Predicate Abstraction and Refinement Techniques for Verifying Verilog

Edmund Clarke Himanshu Jain Daniel Kroening

June 25, 2004

CMU-CS-04-139

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Model checking techniques applied to large industrial circuits suffer from the state explosion problem. A major technique to address this problem is abstraction. Predicate abstraction has been applied successfully to large software programs. Applying this technique to hardware designs poses additional challenges. This paper evaluates three techniques to improve the performance of SAT-based predicate abstraction of circuits: 1) We partition the abstraction problem by forming subsets of the predicates. The resulting abstractions are more coarse, but the computation of the abstract transition relation becomes easier. 2) We evaluate the performance effect of lazy abstraction, i.e., the abstraction is only performed if required by a spurious counterexample. 3) We use weakest preconditions of circuit transitions in order to obtain new predicates during refinement. We provide experimental results on publicly available benchmarks from the Texas97 benchmark suite.

This research was sponsored by the Gigascale Systems Research Center (GSRC), the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of GSRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

Keywords: Predicate Abstraction, Verilog, SAT

1 Introduction

Formal verification techniques are widely applied in the hardware design industry. Introduced in 1981, *Model Checking* [9, 10] is one of the most commonly used formal verification techniques in a commercial setting. However, Model Checking suffers from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [6]. 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 behaviors of the system.

In the hardware domain, the most commonly used abstraction technique is *localization reduction*. The abstract model is created from the given circuit by removing a subset of the latches together with the logic required to compute their next state. The latches that are removed are called the *invisible latches*. The latches remaining in the abstract model are called *visible latches*. Intuitively, the visible latches are most relevant to the property in consideration.

Localization reduction is a *conservative* over-approximation of the original circuit. This implies that if the abstraction satisfies the property, the property also holds on the original, concrete circuit. The drawback of the conservative abstraction is that when model checking of the abstraction fails, it may produce a counterexample that does not correspond to any concrete counterexample. This is usually called a *spurious counterexample*.

In order to check if an abstract counterexample is spurious, the abstract counterexample is simulated on the concrete machine. This is called the *simulation* step. Like in Bounded Model Checking (BMC), the concrete transition relation for the design and the given property are jointly unwound to obtain a Boolean formula. The number of unwinding steps is given by the length of the abstract counterexample. As in BMC, the Boolean formula is then checked for satisfiability using a SAT procedure such as Chaff [23]. If the instance is satisfiable, the counterexample is real and the algorithm terminates. If the instance is unsatisfiable, the abstract counterexample is spurious, and *abstraction refinement* has to be performed.

The basic idea of the abstraction refinement techniques is to create a new abstract model which contains more detail in order to prevent the spurious counterexample. This process is iterated until the property is either proved or disproved. It is known as the *Counterexample Guided Abstraction Refinement* framework, or CEGAR for short [22, 2, 7, 14]. In case of localization reduction, the refinement is done by moving more latches from the set of invisible latches to the set of visible latches.

In the software domain, the most successful abstraction technique for large systems is *Predicate abstraction* [16, 13]. 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.

While localization reduction is a special case of predicate abstraction, predicate abstraction can result in a much smaller abstract model. As an example, assume a circuit contains two sets of latches, each encoding a number. Predicate abstraction can keep track of a numerical relation between the two numbers using a single predicate, and thus, using a single state bit in the abstract model. In contrast to that, localization reduction typically turns all the bits in the two words into visible latches, and thus, the abstraction is identical to the original model.

Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [2] and promoted by the success of the SLAM project [1]. The goal of this project is to verify that Windows device drivers obey API conventions. The abstraction of the program is computed by using a theorem prover such as Simplify [15] or Zapato [4].

When applying predicate abstraction to circuits, two problems arise: Most model-checkers used in the hardware industry use a very low level design, usually a net-list, for verification purposes. However, predicate abstraction is only effective if the predicates can cover the relationship between multiple latches. This typically requires a word level model given in register transfer language (RTL), e.g., in Verilog. The RTL level languages are similar to languages used in the software domain, such as ANSI-C.

The second problem concerns with the use of theorem provers for computing the predicate abstraction. Theorem provers model the variables using unbounded integer numbers. Overflow or bit-wise operators are not modeled. However, hardware description languages like Verilog provide an extensive set of bit-wise operators. For hardware design, the use of these bit-level constructs is ubiquitous.

In [12], a SAT-based technique for predicate abstraction of circuits given in Verilog is introduced. The first step is to obtain predicates from the control flow guards in the Verilog file. The circuit is then synthesized and transformed into net-list-level. The use of a SAT solver like ZChaff [23] in order to perform the abstraction allows to support all bit-level constructs. However, there approach suffers from two drawbacks. 1) Each transition in the abstract model is computed by a new SAT solver run. Thus, the learning done by a SAT solver in the form of conflict clauses is lost when computing other transitions. 2) If refinement becomes necessary, only bit-level predicates are introduced. This way of refinement closely resembles refinement techniques for localization reduction. In contrast to that, predicate abstraction tools for software, such as SLAM, use weakest preconditions to derive new word-level predicates.

Predicate abstraction tools for software predicate abstraction use multiple heuristics in order to reduce the cost of calling the theorem prover while computing the abstraction. The SLAM tool limits the number of predicates in a particular query, i.e., it partitions the set of predicate into smaller subsets. This speeds up the abstraction process, but the resulting abstraction contains additional spurious behavior. If the SLAM toolkit encounters a spurious counterexample, it first assumes that it is caused by a lack of predicates, and attempts to find new predicates. If no new predicates are found, the counterexample is caused by the partitioning of the predicates during the abstraction. In this case, a separate refinement algorithm (called Constrain [3]) is invoked. Note that this step only addresses spurious behavior due to an inexact abstraction, as opposed to spurious behavior caused by insufficient predicates.

Software predicate abstraction tools abstract the individual statements or basic blocks separately. Thus, only a small number of predicates is typically affected, and a syntactic predicate partitioning works well. In contrast to that, even RTL-level circuits are monolithic. Each transition consists of simultaneous assignments to all latches. Thus, the syntactic partitioning of the predicates might result in no reduction of the number of predicates.

In the BLAST toolkit [17], the abstraction is completely demand-driven. It is only performed when a spurious counterexample is encountered. The abstraction is only performed to the extent necessary to remove the spurious behavior. This is called *lazy abstraction*.

Contribution This paper applies three techniques from the software predicate abstraction domain to the abstraction of circuits given in Verilog RTL. We partition the given set of predicates into clusters of related predicates. The abstraction is computed separately with respect to the predicates in each cluster. Since each cluster contains only a small number of predicates, the computation of the abstraction becomes easier.

As described above, the highly concurrent nature of hardware limits the benefits of this technique. We therefore also evaluate lazy abstraction in the context of predicate abstraction for circuits.

When a spurious counterexample is encountered, we first check whether it is caused by insufficient predicates or caused by the lazy abstraction. If the counterexample is caused by lazy abstraction, we compute the unsatisfiable core of the SAT instance corresponding to one abstract transition. We use the unsatisfiable core in order to extract a small set of predicates that eliminates the counterexample. The fewer predicates are found, the more spurious counterexamples can be eliminated.

If the spurious counterexample is caused by insufficient predicates, we use a refinement technique used by software predicate abstraction tools: we compute the weakest precondition of the property with respect to the transition function given by the circuit. In order to ensure that the predicates generated do not become too large, we simplify the weakest pre-conditions using values from the spurious counterexample. To the best of our knowledge, this is the first time weakest preconditions of circuits are used for refinement of predicate abstractions.

We formally describe the semantics of the Verilog subset we are handling. We report experimental results using benchmarks from the Texas97 benchmark suite.

Related work In [12], SAT-based predicate abstraction is applied to hardware verification. The authors present two SAT-based algorithms to refine the abstract models. They distinguish spurious transitions (caused by incomplete abstraction) and spurious counterexamples (caused by insufficient predicates). Spurious transitions are eliminated by constraining the abstract model, while spurious counterexamples are eliminated by adding new *separating* predicates. In contrast to our work, they carry out the predicate abstraction of the net-lists using bit-level predicates.

Henzinger et al. [18] present a technique for constructing parsimonious abstractions of C programs using proofs of unfeasibility of abstract counterexamples. A predicate abstraction is parsimonious if at each control location, it tracks only those predicates which are required for proving correctness. They report that on the average the number of predicates tracked at each control location are quite small. However, they make use of a theorem prover for computing the abstractions and do not deal with bit-wise constructs.

In [11], a SAT solver is used to compute an abstraction of an ANSI-C program. The main idea is to form a SAT equation containing all the predicates, a basic block, and two symbolic variables for each predicate, one variable for the state before the execution of the basic block, and one variable for the state after its execution. The SAT solver is then used to obtain all satisfying assignments in terms of the the symbolic variables. However, the runtime of this process typically grows exponentially in the number of predicates. The technique has also been applied to SpecC [20], which is a concurrent version of ANSI-C.

In [12], spurious transitions are eliminated by generating a constraint from SAT based *conflict-analysis*. Since conflict analysis is used by a SAT solver for generating an unsatisfiable core, our use of unsatisfiable cores is similar to the approach in [12]. In [21], the unsatisfiable cores are used for extracting small abstracted formulas within an abstraction refinement loop for deciding the satisfiability of Presburger formulas.

Outline In section 2, we formalize the semantics of the subset of Verilog that we handle. Section 3 describes SAT-based predicate abstraction with the help of an example. Techniques for partitioning the given set of predicates is given in Section 4. We present techniques for abstraction refinement in section 5. Finally, we report experimental results in section 6.

2 Formal Semantics of Verilog RTL

The Verilog hardware description language is used to model digital circuits at various levels, ranging from high-level behavioral Verilog to low-level net-lists. The Verilog standard [19] describes the semantics of the Verilog language informally by means of a discrete event execution model.

We formalize the Verilog semantics for a particular special case: synthesizable Verilog with one single clock `clk`. We assume the clock is only used within either `posedge` or `negedge` event guards, but not both. The edge descriptor is denoted by E `clk`.

We use the following formalism to model the concrete circuit: A transition system $T = (S, I, R)$ consists of a set of states S, a set of initial states $I \subseteq S$, and a transition relation R, which relates a current state $s \in S$ to a next-state $s' \in S$.

We assume that the module structure of the design is already flattened. Let s be a Verilog module item. A Verilog module item can either be a continuous assignment, or an `initial` or an `always` block.

Continuous assignment Only one continuous assignment per network is allowed. Let w_i be the network that is assigned by continuous assignment i, and e_i the value that is assigned. For each such continuous assignment, we add the constraint $w_i = e_i$ to A:

$$A \quad := \quad \bigwedge_i w_i = e_i$$

Initial and Always The statements in the `initial` and `always` blocks define the initial values of latches and the transition function (next state function) of the latches.

Note that `always` blocks can also be used to define combinational logic. Consider the following example:

```
reg r;
input i;

always @(i) r=!i;
```

This defines combinational logic: `r` is the negation of the input `i`. There will be no latch corresponding to `r` in the circuit. On the other hand, the following example illustrates the case of a latch:

```
reg [31:0] r;
input clk;

always @(posedge clk) r=r+1;
```

Let $\mathcal{V}$ denote the set of variables, as given in the Verilog file. We distinguish the two cases by examining the events given as event guards. If the clock event E `clk` is used to guard an assignment to a register, it is considered to be a latch, and combinational logic otherwise. Let $\mathcal{L} \subseteq \mathcal{R}$ denote the set of true latches. The set of states S of the state machine is then defined to be

$$S \quad := \quad \{0,1\}^{|\mathcal{L}|}$$

For a state $s \in S$, we denote the value of an expression e in that particular state by $s(e)$. The set of variables that are not latches is denoted by $\mathcal{C}$:

$$\mathcal{C} \quad := \quad \mathcal{V} \backslash \mathcal{L}$$

In order to define the semantics of the statements in the `initial` and `always` blocks, we define the notion of a *process state*. A process state ϕ is a mapping from the variables $r \in \mathcal{V}$ into a pair of expressions. We denote the first member of the pair by $\phi_c(r)$ and the second member of the pair by $\phi_f(r)$. The expression $\phi_c(r)$ is called the *current value*, while $\phi_f(r)$ is called the *final value* of r.

The two differ in order to distinguish non-blocking assignments from blocking assignments. Non-blocking assignments only update the final value, but not the current value, while blocking assignments update both.

For an expression e, $\phi_c(e)$ denotes the evaluation of e in the current state ϕ_c, i.e., all variables v that are found in e are replaced by $\phi_c(v)$.

We aim at obtaining the state after the execution of the statements. For this, we define the function $\sigma(\phi, p)$. The function takes a process state ϕ and statement p as argument and returns the new process state after the execution of p. Formally, the function is defined by means of a case-split on p.

- If p is an `if` statement with then-branch p' and else branch p'', the function σ is applied recursively to p' and p''. The value of the branching guard g evaluated in the state ϕ_c is used to select the correct branch.

$$\sigma(\phi, \texttt{if}(g)\ p'\ \texttt{else}\ p'') \quad := \quad \begin{cases} \sigma(\phi, p') & : \quad \phi_c(g) \\ \sigma(\phi, p'') & : \quad \text{otherwise} \end{cases}$$

- If p is a sequential composition of p' and p'' (by means of a `begin end` block), then the function is first applied to p' using the process state ϕ. The process state resulting from this is passed to the application of the function σ to p''.

$$\sigma(\phi, p'\mathtt{;}\, p'') \quad := \quad \sigma(\sigma(\phi, p'), p'')$$

- If p is a blocking assignment of some expression e to register r, then the new state is the old state where both the current and the final value of r is the value of the expression evaluated in the current state.

$$\sigma(\phi, r\mathtt{=}e) \quad := \quad \lambda x \in \mathcal{R} : \begin{cases} (\phi_c(e), \phi_c(e)) & : \quad r = x \\ \phi(x) & : \quad \text{otherwise} \end{cases}$$

- If p is a non-blocking assignment of some expression e to register r, then the new state is the old state where the final value of r (the second member of the pair) is the value of the expression evaluated in the current state. The current value of r (the first member of the pair) remains unchanged.

$$\sigma(\phi, r\mathtt{<=}e) \quad := \quad \lambda x \in \mathcal{R} : \begin{cases} (\phi_c(r), \phi_c(e)) & : \quad r = x \\ \phi(x) & : \quad \text{otherwise} \end{cases}$$

- If p is an event guard statement $\mathtt{@}(G)\ p'$ or a delay statement $\mathtt{\#}d\ p'$, we simply recursively apply σ to p'.

$$\begin{aligned} \sigma(\phi, \mathtt{@}(G)\ p') \quad &:= \quad \sigma(\phi, p') \\ \sigma(\phi, \mathtt{\#}d\ p') \quad &:= \quad \sigma(\phi, p') \end{aligned}$$

Initial State Let p^I denote all the statements in `initial` blocks. The process state before the execution of any statement is denoted by ι^I. It is undefined, i.e., no assumption about the value of any register is made unless it is explicitly initialized. The process state after the execution of p^I is denoted by Φ^I:

$$\Phi^I \quad := \quad \sigma(\iota^I, p^I)$$

The set of initial states I is defined as follows: for each latch $r \in \mathcal{L}$, we require that the initial value of r is the final value of r after the execution of p^I.

$$I \quad := \quad \{s \in S \mid \bigwedge_{r \in \mathcal{L}} s(r) = \Phi^I_f(r)\}$$

Next State Let p^R denote all the statements in `always` blocks. The process state before the execution of any statement is denoted by ι^R, which assigns the previous value to all latches $r \in \mathcal{L}$. It is undefined for the variables $v \in \mathcal{C}$ that are used for combinational logic only.

$$\forall v \in \mathcal{L} : \quad \iota^R(v) \quad := \quad (v, v)$$

The process state after the execution of p^R is denoted by Φ^R:

$$\Phi^R \quad := \quad \sigma(\iota^R, p^R)$$

The transition relation $R(s,s')$ is defined as follows: for each latch $v \in \mathcal{L}$, we require that the next state value of v is the final value of v after the execution of p^R. For each variable $v \in \mathcal{C}$, we require that the current state value of v is the final value of v after the execution of p^R. Also, we add the constraints A defined above for the continuous assignments as requirement for the current state.

$$\begin{array}{rcl} R(s,s') & := & \bigwedge_{v\in\mathcal{L}} s'(v) = s(\Phi^R_f(v)) \\ & \wedge & \bigwedge_{v\in\mathcal{C}} s(v) = s(\Phi^R_f(v)) \\ & \wedge & s(A) \end{array}$$

Example To illustrate the difference between a blocking assignment and a non-blocking assignment, consider the following example:

```
reg r, q;
input clk;

always @(posedge clk) begin
  r<=q;
  q<=r;
end
```

When applying σ to the two non-blocking assignments above, we obtain

$$\begin{array}{rcl} \Phi^R(\mathtt{r}) & = & \mathtt{q} \\ \Phi^R(\mathtt{q}) & = & \mathtt{r} \end{array}$$

and thus, `r` and `q` are swapped. If the non-blocking assignments are replaced by blocking assignments, we obtain a different result:

$$\begin{array}{rcl} \Phi^R(\mathtt{r}) & = & \mathtt{q} \\ \Phi^R(\mathtt{q}) & = & \mathtt{q} \end{array}$$

Notation In order to compute word-level predicates, we group the latches in $\mathcal{L}$ defined above into word-level registers. The latches that are grouped are the latches given by a single declaration in Verilog. Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ denote the set of registers. For example, the state of the Verilog program in Fig. 1 is defined by the value of the registers `x` and `y`, and each of them has a storage capacity of 8 bits.

Note that we consider the external inputs to be registers without a next-state function. Let $\mathcal{Q} \subseteq \mathcal{R}$ denote the set of registers that are not external inputs, i.e., have a next-state function. We denote the next-state function of a word-level register $r_i \in \mathcal{Q}$ by $f_i(r_1, \ldots, r_n)$, or $f_i(\bar{r})$ using vector notation. Any occurrences of network identifiers (continuous assignments) or combinatorial registers are replaced by their respective value.

```
module main (clk);
input clk;
reg [7:0] x,y;

initial x = 1;
initial y = 0;

always @ (posedge clock) begin
   y <=x;
   if (x<100) x<=y+x;
end
endmodule
```

Figure 1: A Verilog program.

Using the word-level next-state functions f_i, the transition relation defined above can be re-written. $R(\bar{r},\bar{r}')$ relates the current state $\bar{r} \in S$ to the next state $\bar{r}'$ and is defined as follows:

$$R(\bar{r},\bar{r}') \quad := \quad \bigwedge_{r_i \in Q} (r_i' \Leftrightarrow f_i(\bar{r}))$$

For example, the next state function for the register x in Fig. 1 is given as follows: if the value of x in the current state is less than 100, then the value of x in the next state is equal to the sum of current values of x and y, that is $x+y$. If the value of x is greater than or equal to 100, then the value of x in the next state remains unchanged. The value of y in the next state is equal to the value of x in the current state. We use the trinary choice operator $c?g:h$ to denote a function which evaluates to g when the condition c is true, otherwise it evaluates to h. Thus, the next state functions for x and y and the transition relation are given as follows:

$$\begin{aligned} f_x(x,y) &:= ((x \ < \ 100) \ ? \ (x+y) \ : \ x) \\ f_y(x,y) &:= x \\ R(x,y,x',y') &:= (x' = ((x \ < \ 100) \ ? \ (x+y) \ : \ x)) \wedge (y' = x) \end{aligned}$$

Note that we do not flatten the registers x,y to individual bits. Thus, we have a next state function for the whole registers x,y and not for the individual bits in x,y.

3 Predicate Abstraction

In predicate abstraction [16], the variables of the concrete program are replaced by Boolean variables that correspond to a predicate on the variables in the concrete program. These predicates are functions that map a concrete state $\bar{r} \in S$ into a Boolean value. Let $\mathtt{B} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates over the given program. When

Figure 2: The value of x and y in different states

applying all predicates to a specific concrete state, one obtains a vector of Boolean values, which represents an abstract state $\bar{b}$. We denote this function by $\alpha(\bar{r})$. It maps a concrete state into an abstract state and is therefore called an *abstraction function*.

We perform an existential abstraction [8], i.e., the abstract model can make a transition from an abstract state $\bar{b}$ to $\bar{b}'$ iff there is a transition from $\bar{r}$ to $\bar{r}'$ in the concrete model and $\bar{r}$ is abstracted to $\bar{b}$ and $\bar{r}'$ is abstracted to $\bar{b}'$. We call the abstract machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\begin{aligned} \hat{R} \quad := \{(\bar{b},\bar{b}') \,|\, \exists \bar{r},\bar{r}' \in S: \quad & R(\bar{r},\bar{r}')\wedge \\ & \alpha(\bar{r}) = \bar{b} \wedge \alpha(\bar{r}') = \bar{b}'\} \end{aligned} \tag{1}$$

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

$$\hat{P}(\bar{b}) \quad :\Longleftrightarrow \quad \forall \bar{r} \in S : (\alpha(\bar{r}) = \bar{b}) \Longrightarrow P(\bar{r})$$

The same abstraction is also used for the initial state $I(\bar{r})$. Thus, if $\hat{P}$ holds on all reachable states of the abstract model, P also holds on all reachable states of the concrete model.

Example Consider the Verilog program in Fig. 1. We wish to show that the value of the register x is always less than 200. That is, we want to prove that the given program satisfies the safety property $\mathbf{AG}(x < 200)$, where **AG** is a CTL operator which stands for *always globally*. Intuitively, the property holds because the value of x follows a sequence starting from 1 to 144. Upon reaching the value 144, the guard in the next state function for x becomes false, and its value remains unchanged. The values of x and y in each state are shown in Fig. 2.

We follow the counterexample guided abstraction refinement (CEGAR) framework in order to prove this property. The first step of the CEGAR loop is to obtain an abstraction of the given program. We use predicate abstraction for this purpose.

SAT based predicate abstraction Most tools using predicate abstraction for verification use general-purpose theorem provers such as Simplify [15] to compute the abstraction. This approach suffers from the fact that errors caused by bit-vector overflow may remain undetected. Furthermore, bit-vector operators are usually treated by means of uninterpreted functions. Thus, properties that rely on these bit-vector operators cannot be verified. However, we expect that Verilog designs typically use an abundance of bit-vector operators, and that the property of interest will depend on these operations.

In [11], the authors propose to use a SAT solver to compute the abstraction of a sequential ANSI-C program. This approach supports all ANSI-C integer operators, including the bit-vector operators. We use a similar technique for computing the abstraction of the Verilog programs.

A symbolic variable b_i is associated with each predicate π_i. Each concrete state $\bar{r} = \{r_1, \ldots, r_n\}$ maps to an abstract state $\bar{b} = \{b_1, \ldots, b_k\}$, where $b_i = \pi_i(\bar{r})$. If the concrete machine makes a transition from state $\bar{r}$ to state $\bar{r}' = \{r'_1, \ldots, r'_n\}$, then the abstract machine makes a transition from state $\bar{b}$ to $\bar{b}' = \{b'_1, \ldots, b'_k\}$, where $b'_i = \pi_i(\bar{r}')$.

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:

$$\hat{R} \quad = \quad \{(\bar{b}, \bar{b}') \mid \exists \bar{r}, \bar{r}' \; : \; \Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}')\} \tag{2}$$

$$\Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}') \quad := \quad \bigwedge_{i=1}^{k} b_i = \pi_i(\bar{r}) \; \wedge \; R(\bar{r}, \bar{r}') \; \wedge \; \bigwedge_{i=1}^{k} b'_i = \pi_i(\bar{r}') \tag{3}$$

The set of abstract transitions $\hat{R}$ is computed by transforming $\Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}')$ into conjunctive normal form (CNF) and passing the resulting formula to a SAT solver. Suppose the SAT solver returns $\bar{r}, \bar{r}', \bar{b}, \bar{b}'$ as the satisfying assignment. We project out all variables but $\bar{b}$ and $\bar{b}'$ from this satisfying assignment to obtain one abstract transition $(\bar{b}, \bar{b}')$. Since we want all the abstract transitions, we add a blocking clause to the SAT equation that eliminates all satisfying assignments with the same values for $\bar{b}$ and $\bar{b}'$. This process is continued until the SAT formula becomes unsatisfiable. The satisfying assignments obtained form the abstract transition relation $\hat{R}$. As described in [11], there are numerous ways to optimize this by computation. These techniques are beyond the scope of this article.

An abstract state $\bar{b}$ is an initial state in the abstract model, if there exists a concrete state $\bar{r}$ which is an initial state in the concrete model and maps to $\bar{b}$.

$$\hat{I} \quad = \quad \{\bar{b} \mid \exists \bar{r} : \; \bigwedge_{i=1}^{k} b_i = \pi_i(\bar{r}) \; \wedge I(\bar{r})\} \tag{4}$$

Using this definition, the abstract set of initial states can be enumerated by using a SAT solver.

Example: Continuing our example, the concrete transition relation of the Verilog program in Fig. 1 is given as follows:

$$R(x, y, x', y') \quad := \quad (x' \Leftrightarrow ((x \; < \; 100) \; ? \; (x + y) \; : \; x)) \wedge (y' \Leftrightarrow x)$$

We want to prove that the concrete system (Verilog program) satisfies $\mathbf{AG}(x < 100)$. In order to perform predicate abstraction we need a set of predicates. For our example, we take $\{x < 200, x < 100, x + y < 200\}$ as the set of predicates. We associate symbolic variables b_1, b_2, b_3 with each predicate, respectively. The following equation is converted to CNF and passed to a SAT solver:

$$(b_1 \Leftrightarrow (x < 200)) \; \wedge \; (b_2 \Leftrightarrow (x < 100)) \; \wedge \; (b_3 \Leftrightarrow (x + y < 200)) \wedge$$

```
MODULE main

VAR b1:  boolean; //stands for x<200
VAR b2:  boolean; //stands for x<100
VAR b3:  boolean; //stands for x+y<200

INIT (b1 & b2 & b3)

TRANS (b1 & !b2 & !b3 & next(b1) & !next(b2) & !next(b3)) |
      (b1 & b2 & !b3 & !next(b1) & !next(b2)) |
      (b1 & b2 & b3 & next(b1) & next(b3)) |
      (b1 & !b2 & next(b1) & !next(b2) & next(b3)) |
      (!b1 & !b2 & !next(b1) & !next(b2)) |
      (b1 & b3 & next(b1) & !next(b2) & !next(b3))

SPEC AG (b1)
```

Figure 3: Abstraction of Verilog program in Fig. 1 using predicates $x < 200$, $y < 100$, and $x + y < 200$. The output is in the format accepted by NuSMV model checker. It is generated automatically by our tool.

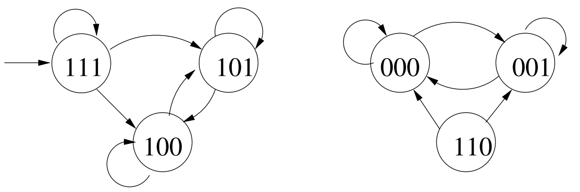

Figure 4: Finite state machine for abstract model in Fig. 3. Abstract states 010 and 011 are not possible, as this would require $x < 200$ to be false and $x < 100$ to be true in the same state.

$$R(x, y, x', y') \;\wedge$$
$$(b'_1 \Leftrightarrow (x' < 200)) \;\wedge\; (b'_2 \Leftrightarrow (x' < 100)) \;\wedge\; (b'_3 \Leftrightarrow (x' + y' < 200)$$

The abstract transition relation obtained is given by the `TRANS` statement in Fig. 3. It is a disjunction of cubes. The cube (`b1` & `!b2` & `!b3` & `next(b1)` & `!next(b2)` & `!next(b3)`) gives the transition from the abstract state in which b_1 is true and b_2, b_3 are false to the same abstract state ($100 \to 100$ for short). Intuitively, this abstract transition is possible because $b_2 = 0$ in the current abstract state, which means that $x \geq 100$ in the concrete system. So the value of the register x in the next state (x') is x and the value of the predicates $x < 200$ and $x < 100$ in the next state remains unchanged. The value of register y becomes equal to x, so $y' = x$. Even though both x' and y' range between 100 and 200, $x' + y'$ can be less than 200, due to arithmetic overflow. Thus, the transition $100 \to 100$ is possible. The other possible transitions are shown in Fig. 4.

The equation passed to the SAT solver for computing the initial set of abstract states is as follows:

$$(b_1 \Leftrightarrow (x < 200)) \wedge (b_2 \Leftrightarrow (x < 100)) \wedge (b_3 \Leftrightarrow (x + y < 200)) \wedge$$
$$(x = 1) \wedge (y = 0)$$

The abstract set of initial states produced is given by the `INIT` statement in Fig. 3. There is only one abstract initial state in which all the boolean variables b_1, b_2, b_3 are true.

The property $\mathbf{AG}(x < 100)$ is abstracted by using the boolean variable b_1 for the predicate $(x < 100)$. The abstracted property is given by the `SPEC` statement in the Fig. 3. The abstract model satisfies the property `AG (b1)`, as the only states reachable from the initial abstract state (111) are $\{111, 101, 100\}$ (Fig. 4). Since the property holds on the abstract model, we can conclude that the property $\mathbf{AG}(x < 100)$ holds on the Verilog program in Fig. 1.

This examples demonstrates the advantage of working with word-level predicates, such as $x + y < 200$. Even if the sizes of the registers x, y are increased, only 3 word-level predicates are needed for proving the property. This is not the case with the approach presented in [12], where the design is flattened to the net-list level and predicate abstraction is carried out using bit-level predicates.

4 Predicate Partitioning

4.1 Computing Multiple Abstract Transition Relations

We call the computation of the exact existential abstraction as described in the previous section the *Monolithic approach*. In the worst case, the number of satisfying assignments generated from equation (3) is exponential in the number of predicates. In practice, computing abstractions using the monolithic approach can be very slow even for a small number of predicates.

The speed of the computation of the abstraction can be improved if we do not aim at the the most precise abstract transition relation. That is, we allow our abstraction to be an over-approximation of the abstract transition relation generated by the monolithic approach. The SLAM toolkit, for example, limits the number of predicates in each theorem prover query. Thus, the set of the predicates and their next-state state versions is partitioned into smaller sets of related predicates. We call these sets clusters, and denote them by $C_1, \ldots, C_l$, with $C_j \subseteq \{\pi_1, \ldots, \pi_k, \pi'_1, \ldots, \pi'_k\}$.

The equation for abstracting the transition system with respect to C_j is given as follows:

$$\bigwedge_{\pi_i \in C_j} b_i = \pi_i(\bar{r}) \wedge R(\bar{r}, \bar{r}') \wedge \bigwedge_{\pi'_i \in C_j} b'_i = \pi_i(\bar{r}')$$

We abstract the transition system with respect to each cluster. This results in a total of l abstract transition relations $\hat{R}_1, \ldots, \hat{R}_l$, which are conjuncted to form $\hat{R}$:

$$\hat{R} \quad := \quad \bigwedge_{i=1}^{l} \hat{R}_i \tag{5}$$

Intuitively, we obtain an over-approximation because now each SAT equation will have fewer predicates and hence less information about the variables of concrete system. We refer to this technique as *predicate partitioning*. We evaluate two different syntactic predicate partitioning techniques, *cone partitioning* and partitioning for *lazy abstraction*.

Let $var(e)$ denote the set of variables appearing in an expression e. For example, $var(x' + y' < 200)$ is $\{x', y'\}$.

In [7], two formulas g_1 and g_2 are said to interfere iff $var(g_1) \cap var(g_2) \neq \emptyset$. The authors use this notion of interference to partition the set of formulas into various formula clusters. This technique can be used for partitioning the set of predicates. However, our experiments indicate that this results in clusters that are too large. Thus, we make the syntactic conditions for keeping the two predicates together stronger, which leads to a smaller number of predicates per cluster.

Syntactic cone partitioning Given a formula g' in terms of next state variables $\bar{r}'$, the current state variables $\bar{r}$ that affect the value of the variables in $var(g')$ are denoted by $cone(g')$. The set of variables $cone(g')$ is similar to one step of cone of influence. It is defined as follows: The variables in the next-state functions for the registers mentioned in g' form the cone of g'. Recall that the next-state function of a particular register r_i is given by $f_i(\bar{r})$.

$$cone(g') \quad := \quad \bigcup_{r_i' \in var(g')} var(f_i(\bar{r}))$$

For example, let g' be $a' + b' < c'$. Let the next state functions for a', b', c' be $a + b$, c, $i + x$, respectively. Here, $var(g') = \{a', b', c'\}$ and $cone(g') = \{a, b, c, i, x\}$.

The clusters of the predicates and their next-state versions $\{\pi_1, \ldots, \pi_k, \pi_1', \ldots, \pi_k'\}$ are created by the following steps:

1. The next-state predicates that have identical cone sets are kept in a single cluster. That is, if $cone(\pi_i') = cone(\pi_j')$ then π_i' and π_j' are kept in the same cluster. Let $C_1', \ldots, C_l'$ be the clusters of $\{\pi_1', \ldots, \pi_k'\}$ obtained after this step. Since all the predicates in a given cluster C_i' have the same cone, we define $cone(C_i')$ as the cone of any element in C_i'.

2. The final set of clusters is given by $\{C_1, \ldots, C_l\}$. Each C_i contains all the next-state predicates from C_i' and the current-state predicates that mention variables in the cone of C_i'. Formally, C_i is defined as follows:

$$C_i \quad := \quad C_i' \cup \{\pi_j \mid var(\pi_j) \subseteq cone(C_i')\}$$

Example: Let the set of current-state and next-state predicates be $\{x < 200, y = 100, z > 100, x' < 200, y' = 100, z' > 100\}$. Let the next state functions be as follows: $x' = y + z$, $y' = x$, and $z' = x$. After the first step of the algorithm the clusters will be $C'_1 = \{x' < 200\}$ and $C'_2 = \{y' = 100, z' > 100\}$. The predicates $y' = 100$ and $z' > 100$ are kept in the same cluster, as they have the identical cone set $\{x\}$. Since $cone(C'_1) = \{y, z\}$ and $cone(C'_2) = \{x\}$, the clusters obtained after the second step of the algorithm are $C_1 = \{y = 100, z > 100, x' < 200\}$ and $C_2 = \{x < 200, y' = 100, z' > 100\}$.

Observe how the predicates in a given cluster affect each other. For example, in C_2, if $x < 200$ is false, then we know that $y' = 100$ will be false and and $z' > 100$ will be true in the next state.

Let $R(x, y, z, x', y', z')$ denote the transition relation. If we associate the symbolic variables b_1, b_2, b_3, b'_1, b'_2, b'_3 with the predicates $x < 200, y = 100, z > 100, x' < 200, y' = 100$, and $z' > 100$, respectively, then the equation for abstracting the transition relation with respect to C_2 is as follows:

$$(b_1 \Leftrightarrow (x < 200)) \wedge R(x, y, z, x', y', z') \wedge$$
$$(b'_2 \Leftrightarrow (y' = 100)) \wedge (b'_3 \Leftrightarrow (z' > 100))$$

The above equation has 4 satisfying assignments for b_1, b'_2, b'_3. The abstraction using C_1 produces 8 satisfying assignments for b_2, b_3, b'_1. Thus, the total number of satisfying assignments generated is 12. On the other hand, the monolithic approach will keep all the predicates together, resulting in 32 assignments. This example shows the advantage of predicate partitioning. In this example, it turns out that even with partitioning the abstraction obtained is same as that computed by the monolithic approach. Since cone partitioning attempts to keep all related predicates together, the abstractions produced are not much coarser than those produced by the monolithic approach. However, in general there is no bound on the number of predicates in a given cluster. In the worst case there might be a cluster containing most of the current-state and next-state predicates.

4.2 Syntactic Partitioning for Lazy Abstraction

The idea of lazy abstraction [17] is to defer the abstraction until required by a spurious counterexample. We therefore use a very inexpensive syntactic partitioning to compute a very coarse initial abstraction. This is done to compute initial abstractions of large circuits quickly.

There are many ways to perform a partitioning for lazy abstraction. One simple technique is to create k clusters, each containing exactly one current-state predicate π_i. We follow a variant of this technique: all current-state predicates that contain the same set of variables are kept in the same partition. That is, if $var(\pi_i) = var(\pi_j)$, then π_i and π_j are kept in the same partition. This is useful if the given set of predicates contains many mutually exclusive (or related) predicates such as $x = 1, x = 2, x = 3$. Keeping these predicates in separate clusters will result in an exponential number of contradicting abstract states, such as an abstract state in which both $x = 1$ and $x = 2$ are true. The next-state predicates are not used for computing the abstraction.

As an example, let the set of current-state predicates be $\{x < 200, x = 100, y = 100, z > 100\}$. The clusters produced for lazy abstraction are $C_1 = \{x < 200, x = 100\}$, $C_2 = \{y = 100\}$, $C_3 = \{z > 100\}$.

Once the abstraction of the concrete system is obtained, we model-check it using the NuSMV model-checker. Fig. 3 shows an abstract model. If the abstract model satisfies the property, the property also holds on the original, concrete circuit. If the model checking of the abstraction fails we obtain a counterexample from the model-checker. In order to check if an abstract counterexample corresponds to a concrete counterexample, a *simulation* step is performed. This is done using the standard technique of bounded model checking [5]. If the counterexample cannot be simulated on the concrete model, it is called a *spurious counterexample*. The elimination of spurious counterexamples from the abstract model is described in the next section.

5 Abstraction Refinement

5.1 Spurious Transitions and Spurious Prefixes

When refining the abstract model, we distinguish between two cases of spurious behavior, as done in [12]:

1. **Spurious transitions** are abstract transitions which do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise abstraction as computed by the monolithic approach. However, as we noted earlier, computing the most precise abstract model is expensive and thus, we make use of the various partitioning techniques. These techniques can typically result in many spurious transitions.

2. **Spurious prefixes** are prefixes of the abstract counterexample that do not have a corresponding concrete path. This happens when the set of predicates is not rich enough to capture the relevant behaviors of the concrete system, even for the most precise abstraction.

In contrast to SLAM, we first check whether any transition in the abstract trace is spurious or not. If a spurious transition is found, it is eliminated from the abstract model by adding a constraint. If all the transitions in the abstract trace are not spurious, then new predicates are generated by computing the weakest precondition of the given property with respect to the transition function of the circuit. Fig. 5 shows how our abstraction and refinement loop differs from that of SLAM.

An abstract counterexample is a sequence of abstract states $\bar{s}(1), \ldots, \bar{s}(l)$, where each abstract state $\bar{s}(j)$ corresponds to a valuation of the k predicates $\pi_1, \ldots, \pi_k$. The value of π_i in a state $\bar{s}$ is denoted by $\bar{s}_i$. We use π_i' to denote the next state version of π_i.

In order to check if an abstract transition $\bar{s}$ to $\bar{t}$ can be simulated on the concrete model, we create a SAT instance given by the following equation:

$$\bigwedge_{i=1}^{k} \pi_i = \bar{s}_i \;\wedge\; R(\bar{r}, \bar{r}') \;\wedge\; \bigwedge_{i=1}^{k} \pi_i' = \bar{t}_i$$

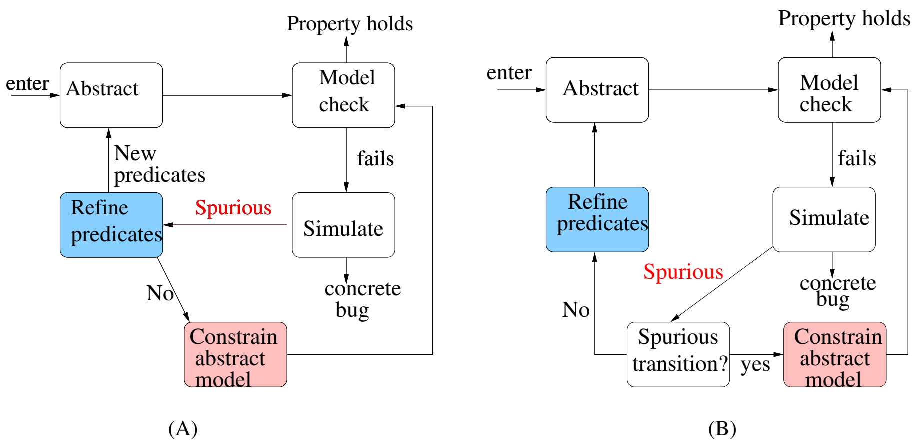

Figure 5: Abstraction refinement loop in (A) SLAM, (B) in [12] and this paper.

The equation above is transformed into CNF and passed to a SAT solver. If the SAT solver detects the equation to be satisfiable, the abstract transition can be simulated on the concrete model. Otherwise, the abstract transition is spurious. In this case, the spurious transition can be removed from the abstract model by adding a constraint to the abstract model.

5.2 Refining spurious transitions

When generating the CNF instance for the simulation of the abstract transition $\bar{s}$ to $\bar{t}$, we store the mapping of each predicate π_i, π'_i to the corresponding literal l_i, l'_i in the CNF instance. If the abstract transition is spurious, the CNF instance is unsatisfiable. In this case, we make use of the ZChaff SAT solver [25] for extracting an unsatisfiable core from the given CNF instance. An *unsatisfiable core* of a CNF instance is a subset of the original set of clauses that is also unsatisfiable. Current state-of-the-art SAT-solvers like ZChaff [23] are quite effective at producing very small unsatisfiable cores.

Let us denote the set of current-state predicates whose corresponding CNF literal l_i appears in the unsatisfiable core as X. We have a similar set for the next-state predicates, which we call Y. Intuitively, the predicates in X and Y taken together are sufficient to prove that the abstract transition $\bar{s}$ to $\bar{t}$ is spurious. All the abstract transitions where the predicates in X and Y have the same truth value as given by the states $\bar{s}$ and $\bar{t}$, respectively are spurious. These spurious transitions are eliminated by adding a constraint to the abstract model. Let b_i and b'_i be the variables used for the predicates π_i and π'_i in the abstract model. The constraint added to the abstract model is as follows:

$$\neg\left(\bigwedge_{\pi_i \in X} b_i = \bar{s}_i \;\wedge\; \bigwedge_{\pi'_i \in Y} b'_i = \bar{t}_i\right)$$

Example: Consider the abstract transition from $\bar{s} = \{b_1 = 0, b_2 = 0, b_3 = 1, b_4 = 1\}$ to $\bar{t} = \{b'_1 = 0, b'_2 = 0, b'_3 = 0, b'_4 = 0\}$, where b_1, b_2, b_3, and b_4 correspond to the predicates $x = 1, x = 2, y = 1, y = z$, respectively. Let the next state functions be $x' = z$, $y' = x$, $z' = y$. Observe that in the state $\bar{s}$, the predicates $y = 1$ and $y = z$ are true and thus, $z = 1$. This implies that $x' = 1$ and b'_1 must hold in $\bar{t}$. However, b'_1 is false in $\bar{t}$ and thus, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. This counterexample can be eliminated by adding the following constraint to the abstract model:

$$\neg(\neg b_1 \wedge \neg b_2 \wedge b_3 \wedge b_4 \wedge \neg b'_1 \wedge \neg b'_2 \wedge \neg b'_3 \wedge \neg b'_4)$$

However, the above constraint removes just one spurious transition. By making use of an unsatisfiable core, we can make the constraint more general, thereby eliminating many spurious transitions at the same time. In this example, the cause of the spurious behavior is due to $b_3 = 1, b_4 = 1$, and $b'_1 = 0$. The unsatisfiable core technique described above will discover this fact. Now we can eliminate this abstract counterexample and 31 more spurious transitions by adding the following constraint to the abstract model:

$$\neg(b_3 \wedge b_4 \wedge \neg b'_1)$$

In practice, we observe that the constraints generated using the unsatisfiable cores are very small (of size 5 to 6), as compared to the total number of current-state and next-state predicates. Thus, this technique is very effective in removing multiple spurious transitions.

5.3 Refining spurious prefixes

In [12], the elimination of spurious prefixes is done by adding a monolithic bit-level predicate. This predicate is called a *separating* predicate and is computed by using a SAT based conflict dependency analysis. In contrast, we make use of weakest preconditions as done in software verification. We generate new word-level predicates from the weakest pre-condition of the given property with respect to the transition function given by the RTL level circuit.

Weakest pre-conditions In software verification, the weakest pre-condition $wp(st, \gamma)$ of γ is usually defined with respect to a statement *st* (e.g., an assignment). It is the weakest formula whose truth before the execution of *st* entails the truth of γ after *st* terminates. In case of hardware, each state transition can be viewed as a statement where the registers are assigned values according to their next-state functions.

Recall that the set of registers that have a next-state function is denoted by $\mathcal{Q}$. For example, external inputs do not appear in this set. The next-state function for register $r_i \in \mathcal{Q}$ is given by $f_i(\bar{r})$. We use $\bar{f}$ to denote the vector of the next state functions for the registers in $\mathcal{Q}$. For any expression e, the expression $e[\bar{x}/\bar{y}]$ denotes the simultaneous substitution of each x_i in e by y_i from $\bar{y}$. Note that x_i and y_i might themselves be expressions.

The weakest precondition of the property $\gamma(\bar{r})$ with respect to one concrete transition is defined as follows:

$$wp_1(\bar{f}, \gamma(\bar{r})) \quad := \quad \gamma(\bar{r})\ [\bar{r}/\bar{f}]$$

The weakest precondition with respect to i consecutive concrete transitions is defined inductively as follows:

$$wp_i(\bar{f},\, \gamma)) \quad := \quad wp_1(\bar{f},\, wp_{i-1}(\bar{f},\, \gamma)) \;\; (i > 1)$$

In order to refine a spurious prefix of length $l > 0$, we compute $wp_l(\bar{f},\gamma)$. Intuitively, γ holds iff $wp_l(\bar{f},\gamma)$ holds before after l transitions. Refinement corresponds to adding the boolean expressions occurring in $wp_l(\bar{f},\gamma)$ to the existing set of predicates. The abstraction created with respect to the new set of predicates results in a model that does not contain this spurious prefix.

In case of circuits the weakest pre-condition is always computed with respect to the same transition function $\bar{f}$ and thus, we may omit it as an argument in $wp_i(\bar{f},\gamma)$.

Example: Let the property be $x < 200$. Let the next state functions for the registers x and y be $((x < 100)?(x+y) : x)$ and x, respectively. Suppose we obtain an spurious prefix of length equal to 1. The weakest pre-condition computed is given as follows:

$$wp_1(x < 200) \quad := \quad (((x \;<\; 100)\;?\;(x+y)\;:\;x\;) <\; 200)$$

We add the boolean conditions occurring in wp_1 to our set of predicates. Thus, we add $x < 100$ and $(((x \;<\; 100)\;?\;(x+y)\;:\;x\;) <\; 200)$ as the new predicates.

Simplifying the weakest pre-conditions The problem with the approach above is that the predicates generated can become very complex when the spurious prefix is large. This will adversely affect the future iterations of the abstraction refinement loop. In software verification, this problem is solved by computing the weakest pre-condition with respect to the statements appearing in the spurious counterexample trace. In our case, this amounts to simplifying the weakest pre-conditions at each step.

We exploit the fact that many of the control flow guards in the Verilog are also present in the current set of predicates. The abstract trace assigns truth values to these predicates in each abstract state. In order to simplify the weakest pre-conditions, we substitute the guards in the weakest pre-conditions with their truth values. Furthermore, we do not add all the boolean expressions occurring in the weakest pre-condition as the new predicates.

Let $g(\bar{r})$ be a boolean expression. We denote the set of conditions (guards) occurring in g by $\mathcal{G}$. For example, the set of conditions in $(((x \;<\; 100)\;?\;(x+y)\;:\;x\;) <\; 200)$ is $\{x < 100\}$. The conditions in g that also occur in the current set of predicates is given by $\mathcal{G}' = \mathcal{G} \cap \{\pi_1, \ldots, \pi_k\}$.

Let *simplify* be a function that takes as input a boolean formula $g(\bar{r})$ and an abstract state $\bar{t}$. This function returns another boolean formula $g'(\bar{r})$, where all the guards in $\mathcal{G}'$ are substituted by their truth values in the state $\bar{t}$. Intuitively, this amounts to following the control flow inside $g(\bar{r})$ using the truth value of guards from the state $\bar{t}$. Formally, we have the following definition:

$$\mathit{simplify}(g(\bar{r}),\bar{t}) \quad := \quad g(\bar{r})[\pi_i/\bar{t}_i] \quad (\forall \pi_i \in \mathcal{G}')$$

Example: Suppose our current set of predicates is $\{x < 200, x < 100\}$. Let $\bar{t}$ be an abstract state in which $x < 200$ is false and $x < 100$ is true. Let $g(x,y)$ be the formula $(((x\ <\ 100)\ ?\ (x+y)\ :\ x\ ) <\ 200)$. In this example, $\mathcal{G} = \mathcal{G}' = \{x\ <\ 100\}$. The call to simplify with $g(x,y)$ and $\bar{t}$ as the arguments will return:

$$((1?\ (x+y)\ :\ x\ ) <\ 200) \quad = \quad x+y < 200$$

Let the spurious prefix be $\bar{t}(0), \ldots, \bar{t}(l)$ with $l \geq 1$ and let the property be γ. The weakest precondition wp_i is the formula that should hold before i concrete transitions. The abstract state $\bar{t}(l-i)$ provides the truth values for the predicates just before the these i transitions. Thus, $wp_i(\gamma)$ is simplified using the predicate values from the abstract state $\bar{t}(l-i)$. Formally, the *simplified* version of the weakest pre-conditions is defined as follows:

$$\begin{array}{rcl} wp_1(\gamma) & := & \mathit{simplify}(\gamma\,[\bar{r}/\bar{r}']\,[\bar{r}'/\bar{f}],\ \bar{t}(l-1)) \\ wp_i(\gamma) & := & \mathit{simplify}(wp_1(wp_{i-1}(\gamma)),\ \bar{t}(l-i)) \quad (1 < i \leq l) \end{array}$$

The new set of predicates for refinement is obtained from wp_l. This is done by taking only the guards of the trinary conditional operator, and other predicates not containing the conditional operator.

Example: We continue our example in Fig. 1. We want to prove the property that always globally $x < 200$. In Fig. 3, an abstraction of this program using three predicates $x < 200, x < 100, x+y < 200$ is presented. The property $\mathbf{AG}(x < 200)$ is proved using this abstraction. We now describe how these predicates are discovered automatically.

We take the predicates occurring in the property itself as the initial set of predicates. Thus, our initial abstraction is created with respect to the predicate $x < 200$ only. Model-checking the abstract model produces a counterexample of length one. It turns out that this counterexample is a spurious prefix with length one. The weakest pre-condition wp_1 of $x < 200$ is given as follows:

$$wp_1(x < 200) \quad = \quad (((x\ <\ 100)\ ?\ (x+y)\ :\ x\ ) <\ 200)$$

The only new predicate obtained from $wp_1(x < 200)$ is $x < 100$. Note that we do not take the entire weakest precondition as a new predicate. The new set of predicates is $\{x < 200, x < 100\}$. Once again, the abstraction and model-checking step is carried out. This time we obtain another spurious prefix of length one. We also obtain the truth value of the predicate $x < 100$ in the abstract states $\bar{t}(0)$ and $\bar{t}(1)$. Suppose the predicate $x < 100$ is true in $\bar{t}(0)$. The simplified weakest pre-condition obtained is given as follows:

$$wp_1(x < 200) \quad := \quad ((1?\ (x+y)\ :\ x\ ) <\ 200) = x+y < 200$$

Simplifying $wp_1(x < 200)$ yields a new predicate $x+y < 200$. Note that this predicate is not present as a guard in the program, nor in the property. Using the new set of predicates $\{x < 200, x < 100, x+y < 200\}$, we obtain the abstraction shown in Fig. 3. The abstract property holds on this abstraction and thus, $\mathbf{AG}(x < 200)$ holds on the concrete program in Fig. 1.

Benchmark	Lines of code	Latches	Variable Bits
`cache coherence (cc2)`	549	43	170
`mpeg`	1215	567	800
`SDLX`	898	41	81
`Miim`	841	83	237
`PI-Bus (pi)`	1020	312	863

Table 1: Benchmark characteristics

6 Experimental Results

We report experimental results for the Texas97 and VIS [24] benchmark suite to evaluate the performance of various techniques for predicate partitioning and the abstraction refinement. The experiments are performed on a 1.5 GHZ AMD machine with 3 GB of memory running Linux. A time limit of two hours was set for each run. The user only needs to provide the Verilog file and the property to be checked as the input. All the phases of the CEGAR loop, namely abstraction, model checking, simulation, and refinement are completely automatic.

The benchmark characteristics are given in table 1. We report the number of lines of code, the total number of latches, and the total number of Verilog variable bits (combinational elements and inputs) for each benchmark. The experimental results are summarized in table 2. For each algorithm, it contains the total run-time, the final number of predicates, and the total number of refinement iterations. Table 6 gives the breakup of total run-time for the experiments in table 2 in terms of the time spent on the abstraction computation, model checking time and the time spent on the simulation and refinement.

We report these results for four different algorithms. The columns marked with ”Monolithic” contain the results for a precise existential abstraction without any partitioning. All experiments use refinement with weakest preconditions.

The columns labeled with ”CONE” contain results using cone partitioning for the set of predicates. The predicates are partitioned into clusters of related current-state and next-state predicates.

The performance of abstraction refinement with lazy abstraction is summarized in the columns labeled with ”Lazy”. Initially, the abstraction is performed by keeping only a few next state predicates, which share exactly the same set of latches and other variables, together.

The columns labeled with ”Constrain” contain the results for abstraction refinement where a check for spurious transitions is added. If a spurious transition is detected in the abstract counterexample, it is eliminated by directly constraining the abstract model. As discussed earlier, we make use of unsatisfiable cores for this step. The check is performed before refinement using weakest preconditions is done.

Summary of Results The `cache coherence (cc2)` benchmark is the design of a two processor write-back-cache system. In this benchmark, the best performance is obtained by partitioning the predicates using the CONE technique. Both CONE and CONE+Constrain have similar performance on this benchmark. Using lazy abstraction, a high number of refinement iterations is observed, as the lazy abstraction is creating overly coarse abstractions.

The `mpeg` benchmark is a design of an MPEG decoder and contains a decoder which accepts a bitstream as an input and produces a video/audio data stream as output. We verify two different safety properties (`mpeg1` and `mpeg2`) for the mpeg benchmark. The monolithic approach of keeping all the predicates together results in the best performance on `mpeg1`. This is due to the fact that only nine predicates are required to prove the property. However, in `mpeg2`, generating the most precise abstract transition relation becomes the bottleneck. For `mpeg2`, CONE+Constrain results in the best performance. Note the consistent performance of CONE+Constrain on both `mpeg1` and `mpeg2`.

The `Miim` benchmark contains the design of an Ethernet core that implements the network protocols CSMA/CD for transmission and reception of frames. Only Lazy+Constrain is able complete this example within the timeout. All other techniques fail because they are unable to compute an abstract transition relation. In this case, the cone partitioning results in almost no reduction; most of the predicates are put in one partition.

The `SDLX` benchmark is the design of a sequential DLX processor that uses a load-store architecture. The cone partitioning and refinement using weakest preconditions has the best runtime. The performance of the Lazy+Constrain is the worst of all. Since the abstraction produced are very coarse most of the time is spent on refinement.

The Peripheral Interconnect bus (PI-Bus) is a high speed industry standard on-chip bus for use on micro-controllers and systems on a chip. Since a very coarse abstraction with no refinement is sufficient for proving the property, lazy abstraction has the best runtime. All other techniques create a more precise abstraction, which is not needed in this case.

Benchmark	Monolithic			CONE			CONE+Constrain			Lazy+Constrain		
	T	P	I	T	P	I	T	P	I	T	P	I
`cc`	1741s	23	3	390s	47	4	397s	47	4	885s	47	119
`mpeg1`	35s	9	2	855s	41	3	130s	9	4	406s	9	15
`mpeg2`	2945s	24	3	325s	35	4	208s	24	4	1951s	24	37
`SDLX`	1141s	25	1	61s	39	2	223s	39	2	5876s	39	133
`Miim`	$>$2h	25	3	$>$2h	25	3	$>$2h	25	3	70s	25	27
`pi`	58s	10	1	21s	10	1	21s	10	1	12s	10	1

Table 2: Experimental results: The ”T” columns contain the total runtime in seconds, the ”P” columns show the final number of predicates, and the ”I” columns contain the total number of refinement iterations. ”CONE” denotes cone predicate partitioning, ”Constrain” is refinement of spurious transitions using UNSAT cores. For the entries where the timeout occurred we report the number of predicates and the number of iterations completed before the timeout.

Bench-mark	Monolithic			CONE			CONE+Constrain			Lazy+Constrain		
	Abs	MC	SR	Abs	MC	SR	Abs	MC	SR	Abs	MC	SR
`cc`	1701	12	28	269	35	87	269	34	95	23	37	825
`mpeg1`	18	1	16	810	1.7	43	29	0.3	101	24	1	381
`mpeg2`	2520	380	46	251	0.7	74	113	0.4	95	67	4.6	1881
`SDLX`	1134	7	0	51	1.6	9	53	34	136	48	2788	3039
`Miim`	-	-	-	-	-	-	-	-	-	12	1.5	56
`Pi`	58	0	0	21	0	0	21	0	0	12	0	0

Table 3: Experimental results: All times are reported in seconds. The "Abs" columns contain the time spent in computing the abstraction, the "MC" columns show the time spent on model checking the abstract model, and the "SR" is the time spent during simulation and refinement. "CONE" denotes cone predicate partitioning, "Constrain" is refinement of spurious transitions using UNSAT cores. A dash "- " indicates a time-out of 2 hours.

7 Conclusions and Future Work

While there are a lot of results on predicate abstraction in the software domain, there is only little research on predicate abstraction in the hardware domain. This paper evaluates three methods to improve the performance of SAT-based predicate abstraction on circuits. The methods have been presented before in the context of abstraction of ANSI-C programs.

When abstracting a basic block, tools like SLAM do not consider all possible abstract transitions. Instead, subsets of the predicates are formed. This reduces the computational effort during abstraction, but may result in additional spurious behavior. Experimental results on Verilog circuits show that this technique is also useful in the hardware context.

In [17], the authors propose to defer the expensive task of program abstraction until a spurious counterexample is found. This is called lazy abstraction. We evaluate the benefit of lazy abstraction in the context of circuits. It turns out that refining the coarse abstractions produced during lazy abstraction can become a bottleneck. We make use of unsatisfiable cores in order to eliminate multiple spurious transitions.

However, the spurious trace may also be caused by insufficient predicates. In the software domain, tools typically use weakest preconditions to compute new predicates that eliminate the spurious behavior. This technique has previously not been applied to hardware, despite of the fact that high-level RTL closely resembles languages like ANSI-C. Our experimental results show that this technique is very effective in discovering new word-level predicates for refinement.

Future research will focus on use of unsatisfiable cores for discovering new predicates. We would like to experiment with the use of interpolants [18] for deriving new predicates. This paper used syntactic techniques for predicate partitioning, the use of semantic techniques for this purpose is yet to be investigated.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

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

[3] Thomas Ball, Byron Cook, Satyaki Das, and Sriram Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer-Verlag, 2004.

[4] Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification*. Springer-Verlag, 2004.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

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

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

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

[10] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[11] Edmund Clarke, Daniel Kroening, Natalia Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. In *Proc. of the Model Checking for Dependable Software-Intensive Systems Workshop, San-Francisco, USA*, 2003.

[12] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based predicate abstraction for hardware verification. In *Proceedings of SAT'03*, 2003.

[13] M. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.

[14] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.

[15] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.

[16] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–244. ACM Press, 2004.

[19] *IEEE Standard Verilog Hardware Description Language*. IEEE Standard 1364. IEEE Computer Society Press, 2001.

[20] H. Jain, D. Kroening, and E.M. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE 2004*. IEEE, 2004.

[21] Daniel Kroening, Joel Ouaknine, Sanjit Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Computer Aided Verification*, 2004.

[22] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[23] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[24] http://vlsi.colorado.edu/∼vis.

[25] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer-Verlag, 2003.