Hybrid CEGAR: Combining Variable Hiding and Predicate Abstraction

Chao Wang
NEC Laboratories America

Hyondeuk Kim
University of Colorado

Aarti Gupta
NEC Laboratories America

Abstract

Variable hiding and predicate abstraction are two popular abstraction methods to obtain simplified models for model checking. Although both methods have been used successfully in practice, no attempt has been made to combine them in counterexample guided abstraction refinement (CEGAR). In this paper, we propose a hybrid abstraction method that allows both visible variables and predicates to take advantages of their relative strengths. We use refinement based on weakest preconditions to add new predicates, and under certain conditions trade in the predicates for visible variables in the abstract model. We also present heuristics for improving the overall performance, based on static analysis to identify useful candidates for visible variables, and use of lazy constraints to find more effective unsatisfiable cores for refinement. We have implemented the proposed hybrid CEGAR procedure. Our experiments on public benchmarks show that the new abstraction method frequently outperforms the better of the two existing abstraction methods.

1. Introduction

Variable hiding [18, 15] and predicate abstraction [8] are two frequently used abstraction techniques in the counterexample guided abstraction refinement framework (CEGAR [15, 4, 2]). Both methods create over-approximated models, and therefore are conservative with respect to universal properties such as LTL [20]. Since the abstract model may have more behaviors than the concrete model, if a property holds in the abstract model, it also holds in the concrete model; however, if a property fails in the abstract model, it may still be correct in the concrete model. The abstraction refinement loop consists of three phases: abstraction, model checking, and refinement. Typically, one starts with a coarse initial abstraction and applies model checking. If the property fails in the abstract model and the model checker returns an abstract counterexample, a concretization procedure is used to check whether a concrete counterexample exists. If a concrete counterexample does not exist, the abstract counterexample is *spurious*. Spurious counterexamples are used during refinement to identify the needed information currently missing in the abstraction.

The variable hiding abstraction [18, 15, 4], or localization reduction, partitions the set of state variables of the model into a *visible* subset and an *invisible* subset. In the abstract model, the transition functions of visible variables are preserved as is, and the invisible variables are abstracted as *pseudo-primary inputs*. Since the invisible variables are left unconstrained, the abstract model has all possible execution traces of the original model, and possibly more. The cone-of-Influence (COI) reduction can be regarded as a special case of variable hiding abstraction, wherein variables in the transitive fan-in of the property variables are marked as visible. Program slicing in software analysis is similar to COI reduction, and can be viewed as another special case. Compared to COI reduction, which produces an *exact* model for deciding the given property, variable hiding in general is more aggressive and may lead to spurious counterexamples.

In variable hiding, the abstraction computation is efficient. Given a set of visible variables, the abstract model can be built directly from a textual description of the original system, without the need for computing the concrete transition relation in the first place. This is advantageous because in practice the concrete transition relation may be too complex to compute. However, in variable hiding only *existing* state variables and transition functions can be used to construct the abstract model, which in general limits the chance of finding a concise abstraction. Despite this restriction, variable hiding has been relatively successful in abstracting large hardware designs [4, 23], especially when combined with the use of SAT solvers [5, 19, 9, 1, 17, 24]. This is because the models tend to be well-partitioned and as a result, system properties often can be localized to a few submodules.

Predicate abstraction [8] is more flexible than variable hiding since it allows a choice of predicates for abstraction, and has been used to verify both software and hardware [2, 6, 12, 14]. In predicate abstraction, a finite set of predicates is defined over the set X of concrete state variables and each predicate corresponds to a fresh Boolean variable $p_i \in P$. With these predicates, the model is mapped from the concrete state space (induced by X) into an abstract state space (induced by P). The main disadvantage of predication abstraction is the expensive abstraction computation. Unlike in variable hiding, this computation is not compositional; the worst-case complexity is exponential in the number of predicates. When the number of predicates is large, the abstraction computation time often goes up significantly. Cartesian abstraction [7, 3] has been proposed to alleviate this problem; however, it leads to a further loss of accurracy in the abstraction.

Traditional hardware models are well structured, in that existing state variables and transition functions are often sufficient for constructing a concise abstraction for most user-defined properties. In this case, exploiting the extra flexibility provided by predicate abstraction may not be very crucial. However, with the increasing use of higher level modeling and description languages in today's hardware design practice, the functional and structural partitionings may no longer directly correspond with each other, and as a result, the correctness of a property may not be easily localized to a few variables or submodules. In such cases, predicate abstraction is generally more effective. Furthermore, for system-level designs the boundary between hardware and software is getting blurred, and there is a need for abstraction method that work well on both.

We believe that variable hiding and predicate abstraction can be

.

regarded as two extremes that have complementary strengths. In this paper, we propose a hybrid approach that explores the spectrum between the two extremes, to provide more robust and concise abstractions. Specifically, we propose a hybrid abstraction that allows both visible state variables and predicates in the same abstract model. We present algorithms for optimizing the abstraction computation, and for deciding when to add more visible state variables and when to add more new predicates within a CEGAR framework. We also present heuristics for improving the overall performance, based on static analysis to identify useful candidates for visible variables, and use of lazy constraints to find more effective unsatisfiable cores for refinement. We have implemented the new hybrid abstraction and CEGAR framework, and demonstrate its application for verifying word-level Verilog designs. Our experimental results show that the new method matches the better of the two existing abstraction methods, and outperforms them both in many cases. We believe this is due to the hybrid abstract model being more concise than either extreme when allowed to have both visible variables as well as predicates.

Although we focus on hardware verification, the main ideas (and indeed our prototype implementation of hybrid CEGAR) are directly applicable to verifying software programs also. The flexibility in our hybrid approach provides a uniform way to handle models derived from both hardware and software, and results in effective and concise abstractions automatically. In the remainder of the paper, we will briefly mention handling of software programs in our model representation, but will focus more on details related to verifying hardware designs described in word-level Verilog.

2. Abstraction Methods

Let $X = \{x_1, \ldots, x_m\}$ be a finite set of variables representing the current state of the model, and $X' = \{x'_1, \ldots, x'_m\}$ be the set of variables representing the next state; then a valuation $\tilde{X}$ or $\tilde{X}'$ of the state variables represents a state. A model is denoted by the tuple $\langle T, I\rangle$, where $T(X, X')$ is the transition relation and $I(X)$ is the initial state predicate. $\tilde{X}$ is an initial state if $I(\tilde{X})$ holds; similarly, $(\tilde{X}, \tilde{X}')$ is a state transition if $T(\tilde{X}, \tilde{X}')$ is true. In symbolic model checking, the transition relation of a model and the state sets are represented symbolically by Boolean functions in terms of a set of state variables. For hardware models, all state variables are assumed to belong to finite domains. The *concrete* transition relation $T(X, X')$ is defined as follows,

$$T \;=\; \bigwedge_{i=1}^{m} T_i(X, X') \;\;,$$

where T_i is an *elementary* transition relation. Each $x_i \in X$ has an elementary transition relation T_i, defined as $x'_i = \delta_i(X)$, where $\delta_i(X)$ is the transition function of x_i.

Variable hiding marks a subset $X_v = \{x_1, \ldots, x_n\} \subseteq X$ of state variables as *visible*. The set of remaining variables (called *invisible* variables) is denoted by $X_{inv} = (X \setminus X_v)$. For $x_i \in X_{inv}$, let $T_i = \mathsf{true}$. The abstract model (via variable hiding) is defined as $\langle T_{\mathcal{V}}, I_{\mathcal{V}}\rangle$ such that,

$$T_{\mathcal{V}} = \textstyle\bigwedge_{i=1}^{n} T_i(X, X')$$
$$I_{\mathcal{V}} = \exists\, X_{inv} \;.\; I(X)$$

$T_{\mathcal{V}}(X, {X_v}')$ may depend on some *invisible* current-state variables in X_{inv}, which are treated as free inputs. In model checking, free inputs are existentially quantified during image computation. One can explicitly remove X_{inv} variables by existential quantification,

$$\widehat{T}_{\mathcal{V}} = \;\exists\, X_{inv} \;.\; T_{\mathcal{V}}(X, {X_v}')$$

However, this may cause a further loss of accuracy since $T_{\mathcal{V}} \subseteq \widehat{T}_{\mathcal{V}}$. In practice, using $T_{\mathcal{V}}$ as opposed to $\widehat{T}_{\mathcal{V}}$ in model checking often gives better results.

In predicate abstraction, we consider a set $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ of predicates over variables in X. A new set $P = \{p_1, \ldots, p_k\}$ of Boolean state variables are added for the predicates such that p_i is true iff $\mathcal{P}_i(X)$ evaluates to true. The abstract model (via predicate abstraction) is defined as $\langle T_{\mathcal{P}}, I_{\mathcal{P}}\rangle$ such that,

$$T_{\mathcal{P}} = \exists X, X' \;.\; T(X, X') \wedge \textstyle\bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X) \wedge p'_i \leftrightarrow \mathcal{P}'_i(X)$$
$$I_{\mathcal{P}} = \exists X \;.\; I(X) \wedge \textstyle\bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X)$$

The mapping from T to $T_{\mathcal{P}}$, or *predicate image* computation, is expensive. Most existing tools [6, 16, 12] developed for hardware verification use either BDDs or a SAT solver to compute the predicate image. For instance, one can build a Boolean formula for $T(X, X') \wedge \bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X) \wedge p'_i \leftrightarrow \mathcal{P}_i(X')$ as the input to a SAT solver; $T_{\mathcal{P}}(P, P')$ is obtained by enumerating all the satisfying solutions of the formula in terms of variables in P and P'.

In the worst case, the number of satisfying assignments in $T_{\mathcal{P}}$ is exponential in the number of predicates. Abstraction computation may become intractable when the number of predicates is large. In such cases, one has to resort to a less precise abstract transition relation $\widehat{T}_{\mathcal{P}}$ (such that $T_{\mathcal{P}} \subseteq \widehat{T}_{\mathcal{P}}$). In Cartesian abstraction [7, 3], for instance, the set P is partitioned into smaller subsets where predicate images are computed separately for each individual subset, and the resulting relations are conjoined together to obtain $\widehat{T}_{\mathcal{P}}$.

3. The Cost of Abstractions

We evaluate the conciseness of abstraction in terms of the number of Boolean state variables in the abstract model. In model checking, the state space is exponential in the number of state variables, making the number of state variables an effective indicator of the hardness of model checking.

3.1 The Cost of a Predicate

In variable hiding abstraction, a visible variable $x_i \in X_v$ with domain $dom(x_i)$ has a cost equal to $log|dom(x_i)|$, where $|dom(x_i)|$ is the cardinality of the set. We assume that binary encoding is used for x_i in the concrete model and $log|dom(x_i)|$ is the number of bits for encoding x_i. The cost of an invisible variable is 0. In predicate abstraction, since all variables in P are in the Boolean domain, the cost of each $p_i \in P$ or each corresponding predicate $\mathcal{P}_i(X)$ is 1. To facilitate the comparison of predicate abstraction with variable hiding, we distribute the cost of p_i (which is 1) evenly to the concrete state variables in $\mathcal{P}_i(X)$ as follows: If there are l supporting X variables appearing in the expression $\mathcal{P}_i(X)$, the predicate adds a cost of $(1/l)$ to each of these variables. When there are visible variables, we distribute the cost of a predicate evenly to its supporting invisible variables only. If all the variables appearing in $\mathcal{P}_i(X)$ are already made visible, then the predicate is redundant since adding it will not improve the accuracy of the abstraction.

Example 1 *The predicate $\mathcal{P}_1 : (u + v > 10)$ adds $1/2$ each to the costs of u and v; the predicate $\mathcal{P}_2 : (u - 2v \leq 3w)$ adds $1/3$ each to the costs of u, v, and w.*

Example 2 *When u is a visible variable, the predicate $\mathcal{P}_1 : (u + v > 10)$ adds 1 to the cost of v, the predicate $\mathcal{P}_2 : (u - 2v \leq 3w)$ adds $1/2$ each to the costs of v and w, and $\mathcal{P}_3 : (u \neq 0)$ is redundant.*

The total cost distributed to a concrete state variable $x_i \in X$ by predicates, denoted by $cost_{\mathcal{P}}(x_i)$, is the sum of the costs incurred

by all the predicates in which x_i appears. Recall that in variable hiding, the cost of $x_i \in X$ is $log|dom(x_i)|$ when it is visible. Therefore, if $cost_{\mathcal{P}}(x_i) > log|dom(x_i)|$, then predicate abstraction is considered to be less concise, since making x_i visible requires less Boolean state variables than representing the predicates. On the other hand, if $cost_{\mathcal{P}}(x_i) < log|dom(x_i)|$, then predicate abstraction is considered to be more concise.

3.2 The Cost of a Visible Variable

Variable hiding can be viewed as a special case of predicate abstraction, wherein all possible valuations of a visible variable are provided as predicates.

In predicate abstraction, $T_{\mathcal{P}}(P, P')$ is defined in the abstract state space; however, it can be mapped back to the original state space as follows, $T_{\mathcal{P}}(Y, Y') =$

$$\exists(P, P') \,.\; T_{\mathcal{P}}(P, P') \wedge \bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(Y) \wedge p_i' \leftrightarrow \mathcal{P}_i(Y')$$

Here Y and Y' are used to represent the same sets of state variables as X and X'. According to the mapping from $T(X, X')$ to $T_{\mathcal{P}}(P, P')$, we have $T_{\mathcal{P}}(Y, Y') =$

$$\exists(X, X') \,.\, T(X, X') \wedge \bigwedge_{i=1}^{k} \mathcal{P}_i(Y) \leftrightarrow \mathcal{P}_i(X) \wedge \mathcal{P}_i(Y') \leftrightarrow \mathcal{P}_i(X')$$

This equation is interpreted as follows: In order to allow all the visible variables in $T(X, X')$ to be preserved, while existentially quantifying invisible variables, one can define a set of new predicates for each $x_i \in X_v$ as follows: let $dom(x_i) = \{d_1, \ldots, d_l\}$, the set of predicates is $\{(x_i = d_1), (x_i = d_2), \ldots, (x_i = d_l)\}$.

However, preserving a visible variable x_i using these predicates may be inefficient since it requires $|dom(x_i)|$ new Boolean state variables, one for each predicate $(x_i = d_j)$. In contrast, making x_i visible only requires $log|dom(x_i)|$ Boolean state variables. If all these predicates (representing valuations of $x_i \in X_v$) are needed in order to decide the property at hand, then variable hiding provides an exponentially more concise abstraction.

4. Hybrid Abstraction

We present a hybrid abstraction method that allows visible variables and predicates to be in the same abstract model. Given a set $X_v = \{x_1, \ldots, x_n\}$ of visible variables and a set $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ of predicates, together with a set $P = \{p_1, \ldots, p_k\}$ of fresh Boolean variables, we define $T_{\mathcal{H}}(X, P, X', P')$, the new hybrid abstract transition relation, as follows,

$$T_{\mathcal{H}} \;=\; T_{\mathcal{V}}(X, X_v') \wedge T_{\mathcal{P}}(P, P') \wedge \bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X)$$

The model can be viewed as a parallel composition of two abstract models $T_{\mathcal{V}}$ and $T_{\mathcal{P}}$, defined in terms of X_v and P variables, and connected through the correlation constraint $\bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X)$. Without loss of generality, we assume that for every predicate $\mathcal{P}_i(X)$, at least one of its supporting X variables is invisible. (If all supporting X variables in $\mathcal{P}_i$ are visible, we remove the redundant predicate $\mathcal{P}_i$.)

Since adding the correlation (third conjunct in the above formula) can make model checking significantly more expensive (due to a large BDD for $T_{\mathcal{H}}$), we choose to use a less precise abstraction

$$\widehat{T}_{\mathcal{H}} \;=\; T_{\mathcal{V}}(X, X_v') \wedge \widehat{T}_{\mathcal{P}}(P, P')$$

Note that in addition to removing the correlation constraint between X and P, we also replace $T_{\mathcal{P}}$ by $\widehat{T}_{\mathcal{P}}$ (Cartesian abstraction) — this removes the potential correlation among P variables. The advantage of using $\widehat{T}_{\mathcal{P}}$ is that it is cheaper to compute. We can use the *syntactic cone partitioning* method [3, 12] to enumerate the elementary transition relation of each predicate separately. That is, each next-state predicate $\mathcal{P}_i(X')$ is clustered with all the current-state predicates $\mathcal{P}_j(X)$ such that the supporting X variables of $\mathcal{P}_j(X)$ affect the next-state values of the supporting X' variables of $\mathcal{P}_i(X')$. If the correlation among some P variables is missing because of this Cartesian abstraction, we will add it back if needed during refinement.

The loss of both kinds of correlation constraints can cause *spurious transitions* to appear in the abstract model. An abstract transition (s, s'), where s and s' are valuations of variables in $(P \cup X_v)$ and $(P' \cup X_v')$, respectively, is spurious if no concrete transition exists between (s, s'). There are two possible reasons for a *spurious counterexample* to appear: (1) there are spurious transitions because the abstraction computation in $\widehat{T}_{\mathcal{H}}$ is not precise; and (2) there are spurious counterexample segments because the sets of predicates and visible state variables are not sufficient. Note that a counterexample segment may be spurious even if none of its transition is spurious. During refinement, we first remove spurious transitions, by identifying some needed constraints over variables in X_v, X_v', P, and P' and conjoining them with $\widehat{T}_{\mathcal{H}}$.

4.1 Refinement for Spurious Transitions

For a spurious transition (s, s'), there is no concrete transition between s and s' but $\widehat{T}_{\mathcal{H}}(s, s')$ is `true`. Let the abstract state $s = \{\bar{p}_1, \ldots, \bar{p}_k, \bar{x}_1, \ldots, \bar{x}_n\}$ be a valuation of variables in $P \cup X_v$ and s' be a valuation of the variables in $P' \cup X_v'$, then (s, s') is spurious iff the formula $R(X, P, X', P')$, defined below, is not satisfiable.

$$T \wedge \bigwedge_{i=1}^{n} x_i = \bar{x}_i \wedge x_i' = \bar{x}_i' \wedge \bigwedge_{i=1}^{k} \mathcal{P}_i(X) \leftrightarrow \bar{p}_i \wedge \mathcal{P}_i(X') \leftrightarrow \bar{p}_i'$$

We build a Boolean formula R for each abstract transition in the given counterexample, and use a SAT solver to check its satisfiability. If the formula is not satisfiable, then the transition is spurious.

Removing the spurious transition requires the addition of a constraint $r(X, P, X', P')$, i.e., conjoining $\widehat{T}_{\mathcal{H}}$ with r. The additional constraint r is defined as follows,

$$r \;=\; \neg\Big(\bigwedge_{i=1}^{n} x_i = \bar{x}_i \wedge x_i' = \bar{x}_i' \wedge \bigwedge_{i=1}^{k} p_i \leftrightarrow \bar{p}_i \wedge p_i' \leftrightarrow \bar{p}_i'\Big)$$

The constraint r can be strengthened by dropping the equality constraints on some irrelevant X and P variables. The irrelevant variables can be determined by analyzing the UNSAT core reported by the SAT solver. An UNSAT core of a Boolean formula is a subset of the formula that is sufficient for proving the unsatisfiability. If certain subformulas in R, such as $x_i = \bar{x}_i$ and $\mathcal{P}_i(X) \leftrightarrow \bar{p}_i$, do not appear in the UNSAT core, then we can drop those equality constraints from r. The strengthened version of r is guaranteed to remove the spurious transition at hand.

4.2 Refinement for Spurious Segments

If there is no spurious transition in a spurious counterexample, more predicates or visible variables are needed to refine the abstract model. We adopt the following notation: let $X^j \cup P^j$ be the copy of $(X \cup P)$ at the j-th time frame. If the counterexample $s_0, \ldots, s_l$ is spurious, the following formula is unsatisfiable,

$$\bigwedge_{j=0}^{l-1} R(X^j, P^j, X^{j+1}, P^{j+1})$$

Note that each R is satisfiable by itself (R is defined in 4.1). We can remove the spurious counterexample by using a weakest precondition (WP) based refinement method [2, 11]. Since the weakest precondition computation relies on the underlying representation of the concrete model, we postpone the discussion of refinement to the next section, after we present this representation.

5. The Hybrid CEGAR Procedure

In this section, we present our hybrid CEGAR procedure based on models represented as Control and Data Flow Graphs (CDFGs). We formally define the CDFG representation first. Intuitively, this representation allows a separation between control and data state, such that control states are represented explicitly in terms of basic blocks (with guarded transitions between blocks) and data states are represented implicitly in terms of symbolic data variables (with assignments that update data state). This provides a natural representation for software programs, where control states correspond to control locations of the program and data states to values of program variables. For hardware models, we pick Verilog as a representative HDL, and describe how to obtain CDFGs from word-level Verilog designs—this has certain features that impact the proposed abstraction and refinement techniques.

The CDFG is a concrete model, serving as input to our hybrid CEGAR procedure. We compute the hybrid abstract model directly from the CDFG model, with respect to a set X_v of visible variables and a set $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ of predicates.

5.1 Transforming Verilog Designs into CDFGs

We transform the Verilog design through rewriting to a functionally equivalent reactive program. This reactive program is formally represented as a control and data flow graph (CDFG).

Definition 1 *A control and data flow graph (CDFG) is a 5-tuple $\langle B, E, V, \Delta, \theta \rangle$ such that*

- *$B = \{b_1, \ldots, b_L\}$ is a finite set of basic blocks, where b_1 is the entry basic block.*
- *$E \subseteq B \times B$ is a set of edges representing transitions between basic blocks.*
- *V is a finite set of variables that consists of actual variables in the design and auxiliary variables added for modeling the synchronous semantics of hardware description.*
- *$\Delta : B \rightarrow 2^{S_{asgn}}$ is a labeling function that labels each basic block with a set of parallel assignments. S_{asgn} is the set of possible assignments.*
- *$\theta : E \rightarrow S_{cond}$ is a labeling function that labels each edge with a conditional expression. S_{cond} is the set of possible conditional expressions.*

Example 3 *The Verilog example in Figure 1 computes Fibonacci numbers (taken from [12]). The equivalent CDFG is on the right. To maintain the synchronous semantics, we add the variable a_NS to hold the next-state value of the* reg *type variable a. The loop body corresponds to the execution of the* `always` *block exactly once (in a clock cycle). Since* `a <= b+a` *is a non-blocking assignment, i.e., a gets the current value of $(b + a)$ at the next clock cycle (not immediately), when translating the assignment* `b <= a`*, we do not substitute a by a_NS. Note that if it were a blocking assignment* `b = a+b` *and* `b = a` *in the Verilog description, we would have translated them into* `a_NS = b+a` *and* `b = a_NS`*.*

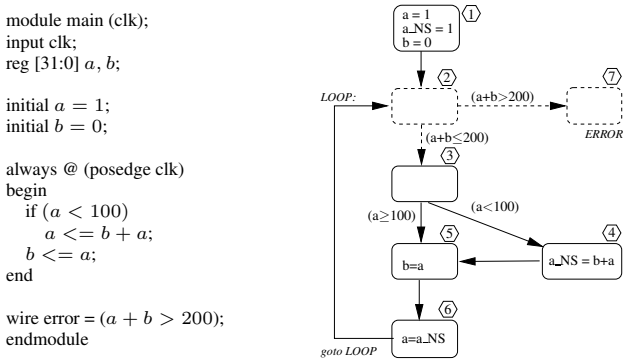

Figure 1: Rewriting the Verilog design into a CDFG.

In Figure 1, each rectangle in the CDFG is a basic block and the edges are labeled by conditional expressions. For example, the transition from block 3 to block 4 is guarded by `(a<100)`. Edges not labeled by any condition are assumed to have a `true` label. Block 1 is the entry block and block 7 is the error block. Reachability properties in the Verilog model are translated into assertion checks at the beginning of the loop. For example, $\mathsf{G}(a + b \leq 200)$ is translated into `if(a+b>200) ERROR`. The verification problem consists of checking whether the ERROR block is reachable from the entry block. More complex properties (PSL or LTL) can be handled by first synthesizing them into monitors followed by our Verilog-to-CDFG translator.

The transformation from Verilog designs to CDFG representations is made easy by introducing the $\_NS$ variables. Our CDFG is a representation similar to a software program, except that it has a single infinite loop to emulate the reactiveness of a hardware model. A clock cycle in the Verilog model corresponds to the execution of the loop body of the CDFG exactly once. Procedural statements from all the `always` blocks are sequentialized inside the infinite loop. Due to the addition of extra $\_NS$ variables for the reg type variables, e.g., a_NS for a as in Figure 1, sequentialization of multiple synchronously running `always` blocks may take an arbitrary order. In our implementation, we choose an order that can minimize the number of added $\_NS$ variables, since such optimizations reduce the size of the concrete model and therefore speed up model checking.

We choose this CDFG model in order to *directly* apply weakest-precondition based predicate abstraction refinement algorithms [2, 11] that have been developed for software programs. In the traditional synchronous model, these WP-based refinement algorithms are not directly applicable. Note that a synchronous model for Verilog designs is equivalent to summarizing all the statements in the loop body of our CDFG model, and creating a single basic block with a self loop, and with a set of parallel running assignments, one for each register variable. Such a synchronous circuit model (as opposed to a reactive program) is used in [12], where significant modifications have to be made to the WP-based refinement algorithm. Even with these modifications, one has to simultaneously consider all possible branches inside each clock cycle, making the WP computation likely to blow up. In contrast, in our CDFG representation, the weakest precondition computation can be localized to a single execution path, therefore offering the possibility of creating abstraction at a finer granularity. Abstraction computation is also faster in the reactive model since SAT enumeration can be applied to assignments in each individual block of the CDFG, as opposed to all assignments of the single block of a synchronous model simultaneously.

5.2 Hybrid Abstraction Refinement for CDFGs

We add a special state variable x_{pc}, the program counter (PC), to represent the control locations of the CDFG; the domain of x_{pc} is the set B of basic blocks. We assume that the set X of state variables of the model is $\{x_{pc}\} \cup V$. In the sequel, x_{pc} is always the first element of X and therefore x_{pc} and x_1 are interchangeable. We define the initial states of the model as $I = (x_1 = b_1)$, i.e., all possible valuations of V in the entry block b_1. If the error block is $b_{Err} \in B$, the property to be verified is $\mathsf{G}(x_1 \neq b_{Err})$. The set of parallel assignments in each basic block $b_j \in B$, denoted $\Delta(b_j)$, is written as $x_2, \ldots, x_m \leftarrow e_{2,j}, \ldots, e_{m,j}$, where $e_{m,j}$ is the expression assigned to x_m in block b_j. The guard $c_{j,k} = \theta(b_j, b_k)$ is the edge label from block b_j to block b_k; if there is no such edge, $c_{j,k} = \mathsf{false}$.

5.2.1 Trading Predicates for Visible Variables

In our hybrid abstraction, x_1 is always visible. We can start with $X_v = \{x_1\}$ and $P = \{\ \}$, and add new predicates using WP-based refinement. At the same time, we check whether it is advantageous to trade some existing predicates for visible variables as follows:

- *Add new visible variables:* For all $x_i \in X_{inv}$, if the total cost distributed to x_i by predicates is larger than $log|dom(x_i)|$, we make x_i visible, i.e., we add x_i in X_v.
- *Remove redundant predicates:* For a predicate $\mathcal{P}_i(X)$ whose supporting X variables are all visible, we remove the predicate and remove the corresponding p_i from P.
- *Modify correlation constraints:* For all existing correlation constraints $r(X_v, P, {X_v}', P')$, if p_i and p_i' are in the support of r, but $\mathcal{P}_i(X)$ has been declared as redundant and removed, we existentially quantify p_i and p_i' from r, i.e., we use $\exists(p_i, p_i')\ .\ r(X_v, {X_v}', P, P')$ instead.

Our initial hybrid abstract transition relation is $\widehat{T}_{\mathcal{H}} = T_{\mathcal{V}} \wedge \widehat{T}_{\mathcal{P}}$. Given a set X_v, we compute $T_{\mathcal{V}} = \bigwedge_i^n T_i$ as follows: For $x_1 \in X_v$, T_1 represents the control flow logic,

$$T_1 = \bigvee_{j=1}^{L} \bigvee_{k=1}^{L} (x_1 = b_j) \wedge (x_1' = b_k) \wedge c_{j,k}$$

Since invisible variables are treated as free inputs, if $c_{i,k}(X)$ contains invisible variables, the guard is nondeterministically chosen to be true or false (corresponding to `if(*)`). For $x_i \in X_v$ such that $i \neq 1$,

$$T_i \ = \ \bigvee_{j=1}^{L} (x_1 = b_j) \wedge (x_i' = e_{i,j}) \ ,$$

wherein $e_{i,j}(X)$ is the RHS expression assigned to x_i in block j. If there is no explicit assignment to x_i, then $e_{i,j} = x_i$.

Correlations between X and P variables, as well as correlations among P variables, are added lazily during refinement if spurious transitions occur. We have explained our refinement method for removing spurious transitions in Section 4.1. In the following section, we focus on removing spurious counterexamples by adding new visible variables and predicates to X_v and P, respectively.

5.2.2 Computing New Predicates in CDFGs

We use a weakest precondition based refinement algorithm for finding new predicates [2, 11]. We include its description in this section for the sake of completeness.

Given a spurious counterexample with no spurious transition, first, we identify a subset of conditional expressions (guards) that are needed to prove the infeasibility of a concrete path. We focus on one path in the CDFG, $blk_1, \ldots, blk_n$, determined by the counterexample. In this path, a basic block may appear more than once. The sequence of statements $\pi = st1, st2, \ldots$ corresponding to this path consists of two kinds of statements: a basic block blk_i corresponds to a set of parallel assignments $\Delta(blk_i)$, and a transition (blk_i, blk_{i+1}) corresponds to a branching statement `assume(c)` where $c = \theta(blk_i, blk_{i+1})$.

Example 4 *A spurious counterexample segment in Figure 1 corresponds to the sequence of basic blocks 1,2,3,5,6,2,7. The sequence of program statements is shown below:*

blocks	*transitions*	*statements*	*in UNSAT*
b_1		$a = 1;$	
		$a\_NS = 1;$	*yes*
		$b = 0;$	*yes*
	$b_1 \rightarrow b_2$		
b_2			
	$b_2 \rightarrow b_3$	*assume*$(a + b \leq 200);$	
b_3			
	$b_3 \rightarrow b_5$	*assume*$(a \geq 100);$	
b_5		$b = a;$	
	$b_5 \rightarrow b_6$		
b_6		$a = a\_NS;$	*yes*
	$b_6 \rightarrow b_2$		
b_2			
	$b_2 \rightarrow b_7$	*assume*$(a + b > 200);$	*yes*

Recall that we use a SAT solver to check the feasibility of a counterexample segment, where an unsatisfiable formula indicates that the counterexample is spurious (Section 4.2). For each $c = \theta(blk_i, blk_{i+1})$, we check whether c appears in the UNSAT core. If it appears in the UNSAT core, then the guard $c(X)$ is chosen and its weakest precondition $WP(\pi, c)$ is computed with respect to the spurious prefix π. $WP(\pi, \phi)$ is the weakest condition whose truth before the execution of π entails ϕ after the execution. Let $f(V/W)$ denote the substitution of W with V in function $f(W)$. $WP(\pi, \phi)$ is defined as follows: (1) for an assignment `s:(v=e)`, $WP(s, \phi) = \phi(e/v)$; (2) for a conditional statement `s:assume(c)`, $WP(s, \phi) = \phi \wedge c$; (3) for a sequence of statements `st1;st2`, $WP(st1; st2, \phi) = WP(st1, WP(st2, \phi))$. Refinement corresponds to adding the new predicates appearing in $WP(\pi, c)$ to the abstract model.

In this example, suppose that the guard $(a + b > 200)$ appears in the UNSAT core and π is the sequence of statements in blocks 1,2,3,5,6 and 2. Then $WP(\pi, a+b > 200)$ provides the following new predicates: $\mathcal{P}_1 :\ (a + b > 200)$, $\mathcal{P}_2 :\ (a\_NS + b > 200)$, $\mathcal{P}_3 :\ (a\_NS + a > 200)$. Adding these predicates will remove the spurious counterexample, because in block 1, $\mathcal{P}_3 = \mathsf{false}$; in blocks 5 and 6, $\mathcal{P}_2 = \mathcal{P}_3$; $\mathcal{P}_1 = \mathcal{P}_2$; this makes the transition from block 2 to block 7, guarded by $(\mathcal{P}_1)$, evaluate to false.

In our method, new predicates are directly added by the refinement algorithm, while visible variables are derived indirectly from the existing set of predicates by trading in predicates. An alternative is to selectively make some of the variables in the UNSAT core visible directly.

5.3 Eagerly Adding Syntactic Constraints

Recall that in $\widehat{T}_{\mathcal{H}}$, the constraints $\bigwedge_{i=1}^{k} p_i \leftrightarrow \mathcal{P}_i(X)$ are left out completely, to make the abstraction computation cheaper. Although some of the needed correlation constraints can be lazily added during refinement of spurious transitions (Section 4.1), this process can sometimes be inefficient due to the model checking expenses and number of refinement iterations. Therefore, we eagerly add certain cheaper constraints to $\widehat{T}_{\mathcal{H}}$ upfront.

We use the following syntactic rules to decide which constraints to add. We assume that $\widehat{T}_{\mathcal{H}} = T_{\mathcal{V}} \wedge \widehat{T}_{\mathcal{P}} = \bigwedge_{i=1}^{n} T_i \wedge \bigwedge_{i=1}^{k} T_{p_i}$.

(Rule 1) for $x_1 \in X_v$ (the PC variable),

$$T_1 = \bigwedge_{j=1}^{L} \bigwedge_{k=1}^{L} (x_1 = b_j) \wedge (x_1' = b_k) \wedge c_{j,k}$$

We process the conditional expressions $c_{j,k}(X)$ as follows:

- if $c_{j,k}$ is a constant (true or false), or all the supporting X variables of $c_{j,k}(X)$ are visible, then do not change it;
- else if $c_{j,k}(X)$ is syntactically equivalent to (the negation of) a predicate $\mathcal{P}_l(X)$, then replace it by (the negation of) p_l;
- otherwise, replace it with $(*)$, by adding a fresh primary input indicating a nondeterministic choice.

Note that in the third case, over-approximation of $\exists\, X_{inv}\ .\, C_{j,k}(X) \wedge \bigwedge_{i=0}^{k} p_i \leftrightarrow \mathcal{P}_i(X)$ is used; however, there is no approximation in the first two cases.

(Rule 2) for $x_i \in X_v$ such that $i \neq 1$ (non-PC variables),

$$T_i \ = \ \bigvee_{j=1}^{L} (x_1 = b_j) \wedge (x_i' = e_{i,j})$$

We choose not to approximate $e_{i,j}(X)$, the expression assigned to x_i in block j. We use $e_{i,j}$ as is, even if there are invisible variables in its support—these invisible variables become pseudo-primary inputs.

(Rule 3) for $p_i \in P$ (predicate variables), T_{p_i} is the elementary transition relation of p_i. We localize the computation of T_{p_i} to the computation of $T_{p_i,j}$ in each basic block j (similar to $x_i' = e_{i,j}$ for computing T_i),

$$T_{p_i} = \bigvee_{j=1}^{L} (x_1 = b_j) \wedge T_{p_i,j}$$

where $T_{p_i,j} = \exists\, X_{inv}\ .W P_j(\mathcal{P}_i) \wedge \bigwedge_{l=1}^{k} p_l \leftrightarrow \mathcal{P}_l(X)$. We use $W P_j(\mathcal{P}_i)$ to denote the weakest precondition of $\mathcal{P}_i(X)$ with respect to the assignments in block b_j. Since the existential quantification ($\exists\, X_{inv}\ .$) is expensive, we compute $T_{p_i,j}$ as follows:

- if $W P_j(\mathcal{P}_i)$ is a constant (true or false), or in the expression of $W P_j(\mathcal{P}_i)$ all the supporting X variables are already visible, then $T_{p_i,j} = (p_i' \leftrightarrow W P_j(\mathcal{P}_i))$;
- else if $W P_j(\mathcal{P}_i)$ is equivalent to (the negation of) another predicate $\mathcal{P}_l(X)$ or its negation, then $T_{p_i,j}$ equals (the negation of) the formula $(p_i' \leftrightarrow p_l)$;
- else if enumerating the solutions of p_i' and P variables for $p_i' \leftrightarrow W P_j(\mathcal{P}_i) \wedge \bigwedge_{l=1}^{k} p_l \leftrightarrow \mathcal{P}_l$ is feasible, we use the enumeration result instead. The result represents a relation over p_i' and P;
- otherwise, let $T_{p_i,j}$ be $p_i' = (*)$—by adding a fresh primary input to indicate a nondeterministic choice.

These heuristics are optional in that they do not affect the completeness of the overall CEGAR procedure. However, in practice they are very effective in reducing the spurious transitions, and hence avoiding the associated costs of model checking and large number of refinement iterations.

6. Additional Heuristics for Refinement

In this section, we discuss some additional heuristics to improve our hybrid CEGAR procedure. These are based on a static identification of candidate variables to make visible quickly, and a lazy constraint technique to improve the quality of the unsatisfiable cores used for the purpose of refinement.

6.1 Static Identification of Visible Variables

Before the CEGAR loop starts, we can use a simple static analysis on the CDFG to heuristically compute a small set of promising candidates of visible variables, i.e., variables that are likely to be made visible during the refinement process. In particular, we use the heuristic that for a state variable v, if (1) the next-state value of v is determined by some arithmetic expression over the current-state value of v, and (2) the variable v appears in some conditional expression guarding an error block, then we consider v as a promising candidate visible variable.

However, we do not add these precomputed candidates as visible variables upfront, since static analysis alone is not a good indicator that these variables are needed to verify the property at hand. Instead, during refinement, if a candidate variable v appears in the support of a predicate $\mathcal{P}_l(X)$ in the UNSAT core, then we immediately add v as a visible variable even if its accumulative cost $cost_P(v)$ is not yet large enough.

```
always @(posedge clk) begin
  ...
  if ( v > 1024 ) begin
    if (...)
      set the error bit;
  end
  ...
  v = v + x ;
end
```

Figure 2: Precomputing candidates of visible variables.

In other words, we bypass the step of first generating new predicates based on WP-based analysis. This is because in the subsequent refinement iterations, it is likely that a large number of new predicates (corresponding to the WP of $\mathcal{P}_l$) are needed, due to the nature of v's transition function. In Fig. 2, for instance, if the predicate `(v<1024)` is in the UNSAT core, the subsequent refinements will add `(v+x<1024),(v+2x<1024),...` as predicates—this is precisely the situation we want to avoid. In the hybrid abstraction, we avoid the situation by adding v as a visible variable immediately after the addition of the new predicate `(v<1024)`.

6.2 Lazy Constraints in UNSAT Core

Recall that we use an UNSAT core derived by the SAT solver for refinement, both for spurious transitions (by identifying correlation constraints in the UNSAT core) and for spurious segments (by identifying the conditional expressions in the UNSAT core). There are often multiple UNSAT cores for the same unsatisfiable problem, and the SAT solver by default may not generate an UNSAT core that is better for refinement.

Consider the example in Figure 3, where a spurious counterexample is shown on the left. Imagine that, for instance, lines 4 and 8 have complex loop bodies guarded by the conditions in lines 3 and 7, respectively; and the loop bodies contain `i=i+1` and `j=j+1`. For this spurious counterexample, there are four UNSAT cores:

- Lines 1, 2 and 3,
- Lines 5, 6 and 7,

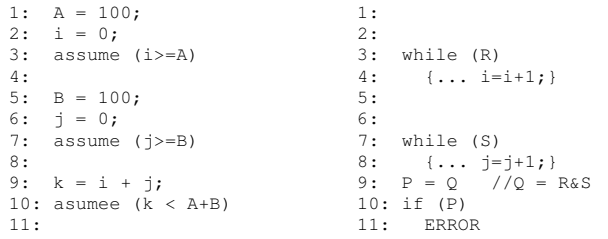

Figure 3: UNSAT core may not generate a good refinement.

- Lines 1, 2, 5, 6, 9 and 10,
- Lines 3, 7 and 10.

Although any of these UNSAT cores can be used to remove the spurious counterexample, the last one is better since it immediately proves that `ERROR` is not reachable, as shown on the right. The weakest precondition of `P:(k<A+B)` is `Q:(i+j<A+B)`, which is implied when both `R:(i<A)` and `S:(j<B)` are true. With these four predicates, we can prove the property.

However, modern SAT solvers are likely to report one of the first three UNSAT cores, due to the eager unit clause propagation used during pre-processing to handle the assignments to constants (lines 1, 2, 5, and 6). In this example, WP computation has to consider the (potentially complex) loop bodies. For instance, if the loops contains `i=i+1` and `j=j+1`, then using the first UNSAT core will result in 1024 predicates derived from `R`.

We heuristically avoid this situation by formulating the satisfiability problem in a slightly different way. For each assignment statement of the form st_i: `v:=const` in the spurious counterexample, the constraint in the corresponding SAT problem is $(v^i = const)$. We change this constraint to :

$$(v^i = const \vee q) \wedge (v^i = const \vee \neg q)$$

where q is a fresh Boolean variable. Note that the new constraint implies $(v^i = const)$. However, the presence of the extra variable q prevents the SAT solver from eagerly propagating the unit clauses due to $(v^i = const)$ during pre-processing. This reduces the chances of such constant assignments appearing in the UNSAT core reported by the SAT solver. Therefore, although this approach does not guarantee that the UNSAT core generated by the SAT solver provides the best refinement solution, it can significantly increase the chance of getting one.

This approach is similar to the lazy constraint method in [10], where it was shown to be effective for finding good variable (latch) hiding abstractions. Here, we apply it in the context of predicate abstraction. Furthermore, the lazy constraints in [10] were applied at the bit-level, for modeling only the initial state values of latches. In contrast, we apply them at the word-level, to assignment statements appearing anywhere in the high-level description of the design or program. Another difference to note is that the earlier work used lazy constraints for the purpose of proof-based abstraction. In that setting, the use of lazy constraints can sometimes be expensive, especially on large problems corresponding to large concrete designs. In our setting, we use lazy constraints only during refinement, where the problem of checking the feasibility of an abstract counterexample is significantly smaller.

7. Experiments

We have implemented the hybrid CEGAR procedure for models represented as CDFGs. We evaluated the proposed techniques by comparing hybrid abstraction with the two existing abstraction methods—variable hiding and predicate abstraction—in the same CEGAR procedure. For the purpose of controlled experiments, the model checking algorithms are kept the same; both predicate abstraction and hybrid abstraction use the same weakest precondition based refinement algorithm to find new predicates, and variable hiding uses an UNSAT core based refinement algorithm to identify new visible variables. In our implementation, we used CUDD [21] for BDD operations and a circuit SAT solver for SAT related operations. Our experiments were conducted on a workstation with a 3 GHz Pentium 4 and 2GB of RAM running Red Hat Linux.

We have instrumented a public Verilog front-end tool (called Icarus Verilog) to translate Verilog designs into functionally equivalent CDFGs. Our benchmarks include examples from [13] and the VIS Verilog benchmarks [22]. All examples are available in public domain. For these examples, we check invariant properties, which are expressed as reachability of an error block. Among the test cases, *AR* is an example computing the Fibonacci numbers (we set the parameterized bit-width to 32, although in the original versions, the bit-vectors have sizes of 500, 1000, and 2000 in all arithmetic operations); *pj_icram* is an example which models a RAM unit of the PicoJava microprocessor; *pj_icu* is an example which models the Instruction Control Unit of the PicoJava microprocessor. The *sdlx* example is a sequential DLX processor that uses a load-store architecture. The *arbiter* example is a Tree Arbiter model from [22], which has a counter of 8-bit width. *tloop* is a model containing three concurrently running submodules with long counters. The *itc99* examples are the Verilog versions (from [22]) of the Torino benchmarks in ITC99-T.

The first three columns of Table 1 provide statistics on the examples: the first column shows the names of the designs; the second column shows the numbers of binary state variables (or registers) in the cone of influence, and the third column indicates whether the property is true. The next three columns compare the CPU time of the CEGAR procedure with different abstraction methods—**varhide** denotes variable hiding, **predabs** denotes predicate abstraction, and **hybrid** denotes the hybrid abstraction. The next three columns compare the number of iterations of the CEGAR procedure needed to prove the properties. The next three columns compare the final abstract models in terms of (**Vars/Preds**), i.e., the number of visible variables and the number of predicates, respectively. (Here a final abstract model is a model on which the property can be decided.) The last three columns show the results for the VCEGAR tool [13]—the CPU time, the number of iterations, and the size of the final abstract models. We ran all the experiments using the latest binary of VCEGAR (version 1.1).

Overall, the hybrid abstraction makes the CEGAR procedure more robust. The performance of **hybrid** consistently matched the better of the two existing methods **varhide** and **predabs**. For half of the examples, **hybrid** obtained the best runtime performance among the three. We believe that this is due to the hybrid model being more concise than either of the two extremes. It is interesting to note that even though our currently implemented refinement approach is slightly biased toward predicates (converting predicates to visible variables, and not vice versa), the final abstract model in all examples included a non-trivial number of visible variables (other than x_{pc}). Note also that our implementation of pure predicate abstraction has a runtime performance comparable to VCEGAR, although it computes abstractions at a significantly finer granularity.

More specifically, note that predicate abstraction timed out on the *arbiter* example, since a large number of predicates of the form `(i+counter <=127)` such that $i = 1, 2, \ldots$ is required (exponential in the bit-width of variable `counter`). The *itc99-d* example is also hard for pure predicate abstraction, since it has a very long counterexample and requires a large number of predicates. Pure

Table 1: Comparing the three abstraction methods in the same CEGAR procedure (TO—timed out after 1 hour)

Test Case			CPU Time (s)			Iterations			Vars / Preds			VCEGAR		
name	bvars	prop	varhide	predabs	hybrid	varhide	predabs	hybrid	varhide	predabs	hybrid (v/p)	Time	Iters	Preds
AR	96	T	3.4	0.5	0.5	3	6	5	96	6	0 / 6	0.5	3	4
pj_icram	243	T	4.4	3.5	3.6	8	8	9	107	21	13 / 8	21.5	2	3
pj_icu	8060	T	84	68	23	2	2	2	1228	46	37 / 9	0.7	2	6
sdlx	124	T	39	20	14	14	15	15	42	28	24 / 4	42.6	20	43
tloop	127	T	TO	3.3	3.1	-	6	6	-	15	9 / 6	TO	-	-
arbiter	121	T	435	TO	401	13	-	20	50	-	13 / 37	TO	-	-
itc99-a	9	F	0.2	0.6	0.3	3	5	4	2	6	4 / 2	0.6	2	12
itc99-b	74	T	32	73	17	11	13	11	60	47	32 / 3	7.5	8	35
itc99-c	71	F	7.9	18	9.0	7	10	8	24	28	22 / 1	2.7	4	17
itc99-d	71	F	225	TO	692	11	-	16	65	-	45 / 8	TO	-	-

variable hiding abstraction worked well on these two examples, because it is able to localize the property to a small subset of variables (the final abstract model for *arbiter*, including the variable `counter`, has 50 Boolean state variables). The hybrid abstraction uses the same WP-based refinement algorithm as in predicate abstraction, but achieved a runtime performance and final sizes similar to variable hiding.

On the other hand, pure variable hiding was the slowest on the *AR* example, since it added all the variables of the model to prove the property (the final abstraction has 96 Boolean state variables). In contrast, both predicate abstraction and hybrid abstraction produced much smaller final abstract models (with only 6 Boolean state variables). Variable hiding also timed out on the *tloop* example, which has a CDFG structure similar to the one in Figure 3; variable hiding is inefficient for this example because the abstract model contains several complex arithmetic operations (large adders). Our implementations of both predicate abstraction and hybrid abstraction completed this example. VCEGAR did not complete the *tloop* example because its refinement is based on the standard UNSAT core reported by zChaff, which results in the addition of a number of predicates. In contrast, we used the lazy constraint heuristic refinement described in Section 6.2 to obtain a more useful UNSAT core. This allowed us to build a simpler abstract model and therefore complete this example quickly.

8. Conclusions

We have presented a hybrid abstraction method that combines variable hiding with predicate abstraction in the same counterexample guided abstraction refinement loop. We use refinement based on weakest preconditions to add new predicates, and under certain conditions trade in the predicates for visible variables in the abstract model. We present heuristics for improving the overall performance, based on static analysis to identify useful candidates for visible variables, and use of lazy constraints to find more effective refinement. Our experiments show that hybrid abstraction frequently outperforms the existing abstraction methods—it makes the CEGAR procedure more robust. For future work, we will explore the use of more static analysis techniques to speed up the abstraction computation and to help computing better refinements.

References

[1] N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer Aided Design (FMCAD'04)*, pages 260–274, Nov. 2004.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI'01)*, pages 203–213, June 2001.

[3] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, pages 268–283. Springer, 2001. LNCS 2031.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV'00)*, pages 154–169. Springer, 2000. LNCS 1855.

[5] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification (CAV'02)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[6] E. Clarke, M. Talupur, H. Veith, and D. Wang. SAT based predicate abstraction for hardware verification. In *International Conference on Theory and Applications of Satisfiability Testing*, S. Margherita Ligure, Italy, May 2003.

[7] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 51–58, Boston, MA, June 2001.

[8] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV'97)*, pages 72–83. Springer, 1997. LNCS 1254.

[9] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *International Conference on Computer-Aided Design*, pages 416–423, Nov. 2003.

[10] A. Gupta, M. K. Ganai, and P. Ashar. Lazy constraints and SAT heuristics for proof-based abstraction. In *Proceedings of International Conference on VLSI Design*, pages 183–188, Jan. 2005.

[11] H. Jain, F. Ivančić, A. Gupta, and M. Ganai. Localization and register sharing for predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, pages 394–409. Springer, 2005. LNCS 3440.

[12] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL verilog. In *ACM/IEEE Design Automation Conference (DAC'05)*, pages 445–450, New York, NY, USA, 2005. ACM Press.

[13] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. VCEGAR: Verilog counterexample guided abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Mar. 2007.

[14] D. Kroening and N. Sharygina. Image computation and predicate refinement for RTL verilog using word level proofs. In *Design, Automation and Test in Europe (DATE'07)*, Mar. 2007.

[15] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[16] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, pages 141–153. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[17] B. Li, C. Wang, and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *Software Tools for Technology Transfer*, 2(7):143–155, 2005.

[18] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, July 1993.

[19] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17. Springer, 2003. LNCS 2619.

[20] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[21] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[22] VIS verification benchmarks. http://vlsi.colorado.edu/∼vis.

[23] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In *International Conference on Computer-Aided Design*, pages 408–415, Nov. 2003.

[24] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle. Dynamic abstraction using SAT-based BMC. In *ACM/IEEE Design Automation Conference (DAC'05)*, pages 754–757, San Jose, CA, June 2005.