Image Computation and Predicate Refinement for RTL Verilog using Word Level Proofs*

Daniel Kroening
ETH Zurich

Natasha Sharygina
University of Lugano

Abstract

Automated abstraction is the enabling technique for model checking large circuits. Predicate Abstraction *is one of the most promising abstraction techniques. It relies on the efficient computation of predicate images and the right choice of predicates. Existing algorithms use a* net-list-level *circuit model for computing predicate images. 1) This paper describes a proof-based algorithm that computes an over-approximation of the predicate image at the* word-level*, and thus, scales to larger circuits. 2) The previous work relies on the computation of the weakest preconditions in order to refine the set of predicates. In contrast to that, we propose to extract predicates from a word-level proof to refine the set of predicates.*

1 Introduction

Model checking [7] is a popular approach to formal verification in the hardware design industry. Algorithms developed to avoid the state space explosion enable the verification of large circuits. One principal method in state space reduction 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 behavior of the system.

In hardware verification, the most commonly used abstraction technique is *localization reduction* [16, 20, 4]. The abstract model is created from the given circuit by removing a large number latches together with the logic required to compute their next state. The removed latches are the ones that are irrelevant to establishing correctness of the particular property.

Localization reduction is a *conservative* over-approximation of the original circuit for reachability properties. This implies that if the abstraction satisfies the property, the property also holds on the original circuit. However, when model checking of the abstraction fails, it may produce a counterexample that does not correspond to any concrete design specification. This is called a *spurious counterexample*. In order to detect if the counterexample is spurious, it is *simulated* on the concrete design. If the counterexample is found to be real, the algorithm terminates. If the abstract counterexample is spurious, *abstraction refinement* has to be performed.

The basic idea of abstraction refinement techniques is to create a new abstract model that contains more detail (e.g., more visible latches) in order to prevent the spurious counterexample. For finite-state models this process is iterated until the property is either proved or disproved. If the refinement is based on the abstract counterexample, the algorithm implements *Counterexample Guided Abstraction Refinement* [16, 2, 5].

The effectiveness of localization reduction is limited if the property actually depends on a large percentage of the latches in the design. In this case, a large part of the original circuit has to be contained in the abstract model, and the verification of the abstract model becomes the bottleneck.

In software verification, the most successful abstraction technique for large systems is *predicate abstraction* [10]. 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.

Predicate abstraction has recently been shown to be effective for hardware verification as well [9, 12]. Traditionally, model-checkers used in the hardware industry operate at the *net-list* level. However, predicate abstraction is only effective if the predicates cover the relationship between multiple latches. This typically requires a *word-level* model given in register transfer language (RTL), e.g., in Verilog. When applying predicate abstraction to circuits, two problems arise:

1) If done precisely, the computation of the abstract transition relation is exponential in the number of predicates. In software, the abstraction is performed separately for each control location, and the number of predicates for a given location is usually small. In hardware, the abstract transition relation has to be computed for the whole circuit at once

*This research is supported by SRC contract 2006-TJ-1539.

due to the high degree of concurrency. Thus, the computation of the abstract model becomes prohibitively expensive.

2) Abstraction refinement is performed by computing new predicates. In software, this is commonly done by computing weakest liberal preconditions. If applied to hardware as described in [12], this approach corresponds to a partial unwinding of the circuit as in BMC. It is ineffective if the property depends on counters.

Contribution This paper addresses the problems above and enables the application of predicate abstraction to larger designs. We propose new techniques for both the abstraction and refinement procedures that are used for word-level abstraction of circuits: 1) proof-based word-level predicate image computation and 2) proof-based word-level predicate refinement for circuits given in Verilog RTL.

Proof-based word-level image computation. In order to build abstract models of large circuits efficiently, we propose to use proof-based predicate image computation for bit-vector logic. It extends the algorithm proposed for software verification in [15] by providing features typical to hardware description (i.e., large Boolean structures in the concrete transition relation). The algorithm scales better both in the number of predicates and the circuit size than the existing approach that is based on all-SAT.

Proof-based word-level refinement. We propose a word-level algorithm for the refinement of the set of predicates based on a word-level proof of unsatisfiability of the simulation instance. In the presence of wide counters, the proof-based refinement results in a faster convergence of a refinement loop (as compared to traditional net-list level techniques) due to better predicates.

To the best of our knowledge, our algorithm is the first to apply proof-based predicate abstraction and proof-based predicate refinement to hardware designs.

The paper is organized as follows. Section 3 provides background on SAT-based predicate abstraction. Techniques for obtaining word-level proofs for bit-vector formulas are given in Section 4. Section 5 describes how to transform such proofs into predicate images. Section 6 describes how to extract word-level predicates from proofs of unsatisfiability for predicate refinement. Experimental results are reported in Section 7.

The formal semantics of the subset of Verilog we handle can be found in a technical report [8].

2 Related Work

Related Work For hardware verification, Clarke et al. [9] introduce a SAT-based technique for predicate abstraction of circuits given in Verilog. The circuit is synthesized and transformed into net-list level. A SAT solver is used to compute the abstraction, which allows to support all bit-level constructs. However, if refinement becomes necessary, only bit-level predicates are introduced.

Andraus et al. [1] present a scheme for automatic abstraction of behavioral RTL Verilog to the CLU language used by the UCLID system [3]. However, the abstractions produced by their approach can be coarse as the semantics of the bit-vector operators are not taken into account when computing the abstraction. Also, no refinement is done when a spurious counterexample is obtained.

3 Predicate Abstraction

Predicate abstraction maps the variables of the concrete model to Boolean variables that correspond to a predicate on the variables in the concrete model [10]. Formally, these predicates are functions that map a concrete state $s \in S$ into a Boolean value. Let $\mathtt{B}=\{\pi_1,\ldots,\pi_k\}$ be the set of predicates. When applying all predicates to a specific concrete state, one obtains a vector of Boolean values, which represents an abstract state $\hat{s} \in \{0,1\}^k$. We denote this abstraction function by $\alpha(s)$.

Predicate abstraction is typically performed using an existential abstraction [6], i.e., the abstract model can make a transition from an abstract state $\hat{s}$ to $\hat{s}'$ iff there is a transition from a corresponding state s to s' in the concrete model and s is abstracted to $\hat{s}$ and s' is abstracted to $\hat{s}'$. Let $R(s,s')$ denote the concrete transition relation. We call the abstract machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\hat{R} \;:=\{(\hat{s},\hat{s}')\,|\,\exists s,s' \in S:\quad R(s,s')\wedge \alpha(s)=\hat{s}\wedge\alpha(s')=\hat{s}'\} \tag{1}$$

The initial state $I(s)$ is abstracted to $\hat{I}(\hat{s})$ as follows:

$$\hat{I}(\hat{s}) \quad := \quad \exists s \in S.\ (\alpha(s)=\hat{s}\,)\wedge\ I(s)$$

The abstraction of a safety property $P(s)$ is defined as follows: for the property to hold on an abstract state $\hat{s}$, the property must hold on all states s that are abstracted to $\hat{s}$.

$$\hat{P}(\hat{s}) \quad := \quad \forall s \in S.\ (\alpha(s)=\hat{s})\ \Rightarrow\ P(s)$$

Consequently, if $\hat{P}$ holds on all reachable states of the abstract model, P also holds on all reachable states of the concrete model.

As introduced in [9], a SAT solver is used to compute the abstraction. This approach supports all Verilog operators, including the bit-vector operators. The formula that is passed to the SAT solver directly follows from the definition of the abstract transition relation $\hat{R}$ as given in equation 1. The satisfying assignments obtained from the SAT solver form the abstract transition relation $\hat{R}$. Even if implemented using an efficient all-SAT solver, this technique suffers from the fact that the number of satisfying assignments is exponential in the number of predicates. As a result, the computation of the abstract model can be very slow even for a small number of predicates.

Predicate partitioning [12] improves this method by partitioning the set of the predicates and their next-state versions into *clusters* $C_1,\ldots,C_l$, with $C_j \subseteq \{\pi_1,\ldots,\pi_k,\pi'_1,\ldots,\pi'_k\}$, where π'_i denotes the next state version of π_i. The number of satisfying assignments is limited by the size of cluster C_j to $2^{|C_j|}$. Clearly, by limiting the size of C_j, one can compute the abstract transition relation much faster as compared to the exact approach. The drawback of predicate partitioning is that the resulting abstractions contains additional spurious behavior. The coarser the partitioning of the predicates, the more abstraction refinement iterations are required to rule out the spurious transitions. Jhala and McMillan [13] introduce interpolants for approximating $\hat{R}$: The interpolant is constructed using a *proof* that the trace is infeasible. However, their method is implemented for a theory of linear arithmetic over integers, which is not suitable for verifying hardware descriptions.

4 Propositional Encodings using Proofs

We present the most important concepts of propositional encodings of decision problems using proofs [15].

Proofs in any logic follow a pre-defined set of *proof rules*. A proof rule consists of a set of antecedents $A_1,\ldots,A_k$, which are the premises that have to hold for the rule to be applicable, and a consequence C. The rule is written as follows, where γ denotes the "name" of the rule:

$$\frac{A_1,\ldots,A_k}{C}\,\gamma$$

Definition 1 (Proof Step) *A* proof step *is a triple* $\langle r,p,\mathcal{A}\rangle$ *where r is a proof rule, p is a proposition, and* $\mathcal{A}$ *is a (possibly empty) list of antecedents* $A_1,\ldots,A_k$.

A major challenge of constructing a proof for φ is the propositional structure in φ, i.e., the Boolean operators $\wedge$, $\vee$, and so on. If φ represents a word-level encoding of a large circuit, a non-trivial propositional structure is to be expected due to control logic. The existing proof engines have difficulties handling large and non-trivial propositional structures, as they usually perform case-splitting on Boolean subexpressions of φ.

An *Atom* in a given formula φ is a subexpression of φ that does not contain a Boolean operator. In order to address the case-splitting problem in proof construction, Strichman [18] introduced the following method: by assigning propositional variables to each atom in φ, the propositional structure of φ can be disregarded during the construction of the proof. The proof is built *as if all atoms are conjoined*. The prover has to be modified to continue the proof-search even if a contradiction is detected. Strichman's algorithm proceeds by extracting a propositional formula from the proof, which we call *Proof Constraint*. The proof constraint is conjoined with the *Propositional Skeleton* of φ. The resulting formula is equi-satisfiable with φ.

Definition 2 (Propositional Skeleton) *Let* φ *denote a formula. The set of all atoms in* φ *that are not Boolean identifiers is denoted by* $\mathcal{A}(\varphi)$. *The i-th distinct atom in* φ *is denoted by* $\mathcal{A}_i(\varphi)$. *The* Propositional Skeleton φ_{sk} *of a formula* φ *is obtained by replacing all atoms* $a \in \mathcal{A}(\varphi)$ *by fresh Boolean identifiers* $e_1,\ldots,e_\nu$, *where* $\nu = |\mathcal{A}(\varphi)|$. *We denote the identifier to replace atom* $\mathcal{A}_i$ *by* $e(\mathcal{A}_i)$.

The fact that the dependence between the proof steps is directed and acyclic is captured by the following definition.

Definition 3 (Proof Graph) *A* Proof Graph *is a directed acyclic graph in which the nodes correspond to the steps, and there is an edge* (x,y) *if and only if x represents an antecedent of step y.*

Definition 4 (Proof-Step Encoder) *Given a proof step* $s = (r,p,\mathcal{A})$, *its* Proof-Step Encoder *is a function* $e(s)$ *such that:*

$$e(s) = \begin{cases} \mathit{false} & : \quad p = \bot \\ \neg e(p') & : \quad p = \neg p' \\ \mathit{new\ propositional\ variable} & : \quad \mathit{otherwise} \end{cases}$$

For a proof step $s = (r,p,\mathcal{A})$, *we denote by* $c(s)$ *the constraint that the encoders of the antecedents imply the encoder of the consequence p:*

$$c(s) := (\bigwedge_{a\in\mathcal{A}} e(a)) \longrightarrow e(p)$$

Definition 5 (Proof Constraint) *A* proof *P is a set of proof steps* $\{s_1,\ldots,s_n\}$ *in which the antecedence relation is acyclic. The* Proof Constraint $c(P)$ *induced by P is the conjunction of the constraints induced by its steps:*

$$c(P) := \bigwedge_{s\in P} c(s)$$

Theorem 1 ([15]) *For a proof P and formula* φ, φ *implies* $\varphi_{sk} \wedge c(P)$.

This generalizes the idea of [18] to any proof-generating decision-procedure:

- All atoms $\mathcal{A}(\varphi)$ are passed to the prover *completely disregarding the Boolean structure* of φ.
- For completeness, the prover must be modified to obtain *all* possible proofs, i.e., must not terminate even if the empty clause is resolved. A proof P is obtained.
- Build φ_P as $\varphi_{sk} \wedge c(P)$.

Axiomatizing Bit-Vector Arithmetic The Verilog HDL offers a very rich set of bit-vector operators, including bitwise Boolean operators, bit-vector extraction and concatenation, and reduction operators. Even if great care is taken to construct a very small set of axioms, the number of (potential) 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.

For capacity reasons, we therefore propose to use an *incomplete* prover. Any derivation found by the prover is still correct, but the prover may miss a proof for a derivable fact. We use the technique proposed in [15] in order to limit the proof size. Note that this does not make the overall algorithm incomplete: in case a proof is missed, the corresponding spurious transition is removed later on by the abstraction refinement procedure.

5 Predicate Images from Proofs

As described in Section 3, the computation of $\hat{R}$ (Eq. 1) using all-SAT is exponential in the number of predicates k. As proposed in [15] for software verification, an over-approximation $\hat{R}'$ of $\hat{R}$ can be extracted from a *proof of validity* of the following formula:

$$R(s,s') \wedge \alpha(s) = \hat{s} \wedge \alpha(s') = \hat{s}' \qquad (2)$$

In case of software, the propositional structure of R is typically trivial, as the abstraction is performed separately for each control flow location. Circuits, however, usually have a transition relation with a very complex Boolean structure due to the high degree of synchronous concurrency. The existing work on predicate abstraction in the domain of software verification is therefore not directly applicable.

In order to address this problem, our technique handles the propositional structure and the atoms in R separately. The facts (atoms) we give 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 $R(x,x')$.

As a running example, consider the following Verilog HDL fragment implementing a counter c with 7 bits:

```
initial c=0;

always @(posedge clk)
  if(c!=64 && issue && !retire)
    c=c+1;
  else if(c!=0 && !issue && retire)
    c=c-1;
```

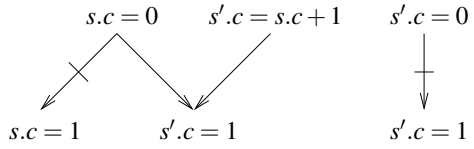


Figure 1. Example of a partial proof of Eq. 2 for the running example. An arrow $a \rightarrow b$ denotes the fact that a is an antecedent of b, a crossed out arrow denotes the fact that a is an antecedent of $\neg b$.

Thus, the concrete transition relation $R(s,s')$ is:

$$s'.c = \begin{cases} s.c+1 & : \quad s.c \neq 64 \wedge s.issue \wedge \neg s.retire \\ s.c-1 & : \quad s.c \neq 0 \wedge \neg s.issue \wedge s.retire \\ s.c & : \quad \text{otherwise} \end{cases}$$

which, after lifting, is

$$\begin{array}{ll} & (s.c \neq 64 \wedge s.issue \wedge \neg s.retire) \rightarrow s'.c = s.c+1 \\ \wedge & (s.c \neq 0 \wedge \neg s.issue \wedge s.retire) \rightarrow s'.c = s.c-1 \\ \wedge & (\neg(s.c \neq 64 \wedge s.issue \wedge \neg s.retire) \wedge \\ & \neg(s.c \neq 0 \wedge \neg s.issue \wedge s.retire)) \rightarrow s'c. = s.c \end{array}$$

The atoms $\mathcal{A}(R)$ are $\{s.c = 0, s.c = 64, s'.c = s.c + 1, s'.c = s.c - 1, s'.c = s.c\}$, and are encoded with the fresh variables $e_1, \ldots, e_5$: $e(s.c = 0) = e_1$, $e(s.c = 64) = e_2$, $e(s'.c = s.c+1) = e_3$, $e(s'.c = s.c-1) = e_4$, and $e(s'.c = s.c) = e_5$. Note that the Boolean variables that form the control of the circuit are not passed to the prover at all. The propositional skeleton φ_{sk} of φ is:

$$\begin{array}{ll} & (\neg e_2 \wedge s.issue \wedge \neg s.retire) \rightarrow e_3 \\ \wedge & (\neg e_1 \wedge \neg s.issue \wedge s.retire) \rightarrow e_4 \\ \wedge & (\neg(\neg e_2 \wedge s.issue \wedge \neg s.retire) \wedge \\ & \neg(\neg e_1 \wedge \neg s.issue \wedge s.retire)) \rightarrow e_5 \end{array}$$

Suppose the predicates of interest are $\pi_1 \iff (c = 0)$ and $\pi_2 \iff (c = 1)$. This yields the atom $s'.c = 0$ for the next-state version of π_1, and $s.c = 1$ and $s'.c = 1$ for π_2. Let these atoms be encoded as follows: $e(s'.c = 0) = e_6$, $e(s.c = 1) = e_7$, and $e(s'.c = 1) = e_8$.

A possible (partial) proof of Eq. 2 is shown in Fig. 1. It is encoded as φ_P as follows:

$$(e_1 \rightarrow \neg e_7) \wedge (e_6 \rightarrow \neg e_8) \wedge ((e_1 \wedge e_3) \rightarrow e_8)$$

The over-approximation $\hat{R}'(\hat{s},\hat{s}')$ of $\hat{R}$ is

$$\begin{array}{c} \hat{s}_1 = e_1 \wedge \hat{s}_2 = e_7 \wedge \hat{s}'_1 = e_6 \wedge \hat{s}'_2 = e_8 \wedge \\ \exists e_2, \ldots, e_5, s.issue, s.retire. \varphi_{sk} \wedge \varphi_P \end{array}$$

Thus, we replaced the existential quantification of the concrete program variables c, $s.issue$, and $s.retire$ by an existential quantification over 7 Boolean variables. Note that the complexity of this operation no longer depends on the width of the counter c. In order to obtain a closed form for $\hat{R}'$, the Boolean quantification has to be performed. The details of this step are beyond the scope of this paper.

6 Abstraction Refinement

Once the abstract model $\hat{T}$ of the circuit is computed, it is passed to a BDD-based Model Checker such as SMV. If the property holds on $\hat{T}$, it also holds on the original circuit, and the algorithm terminates. Otherwise, an abstract counterexample is obtained from the model checker. This abstract counterexample need not correspond to a concrete counterexample due to the over-approximation in $\hat{T}$. It is therefore simulated on the concrete model, which corresponds to the following SAT-instance:

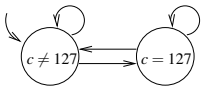
$$\neg p(s_l) \quad \wedge \quad \bigwedge_{t=0}^{l-1} R(s_t, s_{t+1}) \tag{3}$$

If this instance satisfiable, a counterexample trace is extracted and the algorithm terminates. If not so, it is necessary to refine the set of predicates. In [12], weakest preconditions of the property are used to generate new word-level predicates. The main disadvantage of this approach is that it corresponds to a syntactic unwinding of the circuit. This produces undesirable predicates in the presence of counters in the design.

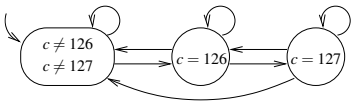
We propose the following algorithm instead of a refinement by means of weakest preconditions:

1. Let l denote the length of the abstract counterexample. Build a formula ϕ that corresponds to an unwinding with l states (i.e., BMC of depth l) and constrain it with the abstract counterexample $\hat{s}(1), \ldots, \hat{s}(l)$.
2. Initialize the bound β with 0.
3. Attempt to refute ϕ using a word-level prover that is limited to use β new facts in the proof.
4. If the proof succeeds, extract the atomic predicates used in the proof, and add these predicates to the set of predicates used for abstraction.
5. Otherwise, increase β, and continue with step 3.

We illustrate this algorithm by continuing the running example from Section 5. Assume we check the property $c \neq 127$. The refinement loop starts with the predicate $\pi_1 \iff c = 127$, which results in the following abstract model:



This model has a counterexample with two states $(\langle \neg\pi_1 \rangle, \langle \pi_1 \rangle)$. The refinement algorithm proposed in [12] computes the weakest precondition of $c = 127$, which is $c = 126$. Let π_2 denote this predicate. This results in an abstract model with three reachable states:



This model contains a counterexample with three states $(\langle \neg\pi_1, \neg\pi_2 \rangle, \langle \neg\pi_1, \pi_2 \rangle, \langle \pi_1, \neg\pi_2 \rangle)$, the weakest precondition is computed, and the predicate $c = 125$ is added. The algorithm continues until the predicate $c = 64$ is added, which proves the property. In total, 63 refinement iterations are needed in order to obtain the predicates for the proof.

McMillan and Jhala [17, 11] extended a software model checker by a refinement algorithm using Craig interpolants extracted from the proof of unsatisfiability of Eq. 3.

In order to prevent predicates that contain variables from more than one simulation step, we never generate new facts that involve variables from more than one time-frame. Continuing our example above for the case of the counterexample of length 2, we pass the following facts to the prover for the time-frames $t = 0$ and $t = 1$, respectively:

$t = 0$	$t = 1$
$c_1 = c_0 + 1$	
$c_0 = 0$	$c_1 = 127$
$c_0 \neq 64$	

The atoms used in the proof of unsatisfiability of Eq. 3 are used as new predicates. Adding these atoms to the set of predicates guarantees that the same spurious prefix is not found again. Continuing our example, one possible proof uses the fact $c_0 = 126$, which is a contradiction to the fact $c_0 = 0$. The facts passed to the prover for the counterexample of length 3 are:

$t = 0$	$t = 1$	$t = 2$
$c_1 = c_0 + 1$		
$c_0 = 0$	$c_2 = c_1 + 1$	$c_2 = 127$
$c_0 \neq 64$	$c_1 \neq 64$	

Note that there also exists a proof that corresponds to an enumeration of the values of c as illustrated above, i.e., a proof using the facts $c_0 = 0$, $c_1 = 1$, and $c_2 = 2$. Unless care is taken, the predicates from the the proof could actually match those from a weakest precondition. However, the prover can be forced to *generalize* the proof that Eq. 3 is unsatisfiable. McMillan et al. achieve this generalization by limiting the size of constants that occur in the proof [14]. The size is gradually increased until a proof is found.

In hardware verification, we expect very large constants, e.g., for use as bit-vector masks, and thus, use a different approach. Continuing our example, the facts $c_2 \leq 64$, $c_1 \leq 64$, and $c_0 \leq 64$ are suitable to construct a proof of unsatisfiability, but the facts of the form $c = 0$, $c = 1$, and so on, should be avoided. Instead of bounding the size of constants used, we limit the number of newly introduced atoms to a bound β. The comparison is done without the time-frame, i.e., an atom is not considered new if it is used in a different time-frame already. We start with $\beta = 1$, and if no proof is found, increase β until a proof is found. In the example above, $\beta = 1$ is sufficient to obtain a proof,

Bench-mark	Latches	Predicate Partitioning+WP					Proof-based Image+Refinement				
		Time	Abs	MC	Ref	P/I	Time	Abs	MC	Ref	P/I
M512B	4137	107.1	2.2	0.8	104.1	3/8	75.9	4.1	0.6	71.2	3/5
M1KB	8234	180.8	9.3	0.8	170.7	3/8	115.2	4.1	0.6	110.5	3/5
M2KB	16427	450.7	24.0	0.9	425.3	3/8	306.5	4.1	0.6	301.8	3/5
M4KB	32796	843.3	37.0	0.8	805.5	3/8	525.9	4.1	0.6	521.2	3/5
AR100	202	3.5	2.8	0.1	0.6	3/3	4.1	3.6	0.1	0.4	3/3
AR200	402	9.6	8.4	0.1	1.1	3/3	4.6	3.6	0.1	0.9	3/3
AR500	1002	32.2	29.3	0.1	2.8	3/3	6.1	3.6	0.1	2.4	3/3
AR1000	2002	122.6	116.8	0.2	5.6	3/3	7.9	3.6	0.1	4.2	3/3
ROB	5136	*	n/a	n/a	n/a	n/a	15.7	5.1	0.1	10.5	6/4

Table 1. Experimental results. All run-times are in seconds. The column "Latches" contains the total number of latches in the cone of influence of the property. The "Time" column contains the total time, the "Abs", "MC", and "Ref" columns show the time taken for abstraction, model checking, and refinement (which includes simulation), respectively. The P/I column shows the number of predicate and refinement iterations, respectively. A * denotes that the 1 hour timeout was exceeded.

namely using $c \leq 64$, whereas the proof that corresponds to unwinding requires $\beta = 2$.

7 Experimental Results

The experiments are performed on a 2.8 GHZ Intel machine with 4 GB of memory running Linux. A time limit of one hour and a memory limit of 700 MB was set for each run. We compare the improved techniques proposed in this paper with the results obtained using image computation based on predicate partitioning with all-SAT and predicate refinement using weakest preconditions.

In order to permit comparison, we use the same benchmarks as in [12]: parts of the Instruction Cache RAM (ICRAM) from the Sun PicoJava II microprocessor [19] (M series benchmarks), and arithmetic circuits. Both benchmarks are parameterized, either in the size of the memory or the width of the words on which arithmetic is performed on. The experimental results are summarized in Table 1. We provide the results from [12] for reference. Note that these results were obtained on a different (slower!) machine. As a SAT-solver, we use Booleforce[1]. The columns marked with "Predicate Partitioning+WP" contain the results of applying the techniques presented in [12], whereas the columns marked with "Proof-based Image+Refinement" contain the results of applying the techniques presented in this paper.

The performance of the M-series benchmarks improves despite of the fact that the algorithm described in [12] spends most time in simulation/refinement. This is due to the fact that the predicate image generated by the prover is more precise, and thus, less refinement is needed. As expected, the performance of the abstraction-intensive AR-series benchmarks benefits dramatically from the proof-based abstraction. In particular, the run-time of the abstraction phase no longer depends on the bit-width. The run-time of the refinement phase still depends on the number of bits, as a net-list-level model is used for simulation. No benefit is obtained from the proof-based refinement, as the weakest-precondition method already computes optimal predicates.

In order to quantify the benefit of the proof-based predicate refinement, we added the benchmark ROB that requires a generalization for a deep counter, similar to the running example used in this paper. The method based on the weakest preconditions unwinds the counter, and times out. The proof-based predicate refinement algorithm generates the appropriate predicate for the counter, and shows the property efficiently.

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

References

[1] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *DAC*, pages 218–223, 2004.

[2] T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, 2000.

[3] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, 2002.

[4] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *FMCAD*, 2002.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. H. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[6] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL*, 1992.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] E. Clarke, H. Jain, and D. Kroening. Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical Report CMU-CS-04-139, Carnegie Mellon University, 2004.

[9] E. Clarke, M. Talupur, and D. Wang. SAT based predicate abstraction for hardware verification. In *SAT*, 2003.

[10] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254, pages 72–83, 1997.

[11] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM Press, 2004.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proceedings of DAC 2005*, pages 445–450, 2005.

[13] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV*, pages 39–64. Springer, 2005.
[14] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.
[15] D. Kroening and N. Sharygina. Approximating predicate images for bit-vector logic. In *TACAS*, LNCS, pages 242–256. Springer, 2006.
[16] R. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
[17] K. L. McMillan. An interpolating theorem prover. In *Proceedings of TACS*, volume 2988, pages 16–30. Springer, 2004.
[18] O. Strichman. On solving Presburger and linear arithmetic with SAT. In *FMCAD*, volume 2517, pages 160–170. Springer, 2002.
[19] http://www.sun.com/processors/technologies.html.
[20] D. Wang, P. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *DAC*, pages 35–40, 2001.