Word Level Predicate Abstraction and Refinement Techniques for Verifying RTL Verilog

Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke, *Fellow, IEEE*

***Abstract*—As a first step, most model checkers used in the hardware industry convert a high-level register transfer level (RT-level/RTL) design into a netlist. However, algorithms that operate at the netlist level are unable to exploit the structure of the higher abstraction levels, and thus, are less scalable. The RT-level of a hardware description language such as Verilog is similar to a software program with special features for hardware design such as bit-vector arithmetic and concurrency.**

This paper uses predicate abstraction, a software verification technique, for verifying RTL Verilog. There are two challenges when applying predicate abstraction to circuits: 1) The computation of the abstract model in presence of a large number of predicates, and 2) the discovery of suitable word-level predicates for abstraction refinement. We address the first problem using a technique called *predicate clustering*. We address the second problem by computing *weakest preconditions* of Verilog statements in order to obtain new word-level predicates during abstraction refinement. We compare the performance of our technique with localization reduction, a netlist level abstraction technique, and report significant improvements on a set of benchmarks.

***Index Terms*—Register transfer level (RTL), verification, model checking, predicate abstraction, refinement, satisfiability (SAT).**

I. Introduction

MOST new hardware designs are implemented at a high level of abstraction, e.g., using the *register transfer level (RT-level/RTL)*, or even at the system level. The RT-level of a hardware description language such as Verilog is very similar to a software program in ANSI-C, and offers special features for hardware designers such as bit-vector arithmetic and concurrency. However, most formal verification tools used in the hardware industry still operate on a low-level design representation called a *netlist*. This is due to lack of automated verification techniques at the RT-level. Converting a high-level RTL design into a netlist results in a significant loss of structure present at the RT-level. This makes verification at the netlist level less scalable.

Manuscript received July 27, 2006; revised November 6, 2006, March 27, 2007, July 03, 2007. This paper is an extended version of [27]. This research was sponsored by the Gigascale Systems Research Center (GSRC), the National Science Foundation (NSF), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), the Defense Advanced Research Projects Agency, the Army Research Office (ARO), and the General Motors Collaborative Research Lab at CMU. This paper was recommended by Associate Editor Andreas Kuehlmann.

H. Jain and E. M. Clarke are with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 15232 USA.

D. Kroening is with the Computer Systems Institute, ETH Zürich, Switzerland.

N. Sharygina is with the Informatics Department, University of Lugano, Switzerland.

A. Model Checking and Abstraction

Model checking [12], [14] is an automatic technique for the verification of concurrent systems. It has been used successfully in practice to verify complex circuit designs and communication protocols. Model checking systematically explores the state space of a given design and checks that each reachable state satisfies the property of interest. When the design fails to satisfy a desired property, the process of model checking produces a *counterexample* that demonstrates a behavior which falsifies the property. The properties (formal specifications) are usually described in linear temporal logic (LTL) or computational tree logic (CTL). By making use of *symbolic* algorithms [9], [5] based on Binary Decision Diagrams (BDDs) [8] and fast satisfiability solvers (SAT solvers) [30], [33], [32], current model checkers can handle many industrial designs.

The number of states in industrial hardware designs is extremely large. This often results in exorbitant resource requirements during model checking even when symbolic model checking algorithms are used. One principal method for 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.

We focus on abstraction techniques that produce a *conservative* over-approximation of the concrete system. This implies that if the abstraction satisfies a given property, the property also holds on the original concrete system. When model checking of the abstraction fails, it produces an *abstract counterexample*. The drawback of the conservative abstraction is that an abstract counterexample may not correspond to any concrete counterexample (real error). This is usually called a *spurious counterexample* [10].

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. As in *bounded model checking* (BMC) [5], 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. The Boolean formula is then checked for satisfiability using a SAT procedure such as MiniSat [32]. If the instance is satisfiable, the counterexample is real and the procedure terminates. If the instance is unsatisfiable, the abstract counterexample is spurious, and *abstraction refinement* has to be performed.

The basic idea of abstraction refinement techniques is to create a new abstract model that contains more detail in

order to prevent the spurious counterexample [28], [10], [3]. 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 [10].

B. Abstraction Techniques for Software Verification

In the software domain, *predicate abstraction* [22] has emerged as a successful technique for verifying large systems. 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 of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [3] and promoted by the success of the SLAM project. The goal of SLAM is to verify that Windows device drivers obey API conventions. The abstraction is computed using a theorem prover.

C. Abstraction Techniques for Hardware Verification

Most model checkers used in hardware verification operate on a very low level design, usually a netlist. At the netlist level, the most commonly used abstraction technique is *localization reduction* [28], [40], [23]. The abstract model is created from the given circuit by removing a large number of 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 the *visible latches*. The initial abstract model is created by making the latches present in the property visible, and the rest invisible. The refinement is done by moving more latches from the set of invisible latches to the set of visible latches.

Clarke et al. [16] introduce a SAT-based technique for predicate abstraction of netlist level circuits. The use of a SAT solver like zChaff [33] in order to perform the abstraction allows precise modeling of bit-vector semantics. However, their approach suffers from two drawbacks. 1) Each transition in the abstract model is computed by a separate run of the SAT solver. Thus, the learning done by a SAT solver in the form of conflict clauses is lost when computing other transitions in the abstract model. 2) If refinement becomes necessary, only bit-level predicates are introduced. This method of refinement closely resembles refinement techniques for localization reduction.

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 registers, 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, localization reduction typically turns all bits of the two registers into visible latches, and thus, the abstraction is identical to the original model.

Predicate abstraction is only effective if the predicates can cover the relationship between multiple latches. This typically requires a word-level model given in RT-level of a hardware description language. RT-level models are similar to programs written in a language, such as ANSI-C. We apply predicate abstraction to word-level models given in RTL Verilog.

Software verification tools use theorem provers for computing the predicate abstraction. Theorem provers model the variables using unbounded integers. Overflow or bit-wise operators are not modeled. However, hardware description languages like Verilog provide an extensive set of bit-wise operators. For hardware designs, the use of these bit-level constructs is ubiquitous. As in [13], we use a bit-level SAT solver to compute the abstract transition relation. This allows us to precisely model the bit-vector semantics of hardware designs during abstraction computation.

We view our technique as a word-level verification technique since the predicates that are used for computing the abstraction are at the word-level. The abstract model contains the relationships between the word-level predicates and not the individual latches. The use of a bit-level SAT solver as a decision procedure can be replaced by a word-level solver. Such a solver eliminates or reduces the need to flatten a given formula to the bit-level. However, existing word-level solvers for hardware description languages are not yet competitive with bit-level SAT solvers.

D. Contribution

This paper applies predicate abstraction and refinement for verifying circuits given in RTL Verilog. Two problems arise when applying predicate abstraction to circuits: 1) The computation of the abstract model in presence of a large number of predicates, and 2) the discovery of suitable word-level predicates for abstraction refinement.

In order to address the first problem, we divide the 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 more efficient. We refer to this technique as *predicate clustering*. This technique allows us to tune the abstraction step between the two extremes of eager abstraction [13] and lazy abstraction [25]. The eager technique refers to the case when all predicates are within a single cluster, while lazy abstraction corresponds to the case in which many clusters of small cardinality (size) are used for computing the abstraction.

When refining the abstract model using a spurious counterexample, we distinguish between two cases of spurious behavior [16]: *Spurious transitions* are abstract transitions that do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise predicate abstraction, which is computed by the eager approach. However, predicate clustering usually produces coarse abstractions, which can give rise to spurious transitions. *Spurious prefixes* are prefixes of a spurious 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.

When a spurious counterexample is encountered, we first check whether each transition in the counterexample can be

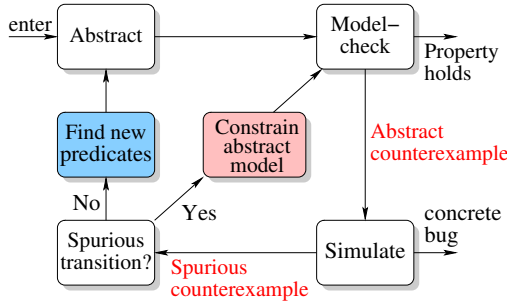


Fig. 1. Abstraction Refinement (CEGAR) Loop in this paper.

simulated on the original program. This is done by creating a SAT instance for the simulation of each abstract transition. If the SAT instance for an abstract transition is unsatisfiable, then the abstract transition is spurious. In this case, we refine the abstraction by adding constraints to the abstract transition relation, which eliminate the spurious transition. We make use of the proof of unsatisfiability of the SAT instance to identify a small subset of the existing predicates that cause the transition to be spurious. The fewer predicates that are found, the more spurious transitions that are eliminated in one step.

When all SAT instances for the simulation of abstract transitions are satisfiable, it means that none of the abstract transitions is spurious due to the clustering. The immediate conclusion is that the spurious counterexample is due to the fact that the predicates used for computing the abstraction were insufficient. For this case, we use the idea of *weakest precondition* from software model checking [34], [3]. We compute the weakest precondition of the property (or existing predicates) with respect to the transition function given by the circuit to obtain new word-level predicates. To the best of our knowledge, syntactic weakest precondition based refinement has not been used before for verifying circuits. We present a technique to avoid the blowup in the size of weakest preconditions when computing new predicates. The overall flow of the various techniques described above is shown in Fig. 1.

E. Further Related Work

Wang [41] proposes a combination of localization reduction and predicate abstraction at the netlist level. This is useful in some cases, for example, when we need to track the value of an n bit counter c in an abstract model precisely. In this case it is inefficient to introduce 2^n predicates of the form $c = v$, where $0 \leq v \leq 2^n - 1$. In localization reduction the value c can be tracked precisely by making each bit in c a visible latch. It is possible to get the benefits of localization reduction in our technique as well by adding $c[i]$ as a predicate.

Andraus et al. [2] present a scheme for automatic abstraction of behavioral RTL Verilog to the CLU language [7]. The CLU language allows modeling using terms, uninterpreted functions, equality, lambda expressions, and counters. In order to remove spurious behaviors from the abstract model a refinement procedure is described in [1]. The techniques in [2], [1] were shown to be useful in context of microprocessor correspondence checking. The techniques described in this paper are different from those in [2], [1] and are geared towards property (assertion) checking of hardware designs.

Predicate discovery for abstraction refinement is still an open area of research. We use weakest preconditions for discovering new predicates. This is sufficient for ensuring that the abstraction refinement loop makes progress. An alternative technique for discovering new predicates is based on interpolation [31]. In order to apply this idea to circuits, an interpolating theorem prover for bit-vector logic is required. At present, it is not known how to build such a prover for bit-vector logic.

A *Pre-image computation* generates a set of states from which it is possible to reach a given set of states with one transition. It is a basic operation in model checking [12] and target enlargement approaches [4]. The idea of computing a pre-image is the same as computing the weakest precondition of a given set of states, although the latter term is more commonly used in software verification. Most existing hardware model checkers compute the pre-image at the netlist level and represent it symbolically using BDDs. As in software verification our use of weakest preconditions or pre-images is at the word (expression) level.

Outline: We describe our way of modeling circuits in Section II. Section III describes SAT-based predicate abstraction with the help of an example. Techniques for clustering the given set of predicates are presented in Section IV. We discuss techniques for abstraction refinement in Section V. We report experimental results in Section VI.

II. Word-Level Transition Functions

Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ denote the set of registers and external inputs in a given Verilog program. For example, the state of the Verilog program in Fig. 2 is defined by the value of the registers x and y, and each of them has a storage capacity of 8 bits. Let S denote the set of states for a given Verilog program. Let $Q \subseteq \mathcal{R}$ denote the set of registers. We denote the next-state function of a register $r_i \in Q$ by $f_i(r_1, \ldots, r_n)$, or $f_i(\bar{r})$ using vector notation, where $\bar{r} = \langle r_1, \ldots, r_n \rangle$. The value of r_i in the next state is given by $f_i(\bar{r})$ as a function of the current state. We use the next-state functions to define the transition relation $R(\bar{r}, \bar{r}')$. It relates the current state $\bar{r} \in S$ to the next state $\bar{r}' \in S$ and is defined as follows:

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

The values of the external inputs ($\mathcal{R} \setminus Q$) are not constrained by the transition relation. The next-state function for the register x in Fig. 2 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 ternary choice operator $c?g : h$ to denote a function that evaluates to g if the condition c is true, and otherwise to h. We denote the next-state functions of x and y by $f_x(x,y)$ and $f_y(x,y)$, respectively, and the transition

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

  initial x = 1;
  initial y = 0;

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

Fig. 2. A Verilog program used as a running example.

relation by $R(x,y,x',y')$.

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

In a *netlist* level representation there is a next-state function for each bit in the registers `x,y`. In contrast, we have a next-state function for the whole registers `x,y` and not for the individual bits in `x,y`. We represent the circuit using *register-level* or *word-level* next-state functions. We formalize the semantics of the subset of Verilog that we handle in a technical report [15].

Example: We use the Verilog program in Fig. 2 as a running example. We wish to show that the value of `x` is always less than 200. That is, we want to prove that the given program satisfies the safety property $\mathbf{G}(x < 200)$, where $\mathbf{G}$ is an LTL operator [12] that stands for *globally*. 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. 3.

We follow the counterexample guided abstraction refinement (CEGAR) framework in order to (dis)prove a given property. The first step of the CEGAR loop is to obtain an abstraction of the given program.

III. Predicate Abstraction

Let S denote the set of all possible valuations of the program variables. In predicate abstraction [22], the variables of the concrete program are replaced by Boolean variables. Each Boolean variable corresponds to a predicate on the variables in the concrete program. Predicates are functions that map concrete states $\bar{r} \in S$ to 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 $\bar{b}$. 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 construct an existential abstraction [11], 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 $\hat{R}$.

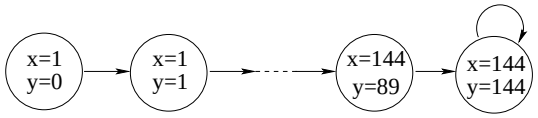


Fig. 3. The state transition graph of the Verilog program in Fig. 2.

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

We refer to a set and its Boolean representation interchangeably. For example, in the above equation $\hat{R}$ denotes a set of abstract transitions. A Boolean (characteristic) function representing this set is denoted as $\hat{R}(\bar{b},\bar{b}')$.

The initial state $I(\bar{r})$ is abstracted as follows: an abstract state $\bar{b}$ is an initial state in the abstract model if there exists a concrete state $\bar{r}$ that is an initial state in the concrete model and is abstracted to $\bar{b}$.

$$\hat{I}(\bar{b}) := \exists \bar{r} \in S:\ (\alpha(\bar{r}) = \bar{b}\ ) \wedge\ I(\bar{r}) \quad (2)$$

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 $\bar{b}$.

$$\hat{P}(\bar{b}) := \forall \bar{r} \in S:\ (\alpha(\bar{r}) = \bar{b}) \Longrightarrow P(\bar{r}) \quad (3)$$

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

The techniques described in the paper can be used to check any LTL safety property. This is because the spurious counterexamples for LTL safety properties are always finite acyclic paths [17]. Such spurious counterexamples can be removed during the refinement phase (Section V). Predicate abstraction can also be used to verify an arbitrary LTL property, including liveness properties, if the transition relation is total. However, this requires removal of counterexamples containing loops and is left for future research.

A. SAT-based Predicate Abstraction

In [13], a SAT solver is used 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 Verilog programs. We describe the computation of the abstract transition relation $\hat{R}$ (Eqn. 1) in more detail below.

Computing $\hat{R}$ using SAT: A symbolic variable b_i is associated with each predicate π_i. Each concrete state $\bar{r} = \langle r_1, \ldots, r_n \rangle$ maps to an abstract state $\bar{b} = \langle b_1, \ldots, b_k \rangle$, where $b_i = \pi_i(\bar{r})$. If the concrete machine makes a transition from state $\bar{r}$ to state $\bar{r}' = \langle r'_1, \ldots, r'_n \rangle$, then the abstract machine makes a transition from state $\bar{b}$ to $\bar{b}' = \langle b'_1, \ldots, b'_k \rangle$, where $b'_i = \pi_i(\bar{r}')$. We refer to $\pi_i(\bar{r})$ as a *current-state* predicate and $\pi_i(\bar{r}')$ as a *next-state* predicate. For example, if $x = y$ is a current-state predicate, then the corresponding next-state predicate is $x' = y'$.

The formula that is passed to the SAT solver directly follows from the definition of the abstract transition relation $\hat{R}$ as given in Eqn. 1:

$$\hat{R} := \{(\bar{b},\bar{b}') \mid \exists \bar{r},\bar{r}'\ :\ \Gamma(\bar{r},\bar{r}',\bar{b},\bar{b}')\}\ \text{, where} \quad (4)$$

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

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 that assign the same values to $\bar{b}$ and $\bar{b}'$, and re-start the solver. This process is continued until the SAT formula becomes unsatisfiable. The disjunction of the abstract transitions obtained gives us the abstract transition relation $\hat{R}$.

The predicates used for abstraction are arbitrary Boolean expressions allowed by the Verilog syntax. Thus, the predicates can involve operators for concatenation, extraction, and so on. For example, `a[3:0]>7` and `ram[{addr,1'b0}]==d[9:2]` are allowed as predicates. Predicates can refer to individual bits in a register. For example, `rg[i]` is a valid predicate, where `rg` is a register and `i` is an index.

Example: We continue our example based on Fig. 2. Assume that $\{x<200, x<100, x+y<200\}$ is the set of predicates. We associate symbolic variables b_1, b_2, b_3 with each predicate, respectively. In order to compute $\hat{R}$ the following equation is converted to CNF and passed to a SAT solver:

$$(b_1 \Leftrightarrow (x<200)) \ \wedge \ (b_2 \Leftrightarrow (x<100)) \wedge$$
$$(b_3 \Leftrightarrow (x+y<200)) \wedge R(x,y,x',y') \ \wedge (b_1' \Leftrightarrow (x'<200)) \wedge$$
$$(b_2' \Leftrightarrow (x'<100)) \ \wedge \ (b_3' \Leftrightarrow (x'+y'<200))$$

The abstract transition relation obtained is given by the SMV [18], [35] `TRANS` statement in Fig. 4. It is a disjunction of cubes. The cube (`b1 & !b2 & !b3 & next(b1) & !next(b2) & !next(b3)`) corresponds to the transition from the abstract state in which b_1 is true and b_2, b_3 are false to the same abstract state ($100 \rightarrow 100$ 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. Subsequently, the value of the register x in the next state (x') is x and the values of the predicates $x<200$ and $x<100$ in the next state remain unchanged. The value of register y becomes equal to x (as $y' = x$). Since both x' and y' range between 100 and 200, $x'+y'$ can be greater than or equal to 200. Thus, the transition $100 \rightarrow 100$ is possible. All possible abstract transitions are shown explicitly in Fig. 5.

In Fig. 5, consider the abstract transitions from any state with $b_2 = 0$ to any state with $b_3 = 1$. There are four such transitions, namely $100 \rightarrow 101$, $101 \rightarrow 101$, $000 \rightarrow 001$, and $001 \rightarrow 001$. In these transitions, $b_2 = 0$ holds in the current abstract state, which means $x \geq 100$ in the concrete system. Thus, in the next state, $x' = y' = x$ holds. At first glance it seems that $x'+y'$ must be greater than or equal to 200 as $x' \geq 100$ and $y' \geq 100$. However, an overflow may occur during the addition of two 8-bit registers x' and y' such that the 8-bit result $x'+y'$ is less than 200. Thus, the predicate

```
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 G b1
```

Fig. 4. Abstraction of the Verilog program in Fig. 2 using the predicates $x<200$, $x<100$, and $x+y<200$. It is in the format accepted by the SMV model checker.

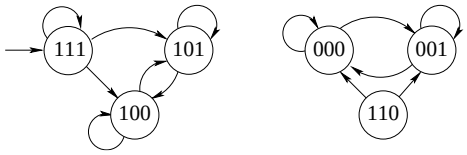


Fig. 5. Finite state machine for the abstract model in Fig. 4. The abstract states 010 and 011 are not possible, as this would require $x<200$ to be false and $x<100$ to be true at the same time.

$x'+y' < 200$ can be true. This explains why $b_2 = 0$ and $b_3' = 1$ in the four abstract transitions mentioned above. Note that such overflows are allowed by the semantics of hardware description languages and must be taken into account when computing the abstraction of hardware designs.

The set of abstract initial states (Eqn. 2) can be enumerated using a SAT solver in a similar manner as $\hat{R}$. The set of abstract initial states is given by the `INIT` statement in Fig. 4. There is only one abstract initial state in which all the Boolean variables b_1, b_2, b_3 are true.

The property $\mathbf{G}(x<200)$ is abstracted using the Boolean variable b_1 for the predicate $(x<200)$. The abstracted property is given by the `SPEC` statement in Fig. 4. The abstract model satisfies the property `G b1`, as the only states reachable from the initial abstract state (111) are $\{111, 101, 100\}$ (Fig. 5). Since the property holds on the abstract model, we can conclude that the property $\mathbf{G}(x<200)$ holds on the Verilog program in Fig. 2.

IV. Predicate Clustering

A. Computing Multiple Abstract Transition Relations

We call the computation of the exact existential abstraction as described in the previous section the *Eager Approach* (Eqn. 4). A single abstract transition relation is computed using all the available predicates. In the worst case, the number of satisfying assignments generated from Eqn. 4 is exponential in the number of predicates. In practice, computing abstractions using the eager approach can be very slow even for a small number of predicates.

The abstraction step can be accelerated if we do not aim at 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 eager approach. Software

predicate abstraction tools abstract the individual statements or basic blocks separately. As only a small number of predicates are typically affected at each statement or basic block, simple heuristics can be used to compute the abstraction quickly. The SLAM toolkit, for example, limits the number of predicates in each theorem prover query. In contrast, each transition in an RT-level circuit consists of simultaneous assignments to all registers. All predicates might change their value in each transition of the circuit. Thus, more sophisticated techniques are needed to compute the predicate abstraction of circuits efficiently.

Our solution to the above problem is as follows: the set of the predicates and their next-state state versions is clustered into smaller sets of related predicates. We call these sets *clusters*, and denote them by $C_1,\ldots,C_l$, with $C_j \subseteq \{\pi_1,\ldots,\pi_k,\pi'_1,\ldots,\pi'_k\}$. Note that we do not require the clusters to be disjoint, that is, they can have common predicates. We abstract the transition system with respect to each cluster $C_1,\ldots,C_l$. This results in a total of l abstract transition relations $\hat{R}_1,\ldots,\hat{R}_l$, which are conjoined to form $\hat{R}$:

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

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

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

The satisfying assignments to the equation above correspond to the abstract transition relation $\hat{R}_j$. The number of satisfying assignments is limited by the size of cluster C_j, that is, to at most $2^{|C_j|}$. Clearly, by limiting the size of C_j, we can compute the abstract transition relations much faster as compared to the eager approach.

We refer to the above technique of generating smaller clusters from a given set of predicates, and using these clusters for computing the abstraction $\hat{R}$, as *predicate clustering*. The following claim states that $\hat{R}$ is an over-approximation of the most precise predicate abstraction.

Proposition 1: If $\hat{Q}$ denotes the abstract transition relation obtained by using the eager approach (Eqn. 4), and $\hat{R}$ denotes the abstract transition relation obtained by predicate clustering (Eqn. 5), then $\hat{Q} \Rightarrow \hat{R}$ or $\hat{Q} \subseteq \hat{R}$ using set notation.

We present a proof in the appendix. We discuss techniques for creating predicate clusters next. Let $var(e)$ denote the set of variables (state elements and inputs) appearing in an expression e. For example, $var(x'+y' < 200)$ is $\{x',y'\}$. If e contains combinational elements, we replace them by their definitions in terms of state elements and inputs before computing $var(e)$.

Clarke et al. [10] call two formulas g_1 and g_2 interfering iff $var(g_1) \cap var(g_2) \neq \emptyset$. The authors use the notion of interference to partition a set of formulas into various formula clusters. This technique can be used for clustering the set of predicates as well. However, our early unreported experiments indicate that this results in clusters that are too large. Thus, we make the conditions for keeping the two predicates together stronger, which leads to a smaller number of predicates per cluster. We evaluate three different techniques for creating predicate clusters used in predicate clustering. Two of these techniques *cone clustering* and clustering for *lazy abstraction* are described below. The third clustering technique *semantic* predicate clustering is described in Section V-A.

B. Syntactic Cone Clustering

This technique clusters next-state predicates with current-state predicates that are related to each other. In order to identify when a next-state predicate is related to a current-state predicate we use a *cone-of-influence*-like computation [12].

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 $cone(g')$. 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 set of registers is denoted by Q. The next-state function of a particular register $r_i \in Q$ is given by $f_i(\bar{r})$.

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

The value of g' in a given state depends on the values of variables in $cone(g')$ from the previous state.

Example: Let g' be $a' < b'$. Let the next-state functions for a',b' be $x+b$, c, respectively. Here, $var(g') := \{a',b'\}$ and $cone(g') := \{x,b,c\}$. Given the values of x,b,c in a state, the value of the predicate $a < b$ in the next state (that is, the value of $a' < b'$) is $x+b < c$. We would like to keep the current-state predicates over the variables $\{x,b,c\}$ and the next-state predicate $a' < b'$ in the same cluster. This allows the value of $a' < b'$ to be tracked precisely in the abstract model.

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 two steps:

1) The next-state predicates that have identical cone sets are kept in a single cluster. Intuitively, these predicates depend on exactly the same set of variables from the previous state and hence, are related to each other. 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 transition relation $R(x,y,z,x',y',z')$ be $x' = y \wedge y' = x \wedge z' = x$. Let the set of predicates be $\{x = 2, y = 1, z > 3, x' = 2, y' = 1, z' > 3\}$. The cone sets for the next-state predicates $x' = 2, y' = 1, z' > 3$ are $\{y\},\{x\},\{x\}$, respectively. After the first step of the clustering, the clusters are $C'_1 := \{x' = 2\}$ and $C'_2 := \{y' = 1, z' > 3\}$. Even though $y' = 1$ and

$z' > 3$ do not share a common set of variables they are kept in the same cluster, as they have the identical cone set $\{x\}$.

Since $cone(C'_1) := \{y\}$ and $cone(C'_2) := \{x\}$, the clusters obtained after the second step of the clustering are $C_1 := \{y = 1, x' = 2\}$ and $C_2 := \{x = 2, y' = 1, z' > 3\}$. Observe how the predicates in a given cluster affect each other. For example, in C_2, if $x = 2$ is true, then we know that $y' = 1$ and $z' > 3$ will be false (as y' and z' equal x). If $x = 2$ is false, then $y' = 1$ can be either true or false and $z' > 3$ can be either true or false. However, both $y' = 1$ and $z' > 3$ cannot be true together.

Since cone clustering attempts to keep all related predicates together, the abstractions produced are not much coarser than those produced by the eager 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.

C. Syntactic Clustering for Lazy Abstraction

The idea of lazy abstraction [25] is to start with a coarse initial abstract model, which is refined on-demand as required by a spurious counterexample. Since a coarse abstract model is computed, the abstraction step is usually very fast. This prevents the abstraction step from becoming a bottleneck when computing the abstraction of large circuits or when a large number of predicates are in use.

A completely lazy abstraction corresponds to using no predicate clusters. Thus, the initial abstract transition relation is simply true (allows all abstract transitions). We follow a variant of this technique: all current-state predicates that contain the same set of variables are kept in the same cluster. That is, if $var(\pi_i) = var(\pi_j)$, then π_i and π_j are kept in the same cluster. This is useful if the given set of predicates contains many mutually exclusive (or related) predicates such as $x = 1, x = 2, x > 2$. Keeping these predicates together in a cluster eliminates a large number of abstract states that do not correspond to any concrete states, also known as *spurious abstract states*. For example, an abstract state in which both predicates $x = 1$ and $x = 2$ are true is spurious.

The next-state predicates are not used in the clusters. Thus, the abstraction produced only contains predicate relationships that hold in each abstract state (not between states). If needed, the relationships between current-state and next-state predicates are discovered lazily using refinement techniques.

Example: Let the set of current-state predicates be $\{x < 2, x = 1, y = 1, z > 1\}$. The clusters produced for lazy abstraction are $C_1 := \{x < 2, x = 1\}$, $C_2 := \{y = 1\}$, and $C_3 := \{z > 1\}$.
Loss of precision: In the above example let the next-state function of y be equal to x (that is $y' = x$). The predicates involving x and y' are not present together in any cluster. Thus, the abstract model generated using lazy abstraction allows an abstract transition from a state where $x = 1$ to a state where $y \neq 1$. This is a *spurious transition* because the value of y in the next state must be equal to the value of x in the previous state. This spurious transition will not occur in the abstraction computed using cone clustering, as the predicates $x = 1$ and $y' = 1$ will be in the same cluster.

The abstract transition relation for a predicate cluster depends on the predicates contained in that cluster. Clusters of small size speed up the abstraction computation at the cost of making the abstraction less precise. If needed, the abstract model obtained can be made more precise by using refinement techniques. The loss in precision due to clustering is beneficial as long as the cost of potential refinement steps is smaller than the cost of computing the most precise abstraction.

Once the abstraction of the concrete system is obtained, we model check it using a model checker for finite state systems like SMV [18], [35]. Fig. 4 shows an abstract model. If the abstract model satisfies the property, the property also holds on the original, concrete circuit. If 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. 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.

V. Abstraction Refinement

When refining the abstract model, we distinguish between two cases of spurious behavior, as done in [16]:

1) **Spurious transitions** are abstract transitions that do not have any corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise abstraction, which is computed by the eager approach. However, as we noted earlier, computing the most precise abstract model is expensive and thus, we make use of various predicate clustering techniques. This can 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.

Given a spurious counterexample we first check if any transition in the counterexample is spurious. If a spurious transition is found, it is eliminated from the abstract model by adding a constraint to the abstract model. If no transition in the counterexample is spurious, then new predicates are generated in order to eliminate a spurious prefix in the counterexample. We treat the entire spurious counterexample as a spurious prefix and do not find the shortest spurious prefix.

An abstract counterexample of length l is a sequence of abstract states $\bar{s}(0), \ldots, \bar{s}(l)$, 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 $\bar{s}_i$. Given an abstract state $\bar{s}$, let $\beta(\bar{s})$ denote the conjunction of predicates (or their negation) depending upon their values in $\bar{s}$. For example, let $\bar{s}$ be an abstract state in which the predicate $x < 2$ is true and the predicate $x = y$ is false. Then $\beta(\bar{s}) = x < 2 \ \wedge \ \neg(x = y)$.

$$\beta(\bar{s}) := \bigwedge_{i=1}^{k} \pi_i \Leftrightarrow \bar{s}_i$$

We write $\beta(\bar{s}, \bar{r})$ to denote that the variables in $\beta(\bar{s})$ refer to the concrete variables $\bar{r}$.

A. Detecting and Removing Spurious Transitions

An abstract transition from $\bar{s}$ to $\bar{t}$ is a spurious transition iff there are no concrete states $\bar{r}, \bar{r}'$ such that $\bar{r}$ is abstracted to $\bar{s}$, $\bar{r}'$ is abstracted to $\bar{t}$, and there is a transition from $\bar{r}$ to $\bar{r}'$. Formally, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious iff the following formula is unsatisfiable:

$$\beta(\bar{s},\bar{r}) \ \wedge \ R(\bar{r},\bar{r}') \ \wedge \ \beta(\bar{t},\bar{r}')$$

We use a CNF SAT solver to check the satisfiability of the above formula. If the formula is satisfiable, the abstract transition can be simulated on the concrete model. Otherwise, the abstract transition is spurious. In this case, the spurious transition is removed from the abstract model by adding a constraint to the abstract model.

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 extract an unsatisfiable core [42] 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 are quite effective at producing small unsatisfiable cores, if they exist.

Let us denote the set of current-state predicates whose corresponding CNF literal l_i appears in the unsatisfiable core by 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 from $\bar{s}$ 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:

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

Proposition 2: Every abstract transition from $\bar{u}$ to $\bar{v}$ such the predicates in X have the same value in $\bar{u}$ and $\bar{s}$, and the predicates in Y have the same value in $\bar{v}$ and $\bar{t}$, is spurious. The constraint above removes all of these spurious transitions from the abstract model.

Example: Let the set of current-state predicates be $\{x<2, x=1, y=1, z>1\}$. Consider the abstract transition from $\bar{s}=\{b_1=1, b_2=1, b_3=1, b_4=1\}$ 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 $x<2$, $x=1$, $y=1$, $z>1$, respectively. Let the next-state function of y be x, i.e., $y'=x$. Observe that in the state $\bar{s}$, $x=1$. This implies that $y=1$ in $\bar{t}$ (as $y'=x$). However, b_3' is false in $\bar{t}$ and thus, the abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. As described in section IV-C, the abstract transition from $\bar{s}$ to $\bar{t}$ can arise when using lazy abstraction. This spurious transition can be eliminated by adding the following constraint to the abstract model [19]: $\neg(b_1 \wedge b_2 \wedge b_3 \wedge b_4 \wedge \neg b_1' \wedge \neg b_2' \wedge \neg b_3' \wedge \neg b_4')$.

However, the constraint above removes just one spurious transition. By examining 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 $b_2=1$, and $b_3'=0$. The unsatisfiable core technique described above is capable of discovering this fact. This allows us to eliminate the abstract transition from $\bar{s}$ to $\bar{t}$ and 63 more spurious transitions by adding the following constraint to the abstract model: $\neg(b_2 \wedge \neg b_3')$. It is very important to remove as many spurious transitions as possible in order to make the CEGAR loop terminate quickly.

Semantic Predicate Clustering: The predicates responsible for making an abstract transition spurious can be treated as a predicate cluster C, which can be used during the abstraction step. Suppose an abstract transition from $\bar{s}$ to $\bar{t}$ is spurious. Let C denote the set of current-state and next-state predicates responsible for this spurious transition as identified by an unsatisfiable core. As described above, the predicates appearing in C are used to remove the spurious transition from $\bar{s}$ to $\bar{t}$. In semantic predicate clustering, C is also added to the existing set of predicate clusters and is used to compute the abstraction (Eqn. 5) in the subsequent iterations. Intuitively, the predicates occurring in C are *semantically* related because a particular assignment of truth values to the predicates in C (as given by $\bar{s}$, $\bar{t}$) can make an abstract transition spurious. Thus, by computing all possible relationships between the predicates in C (during abstraction), we remove all abstract transitions that are spurious due to the predicates in C.

Example: For the spurious transition in the example above, we obtain $C := \{x=1, y'=1\}$. The predicates in C are used to eliminate multiple spurious transitions by adding the constraint $\neg(b_2 \wedge \neg b_3')$. However, even after adding this constraint the abstract model allows another spurious transition from a state $\bar{u}$ where $\neg(x=1)$ to a state $\bar{v}$ where $y=1$ (that is, $y'=1$). In semantic predicate clustering C is added as a predicate cluster. The abstraction step will discover that $b_2 \Leftrightarrow b_3'$ using C. Thus, the spurious transition from $\bar{u}$ to $\bar{v}$ cannot arise.

B. Detecting and Removing Spurious Prefixes

An abstract counterexample $\bar{s}(0), \ldots, \bar{s}(l)$ of length l is a spurious prefix iff there is no concrete execution of l transitions such that at each step the concrete state is consistent with the corresponding abstract state. More formally, let $\bar{r}_0, \ldots, \bar{r}_l$ denote the concrete state variables at each of the $l+1$ states. The initial state of the concrete system is denoted as $I(\bar{r}_0)$.

The abstract counterexample $\bar{s}(0), \ldots, \bar{s}(l)$ is a spurious prefix iff the following formula is unsatisfiable:

$$I(\bar{r}_0) \ \wedge \ \bigwedge_{i=0}^{l-1} R(\bar{r}_i, \bar{r}_{i+1}) \ \wedge \ \bigwedge_{i=0}^{l} \beta(\bar{s}(i), \bar{r}_i)$$

The above formula is unsatisfiable iff there is no sequence of concrete states $\bar{r}_0, \ldots, \bar{r}_l$ such that $\bar{r}_0$ is an initial state, there is a transition from $\bar{r}_i$ to $\bar{r}_{i+1}$ for $0 \leq i < l$, and the predicate

values in each concrete state $\bar{r}_j$ exactly match the predicate values given by the abstract state $\bar{s}(j)$ for $0 \leq j \leq l$.

In [16], spurious prefixes are eliminated by adding a 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 by computing the weakest precondition of the given property with respect to the transition function given by the RT-level circuit.

Weakest Preconditions: In software verification, the weakest precondition $wp(st,\gamma)$ of a predicate 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 and external inputs are assigned new non-deterministic values.

Recall that the set of registers is denoted by Q. The next-state function for register $r_i \in Q$ is given by $f_i(\bar{r})$. We use $\bar{f}$ to denote the vector of the next-state functions for the registers in Q. We use the expression $e[\bar{r}/\bar{f}]$ to denote the simultaneous substitution of each variable $r_i \in Q$ by its corresponding next state function $f_i(\bar{r})$.

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 (we write $\gamma(\bar{r})$ as γ for short):

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

In order to refine a spurious prefix of length $l > 0$, we compute $wp_i(\bar{f},\tau)$ for each $1 \leq i \leq l$, where τ is the safety property we are interested in checking. Intuitively, τ holds after i transitions iff $wp_i(\bar{f},\tau)$ holds before i transitions. Refinement corresponds to adding the Boolean expressions occurring in each $wp_i(\bar{f},\tau)$ to the existing set of predicates.

Proposition 3: The most precise abstraction (Eqn. 4), created with respect to the new set of predicates $\{wp_i(\bar{f},\tau)\}$, does not contain the given spurious prefix[1]. Furthermore, all spurious prefixes of length l are removed.

Propositions 2 and 3 taken together guarantee that the progress is made with each refinement iteration. In case of circuits, the weakest precondition is always computed with respect to the same transition function vector $\bar{f}$ 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 a spurious prefix of length 1. The weakest precondition is computed 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 Preconditions: When the spurious counterexample is long, the weakest precondition computation becomes expensive and the predicates generated can become very complex (see wp_1 above). This adversely affects the abstraction refinement loop. In software verification, this problem is solved by computing the weakest precondition with respect to the statements appearing in the spurious trace only. This is not directly applicable to a synchronous circuit because the statements occurring in the spurious trace correspond to the next-state functions. The next-state functions usually contain many conditional statements. Thus, simply substituting the next-state functions as done above leads to a blowup in the size of weakest preconditions.

Instead, we apply a syntactic simplification to the weakest preconditions at each step. The simplification uses data from the abstract error trace. We exploit the fact that many of the control flow guards in the Verilog code 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 preconditions, we substitute the guards in the weakest preconditions with their truth values. Furthermore, we only add the atomic predicates occurring in the weakest preconditions as new predicates.

In order to formalize the simplification of weakest preconditions we define a procedure *simplify* in Fig. 6. Let the current set of predicates be $\{\pi_1,\ldots,\pi_k\}$. The *simplify* procedure takes as input a formula $g(\bar{r})$ (written as g for short) and an abstract state $\bar{t}$. It replaces all occurrences of predicates $\{\pi_1,\ldots,\pi_k\}$ in g by their truth values in $\bar{t}$.

Example: Suppose our current set of predicates is $\{x < 2, x < 1\}$. Let $\bar{t}$ be an abstract state in which $x < 2$ is true and $x < 1$ is true. Let $g(x,y)$ be the formula $(((x\ <\ 1)\ ?\ (x+y)\ :\ x\ ) <\ 2)$. After calling *simplify* with g and $\bar{t}$ as arguments, g becomes:

$$((1?\ (x+y)\ :\ x\ ) <\ 2) \quad = \quad x+y<2$$

```
// Input: a formula g
// Input: an abstract state t̄ assigning values to {π_1,...,π_k}
// Output: g is simplified (modified in-place)

simplify(g,t̄) {
1: for all operands h in g
2:    simplify(h,t̄);          // recursive simplification
3: Remove the constant conditionals from g
   // For example, replace (0?x:y) by y
4: if ∃π_j.(π_j = g) then    // syntactic equality of expressions
5:    g = t̄_j                // replace g by value of π_j in t̄
}
```

Fig. 6. Simplification of a formula using an abstract state.

[1]In order to provide this guarantee, inputs in a formula need to be replaced by fresh identifiers every time the weakest precondition is computed. The new identifiers can be left as it is in the predicates. They will be existentially quantified during the decision procedure queries. A heuristic that works well in practice is to leave the input variables unchanged during the weakest precondition computation.

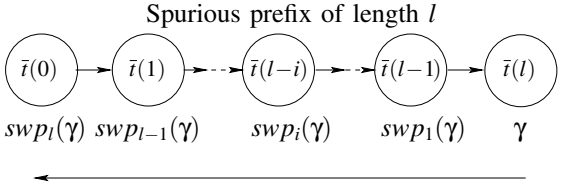


Fig. 7. Simplified weakest precondition computation.

Let $h(x,y)$ be the formula $((x = y)?y : x) = y$. After calling *simplify* with h and $\bar{t}$ as arguments, h remains unchanged.

Definition of Simplified Weakest Precondition: Let the spurious prefix be $\bar{t}(0),\ldots,\bar{t}(l)$ with $l \geq 1$ and the property be γ. The weakest precondition wp_i is a formula that should hold before i concrete transitions for γ to hold after i transitions.

As motivated earlier we want to simplify wp_i using the predicate values from the spurious prefix. We denote the *simplified weakest precondition (swp)* for i steps by swp_i. The abstract state $\bar{t}(l-i)$ provides the truth values of the predicates just before the i transitions leading to the end of spurious prefix. Thus, $swp_i(\gamma)$ is simplified using the predicate values from the abstract state $\bar{t}(l-i)$. Fig. 7 shows the correspondence between the abstract states and swp_i. Formally, swp_i is defined as follows (wp_1 was defined earlier and l is the length of spurious prefix):

$$swp_1(\gamma) \;:=\; simplify(wp_1(\gamma),\ \bar{t}(l-1))$$
$$swp_i(\gamma) \;:=\; simplify(wp_1(swp_{i-1}(\gamma)),\ \bar{t}(l-i)) \quad (1 < i \leq l)$$

The new set of predicates for refinement is obtained from $swp_1,\ldots,swp_l$. This is done by taking only the atomic predicates occurring in the simplified weakest preconditions. After the addition of new predicates we re-compute the predicate clusters.

Example: We continue our example in Fig. 2. We want to prove that $x < 200$ is an invariant. In Fig. 4, an abstraction of this program using three predicates $x < 200, x < 100, x+y < 200$ is presented. The property $\mathbf{G}(x < 200)$ is proved by means of this abstraction. We now describe how these predicates are discovered automatically.

We take the set of predicates occurring in the property itself as the initial set of predicates. Thus, our initial abstraction is created with respect to the predicate $x < 200$. Model checking the abstract model produces a counterexample with one transition from a state in which $x < 200$ to a state in which $\neg(x < 200)$. This counterexample is a spurious prefix (not a spurious transition). The simplified weakest precondition swp_1 of $x < 200$ is:

$$swp_1(x < 200) \;:=\; (((x < 100)\ ?\ (x+y)\ :\ x\ ) <\ 200)$$

The only new predicate obtained from $swp_1(x < 200)$ is $x < 100$. Note that we do not take the entire weakest precondition as a new predicate as it is not atomic. The new set of predicates is $\{x < 200, x < 100\}$. Once again, the abstraction and model checking steps are performed. This time, we obtain another spurious prefix $\bar{t}(0)$, $\bar{t}(1)$ of length one. We also obtain the truth value of the predicate $x < 100$ in the abstract states $\bar{t}(0)$ and $\bar{t}(1)$. Since x is equal to one in the initial state of the system, it turns out that the predicate $x < 100$ is true in $\bar{t}(0)$. The simplified weakest precondition is given as follows:

$$swp_1(x < 200) \;:=\; ((1?\ (x+y)\ :\ x\ ) < 200) = x+y < 200$$

Thus, $swp_1(x < 200)$ yields a new predicate $x+y < 200$. Using the new set of predicates $\{x < 200, x < 100, x+y < 200\}$, we obtain the abstraction shown earlier in Fig. 4. The abstract property holds on this abstraction and thus, $\mathbf{G}(x < 200)$ holds on the concrete program in Fig. 2.

The use of simplification invalidates the progress guarantee: the predicates in the simplified weakest precondition of the given property are not always sufficient to ensure that the spurious prefix is eliminated from the abstract model. For example, we may need the weakest precondition of the guard $x < 100$ in the example above, which is not computed if we substitute $x < 100$ by a Boolean value during simplification.

In order to guarantee progress, we identify a subset of the existing predicates such that computing the weakest precondition of these predicates is sufficient for removing the spurious behavior. As in [26], this is done by using the unsatisfiable core of the SAT instance used for simulating the prefix. This approach identifies a subset of the existing predicates that is responsible for the spurious behavior. If a copy of predicate p in cycle k appears in the unsatisfiable core, we compute the (simplified) weakest precondition of p for k steps ($k \leq l$). In addition we compute the weakest precondition for each predicate used during the simplification (Fig. 6, Line 4).

VI. Experimental Results

The experiments are performed on a 1.86 GHz Intel Xeon (R) machine with 4 GB of memory running Linux. The techniques described in this paper have been implemented in a tool called `VCEGAR` [38]. Our implementation is available for experimentation by other researchers. We use the MiniSat SAT solver [32] as our decision procedure. The abstractions are model checked using a publicly available version of the Cadence SMV model checker [18]. We perform two sets of experiments:

1) We compare the performance of `VCEGAR` with the performance of localization reduction technique implemented in Cadence SMV. The Cadence SMV tool is a netlist based model checker. It implements localization reduction using BDDs and SAT checkers. The results are reported in Section VI-A.
2) We compare three different predicate clustering algorithms: syntactic cone clustering, clustering for lazy abstraction described in Section IV, and semantic predicate clustering (Section V-A). These results are reported in Section VI-B.

In all our experiments we compute the initial abstraction using the atomic predicates appearing in the property. The remaining predicates are discovered automatically using refinement.

A. Comparison with Localization Reduction

The results are summarized in Table I. The column "Latches" contains the total number of latches in the design.

TABLE I
EXPERIMENTAL RESULTS: ALL RUNTIMES ARE IN SECONDS (ROUNDED TO NEAREST INTEGER). A DASH "-" INDICATES A TIMEOUT OF 2 HOURS. A STAR "*" INDICATES THAT THE MODEL CHECKER TERMINATED AND REPORTED A TOO LARGE NUMBER OF BDD VARIABLES

Bench-mark	Latches	Predicate Abstraction						CSMV
		Time	Abs	MC	Ref	P	I	Time
USB1	545	42	1	2	29	17	62/0	149
USB2	545	599	47	147	386	116	146/22	1349
USB3	545	446	46	73	317	114	123/20	2594
ETH0	359	44	2	3	30	21	55/0	-
ETH1	359	127	8	8	102	93	49/2	-
ETH2	359	161	8	16	127	94	109/2	21
M2KB	16427	5	0	0	5	3	2/0	28
M8KB	65694	28	0	0	28	3	2/0	*
M16KB	131117	34	0	0	34	3	2/0	*
N2KB	16427	93	0	0	93	11	9/0	*
N8KB	65694	490	0	0	490	11	9/0	*
N16KB	131117	790	0	0	789	11	9/0	*
AR200	400	1	0	0	1	3	3/2	672
AR3000	6000	12	0	0	12	3	3/2	-
AR4000	8000	17	0	0	16	3	3/2	-

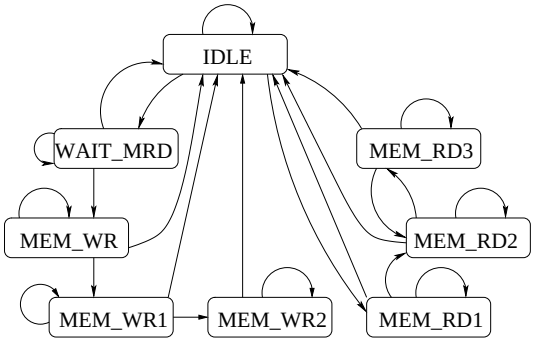


Fig. 8. State machine for the DMA in the USB 2.0 Function core.

The columns marked with "Predicate Abstraction" contain the results of applying the techniques discussed in this paper. The "Time", "Abs", "MC", and "Ref" columns contain the total time, followed by the time taken by abstraction, model checking, and refinement including simulation. The time spent before the start of the CEGAR loop is given by Time-(Abs+MC+Ref). We use lazy abstraction and rely on refinement to do most of the work in these benchmarks. The "P" column contains the final number of predicates. The "I" column gives two numbers separated by a slash: 1) Number of refinement steps in which spurious transitions are removed, and 2) number of refinement steps in which new predicates are added. The sum of these two numbers is the total number of refinement iterations. The results of running Cadence SMV are given in the "CSMV" column. We report the total time taken by Cadence SMV when running with the counterexample-based abstraction refinement option `-absref3`.

Benchmarks: The USB benchmark was used for experimental evaluation of the EverLost tool [20]. It is derived from a USB 2.0 Function core [36] and contains approximately 4000 lines of RTL Verilog. We checked three properties. The first property USB1 checks that the implementation of the internal DMA module simulates the state transition diagram shown in Fig. 8. The property holds and all the predicates required for the proof are present in the property itself. The second property USB2 encodes the following: if the `abort` signal is true in any state of Fig. 8, then the next state will be `IDLE`. This property does not hold because the transition from the `MEM_WR2` state to the `IDLE` state is not guaranteed by the `abort` signal. The third property USB3 excludes the state `MEM_WR2` from the USB2 property. This property holds on the design. The properties USB2 and USB3 contain three and four atomic predicates, respectively. The remaining predicates are discovered through refinement.

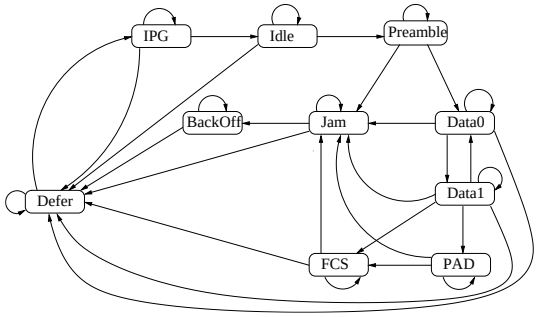


Fig. 9. State machine for the Transmit module in the Ethernet MAC.

The ETH benchmark was also used in [20]. It is the design of a 10/100 Mbps Ethernet MAC [36] and contains approximately 5000 lines of RTL Verilog. The transmit module of the design contains a state machine with ten states (see Fig. 9). The property ETH0 checks that the implementation obeys the state machine description given in Fig. 9. All the predicates required for proving the property are present in the property itself. The property ETH1 checks the outgoing transitions from the state `BackOff`. The property ETH2 checks the outgoing transitions from the state `Jam`. Both ETH1 and ETH2 hold on the design. When checking ETH1 and ETH2 most of the predicates are discovered through refinement.

The ICRAM benchmark is taken from the Instruction Cache RAM unit of the Sun PicoJava II microprocessor [37]. It maintains a RAM of size 16KB (organized as 2048 entries of 64 bits each). If the writing signal `wen0` is enabled the value of data input (`din`) is written to the lower 32 bits of the location addressed by the input address (`addr`). Otherwise, if the writing signal `wen1` is enabled, the value `din` is written to the higher 32 bits of the location addressed by `addr`. This functionality of the ICRAM is encoded in form of eight safety properties using the current-state and next-state of the variables. We use `P.x` to denote the value of a register or input `x` in the previous state. Each property compares eight bits in `P.din` and corresponding bits in ICRAM. A sample property is given below:

`P.wen0`$\rightarrow$`(ram[{P.addr,3'b001}]=P.din[23:16])`

The above property depends on the contents of the RAM. Thus, even after applying techniques such as localization reduction, the system has 16KB ($16 \times 1024 \times 8$) latches. We verified the above property by varying the size of RAM from 2KB to 16KB. These benchmarks start with a prefix "M" in Table I. We also combined all the eight properties for the ICRAM benchmark into a single property. These benchmarks

start with a prefix "N" in Table I. For both "M" and "N" benchmarks the property is proved using only the predicates occurring in the property. No new predicates are discovered.

The benchmarks with names starting with "AR" perform arithmetic operations on two registers x and y as shown in Fig. 2. We verify the invariant $x < 200$. In the ARi benchmark the size of both x, y is i and total number of latches is $2 \times i$. As described in the previous section, this property is proved using the predicates $x < 200$, $x < 100$, $x + y < 200$. The predicate $x < 200$ is obtained from the property and the predicates $x < 100$, $x + y < 200$ are discovered using refinement.

Summary: VCEGAR has a better performance on all but one benchmark reported in Table I. Cadence SMV times out on the ETH0, ETH1, AR3000, AR4000 benchmarks, while the predicate abstraction method is able to complete these benchmarks with better runtimes. Some of the inferences drawn from Table I are as follows:

- The runtime of localization reduction grows exponentially with each newly added latch. This trend is evident in the AR benchmark series. On these benchmarks, localization reduction is not able to reduce the number of latches in the abstract model created.
- When using predicate abstraction, the size of the abstract model remains constant even when the number of latches is increased. For many properties, the number of word-level predicates needed for the proof does not grow as the width of the registers is increased. This trend is visible in the M*, N*, and AR* benchmarks. Thus, the model checking (MC) time is similar across these benchmarks.

A plot of the total time needed by predicate abstraction compared to the number of latches is given in Fig. 10(a) and Fig. 10(b) for the N* and the AR* benchmarks, respectively. These graphs show that the predicate abstraction technique scales well with the increase in the number of latches.

B. Comparing Predicate Clustering Techniques

We report the performance of the CEGAR loop using three different predicate clustering techniques described in Section IV and Section V-A. The benchmark characteristics are given in Table II. We report the number of lines of code, the total number of latches, the total number of Verilog combinational elements and inputs ("CE+I" column), and the total number of properties checked for each benchmark. The benchmarks USB 2.0 and Ethernet MAC were described in the previous section. Other benchmarks are taken from the Texas97 and VIS [39] benchmark suites.

The results are summarized in Table III. The columns labeled with "Cone" contain the results of using syntactic cone clustering in the CEGAR loop. The performance of the CEGAR loop when using clustering for lazy abstraction is summarized in the columns labeled with "Lazy". The "Semantic" column presents the results of using semantic predicate clustering (Section V-A).

For each predicate clustering technique, the "Total", "Abs", "MC", and "Ref" columns contain the total verification time, followed by the time taken by abstraction, model checking, and refinement including simulation. The "Preds" column contains two numbers separated by a slash: 1) The total number of predicates in the last iteration of the CEGAR loop. This includes only the current-state predicates. 2) The maximum number of predicates present in any predicate cluster generated by the predicate clustering technique. The number of refinement iterations is reported in the "I" column. The "Res" column contains T (true) if the property holds, else it contains F (false), followed by the length of the counterexample. In these benchmarks (expect USB1, ETH0) most of the predicates are discovered automatically during refinement phase. Below, we compare the three instantiations of the CEGAR loop, which are "Cone", "Lazy", and "Semantic".

TABLE II
BENCHMARK CHARACTERISTICS

Benchmark	Lines	Latches	CE+I	Properties
mpeg	1215	599	234	2
SDLX	898	41	40	1
Miim	841	83	173	1
ethernet (enet)	610	91	156	2
itc99-b12 (b12)	558	151	723	1
usb-phy (uphy)	1054	44	25	1
USB 2.0 (USB)	4000	545	1686	3
Ethernet MAC (ETH)	5000	359	2363	3

"Cone" versus "Lazy": The "Lazy" technique is able to handle all benchmarks within the timeout, and thus, it is more robust than the "Cone" technique (which timeouts on five problems). When using the "Cone" technique, the SAT-based abstraction becomes the bottleneck. Model checking of abstract models also becomes expensive (see Miim row). This happens because the abstract models created in the "Cone" technique are more detailed and thus harder. However, the properties can usually be checked using coarse (less precise) abstractions created by the "Lazy" technique.

"Semantic" versus "Lazy": In the "Semantic" technique (Section V-A), new predicate clusters are generated as follows: When a spurious transition is found, we identify a set of predicates responsible for spurious behavior. These predicates are treated as a new predicate cluster. In our experiments this cluster is used during abstraction computation only if it has ≤ 6 predicates. In addition, we use the same predicate clusters as for the "Lazy" technique.

The "Semantic" technique consistently requires fewer refinement iterations than the "Lazy" technique. This shows that computing all possible abstract transitions for the predicates responsible for a spurious transition also rules out other spurious transitions. The runtime of both techniques is comparable.

The abstraction computation or abstraction model checking can become a bottleneck when using the "Cone" technique, while a large number of refinement iterations can hurt the performance when using the "Lazy" technique. The "Semantic" technique tries to balance the bottlenecks of both "Cone" and "Lazy" techniques, and thus, seems to be the most scalable.

VII. Conclusion and Future Work

We apply the idea of predicate abstraction from software verification to verify hardware designs at a higher level of abstraction. We show how to reduce the abstraction computation

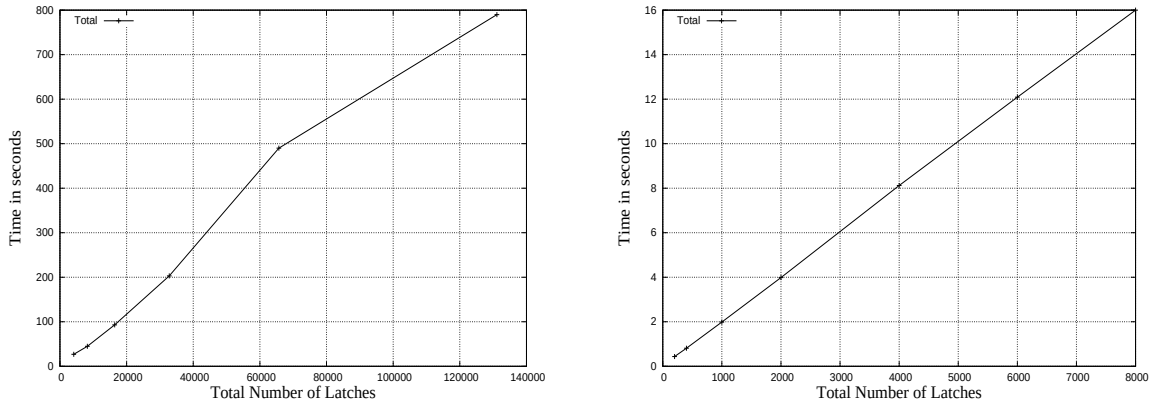


Fig. 10. Runtime of the CEGAR loop with respect to number of latches: (a) N* benchmarks (b) AR* benchmarks.

TABLE III
COMPARING THREE CEGAR LOOPS EACH EMPLOYING A DIFFERENT PREDICATE CLUSTERING METHOD. ALL TIMES ARE REPORTED IN SECONDS (ROUNDED TO NEAREST INTEGER). A DASH "-" INDICATES A TIMEOUT OF 2 HOURS

Bench-mark	Cone						Lazy						Semantic						Res
	Time	Abs	MC	Ref	Preds	I	Time	Abs	MC	Ref	Preds	I	Time	Abs	MC	Ref	Preds	I	
mpeg1	44	25	1	18	31/33	7	41	3	1	37	31/22	24	46	10	1	35	30/22	22	T
mpeg2	51	26	1	23	31/32	9	47	4	1	43	30/22	26	54	11	1	43	31/22	24	T
SDLX	8	4	1	2	32/13	23	14	1	5	8	32/6	83	13	3	3	6	32/6	64	T
Miim	170	49	119	2	23/19	19	8	1	2	6	23/4	55	8	2	1	4	23/6	40	T
enet1	-	-	-	-	-	-	45	2	20	22	48/4	129	45	6	17	21	48/6	121	F(6)
enet2	38	6	5	27	37/11	36	69	2	20	47	37/3	117	66	6	19	41	37/6	99	T
b12	310	181	69	57	50/24	29	132	3	24	103	38/8	148	131	17	13	98	48/8	94	F(14)
uphy	13	1	3	8	42/18	29	24	0	10	13	42/7	100	23	1	10	11	42/7	87	F(36)
USB1	12	1	0	0	17/17	0	42	1	2	29	17/8	62	51	19	1	20	17/8	40	T
USB2	-	-	-	-	-	-	599	47	147	386	116/15	168	547	109	87	333	116/15	139	F(14)
USB3	-	-	-	-	-	-	446	46	73	317	114/15	143	459	97	70	282	114/15	120	T
ETH0	49	15	4	19	21/11	31	44	2	3	30	21/0	55	57	14	3	30	21/6	53	T
ETH1	-	-	-	-	-	-	127	8	8	102	93/0	51	177	48	13	107	93/6	54	T
ETH2	-	-	-	-	-	-	161	8	16	127	94/0	111	172	48	14	100	94/6	95	T

overhead in presence of a large number of predicates. This is done by dividing the set of predicates into clusters of related predicates and the abstraction is computed separately for each cluster. In lazy abstraction the expensive task of program abstraction is deferred until a spurious counterexample is found. We show the benefit of lazy abstraction in the context of hardware verification.

We use unsatisfiable cores in order to eliminate multiple spurious transitions. The spurious trace may also be caused by insufficient predicates. In the software domain, tools typically use weakest preconditions or strongest postconditions 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 models closely resemble programs written in languages like ANSI-C. Our experimental results show that this technique is effective in discovering new word-level predicates for refinement.

Future research will focus on the use of interpolants [24] for deriving new predicates. Memory abstraction techniques [21], [29] can be combined with our technique to handle large memories efficiently. We would also like to experiment with new decision procedures for bit-vector arithmetic [6], [29].

APPENDIX

Proof of Proposition 1: Let the set of predicates be Pr. $\hat{Q}$ denotes the abstraction with respect to Pr. From Eqn. 5, $\hat{R} = \bigwedge_{j=1}^{l} \hat{R}_j$, where $\hat{R}_j$ denotes the abstraction with respect to a cluster C_j and $C_j \subseteq Pr$. The proposition is proved by showing that for all $1 \leq j \leq l$, $\hat{Q} \Rightarrow \hat{R}_j$ or $\hat{Q} \subseteq \hat{R}_j$ using set notation. We will treat $\hat{Q}$ and $\hat{R}_j$ as sets of abstract transitions and show that $\hat{Q} \subseteq \hat{R}_j$. We rewrite the definitions of $\hat{Q}$ and $\hat{R}_j$ as follows:

$$\begin{aligned} \hat{Q} &:= \{(\bar{b},\bar{b}') \mid \exists \bar{r},\bar{r}' \ : \ \delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Pr)\} \\ \hat{R}_j &:= \{(\bar{b},\bar{b}') \mid \exists \bar{r},\bar{r}' \ : \ \delta(\bar{r},\bar{r}',\bar{b},\bar{b}',C_j)\} \end{aligned}$$

where $\delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z)$ relates concrete states $\bar{r},\bar{r}'$, and abstract states $\bar{b},\bar{b}'$ with respect to a set of predicates Z.

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

If $Z_2 \subseteq Z_1$, then $\delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z_1)$ is equivalent to $\delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z_2) \wedge \delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z_1 \backslash Z_2)$. Thus, if $\delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z_1)$, then $\delta(\bar{r},\bar{r}',\bar{b},\bar{b}',Z_2)$ holds. If an abstract transition $(\bar{a},\bar{a}') \in \hat{Q}$, then there exist two concrete states $\bar{x},\bar{x}'$ such that $\delta(\bar{x},\bar{x}',\bar{a},\bar{a}',Pr)$ holds. Since $C_j \subseteq Pr$, it follows from the above that $\delta(\bar{x},\bar{x}',\bar{a},\bar{a}',C_j)$ holds. Thus, $\exists \bar{r},\bar{r}' \ : \ \delta(\bar{r},\bar{r}',\bar{a},\bar{a}',C_j)$ holds and $(\bar{a},\bar{a}') \in \hat{R}_j$. This shows $\hat{Q} \subseteq \hat{R}_j$. As $\hat{Q} \subseteq \hat{R}_j$ for all $1 \leq j \leq l$ and $\hat{R} = \bigcap_j \hat{R}_j$, it follows that $\hat{Q} \subseteq \hat{R}$. □

ACKNOWLEDGMENT

The authors would like to thank all anonymous reviewers for their excellent comments, resulting in an improvement in the quality of this paper.

REFERENCES

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Asia South Pacific design automation conference*, pages 19–24, 2006.
[2] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *DAC*, pages 218–223. ACM Press, 2004.
[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, pages 103–122, 2001.
[4] Jason Baumgartner, Andreas Kuehlmann, and Jacob A. Abraham. Property checking via structural analysis. In *CAV*, pages 151–165, 2002.
[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207. Springer, 1999.
[6] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, pages 358–372. Springer, 2007.
[7] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.
[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
[9] 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.
[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and Veith H. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000.
[11] E. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *POPL*, pages 342–354, 1992.
[12] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, 2004.
[14] 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, 1981.
[15] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Predicate abstraction and refinement techniques for verifying Verilog. Technical Report CMU-CS-04-139, Carnegie Mellon University, 2004.
[16] Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based predicate abstraction for hardware verification. In *SAT*, 2003.
[17] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224. Springer, 2003.
[18] Cadence SMV, http://www.kenmcmil.com/smv.html.
[19] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of LICS*, pages 51–60, 2001.
[20] Flavio M. de Paula and Alan J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC*, June 4–8 2007.
[21] M. K. Ganai, A. Gupta, and P. Ashar. Efficient modeling of embedded memories in bounded model checking. In *CAV*, pages 440–452, 2004.
[22] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254, pages 72–83, 1997.
[23] Anubhav Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006.
[24] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
[26] H. Jain, F. Ivančić, A. Gupta, and M. K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, pages 397–412, 2005.
[27] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In *DAC*, pages 445–450, 2005.
[28] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
[29] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL-Level verification. In *ICCAD*, pages 786–793, 2006.
[30] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, 1996.
[31] Kenneth L. McMillan. An interpolating theorem prover. In *TACAS*, pages 16–30. Springer, 2004.
[32] MiniSat: http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/.
[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.
[34] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, pages 435–449. Springer, 2000.
[35] NuSMV model checker: http://nusmv.irst.itc.it/.
[36] Opencores: http://www.opencores.org/.
[37] Sun PicoJava: http://www.sun.com/processors/technologies.html.
[38] VCEGAR tool: http://www.cs.cmu.edu/∼modelcheck/vcegar/.
[39] VIS model checker, http://vlsi.colorado.edu/∼vis.
[40] 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.
[41] Dong Wang. *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, 2003.
[42] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formulas. In *SAT*, 2003.

Himanshu Jain received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India in 2003. He is currently working towards the Ph.D. degree in computer science at the Carnegie Mellon University, Pittsburgh PA, USA.

His research interests include hardware and software verification, satisfiability solvers, decision procedures, concurrency, and computer security.

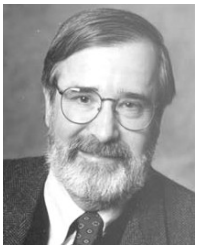
Daniel Kroening received the M.E. and doctoral degrees in computer science from the University of Saarland, Saarbrücken, Germany, in 1999 and 2001, respectively. He joined the Model Checking group in the Computer Science Department at Carnegie Mellon University, Pittsburgh PA, USA, in 2001 as a Post-Doc.

In 2004, he was appointed as an assistant professor at the Swiss Technical Institute (ETH) in Zürich, Switzerland. His research interests include automated formal verification of hardware and software systems, decision procedures, embedded systems, and hardware/software co-design.

Natasha Sharygina received the Ph.D. degree from the University of Texas at Austin, USA in 2002.

Her professional experience includes consulting at Bell Labs, Computing Principles Research (2000–2001), a research fellow position at Carnegie Mellon University (CMU), Software Engineering Institute (2002–2005), and an adjunct assistant professor at CMU, School of Computer Science (since 2002). She is currently an assistant professor at the Department of Informatics at the University of Lugano, Switzerland. Her research interests are in dependable computing, computational logic, automated formal methods, computer security, electronic design automation, and program analysis.

Dr. Sharygina is a recipient of CMU Independent Research and Development grant (2004), Swiss NSF grant (2005), an Independent Swiss Research Foundation career award (2005), and ACM Recognition of Service Award.



Edmund Clarke (M'96-F'05) received the B.A. degree in mathematics from the University of Virginia, Charlottesville, VA, in 1967, the M.A. degree in mathematics from Duke University, Durham, NC, in 1968, and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, in 1976.

He joined the faculty in the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA in 1982. He was appointed Full Professor in 1989. His interests include software and hardware verification and automatic theorem proving. He is a co-developer of Model Checking, an algorithmic verification technique.

Dr. Clarke is a recipient of the ACM Kanellakis Theory and Practice award Award. He was elected to the National Academy of Engineering in 2005. He is a Fellow of the Association for Computing Machinery.