNCS., Springer (2005) 24–38
17. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Computer Aided Verification (CAV). Volume 3576 of LNCS., Springer (2005)
18. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
19. Cousot, P.: Abstract interpretation. Symposium on Models of Programming Languages and Computation, ACM Computing Surveys **28** (1996) 324–328
20. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: European Symposium on Programming (ESOP). Volume 3444 of LNCS. Springer (2005) 21–30
21. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: Correct Hardware Design and Verification Methods (CHARME). Volume 2860 of LNCS. Springer (2003) 19 – 34
22. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
23. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: Computer-Aided Verification (CAV). Volume 2725 of LNCS. Springer (2003) 141–153

24. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: Computer Aided Verification (CAV). Volume 3114 of LNCS., Springer (2004)
25. Das, S., Dill, D.: Successive approximation of abstract transition relations. In: Logic in Computer Science (LICS). (2001) 51–60
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of programming languages (POPL). (2002) 58–70
27. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: Principles of Programming Languages (POPL), ACM (2004) 232–244
28. Strichman, O.: On solving presburger and linear arithmetic with SAT. In: Formal Methods in Computer-Aided Design (FMCAD). Volume 2517 of LNCS. Springer (2002) 160–170
29. Barret, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: Design Automation Conference (DAC), ACM (1998)
30. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: Computer-Aided Verification. Volume 3114 of LNCS. Springer (2004)
31. Wedler, M., Stoffel, D., Kunz, W.: Normalization at the arithmetic bit level. In: Design Automation Conference (DAC), ACM (2005) 457–462
32. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: VLSI Design, IEEE (2002) 741–746
33. Parthasarathy, G., Iyer, M.K., Cheng, K.T., Wang, L.C.: An efficient finite-domain constraint solver for circuits. In: Design Automation Conference (DAC), ACM (2004) 212–217
34. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Symbolic Computation **2** (1986) 293–304
35. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Computer-Aided Verification. Volume 2404 of LNCS. Springer (2002)
36. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM (1992) 102–114
37. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Verification, Model Checking and Abstract Interpretation (VMCAI). Volume 2937 of LNCS. Springer (2004) 267–281
38. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous Boolean programs. In: SPIN. Volume 3639 of LNCS. Springer (2005) 75–90
39. Jain, H., Kroening, D., Sharygina, N., Clarke, E.: Word level predicate abstraction and refinement for verifying RTL Verilog. In: Design Automation Conference (DAC), ACM (2005) 445–450
40. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 3440 of LNCS. Springer (2005) 570–574
41. Reynolds, J.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS), IEEE (2002) 55–74