Formal Verification at Higher Levels of Abstraction

Daniel Kroening
Oxford University Computing Laboratory
kroening@kroening.com

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

***Abstract*— Most formal verification tools on the market convert a high-level register transfer level (RTL) design into a bit-level model. Algorithms that operate at the bit-level are unable to exploit the structure provided by the higher abstraction levels, and thus, are less scalable.**

This tutorial surveys recent advances in formal verification using high-level models. We present word-level verification with predicate abstraction and satisfiability modulo theories (SMT) solvers. We then describe techniques for term-level modeling and ways to combine word-level and term-level approaches for scalable verification.

I. Introduction

Most new hardware designs are implemented at a high level of abstraction, e.g., using the *register transfer level (RT-level/RTL)*, or even at the system level. The RT-level of a hardware description language such as Verilog is very similar to a software program in ANSI-C, and offers special features for hardware designers such as bit-vector arithmetic and concurrency. The word-level modeling of data is encouraged by System-level languages such as SystemVerilog and SystemC, and promises better readability as well as higher simulation performance.

However, most formal verification tools used in the hardware industry still operate on a low-level design representation called a *net-list*. Even if given a RT-level description, the design is first flattened into a net-list. Converting a high-level RTL design into a net-list results in a significant loss of structure present at the RT-level. This makes verification at the net-list level inherently less scalable.

This tutorial surveys recent advances in formal verification using high-level models. We present word-level verification with predicate abstraction and SMT (satisfiability modulo theories) solvers. We then briefly review techniques for term-level modeling and ways to combine word-level and term-level approaches for scalable verification.

A basic building block of formal verification tools is a satisfiability (SAT) solver. Given a formula, it decides if there exists a truth assignment to the variables in the formula such that the formula evaluates to TRUE. Equivalently, given a (part of) a combinational circuit and a signal in the circuit, it decides if there exists a valuation of the primary inputs such that the signal evaluates to TRUE. Bit-level verification tools use a Boolean formula to represent the circuit. At the word-level, the circuit is represented as a finite-precision *bit-vector* formula, which contains arithmetic operators such as addition. Advances in Boolean satisfiability solving and automatic abstraction techniques have recently led to a resurgence of interest in decision procedures (SMT solvers) for finite-precision bit-vector arithmetic.

Model checking [16], [19] is an automatic technique for the verification of concurrent systems. It has been used successfully in practice to verify complex circuit designs and communication protocols. Model checking systematically explores the state space of a given design and checks that each reachable state satisfies the property of interest. When the design fails to satisfy a desired property, the model checking procedure produces a *counterexample* that demonstrates a behavior which falsifies the property. The properties (formal specifications) are usually described in linear temporal logic (LTL) or computational tree logic (CTL).

A straight-forward application of an efficient bit-vector decision procedure in formal verification is *Bounded Model Checking* (BMC) [7], [8]. The design and the property are unwound up to a given depth k to form a bit-vector formula that is satisfiable if there exists a refutation of the property of length k. If the formula is satisfiable, a counterexample can be extracted from the satisfying assignment. BMC is only complete if k exceeds a pre-computed *completeness threshold* [29], which is rarely feasible for industrial designs. In practice, BMC is therefore mostly used for refutation.

There are also word-level techniques geared for verification: *Predicate abstraction* [26] has been very successful in both hardware and software verification. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement was introduced by Ball and Rajamani [4] and promoted by the success of the SLAM project. The goal of SLAM is to verify that Windows device drivers obey API conventions. Predicate abstraction is only effective if the refinement procedure is able to infer sufficiently strong word-level predicates. In the context of assertion checking for hardware designs, this process requires a word-level circuit model.

Finally, term-level abstraction, as implemented in the UCLID system, is an effective way of abstracting away data and functional units to focus on verifying the correctness of control logic. Variables of bit-vector types can be replaced by variables drawn from an infinite domain. Notable applications of UCLID include, for example, the verification of an out-of-order execution unit [31].

1-4244-1382-6/07/$25.00 ©2007 IEEE

Related Work: Andraus et al. [2] present a scheme for automatic abstraction of behavioral RTL Verilog to the CLU language used by the UCLID system [12]. The CLU language allows modeling using terms, uninterpreted functions, equality, lambda expressions, and counters. In order to remove spurious behaviors from the abstract model a refinement procedure is described in [1]. The techniques in [1], [2] were shown to be useful in context of microprocessor correspondence checking.

Word-level reasoning for hardware designs is very common in the automated theorem proving community, and has been applied extensively to many circuits. While we focus on non-interactive, automatic methods, some of the techniques have been transfered into automatic decision procedures. As an instance, Manolios et al. [32] use rewriting-techniques common in the automatic theorem proving in order to decide quantified bit-vector formulas. The authors have implemented their algorithms in the BAT-tool. However, the BAT-tool uses a custom input language, and automated translation from Verilog has not yet been reported.

Outline: We first discuss word-level decision procedures for finite-precision bit-vector arithmetic in Section II. We review word-level predicate abstraction in Section III. We conclude the tutorial with a discussion of techniques for combining word-level and term-level modeling.

II. Bit-Vector Arithmetic

This section gives the syntax for bit-vector arithmetic, and surveys current decision procedures for this theory. We also describe an abstraction-based approach to solving bit-vector arithmetic [11], which is implemented in UCLID and CBMC.

A. Definition

While we are not aware of a standard definition of *bit-vector arithmetic* (it varies according to the application and tools), the fragment we consider here includes finite-precision integer arithmetic with linear and non-linear operators, as well as standard bit-wise operators, such as left shift, logical and arithmetic right shifts, extraction, concatenation, and so forth. We summarize the syntax below:

$$
\begin{array}{rcl}
\mathit{formula} & : & \mathit{formula} \vee \mathit{formula} \mid \mathit{formula} \wedge \mathit{formula} \\
 & & \mid \neg \mathit{formula} \mid \mathit{atom} \\
\mathit{atom} & : & \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \\
\mathit{rel} & : & = \mid \neq \mid \leq \mid \geq \mid < \mid > \\
\mathit{term} & : & \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \sim \mathit{term} \mid \\
 & & \mathit{Constant} \mid \mathit{term}[i:j] \mid \mathit{formula}\ ?\ \mathit{term}\ :\ \mathit{term} \\
\mathit{op} & : & + \mid - \mid * \mid \div \mid \% \mid \\
 & & << \mid >> \mid @ \mid \& \mid \mid \mid \hat{} \\
\end{array}
$$

Standard notation has been used in describing the above grammar, and we only point out certain aspects of the notation. *Terms* denote bit-vectors while *formulas* are Boolean-valued expressions. The expression $\mathit{formula}\ ?\ \mathit{term}\ :\ \mathit{term}$ is an "if-then-else" expression that selects between two terms on the basis of the value of its Boolean first argument. The expression $\mathit{term}[i:j]$ denotes the extraction of bits i through j of the bit-vector expression *term*. The operator % denotes the integer modulo operator, while @ denotes concatenation.

Each bit-vector expression is associated with a type. The type defines the the width of the expression in bits and the encoding (two's complement or binary encoding). Assigning semantics to this language is straightforward, e.g., as done in [9]. Note that all arithmetic operators are finite-precision, and come with an associated operator width. The semantics of the relational operators $>, <, \leq, \geq$, the non-linear arithmetic operators $(*, \div, \%)$ and the right-shift depends on whether an unsigned, binary encoding is used or a two's complement encoding is used. We assume that the type of the expression is clear from the context.

Example *1:* The following formula is valid when interpreted over the integers:

$$(x - y > 0) \iff (x > y) \tag{1}$$

However, if x and y are interpreted as bit-vectors, this equivalence no longer holds, due to possible overflow on the subtraction operation. □

B. Decision Procedures for Bit-Vector Arithmetic

We consider the *satisfiability problem* for bit-vector formulas: given a bit-vector formula ϕ, is there an assignment to the bits in ϕ under which ϕ evaluates to TRUE?

It is easy to see that this problem is NP-complete. The basic technique to decide satisfiability of a bit-vector formula is to replace the word-level operators by bit-level circuit equivalents. E.g., the addition on bit-vectors is replaced by a Boolean formula that corresponds to a ripple-carry adder. The resulting Boolean formula is then passed to a propositional SAT solver such as MiniSat [34]. This technique is called *bit-flattening* or *bit-blasting*. There are several techniques that go beyond this basic idea, surveyed below:

1) Simplification followed by bit-blasting: Most current decision procedures first performing heuristic simplifications on the input formula before bit-blasting it to SAT, with a view of handling arbitrary bit-vector operations. In particular, most of the solvers that competed in the recent SMT-COMP competition for decision procedures in the fixed-width bit-vector category belong to this class. The Cogent [21] procedure decides validity of ANSI-C expressions and belongs to this category. The current version of CVC-Lite [6], [25] pre-processes the input formula using a normalization step followed by equality rewrites before finally bit-blasting to SAT.

Wedler et al. [36] have a similar approach wherein they normalize bit-vector formulas in order to simplify the generated SAT instance. STP [13] is the successor to the CVC-Lite system; it performs several array optimizations, as well as arithmetic and Boolean simplifications on the bit-vector formula before bit-blasting to MiniSat. Yices [24] applies bit-blasting to all bit-vector operators except for equality. MathSAT [10] also falls in this category, and in addition to the techniques described above, uses a layered approach,

which permits an incremental strengthening the model of the arithmetic operators. Experimental results show performance improvements of several orders of magnitude when compared to plain bit-blasting.

2) Canonizer-based procedures: Earlier work on deciding bit-vector arithmetic centered on using a Shostak-like approach of using a canonizer and solver for that theory. The work by Cyrluk et al. [23] and by Barrett et al. on the Stanford Validity Checker [5] fall into this category; the latter differs from the former in the choice of a canonical representation. These approaches are very elegant, but are restricted to a subset of bit-vector arithmetic comprising concatenation, extraction, and linear equations (not inequalities) over bit-vectors.

3) Procedures for modular and bounded arithmetic: The third category of systems focuses on techniques to handle (linear and non-linear) modular arithmetic. The most recent work in this area is by Babić and Musuvathi [3], who encode non-linear operations as non-linear congruences and make novel use of Newton's p-adic method for solving them. However, this approach does not treat some of the operations that we handle such as integer division, and seems harder to extend to new operations. Brinkmann and Drechsler [9] use an encoding of linear bit-vector arithmetic into integer linear programming with bounded variables in order to decide properties of RTL descriptions of circuit data-paths, but do not handle any Boolean operations. Parthasarathy et al. [35] build on this approach by using a lazy encoding with a modified DPLL search, but non-linear bit-vector arithmetic is not supported. Huang and Cheng [27] give an approach to solving bit-vector arithmetic based on combining ATPG with a solver for linear modular arithmetic. This approach is limited in its treatment of non-linear operations which it handles by heuristically rewriting them as linear modular arithmetic constraints.

C. Deciding Bit-Vector Arithmetic with Abstraction

The authors and colleagues proposed an abstraction-based approach to solving bit-vector arithmetic [11]. Briefly, the algorithm operates by iteratively solving under- and over-approximations of the original formula. The method is illustrated in Figure 1. For details, the reader is referred to the paper [11].

The input to the decision procedure is a bit-vector arithmetic formula ϕ. Let there be n bit-vector variables appearing in ϕ, denoted by $\vec{v}_1, \vec{v}_2, \vec{v}_3, \ldots, \vec{v}_n$. Each variable $\vec{v}_i$ has an associated bit-width w_i.

The decision procedure performs the following steps:

1) *Initialization:* For each variable $\vec{v}_i$, we select a corresponding number s_i of Boolean variables to encode it with, where $0 \leq s_i \leq w_i$. We call s_i the *encoding size* of bit-vector variable $\vec{v}_i$.
2) *Generate Under-Approximation and Encode to SAT:* An under-approximation, denoted $\underline{\phi}$, is generated by restricting the values of each $\vec{v}_i$ to range over a set of cardinality 2^{s_i}. Thus, the Boolean encoding of $\vec{v}_i$ will comprise s_i Boolean variables; note, however, that the length of the vector of Boolean variables replacing $\vec{v}_i$ remains w_i.
   There are several ways to generate such an under-approximation and its Boolean encoding. One option is to encode $\vec{v}_i$ using Boolean variables on its s_i low order bits and then zero-extend it to be of length w_i. The other is to "sign-extend" it instead. For example, if $s_i = 2$ and $w_i = 4$, the latter would generate the Boolean vector $[\vec{v}_{i1}, \vec{v}_{i1}, \vec{v}_{i1}, \vec{v}_{i0}]$ (where $\vec{v}_{ij}$ are Boolean variables). Our implementation currently uses the latter encoding, as it enables searching for solutions at both ends of the ranges of bit-vector values.
   A Boolean formula β is then computed from $\underline{\phi}$ using standard circuit encodings for bit-vector arithmetic operators. The width of the operators is left unchanged. The formula β is handed to an off-the-shelf SAT solver. The only feature required of this SAT solver is that its response on unsatisfiable formulas should be accompanied by an unsatisfiable core.
   If the SAT solver reports that β is satisfiable, then the satisfying assignment is an assignment to the original formula ϕ, and the procedure terminates. However, if β is unsatisfiable, we continue on to the next step.
3) *Generate Over-Approximation from Unsatisfiable Core:* The SAT solver extracts an unsatisfiable core $\mathcal{C}$ from the proof of unsatisfiability of β. We use $\mathcal{C}$ to generate an over-approximating *abstraction* $\overline{\phi}$ of ϕ. The formula $\overline{\phi}$ is also a bit-vector formula, but typically much smaller than ϕ.
   The key property of $\overline{\phi}$ is that its translation into SAT, using the same sizes s_i as those that were used for $\underline{\phi}$, would also result in an unsatisfiable Boolean formula. This is achieved by replacing those expressions in ϕ that do not occur in the core $\mathcal{C}$ by new variables.
   The satisfiability of $\overline{\phi}$ is then checked using a *sound and complete decision procedure* $\mathcal{P}$ for bit-vector arithmetic, e.g., a bit-blasting approach.
   If $\overline{\phi}$ is unsatisfiable, we can conclude that so is ϕ. On the other hand, if $\overline{\phi}$ is satisfiable, it must be the case that at least one variable $\vec{v}_i$ is assigned a value that is not representable with s_i Boolean variables (recall the key property enjoyed by $\overline{\phi}$ cited earlier). This larger satisfying assignment indicates the necessary increase in the encoding size s_i for $\vec{v}_i$. Proceeding thus, we increase s_i for all relevant i, and return to Step 2.

Since s_i increases for at least one i in each iteration of this loop, this procedure is guaranteed to terminate in $O(n \cdot w_{max})$ iterations, where $w_{max} = \max_i w_i$. Of course, each iteration involves a call to a SAT solver and a decision procedure for bit-vector arithmetic.

D. Another Step of Abstraction

It is well-known that certain bit-vector arithmetic operators, such as integer multiplication of two variables (of adequately large width) are extremely hard for a procedure based on bit-blasting. However, for many problems involving these

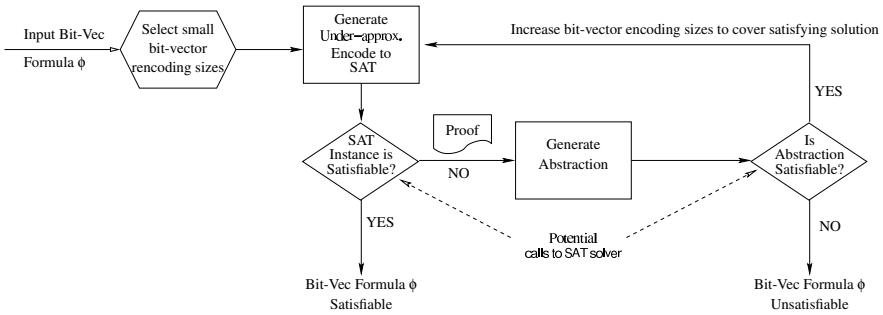

Fig. 1. **Abstraction-based approach to solving bit-vector arithmetic**

operators, it is unnecessary to reason about all of the operators' properties in order to decide the formula. Instead, using a set of rules (based on well-known rewrite rules) we can perform fine-grained abstractions of functions, which often suffice. Such (incomplete) abstractions can be used in the over-approximation phase of the procedure described in the previous subsection, and preserve the soundness and completeness of the overall procedure.

Therefore, UCLID invokes a preprocessing step before building $\overline{\phi}$. In this step, it replaces a subset of "hard" operators by lambda expressions that *partially interpret* those operators.

For example, UCLID replaces the multiplication operator $*_w$ of width w (for $w > 4$, chosen according to the capacity of current SAT engines) by the following lambda expression involving the freshly introduced uninterpreted function symbol $\texttt{mul}_w$:

$$\begin{aligned}\lambda x.\lambda y. \quad & (x = 0 \vee y = 0)\,?0 : \\ & ((x = 1)\,?y : \\ & ((y = 1)\,?x : \texttt{mul}_w(x,y)))\end{aligned}$$

This expression can be seen as partially interpreting multiplication, as it models precisely the behavior of this operator when one of the arguments is 0 or 1, but is uninterpreted otherwise. Experimental results show that this abstraction can be an enabling step when solving hard bit-vector formulas.

III. PREDICATE ABSTRACTION

This section describes predicate abstraction-based approach to model checking word-level descriptions of circuits, proposed by one of the authors and colleagues. The technique is implemented in the VCEGAR tool, which is available for download.[1] For details, the reader is referred to the paper [28].

A. Overview

The number of states in industrial hardware designs is extremely large. This often results in exorbitant resource requirements during model checking even when symbolic model checking algorithms are used. One principal method for state space reduction is *abstraction*. Abstraction techniques reduce the state space by mapping the set of states of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system.

[1] http://www.cs.cmu.edu/~modelcheck/vcegar/

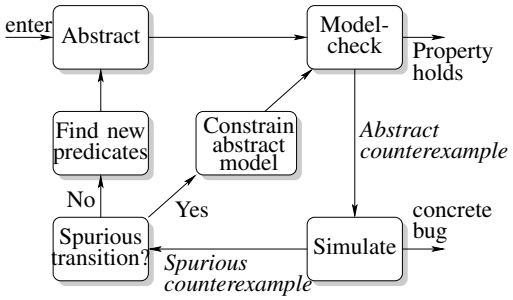

Fig. 2. Abstraction refinement (CEGAR) loop for predicate abstraction of RTL Verilog

Predicate abstraction is usually combined with an automatic abstraction refinement procedure. We describe the basic idea of the *Counterexample Guided Abstraction Refinement* framework, or CEGAR for short [14]. Fig. 2 shows an overview of the individual steps of the CEGAR loop for predicate abstraction.

Predicate abstraction is typically done in a *conservative* manner. This implies that if the abstraction satisfies a given property, the property also holds on the original concrete system, and the CEGAR loop terminates. When model checking of the abstraction fails, it produces an *abstract counterexample*. The drawback of the conservative abstraction is that an abstract counterexample may not correspond to any concrete counterexample (real error). This is usually called a *spurious counterexample* [14].

In order to check if an abstract counterexample is spurious, the abstract counterexample is simulated on the concrete system. This is called the *simulation* step. As in BMC, the concrete transition relation for the design and the given property are jointly unwound to obtain a Boolean formula. The number of unwinding steps is given by the length of the abstract counterexample. The resulting formula can then be checked for satisfiability using a procedure such as described in Sec. II. If the instance is satisfiable, the counterexample is real and the procedure terminates. If the instance is unsatisfiable, the abstract counterexample is spurious, and *abstraction refinement* has to be performed.

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

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

  initial x = 1;
  initial y = 0;

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

Fig. 3. A Verilog program used as a running example

order to prevent the spurious counterexample [4], [14], [30]. This process is iterated until the property is either proved or disproved. Predicate abstraction is only effective if the predicates can cover the relationship between multiple latches. This typically requires a word-level model given in RT-level of a hardware description language. We apply predicate abstraction to word-level models given in RTL Verilog.

B. Word-Level Circuit Models

Let $\mathcal{R} = \{r_1, \ldots, r_n\}$ denote the set of registers and external inputs in a given Verilog program. For example, the state of the Verilog program in Fig. 3 is defined by the value of the registers x and y, and each of them has a storage capacity of 8 bits. Let S denote the set of states for a given Verilog program. Let $Q \subseteq \mathcal{R}$ denote the set of registers. We denote the next-state function of a register $r_i \in Q$ by $f_i(r_1, \ldots, r_n)$, or $f_i(\bar{r})$ using vector notation, where $\bar{r} = \langle r_1, \ldots, r_n \rangle$. The value of r_i in the the next state is given by $f_i(\bar{r})$ as a function of the current state. We use the next-state functions to define the transition relation $R(\bar{r}, \bar{r}')$. It relates the current state $\bar{r} \in S$ to the next state $\bar{r}' \in S$ and is defined as follows:

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

We denote the next-state functions of x and y by $f_x(x,y)$ and $f_y(x,y)$, respectively, and the transition relation by $R(x,y,x',y')$.

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

The first step of the CEGAR loop is to obtain an abstraction of the given circuit.

C. Existential Abstraction

In predicate abstraction [26], the variables of the concrete program are replaced by Boolean variables. Each Boolean variable corresponds to a predicate on the variables in the concrete program. Predicates are functions that map concrete states $\bar{r} \in S$ to Boolean values. Let $\mathtt{B} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates. When applying all predicates to a specific concrete state, one obtains a vector of Boolean values, which represents an abstract state $\bar{b}$. We denote this function by $\alpha(\bar{r})$. It maps a concrete state into an abstract state and is therefore called an *abstraction function*.

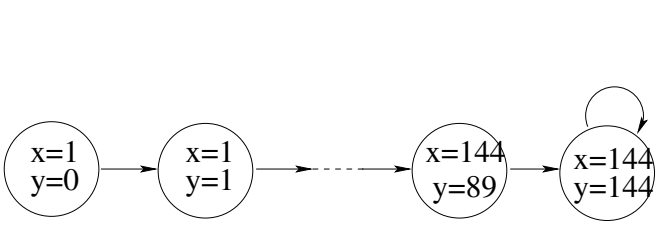

Fig. 4. The state transition graph of the Verilog program in Fig. 3

We construct an abstract transition relation $\hat{R}$ as follows: 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 $\bar{b}'$. The system described by $\hat{R}$ is called an existential abstraction [15]. We call the abstract machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\begin{aligned} \hat{R} \quad := \{(\bar{b}, \bar{b}') \,|\, \exists \bar{r}, \bar{r}' \in S: \quad & \alpha(\bar{r}) = \bar{b} \wedge R(\bar{r}, \bar{r}') \ \wedge \\ & \alpha(\bar{r}') = \bar{b}'\} \end{aligned} \tag{2}$$

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

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

$$\hat{I}(\bar{b}) \quad := \quad \exists \bar{r} \in S: \ (\alpha(\bar{r}) = \bar{b}\ ) \wedge\ I(\bar{r}) \tag{3}$$

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

$$\hat{P}(\bar{b}) \quad := \quad \forall \bar{r} \in S: \ (\alpha(\bar{r}) = \bar{b}) \Longrightarrow P(\bar{r}) \tag{4}$$

Thus, if $\hat{P}$ holds on all reachable states of the abstract model, P also holds on all reachable states of the concrete model. The converse is not true, which is a possible source of spurious counterexamples.

The techniques described above can be used to check any LTL safety property. This is because the spurious counterexamples for LTL safety properties are always finite acyclic paths [20]. Such spurious counterexamples can be removed during the refinement phase (Section III-E).

D. SAT-based Predicate Abstraction

In [17], a SAT solver is used to compute the abstraction of a sequential ANSI-C program. We use a similar technique for computing the abstraction of Verilog programs. We describe the computation of the abstract transition relation $\hat{R}$ (Eqn. 2) in more detail below.

Computing $\hat{R}$ using SAT: A symbolic variable b_i is associated with each predicate π_i. Each concrete state $\bar{r} = \langle r_1, \ldots, r_n \rangle$ maps to an abstract state $\bar{b} = \langle b_1, \ldots, b_k \rangle$, where $b_i = \pi_i(\bar{r})$. If the concrete machine makes a transition from state $\bar{r}$ to state

$\bar{r}' = \langle r'_1, \ldots, r'_n \rangle$, then the abstract machine makes a transition from state $\bar{b}$ to $\bar{b}' = \langle b'_1, \ldots, b'_k \rangle$, where $b'_i = \pi_i(\bar{r}')$. We refer to $\pi_i(\bar{r})$ as a *current-state* predicate and $\pi_i(\bar{r}')$ as a *next-state* predicate. For example, if $x = y$ is a current-state predicate, then the corresponding next-state predicate is $x' = y'$.

The formula that is passed to the decision procedure directly follows from the definition of the abstract transition relation $\hat{R}$ as given in Eqn. 2:

$$\hat{R} \;:=\; \{(\bar{b}, \bar{b}') \mid \exists \bar{r}, \bar{r}' \;:\; \Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}')\} \text{ , where} \qquad (5)$$

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

The set of abstract transitions $\hat{R}$ is computed by passing $\Gamma(\bar{r}, \bar{r}', \bar{b}, \bar{b}')$ to a decision procedure for bit-vector arithmetic as described in Sec. II. Suppose we obtain a satisfying assignment $\bar{r}, \bar{r}', \bar{b}, \bar{b}'$. We project out all variables but $\bar{b}$ and $\bar{b}'$ from this satisfying assignment to obtain one abstract transition $(\bar{b}, \bar{b}')$. Since we want all the abstract transitions, we add a blocking clause to the formula that eliminates all satisfying assignments that assign the same values to $\bar{b}$ and $\bar{b}'$, and re-start the solver. This process is continued until the formula becomes unsatisfiable. The disjunction of the abstract transitions obtained gives us the abstract transition relation $\hat{R}$.

Example: We continue our example based on Fig. 3. Assume that $\{x < 200, x < 100, x + y < 200\}$ is the set of predicates. We associate symbolic variables b_1, b_2, b_3 with each predicate, respectively. In order to compute $\hat{R}$ the following equation is passed to the decision procedure:

$$\begin{gathered}(b_1 \Leftrightarrow (x < 200)) \;\wedge\; (b_2 \Leftrightarrow (x < 100)) \;\wedge \\ (b_3 \Leftrightarrow (x + y < 200)) \wedge R(x, y, x', y') \;\wedge (b'_1 \Leftrightarrow (x' < 200)) \;\wedge \\ (b'_2 \Leftrightarrow (x' < 100)) \;\wedge\; (b'_3 \Leftrightarrow (x' + y' < 200))\end{gathered}$$

The abstract transition relation obtained is given by the SMV [22] `TRANS` statement in Fig. 5. It is a disjunction of cubes. The cube (`b1 & !b2 & !b3 & next(b1) & !next(b2) & !next(b3)`) corresponds to the transition from the abstract state in which b_1 is true and b_2, b_3 are false to the same abstract state ($100 \rightarrow 100$ for short). All possible abstract transitions are shown explicitly in Fig. 6.

The set of abstract initial states (Eqn. 3) can be enumerated using a SAT solver in a similar manner as $\hat{R}$. The set of abstract initial states is given by the `INIT` statement in Fig. 5. There is only one abstract initial state in which all the Boolean variables b_1, b_2, b_3 are true.

The LTL property $\mathbf{G}(x < 200)$ is abstracted using the Boolean variable b_1 for the predicate $(x < 200)$. The abstracted property is given by the `SPEC` statement in Fig. 5. The abstract model satisfies the property `G b1`, as the only states reachable from the initial abstract state (111) are $\{111, 101, 100\}$ (Fig. 6). Since the property holds on the abstract model, we can conclude that the property $\mathbf{G}(x < 200)$ holds on the Verilog program in Fig. 3.

```
MODULE main

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

INIT (b1 & b2 & b3)

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

SPEC G b1
```

Fig. 5. Abstraction of the Verilog program in Fig. 3 using the predicates $x < 200$, $x < 100$, and $x + y < 200$. It is in the format accepted by the SMV model checker.

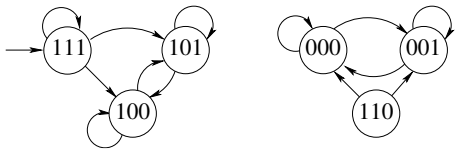

Fig. 6. Finite state machine for the abstract model in Fig. 5. The abstract states 010 and 011 are not possible, as this would require $x < 200$ to be false and $x < 100$ to be true at the same time.

E. Abstraction Refinement

We focus the discussion on spurious prefixes, and refer the reader to [28] for methods to remove spurious transitions. An abstract counterexample of length l is a sequence of abstract states $\bar{s}(0), \ldots, \bar{s}(l)$, where each abstract state $\bar{s}(j)$ corresponds to a valuation of the k predicates $\pi_1, \ldots, \pi_k$. The value of π_i in a state $\bar{s}$ is denoted by $\bar{s}_i$. Given an abstract state $\bar{s}$, let $\beta(\bar{s})$ denote the conjunction of predicates (or their negation) depending upon their values in $\bar{s}$.

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

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

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

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

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

The above formula is unsatisfiable iff there is no sequence of concrete states $\bar{r}_0, \ldots, \bar{r}_l$ such that $\bar{r}_0$ is an initial state, there is a transition from $\bar{r}_i$ to $\bar{r}_{i+1}$ for $0 \leq i < l$, and the predicate values in each concrete state $\bar{r}_j$ exactly match the predicate values given by the abstract state $\bar{s}(j)$ for $0 \leq j \leq l$.

In [18], spurious prefixes are eliminated by adding a bit-level predicate. This predicate is called a *separating* predicate and is computed by using a SAT-based conflict dependency analysis. In contrast, we make use of *weakest preconditions* as done in software verification. We generate new word-level predicates by computing the weakest precondition of the given property with respect to the transition function given by the RT-level circuit.

Example: Let the property be $x < 200$. Let the next state functions for the registers x and y be $((x < 100)?(x+y) : x)$ and x, respectively. Suppose we obtain a spurious prefix of length 1. The weakest precondition wp is computed as follows:

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

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

Predicate discovery for abstraction refinement is still an open area of research. We use weakest preconditions for discovering new predicates. This is sufficient for ensuring that the abstraction refinement loop makes progress. An alternative technique for discovering new predicates is based on interpolation [33]. In order to apply this idea to circuits, an interpolating theorem prover for bit-vector logic is required. At present, it is not known how to build such a prover for bit-vector logic.

IV. Conclusion

Capacity is the main challenge for formal verification tools. Given a high-level model, word-level reasoning can increase the capacity of formal verification tools significantly when compared to a net-list level tool. We discuss decision procedures (SMT solvers) for bit-vector arithmetic, and give an overview of predicate abstraction, a word-level assertion checking technique.

References

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 19–24, 2006.

[2] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Design Automation Conference (DAC)*, pages 218–223. ACM Press, 2004.

[3] D. Babić and M. Musuvathi. Modular Arithmetic Decision Procedure. Technical report, Microsoft Research, Redmond, 2005.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop*, pages 103–122, 2001.

[5] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM Press, 1998.

[6] S. Berezin, V. Ganesh, and D. Dill. A decision procedure for fixed-width bit-vectors. Technical report, Computer Science Department, Stanford University, April 2005.

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

[8] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, 1999.

[9] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of VLSI Design*, pages 741–746, 2002.

[10] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification (CAV)*, LNCS. Springer, 2007.

[11] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 358–372. Springer, 2007.

[12] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.

[13] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

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

[15] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL*, pages 342–354, 1992.

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

[17] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, 2004.

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

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

[20] E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224. Springer, 2003.

[21] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV 2005*, pages 296–300. Springer, 2005.

[22] Cadence SMV, http://www.kenmcmil.com/smv.html.

[23] D. Cyrluk, M. O. Möller, and H. Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer-Aided Verification (CAV '97)*, pages 60–71, 1997.

[24] B. Dutertre and L. de Moura. The Yices SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, September 2006.

[25] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, LNCS. Springer, 2007.

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

[27] C.-Y. Huang and K.-T. Cheng. Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques. In *Design Automation Conference (DAC)*, pages 118–123, 2000.

[28] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Design Automation Conference (DAC)*, pages 445–450. ACM, 2005.

[29] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2575 of *LNCS*, pages 298–309. Springer, 2003.

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

[31] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *LNCS*, pages 142–159. Springer, 2002.

[32] P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for rtl model verification. In *International Conference on Computer-Aided Design (ICCAD)*, pages 786–793. ACM, 2006.

[33] K. L. McMillan. An interpolating theorem prover. In *TACAS*, pages 16–30. Springer, 2004.

[34] MiniSat. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`.

[35] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang. An efficient finite-domain constraint solver for circuits. In *Design Automation Conference (DAC)*, pages 212–217, 2004.

[36] M. Wedler, D. Stoffel, and W. Kunz. Normalization at the arithmetic bit level. In *Design Automation Conference (DAC)*, pages 457–462. ACM Press, 2005.