Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog *

Himanshu Jain
CMU SCS, Pittsburgh, PA 15213

Daniel Kroening
ETH Zürich, Switzerland

Natasha Sharygina
CMU SCS and SEI, Pittsburgh, PA 15213

Edmund Clarke
CMU SCS, Pittsburgh, PA 15213

ABSTRACT

Model checking techniques applied to large industrial circuits suffer from the state space explosion problem. A major technique to address this problem is abstraction. The most commonly used abstraction technique for hardware verification is localization reduction, which removes latches that are not relevant to the property. However, localization reduction fails to reduce the size of the model if the property actually depends on most of the latches. This paper proposes to use predicate abstraction for verifying RTL Verilog, a technique successfully used for software verification. The main challenge when using predicate abstraction is the discovery of suitable predicates. We propose to use weakest preconditions of Verilog statements in order to obtain new predicates during abstraction refinement. This technique has not been applied to circuits before. On benchmarks taken from an industrial microprocessor, we successfully verified safety properties with more than 32,000 latches in the cone of influence. We compare the performance of our technique with a modern model checker that implements localization reduction.

Categories and Subject Descriptors: B.5.2 [Hardware]: Register-Transfer-Level Implementation–Design Aids; J.6 [Computer Aided Engineering]: [Computer-Aided Design]

General Terms: Verification

Keywords: Predicate Abstraction, Verilog, SAT

1. INTRODUCTION

Formal verification techniques are widely applied in the hardware design industry. *Model checking* [10] is one of the most commonly used formal verification techniques in a commercial setting. However, model checking suffers from the state space explosion problem. 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* [18, 24, 6]. The abstract model is created from the given circuit by removing a large number latches together with the logic required to compute their next state. The latches that are removed are called the *invisible latches*. The latches remaining in the abstract model are called *visible latches*. The initial abstract model is created by making the latches present in the property visible, and the rest invisible.

Localization reduction is a *conservative* over-approximation of the original circuit for reachability properties. This implies that if the abstraction satisfies the property, the property also holds on the original circuit. 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 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. As in Bounded Model Checking (BMC) [4], 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 [24]. 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 (e.g., more visible latches) 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 [18, 7, 8, 3, 15, 24].

In the software domain, the most successful abstraction technique for large systems is *predicate abstraction* [16]. 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. When applying predicate abstraction to circuits, two problems arise:

- Most model-checkers used in the hardware industry work on a very low level design, usually a *net-list*. 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.

*This research was sponsored by the Gigascale Systems Research Center (GSRC), the Semiconductor Research Corporation (SRC), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

- The second problem concerns 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 designs, the use of these bit-level constructs is ubiquitous.

Predicate abstraction tools used for software employ multiple heuristics in order to reduce the cost of calling the theorem prover while computing the abstraction. SLAM [3] applies ad-hoc heuristics that limit the number of predicates in a query, i.e., it partitions the set of predicates 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, SLAM concludes that the counterexample is caused by the partitioning of the predicates during the abstraction. In this case, a separate refinement algorithm (called Constrain [2]) is invoked. This step only addresses spurious behavior due to an inexact abstraction, as opposed to spurious behavior caused by insufficient predicates.

In the BLAST tool [17], the abstraction is completely demand-driven. Initially, BLAST uses a very coarse abstraction. Additional abstraction is only performed when a spurious counterexample is encountered. The abstraction is only done to the extent necessary to remove the spurious behavior. This is called *lazy abstraction*.

Contribution. This paper introduces new techniques for word-level predicate abstraction and refinement for circuits given in Verilog RTL. There are two challenges when applying predicate abstraction to circuits: 1) The computation of the abstract model is hard in presence of large number of predicates, and 2) discovery of suitable word-level predicates for abstraction refinement.

In order to address the first problem, we partition 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 partitioning*. We identify eager abstraction [12] and lazy abstraction [17] as special cases of predicate partitioning. The eager technique refers to the case when all predicates are within a single cluster, while lazy abstraction corresponds to the case in which very few predicates are used for computing the abstraction (clusters of small size). As in [12], we use SAT to compute the abstract transition relation. However, the predicate partitioning is also applicable with any other solver (or theorem prover).

Due to partitioning additional spurious counterexamples are introduced which have to be removed during the refinement phase. When a spurious counterexample is encountered, we first check whether each transition in the counterexample can be 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 on the abstract transition relation which eliminates the spurious transition. We make use of the proof of unsatisfiability of the SAT instance to identify a small subset of existing predicates to eliminate the transition. The fewer predicates are found, the more spurious counterexamples can be eliminated in one step.

When all SAT instances for simulation of abstract transitions are satisfiable it means that none of the abstract transitions is spurious due to the partitioning. The immediate conclusion then is that the spurious counterexample is caused by insufficient predicates.

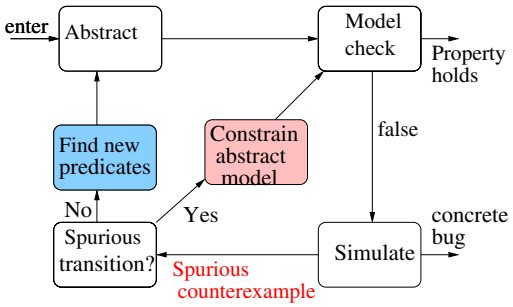

Figure 1: Abstraction-refinement loop in this paper.

In this case, we apply a novel word-level refinement technique: 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, this is the first time syntactic weakest preconditions of circuits have been used for refinement in predicate abstraction. The overall flow of the various techniques described above is shown in Fig. 1.

Related work. Namjoshi et al. [20] use weakest preconditions for extracting finite state abstractions, from possibly infinite state programs. However, no automatic refinement procedure is described for spurious counterexamples. In [13], a SAT-based technique for predicate abstraction of circuits given in Verilog is introduced. The circuit is synthesized and transformed into net-list level. A SAT solver is used to compute the abstraction, which makes it possible to support all bit-level constructs. However, if refinement becomes necessary, only bit-level predicates can be introduced.

Andraus et al. [1] present a scheme for automatic abstraction of behavioral RTL Verilog to the CLU language used by the UCLID system [5]. However, the abstractions produced by their approach can be coarse as there is no direct support for bit-vectors and bit-wise operators in the CLU language. Also no refinement is done when a spurious counterexample is obtained.

Outline. In section 2, we provide the notation used throughout the paper. Section 3 describes the SAT-based predicate abstraction. Techniques for partitioning predicates are given in section 4. We present techniques for word-level abstraction refinement in section 5. We report the experimental results in section 6, and conclude the paper in section 7. The formal semantics of the subset of Verilog we handle can be found in our technical report [11].

2. PRELIMINARIES

Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ denote the set of registers. The state of the Verilog program is given by the valuation of these registers. We consider the external inputs to be registers without a next-state function. Let $Q \subseteq \mathcal{R}$ 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 $r_i \in Q$ by $f_i(r_1, \ldots, r_n)$, or $f_i(\bar{r})$ using vector notation. The transition relation $R(\bar{r}, \bar{r}')$ relates the current state $\bar{r} \in S$ to the next state $\bar{r}'$ and is defined as follows:

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

Example: Consider a register x of size 8 bits. In each clock cycle, if x is less than five, then the value of x is incremented by two, else the value of x remains unchanged. Thus, the next state function of x is given by $((x < 5)?(x+2) : x)$, where ? denotes the choice operator. Note that we have a next state function for the whole register x and not for the individual bits in x.

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 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 [9], 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 $\hat{R}$.

$$\hat{R} \;:= \{(\bar{b},\bar{b}') \mid \exists \bar{r},\bar{r}' \in S: \quad R(\bar{r},\bar{r}') \wedge \alpha(\bar{r}) = \bar{b} \wedge \alpha(\bar{r}') = \bar{b}'\} \tag{1}$$

The initial state $I(\bar{r})$ is abstracted as follows:

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

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 $\bar{b}$.

$$\hat{P}(\bar{b}) \;:=\; \forall \bar{r} \in S: \; (\alpha(\bar{r}) = \bar{b}) \;\Rightarrow\; P(\bar{r})$$

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

SAT-based predicate abstraction. In [12], 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 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. The set of abstract transitions $\hat{R}$ is computed by transforming equation 1 into conjunctive normal form (CNF) and passing the resulting formula to a SAT solver. The satisfying assignments obtained form the abstract transition relation $\hat{R}$.

Example: Let the transition relation $R(x,y,x',y')$ be $x' = y \wedge y' = x$. Let the set of predicates be $\{x = 1, y = 1\}$. The equation for computing the $\hat{R}$ is given as follows:

$$\exists x,y,x',y' : (b_1 \Leftrightarrow (x = 1)) \wedge (b_2 \Leftrightarrow (y = 1)) \wedge R(x,y,x',y') \;\wedge (b'_1 \Leftrightarrow (x' = 1)) \;\wedge (b'_2 \Leftrightarrow (y' = 1))$$

The set of satisfying assignments to the above equation results in $\hat{R} := ((b'_1 \Leftrightarrow b_2) \;\wedge\; (b'_2 \Leftrightarrow b_1))$.

Note that the predicates used for abstraction can be arbitrary Boolean expressions allowed by the Verilog syntax. Thus, the predicates can involve operators for concatenation, extraction etc. For example, `a[3:0]>7`, `ram[{addr,1'b0}]==d[9:2]` are allowed as predicates.

4. PREDICATE PARTITIONING

We call the computation of the exact existential abstraction as described in the previous section the *eager approach*. In the worst case, the number of satisfying assignments is exponential in the number of predicates. As a result computing abstractions using the eager approach can be very slow even for a small number of predicates. The speed of the abstraction computation can be improved 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. The SLAM toolkit, for example, limits the number of predicates in each theorem prover query. Extending the idea in SLAM we partition the set of the predicates and their next-state versions into smaller sets of related predicates. We call these sets *clusters*, and denote them by $C_1, \ldots, C_l$, with $C_j \subseteq \{\pi_1, \ldots, \pi_k, \pi'_1, \ldots, \pi'_k\}$, where π'_i denotes the next state version of π_i. The equation for abstracting the transition system with respect to C_j is given as follows:

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

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

The conjunction of l abstract transition relations $\hat{R}_1, \ldots, \hat{R}_l$ results in the abstract transition relation $\hat{R}$:

$$\hat{R} \;:=\; \bigwedge_{i=1}^{l} \hat{R}_i \tag{2}$$

We refer to the above technique of partitioning the set of predicates in various clusters, and using these clusters for computing the abstraction $\hat{R}$, as *predicate partitioning*.

Claim. If $\hat{Q}$ denotes the transition relation obtained by using the eager approach (Eqn. 1), and $\hat{R}$ denotes the transition relation obtained by predicate partitioning (Eqn. 2), then $\hat{Q} \Rightarrow \hat{R}$.

The above claim is proved by observing that for all $1 \leq j \leq l$, $\hat{Q} \Rightarrow \hat{R}_j$. Thus, $\hat{R}$ is an over-approximation of $\hat{Q}$, and hence, a conservative over-approximation of the original circuit.

We evaluate two different techniques for creating predicate clusters used in predicate partitioning, *cone partitioning* and partitioning for *lazy abstraction*.

Syntactic cone partitioning. This technique clusters a next state predicate with a set of current state predicates if the variables appearing in the current state predicates affect the value of the next state predicate. *Example:* Let the transition relation $R(x,y,x',y')$ be $x' = y \wedge y' = x$. Let the set of predicates be $\{x = 1, y = 1, x' = 1, y' = 1\}$. The value of the predicate $y' = 1$ is affected by the value of x (as y' equals x). Note that the value of $y' = 1$ is not affected by the value of y. Thus, we keep $x = 1$ and $y' = 1$ together in a cluster C_1. Similarly, the other cluster $C_2 := \{y = 1, x' = 1\}$ is obtained.

Syntactic partitioning for lazy abstraction. The idea of lazy abstraction [17] is to defer the abstraction until required by a spurious counterexample. A completely lazy abstraction corresponds to using no clusters. Thus, the initial abstraction is simply `true`. Motivated by this idea, we 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 a coarse abstraction. One simple technique is to create k clusters, each containing exactly one next-state predicate π'_i. We follow a variant of this technique: all next-state predicates that contain the exact same set of variables are kept in the same cluster. This is useful if the given set of predicates contains many mutually exclusive (or related) predicates such as $x' = 1, x' = 2, x' = 3$. 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.

Example: Let the set of next-state predicates be $\{x' < 200, x' = 31, y' = 10, z' > 10\}$. The clusters produced for lazy abstraction are $C_1 := \{x' < 200, x' = 31\}$, $C_2 := \{y' = 10\}$, $C_3 := \{z' > 10\}$.

Once the abstraction of the concrete system is obtained, we model-check it using the NuSMV model-checker [21]. If the abstract model satisfies the property, the property also holds on the original, concrete circuit. If the model checking of the abstraction returns false, 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.

5. ABSTRACTION REFINEMENT

When refining the abstract model, we distinguish between two cases of spurious behavior, as done in [13]: **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 eager 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. **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.

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 $\bar{s}_i$. Recall that π_i' denotes 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:

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

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.

Removing spurious transitions. If the abstract transition is spurious, the CNF instance is unsatisfiable. In this case, we use the ZChaff SAT solver [19] for finding a small subset of clauses in the CNF instance which is also unsatisfiable (called an *unsatisfiable core*). It is computed by making use of the proof of unsatisfiability of the SAT instance [25]. We use the unsatisfiable core to determine a subset of existing predicates which are sufficient to show that the abstract transition is spurious. The spurious transition is removed from the abstract model by adding a constraint in terms of the predicates appearing in the unsatisfiable core.

Example: Consider the abstract transition from $\bar{s} = \{b_1 = 0, b_2 = 1\}$ to $\bar{t} = \{b_1' = 0, b_2' = 0\}$, where b_1, b_2 represent the current state values and b_1', b_2' represent the next state values of predicates $x > 2$, $y = 3$, respectively. Let the next state functions be $x' = y$, $y' = x$. Observe that in $\bar{s}$, the predicate $y = 3$ is true. This implies that $x' = 3$, and thus, b_1' must hold in $\bar{t}$. However, b_1' is false in $\bar{t}$ and thus, the transition from $\bar{s}$ to $\bar{t}$ is spurious. This transition can be eliminated by adding the constraint $\neg(\neg b_1 \wedge b_2 \wedge \neg b_1' \wedge \neg b_2')$ to the abstract model. However, this 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 $b_2 = 1$, and $b_1' = 0$. The unsatisfiable core allows us to discover this fact. Now we can eliminate this abstract transition and many more spurious transitions by adding the following constraint to the abstract model: $\neg(b_2 \wedge \neg b_1')$.

Removing spurious prefixes. In [13], the elimination of spurious prefixes is done by adding a monolithic bit-level predicate. In contrast to that, we make use of weakest preconditions as done in software verification. We generate new word-level predicates from the weakest precondition of the given property with respect to the transition function given by the RTL level circuit as described next.

Weakest preconditions for Verilog. In software verification, the weakest precondition $wp(st, \gamma)$ of a formula 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 Q. That is, external inputs do not appear in this set. 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. 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 $\bar{y}$.

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 counterexample of length $l > 0$, we compute $wp_l(\bar{f}, \tau)$, where τ is the safety property we are interested in checking. Intuitively, τ holds holds after l transitions iff $wp_l(\bar{f}, \tau)$ holds before l transitions. Refinement corresponds to adding the boolean expressions occurring in $wp_l(\bar{f}, \tau)$ to the existing set of predicates.

Example: Let the property be $x < 3$, and the next state function for the register x be $((x < 5)?(x+2) : x)$. Suppose we obtain a spurious counterexample of length equal to 1. The weakest precondition wp_1 of $x < 3$ is given as $(((x\ <\ 5)\ ?\ (x+2)\ :\ x\ ) < 3)$.

Simplifying the weakest preconditions. The problem with the approach above is that 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.

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 file 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 in the weakest precondition as the new predicates (more details in [11]).

Example: Suppose the guard $x < 5$ is present in the current set of predicates. Let the value of $x < 5$ in an abstract state $\bar{s}$ be `true`.

The weakest precondition given as $(((x < 5)\ ?\ (x+2)\ :\ x) < 3)$, can be simplified in $\bar{s}$, by substituting the value of $x < 5$. This results in a new predicate $x+2 < 3$ (or $x < 1$).

With weakest precondition simplification, it is not always enough to compute the weakest precondition of the given property for refinement. For example, we may need the weakest precondition of the guard $x < 5$ in the example above, which will not be computed if we do the simplification of the weakest precondition. Thus, one needs to identify a subset of existing predicates, whose weakest precondition must be computed for removing the spurious behavior. This is done by simulating the entire spurious counterexample. The unsatisfiable core obtained identifies a subset of existing predicates responsible for the spurious behavior. If a copy of predicate p in cycle k appears in the unsatisfiable core, then we compute the weakest precondition of p for k steps.

6. EXPERIMENTAL RESULTS

The experiments are performed on a 1.5 GHZ AMD machine with 3 GB of memory running Linux. A time limit of one hour and a memory limit of 700 MB was set for each run. We compare our technique against a non-commercial version of the Cadence SMV model checker [14]. The Cadence SMV tool is a net-list based model checker, which implements localization reduction.

6.1 Benchmarks and Properties Verified

Our benchmarks are taken from the Instruction Cache Unit (ICU), and the Instruction Cache RAM (ICRAM) unit of the Sun PicoJava II microprocessor [23]. The ICU fetches the instructions from the instruction cache and passes them to the decode unit. We checked the property that in case of a cache read miss the ICU controller implementation simulates a *miss state* transition diagram given in the picoJava-II micro-architecture guide [23].

The ICRAM maintains a RAM of size 16KB (organized as 2048 entries of 64 bits) each. If the write is enabled (icu_ram_we[1:0] = 2'b10), then the value of data input (icu_din) is written to the higher 32 bits of the location addressed by the input address (icu_addr). This functionality of the ICRAM was encoded in form of a safety property using the current and the next state of the variables. Observe that the property depends on the contents of the RAM. Thus, even after applying the techniques such as localization reduction, the system will have 16KB ($16 \times 1024 \times 8$) latches. In order to simplify the problem, we verified the property for the RAM of sizes 512 byte, 1KB, 2KB, and 4KB. These benchmarks are denoted as M512B, M1KB, M2KB, M4KB, in the Table 1, respectively.

The benchmarks starting with "AR" perform arithmetic operations on two registers a and b in each clock cycle. The next state functions of a and b are given as follows: $a' := (a < 100)?(a+b) : a$ and $b' := a$. Initial values of these registers are 1 and 0, respectively. We check the property that $a < 200$ in each clock cycle. The benchmarks AR100, AR200, AR500, AR1000 in Table 1 are variants of this circuit obtained by increasing the size of the registers a and b.

The experimental results are summarized in Table 1. The column "Latches" contains the total number of latches in the cone of influence of the property. We compare two different techniques for verifying these benchmarks. The columns marked with "Predicate Abstraction" contain the results of applying the predicate abstraction and refinement techniques discussed in this paper. The "Time", "Abs", "MC", and "Ref" columns contain the total time, followed by the breakup of the total time into the time taken by abstraction, model checking, and refinement including simulation. The "P/I" column contains the final number of predicates followed by the total number of iterations.

The results of running Cadence SMV are given in the "CSMV" column. Of the various options to Cadence SMV, we found the counterexample-based abstraction refinement option `-absref3` to result in the best performance when checking the various benchmarks. We report the total time taken by Cadence SMV when running with this option.

Benchmark	Latches	Predicate Abstraction					CSMV
		Time	Abs	MC	Ref	P/I	Time
ICU	28	1.3	0.6	0.1	0.6	5/1	0.1
M512B	4137	107.1	2.2	0.8	104.1	3/8	2.3
M1KB	8234	180.8	9.3	0.8	170.7	3/8	7.5
M2KB	16427	450.7	24	0.9	425.3	3/8	25.0
M4KB	32796	843.3	37	0.8	805.5	3/8	-
AR100	202	3.5	2.8	0.12	0.55	3/3	182.4
AR200	402	9.6	8.4	0.12	1.1	3/3	2147
AR500	1002	32.2	29.3	0.12	2.8	3/3	*
AR1000	2002	122.6	116.8	0.16	5.6	3/3	*

Table 1: Experimental results: All runtimes are in seconds. A "*" indicates a timeout of 1 hour. A "-" indicates the model checker terminated due to the large number of BDD variables.

6.2 Summary of Results

On the ICU benchmark, Cadence SMV outperforms predicate abstraction. Since the state space of this benchmark is very small, no abstraction is necessary. On the M512B, M1KB, and M2KB benchmarks, the runtime of Cadence SMV is better than the predicate abstraction runtime. However, Cadence SMV is not able to handle the M4KB benchmark which has a much larger state space. Cadence SMV timeouts on the AR500 and the AR1000 benchmarks, while the predicate abstraction method is able to complete these benchmarks with better runtimes. Some of the inferences drawn from these observations are as follows:

- The runtime of localization reduction grows exponentially with each newly added latch. This trend is visible in the AR100 to AR1000 benchmarks. In these benchmarks, Cadence SMV is not able to reduce the number of latches in the abstract model created, making the model checking step expensive.
- When using predicate abstraction the size of the abstract model remains constant even when the number of latches are increased. This is because for many properties the number of word-level predicates needed for the proof does not grow, as the sizes of the registers appearing in the property is increased. This trend is visible in the M* and the AR* benchmarks, where the number of predicates needed to prove the property does not change as the number of latches is increased. Thus, the model checking (MC) time is similar across M* and across AR* benchmarks.
- The computation of the abstract model using predicate abstraction requires the use of a decision procedure, which is a SAT solver in our case. In general, the problem of computing the precise existential abstraction (Eqn. 1) is itself exponential in the number of predicates and the size of the transition relation (number of latches). However, this complexity is not observed in our experiments due to two reasons: 1) the use of state of art SAT solvers like ZChaff [19] and Siege [22] for computing abstraction, 2) the use of predicate partitioning technique (Sec. 4) to handle the large number of predicates. The experimental results indicate that the abstraction computation time does not grow exponentially with each newly added latch.

A plot of the total time needed by the predicate abstraction technique compared to the number of latches is given in Fig. 2(a) and Fig. 2(b) for the M* and the AR* benchmarks, respectively. Observe that the runtime does not increase exponentially with number of latches. These experiments support the hypothesis that the it-

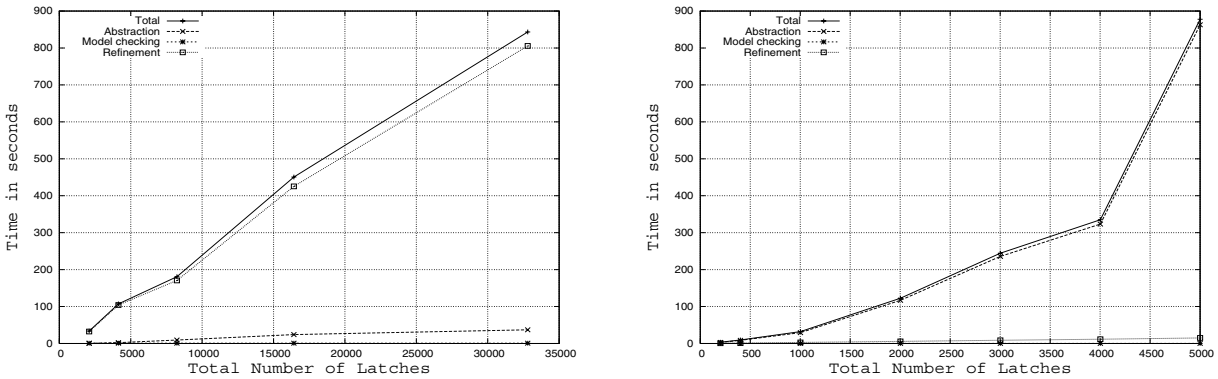

Figure 2: Runtime of the predicate abstraction and refinement with respect to number of latches: (a) M* benchmarks (b) AR* benchmarks

erative predicate abstraction and refinement can *scale to circuits involving thousands of latches.*

Predicate partitioning techniques. We use two different techniques for creating predicate clusters (section 4), namely cone partitioning and partitioning for lazy abstraction. Both techniques are complementary to each other. Cone partitioning attempts to keep all related predicates together, thus, the abstract model produced is more precise as compared to lazy abstraction. However, the time taken for abstraction using cone partitioning can become a bottleneck. In such cases, lazy abstraction works well if the property can be checked using a coarse abstract model. Cone partitioning is used for AR* benchmarks, while the lazy abstraction is used for M* benchmarks. Observe that the total time is dominated by the abstraction time in case of AR* benchmarks, and the refinement time in case of the M* benchmarks. Additional experiments can be found in our technical report [11].

Performance on Vapor benchmarks. The Vapor tool [1] performs abstraction of the Verilog models to the CLU language [5] for verification. In [1], Vapor was used to verify control related properties of the `ITC99` circuits. We found that 18 of the 22 properties of the `ITC-b13` benchmark are proved trivially using predicate abstraction. The time taken is less than one second, and two predicates, one refinement iteration is required on average. The remaining four properties are proved in less than 4 seconds, and 12 predicates, four refinement iterations are required on average. The other `ITC99` circuits reported in [1] are also handled in a straightforward way.

7. CONCLUSIONS

Localization reduction fails if the property depends on too many latches. We overcome this limitation by using a stronger abstraction technique called *predicate abstraction*. We present novel algorithms for computing and refining predicate abstractions of circuits given in RTL Verilog using SAT.

There are two challenges when using predicate abstraction on Verilog: 1) the computation of the abstract model, and 2) how to obtain good predicates. We address the first challenge by introducing *predicate partitioning*, a hybrid between eager abstraction and lazy abstraction [17]. We make use of unsatisfiable cores of SAT instances in order to eliminate multiple spurious transitions caused by an over-approximation of the eager abstraction.

In order to obtain the right set of predicates, we compute new word-level predicates by using weakest preconditions of Verilog RTL. Weakest preconditions are commonly used in the software domain. However, this technique was not applied to hardware before, 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. On the large benchmarks, our new algorithm scales well with the design size and clearly outperforms existing algorithms that use localization reduction. Our techniques are implemented in a tool called `VCEGAR`, which is publically is available from `http://www.cs.cmu.edu/~modelcheck/vcegar`.

8. REFERENCES

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

[2] T. Ball, B. Cook, S. Das, and S.K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS*, pages 388–403, 2004.

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

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Yhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999.

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

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

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

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.

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

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

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

[12] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods In System Design*, 25, 2004.

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

[14] www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[15] S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS*, 2001. June 2001, Boston, USA.

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

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

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

[19] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[20] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV 00*, number 1855 in LNCS, 2000.

[21] http://nusmv.irst.itc.it/.

[22] http://www.cs.sfu.ca/~loryan/personal.

[23] http://www.sun.com/processors/technologies.html.

[24] 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.

[25] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *SAT*, 2003.