Checking Consistency of C and Verilog using Predicate Abstraction and Induction

Edmund Clarke Daniel Kroening

June 25, 2004
CMU-CS-04-131

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

It is common practice to write C models of circuits due to the greater simulation efficiency. Once the C program satisfies the requirements, the circuit is designed in a hardware description language (HDL) such as Verilog. It is therefore highly desirable to automatically perform a correspondence check between the C model and a circuit given in HDL. We present an algorithm that checks consistency between an ANSI-C program and a circuit given in Verilog using Predicate Abstraction. The algorithm exploits the fact that the C program and the circuit share many basic predicates. In contrast to existing tools that perform predicate abstraction, our approach is SAT-based and allows all ANSI-C and Verilog operators in the predicates. We report experimental results on an out-of-order RISC processor. We compare the performance of the new technique to Bounded Model Checking (BMC).

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

Keywords: Predicate Abstraction, Verilog, SAT, Equivalence Checking

1 Introduction

ANSI-C is a language designed for best execution efficiency. This is why C programs are often used as a model for circuits that require extensive testing and simulations. The testing is done using the fast C model. Once the C model satisfies the requirements, it is used as a specification for building the circuit in a language that will yield to an efficient circuit, such as Verilog or VHDL. Due to time-to-market constraints, there is often not enough time to perform the same rigorous evaluation of the Verilog implementation as it was performed for the C model.

Thus, it is highly desirable to determine if the C and Verilog programs are consistent [20].

Related Work There are already multiple different approaches to this problem:

There are tools that take a C program in a specific form as input and translate it into a circuit. The two circuits can then be compared using standard equivalence checkers, as done by Séméria et al. [24]. However, the C program has to be very similar to the circuit, e.g., they must share the same registers and must perform the computations in the same number of steps.

Matsumoto, Saito, and Fujita compare two C-based hardware descriptions [16]. First, the differences are identified syntactically, and then compared using symbolic simulation. The method also assumes very strong similarity of the two descriptions.

In [12], Bounded Model Checking (BMC) [4, 3] is applied to both a circuit and an ANSI-C program. No particular similarity is assumed, and the notion of equivalence can be adapted using C language constructs. However, no attempt is made to abstract the program or the circuit, which limits the capacity of the method. Furthermore, Bounded Model Checking only shows the absence of inconsistencies up to a given bound. Determining if this bound is large enough to guarantee the absence of any inconsistencies is non-trivial [13].

The concept of verifying the equivalence of a software implementation and a synchronous transition system was introduced by Pnueli, Siegel, and Shtrichman [23]. Since the target code is generated automatically by a compiler, the C program is assumed to have a specific form.

With the exception of [12], the related work requires a very strong correspondence of the circuit and the program. However, the programs written for simulation purposes often do not show such a strong correspondence. This means that these programs would have to be rewritten for equivalence checking, which is undesirable. Thus, we would like to be able to compare programs and circuits that achieve the same goal in completely different ways.

The criterion we use for equivalence is input/output equivalence: assuming the circuit and program obtain corresponding input, we want to show that they produce corresponding output. However, if this property is checked cycle-by-cycle, this would require that the C program has to be *cycle accurate*, i.e., it would have to compute all the values the circuit computes in the same number of steps.

We would like to be more flexible about the points in time used for the I/O equivalence check. The user of the framework should be able to customize it for anything from complete cycle-accuracy to an occasional check of computational results. This means that both the circuit and the program should be allowed to perform a possible lengthy computation completely independent from each other. Once each transition system is finished, only the results are compared. The number of transitions required for each machine to obtain the results may not be related at all. Obviously, the time required may depend on input data and the algorithms used by the machines. This flexibility is achieved by distinguishing external and internal transitions. The external transitions of the two machines are synchronized, and the equivalence check is only performed on these transitions.

Contribution We formalize I/O equivalence for transition systems with external and internal transitions, similar to *weak bisimulation* as described by Milner [18]. We describe a method to reduce this equivalence criterion to a safety property of a special product machine of the two transition systems. We then describe how to use predicate abstraction in order to prove the safety property, and thus, the I/O equivalence.

During the abstraction of the transition system, we add the safety property as a constraint to the current state. This is a special form of inductive reasoning, and allows to exploit structural similarities of the two machines automatically. The more the two machines share, the stronger is the constraint. In the special case of two machines that have the exact same set of latches, the problem becomes equivalent to SAT-based combinational equivalence checking.

This approach is less flexible than the approach in the related work: In [12], the ANSI-C program is able to refer to the value of any circuit signal in any given cycle. In contrast to that, the approach proposed in this paper only allows to refer to current signal values, not past values. However, we believe that this is not a strong restriction, and that the benefits of abstraction out-weight this downside. In particular, we are able to conclude that the circuit and program are consistent for any number of steps, not for just a given bound.

Outline In section 2, we formalize the correctness criterion. In section 3, we describe how to reduce it to a safety property of the product machine using given relations for input and output. In section 4, we show possible ways to write circuit specifications in the form of efficient C programs and how to automatically generate the input/output relations for a particular form of correspondence. In section 5, we report experimental results.

2 Formal Equivalence Criterion

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

$L(s)$ is a labeling function: it maps a state $s \in S$ to the action (or event) that is generated by the state. We consider only one action, σ, which is used to synchronize the two machines, and the silent event τ, which denotes an internal transition. No synchronization is done when a machine generates a τ-action. A state s with $L(s) = \sigma$ is called a *visible* state, a state s with $L(s) = \tau$ is a hidden state. Analogously, a transition out of a state labeled with τ is called an invisible or weak transition, and a transition labeled with σ is a visible or strong transition [18].

We call a sequence of states $t(0), \ldots, t(n)$ of a machine a trace of the machine iff the state of $t(0)$ is an initial state, and all subsequent states are related using R:

$$t(0) \in I$$
$$\forall i < n : \; t(i) \, R \; t(i+1)$$

By Vt, we denote the sequence of states where the first state is the first visible state in the sequence t, the second state is the second visible state in t, and so on.

Let the circuit be given by $T_1 = (S_1, I_1, \mathcal{I}_1, R_1, L_1)$, and the ANSI-C program be given by $T_2 = (S_2, I_2, \mathcal{I}_2, R_2, L_2)$. We will describe several restrictions of these transition systems, but note that we do *not* require $S_1 = S_2$, i.e., the registers/latches do not have to correspond to any program variables or vice versa. This is in contrast to the work presented in [24], which assumes a one-on-one mapping of registers and variables.

Instead of comparing the states of the two machines, we propose to check the externally visible I/O behavior only. Informally, in visible states, we require that the outputs match assuming that the inputs have matched so far. We assume that there is a user-provided relation that specifies what matching inputs and outputs are. The relation may be generated automatically for a restricted program syntax, e.g., by means of a variable mapping (section 4). Formally, input is modeled by means of non-determinism in the transition relations R_1 and R_2. The output is assumed to be a function of the current states s_1 and s_2. Thus, it is sufficient to relate the states. Let $\widehat{=}_I$ denote the consistency relation for inputs, and $\widehat{=}_O$ for outputs:

$$\widehat{=}_I \quad : \quad \mathcal{S}_1 \longleftrightarrow \mathcal{S}_2$$
$$\widehat{=}_O \quad : \quad \mathcal{S}_1 \longleftrightarrow \mathcal{S}_2$$

Two traces t_1 of T_1 and t_2 of T_2 are said to be input consistent iff the inputs of all external transitions of the traces are consistent:

$$t_1 \widehat{=}_I t_2 \quad :\Longleftrightarrow \quad \forall i : \; Vt_1(i) \, \widehat{=}_I \, Vt_2(i) \tag{1}$$

Analogously, two traces t_1 of T_1 and t_2 of T_2 are said to be output consistent iff the outputs of all external transitions of the traces are consistent:

$$t_1 \widehat{=}_O t_2 \quad :\Longleftrightarrow \quad \forall i : \; Vt_1(i) \, \widehat{=}_O \, Vt_2(i) \tag{2}$$

Formally, we define two transition systems T_1 and T_2 to be I/O consistent, iff input consistency implies output consistency for all valid traces:

$$T_1 \widehat{=} T_2 \quad :\Longleftrightarrow \quad (t_1 \widehat{=}_I t_2) \Longrightarrow (t_1 \widehat{=}_O t_2) \tag{3}$$

3 Implementation

3.1 The Product Machine

This section describes how we apply counterexample guided abstraction refinement in order to check equivalence as defined in the previous section. We define a specific product machine T_p as follows: The set of states S_p of the product machine is $S_1 \times S_2$. Thus, a state of T_p is a pair of one state of T_1 and one state of T_2. The initial state of the machine must be a pair of initial states of the corresponding machines (no attempt is made to synchronize the initial states).

The transition relation R_p of T_p is constructed as follows: First, we define relations Δ_1 and Δ_2, which take a state $s \in S_x$, a next state $s' in S_x$, and a Boolean value c. If the Boolean value is true, Δ_1 and Δ_2 are identical to the original transition relations R_1 and R_2, respectively. If it is false, only equal states are related to each other, and thus the state of the machine does not change:

$$\Delta_x(s, c, s') \quad := \quad \begin{cases} R(s, s') & : \quad c \\ s = s' & : \quad \text{otherwise} \end{cases}$$

Note that the equality in the definition above is equality of two states in S_x, not a mixture of both transition systems. Intuitively, c is a "clock enable signal" for the transition systems. If not active, the state of the corresponding machine does not change.

A transition system is allowed to make a transition iff the transition is either a τ-transition, or if both transition systems are ready to make a σ-transition. We use c_1 and c_2 as a shorthand for these conditions.

$$\begin{aligned} c_1 \quad &:= \quad (L_1(s_1) = \tau) \vee (L_2(s_2) = \sigma) \\ c_2 \quad &:= \quad (L_2(s_2) = \tau) \vee (L_1(s_1) = \sigma) \end{aligned}$$

We also label the states of the product machine using the labeling function L_p. A state (s_1, s_2) of the product machine is labeled with σ if and only of both transition systems are about perform a σ transition. It is labeled with τ otherwise.

$$L_p(s_1, s_2) \quad := \quad (L_1(s_1) = \sigma \wedge L_2(s_2) = \sigma)$$

If the product machine makes a σ-transition, we require that the inputs of both transition systems are consistent. We use ρ as a shorthand for this restriction:

$$\rho(s_1, s_2) \quad :\Longleftrightarrow \quad L_p(s_1, s_2) = \sigma \Longrightarrow (s_1 \widehat{=}_I s_2)$$

This allows us to define the transition relation as follows: the product machine can make a transition from (s_1, s_2) to (s_1', s_2') iff the states obey the restriction ρ and allow making the steps of the two machines using Δ_1 and Δ_2:

$$\begin{aligned} (s_1, s_2) R_p (s_1', s_2') \quad :\Longleftrightarrow \quad & \rho(s_1, s_2) \wedge \\ & \Delta_1(s_1, c_1, s_1') \wedge \\ & \Delta_2(s_2, c_2, s_2') \end{aligned}$$

Thus, given the machines T_1 and T_2, the product machine can be constructed easily. For all reachable states of the product machine that are labeled with σ, we check that the two states (s_1, s_2) are output consistent:

$$L_1(s_1, s_2) = \sigma \Longrightarrow s_1 \widehat{=}_O s_2 \tag{4}$$

Claim 1 *T_1 and T_2 are I/O equivalent iff $s_1 \widehat{=}_O s_2$ holds for all reachable states (s_1, s_2) of T_p that are labeled with σ, i.e., perform I/O.*

3.2 Using Abstraction

Claim 1 reduces the criterion for I/O equivalence to a safety property on the product machine. We check this safety property using counterexample guided abstraction refinement (CEGAR) [14, 1, 5]. We perform a predicate abstraction [10], i.e., the latches of the circuit and the variables of the program are replaced by Boolean variables that correspond to a predicate on the original variables and latches.

Note that *both* transition systems are abstracted. Using abstraction for checking equivalence requires care in order to avoid false positives. We argue that we do not obtain false positives as we reduce the equivalence criterion to a safety property, which can be verified using overapproximations without risking false positives.

The first step is to obtain an initial abstraction of the product machine. This abstraction is then checked using a symbolic model checker. We perform a save abstraction, i.e., if the property holds on the abstract model, we can conclude that it also holds on the concrete model, and thus, I/O equivalence is shown. If the property does not hold on the abstract model, we expect the model checker to provide a counterexample. This abstract counterexample is then simulated on the concrete machine. This step corresponds to Bounded Model Checking on the concrete machine with additional constraints that are derived from the abstract counterexample.

If the simulation is successful, we obtain a concrete counterexample from the Bounded Model Checker. This counterexample is for the product machine and therefore allows us to extract separate traces for T_1 and T_2 that demonstrate the inconsistency. If the simulation fails, the abstract counterexample is spurious, and the abstraction has to be refined.

Formally, we assume that the algorithm maintains a set of n predicates $p_1, \ldots, p_n$. The predicates are functions that map a concrete state $x \in S_p$ into a Boolean value. When applying all predicates to a specific concrete state, one obtains a vector of n Boolean values, which represents an abstract state $\hat{x}$. We denote this function by $\alpha(x)$. It maps a concrete state into an abstract state and is therefore called *abstraction function*.

We perform an existential abstraction [6], i.e., the abstract machine can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from x to x' in the concrete machine and x is abstracted to $\hat{x}$ and x' is abstracted to $\hat{x}'$. We call the abstract product machine $\hat{T}$, and we denote the transition relation of $\hat{T}$ by $\hat{R}$.

$$\begin{aligned} \hat{R} \quad := \{\hat{x}, \hat{x}' \mid \quad & \exists x, x' \in S_p : x R_p x' \wedge \\ & \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\} \end{aligned} \tag{5}$$

Note that in practice, additional transitions are often added to the abstract transition relation in order to make the computation of $\hat{R}$ easier. This is common for the abstraction of both circuits and programs.

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

$$\hat{P}(\hat{x}) \quad :\Longleftrightarrow \quad \forall x \in S_p : (\alpha(x) = \hat{x}) \Longrightarrow P(x) \tag{6}$$

The same abstraction is also used for the initial state predicate. Thus, if P holds on all reachable states of the abstract machine, P also holds on all reachable states of the concrete machine. This leads to

Claim 2 *T_1 and T_2 are I/O equivalent if the abstraction of eq. (4) holds for all reachable states of $\hat{T}$.*

A state violating the safety property is called a *bad state*.

3.3 Using Induction during Abstraction

As we are checking an invariant, it is straight-forward to make the following restriction of the abstract transition relation: When considering a concrete transition x to x', we can safely assume that the property holds in the state x. Thus, we can use the following transition relation $\hat{R}^{-}$:

$$\hat{R}^{-} \quad := \{\hat{x}, \hat{x}' \,|\, \hat{R}(x, x') \wedge \hat{P}(\hat{x})\} \tag{7}$$

Note that the next state x' is not restricted. Intuitively, we are removing all transitions out of bad states. This restriction is justified as follows: The abstraction of the initial state is not restricted, and it is checked that it satisfies $\hat{P}$. It can now be argued inductively that the restriction to $\hat{R}^{-}$ does not remove paths to bad states, as only transitions *out of* bad states are removed. Transitions into bad states are only removed if they originate from a bad state.

This restrictions allows us to benefit automatically from any parts of the two transition systems that are equal. This applies to both latches and combinational circuitry. The reason for this is the fact that if such latches are present, the property will assert that the corresponding latches/variables are equal. Our tool will then collapse the logic that is shared by both transition systems. In the special case that both transition systems have the exact same set of latches/variables, the problem is reduced to SAT-based combinational equivalence checking. While we do not propose to use our tool for this special case, we benefit from the reduction in case some parts of the transition system are equal.

The following two sections describe how to abstract the program and the circuit given the set of predicates.

3.4 Abstracting the Program

Predicate abstraction of ANSI-C programs in combination with counterexample guided abstraction refinement has become a widely applied technique. It was introduced by Ball and Rajamani [1] and promoted by the success of the SLAM project [2]. The goal of this project is to verify that Windows device drivers obey API conventions. SLAM models the program variables using unbounded integer numbers, and does not take overflow or bit-wise operators into account. The abstraction of the program is computed using a theorem prover such as Simplify [9]. The property checked mainly depends on the control flow, and thus, this treatment is sufficient. However, for C programs that represent a circuit model, we expect extensive use of bit-wise operators, and we expect that the limited range of the variables will be crucial.

Thus, we compute the abstraction not using Simplify or similar tools, but using SAT: this allows us to precisely model the semantics of the bit vector arithmetic as described in the ANSI-C standard. Furthermore, it allows us to support all ANSI-C integer operators, including the bit-wise operators [7].

The control flow structure is not changed during the abstraction, i.e., the abstraction will contain a program counter construction that models the original control flow of the C program. The conversion of all ANSI-C control flow statements including `goto` and `switch` is straight-forward. However, unbounded recursion is not supported, as we are not using a push-down-automaton. However, we do not expect unbounded recursion in programs that serve as circuit model. What remains is the abstraction of the branching conditions and the basic blocks, i.e., sequences of instructions without any control flow statements.

3.4.1 Abstracting the Basic Blocks

A basic block is a sequence of assignment statements. We first transform the basic block into static single assignment form (SSA). If pointer dereferencing operators are used, this requires a standard points-to analysis.

After the transformation into SSA, the assignments in the basic block are turned into equalities. After that, these equalities are conjuncted to form an equation system, which is equivalent to the concrete transition relation for the basic block. We denote it by $\mathcal{T}(\overline{v}, \overline{v}')$.

The abstract transition relation $\mathcal{B}(\hat{x}, \hat{x}')$ relates a current state $\hat{x}$ (before the execution of the basic block) to a next state $\hat{x}'$ (after the execution of the basic block). It is defined using α as follows:

$$\{(\hat{x}, \hat{x}') \,|\, (\alpha(\overline{v}) = \hat{x}) \wedge \mathcal{T}(\overline{v}, \overline{v}') \wedge (\alpha(\overline{v}') = \hat{x}')\} \tag{8}$$

We compute this set using SAT-based Boolean quantification, as described in section 3.7.

3.4.2 Abstracting the Branching Conditions

The expressions used in the branching conditions of the program are ideal candidates for predicates, and thus, the branching condition will often be a Boolean combination of predicates. If this is so, the predicates are simply replaced by their corresponding Boolean variables. If not so, the expression is abstracted using SAT in analogy to a basic block.

3.5 Abstracting the Circuit

Let S_c denote the set of states of the (concrete) circuit, and R_c the concrete transition relation. The abstract transition relation of the circuit can be computed directly using the circuit-part of the relation defined in definition 5:

$$\begin{array}{c}\{(\hat{x}, \hat{x}') \,|\, \exists x, x' \in S_c, : x R_c x' \wedge \\ \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\}\end{array} \tag{9}$$

This set is obtained using a Boolean quantification, as described in section 3.7. If this equation is already too hard for the SAT solver due to the sheer size of the circuit, it can be partitioned into components. The components are then abstracted separately. The final abstract transition relation is then the conjunction of the relations obtained for each part. However, this partitioning may introduce additional spurious behavior.

In [8], a similar approach to the abstraction of hardware is described. The main difference to the approach presented here is that [8] treats the SAT solver like a theorem prover, and enumerates particular abstract transitions instead of performing a Boolean quantification. The idea of using a Boolean quantification for hardware abstraction was introduced by Lahiri, Bryant, and Cook [15]. While we are using a bit-accurate representation of the circuit, [15] is using a word-level representation, which does not permit the use of bit-level operators.

3.6 Simulation and Refinement

In order to check the abstract model, we use SMV. If the property does not hold on the abstract model, SMV returns a counterexample trace. This trace is then simulated on the concrete model. This simulation corresponds to a series of BMC instances with additional constraints. The unwinding bound for the program loop constructs and the circuit can be taken from the abstract counterexample. As the instances are very similar, incremental SAT can be used. If the last BMC instance is satisfiable, the counterexample can be concretized, and the algorithm terminates.

If not so, the set of predicates has to be refined. This is done by computing preconditions of the constraint that causes the counterexample to be spurious.

3.7 Quantification using SAT

For the abstraction of both the circuit and the C program we need to obtain a representation for a set of Boolean vectors x such that a function is true for this argument. The vector x corresponds to the abstract present and next-state. In addition to x, the function also takes an existentially quantified vector y, which is used for intermediate variables for the CNF conversion and for the concrete states.

$$\{x \in \{0,1\}^i \,|\, \exists y \in \{0,1\}^j : f(x,y)\} \tag{10}$$

This corresponds to a quantification of the y variables.

The quantification is done by modifying the SAT solver Chaff [19] as follows: Every time a satisfying assignment for $f(x,y)$ is found, the algorithm records the values of the literals corresponding to x (the variables *not* to be quantified), and then adds a blocking clause in terms of these literals that eliminates all satisfying assignments with the same value for x. The literals in the blocking clauses all have a decision level, since the assignment is complete. The solver then backtracks to the highest of these decision levels and continues its search for further, different satisfying assignments. Eventually, the additional constraints will make the problem unsatisfiable, and the algorithm terminates. The blocking clauses added by the algorithm are a DNF representation of the desired set.

This technique is commonly used in other areas, for example in [17, 11] and was suggested earlier for solving quantified formulae in [21, 22]. In [15], our implementation of this algorithm was applied to predicate abstraction for hardware and software systems. It outperformed BDDs on all software examples. The basic algorithm can be improved by heuristics that try to enlarge the cube represented by each clause. McMillan [17] uses conflict graph analysis in order to enlarge the cube. Gupta et al. [11] use BDDs for the enlargement. However, these techniques are beyond the scope of this article.

4 Circuit Specification using C

4.1 Cycle Accurate C Programs

The equivalence criterion defined in section 2 allows a wide range of styles for the ANSI-C program. This is done by adjusting the relations that define input and output equivalence, and by defining the labeling function L appropriately.

A cycle accurate C model has to compute the values of all latches of the circuit in every cycle. These values have to be stored in specially designated program variables. In our tool, this is done by a separate file which contains an entry for each latch containing the name of the latch in the circuit and the name of the C program variable. Let v_1 and v_2 denote such a pair of a corresponding latch and a variable for all such variables V.

A special "next cycle" command indicates that this computation is finished. It can be used in arbitrary locations. When invoked, the C program makes an externally visible transition. This is done by defining L_2 (the labeling function for the program) to be σ for states that have a program counter value corresponding to the "next cycle" command. L_1 (the labeling function for the circuit) is defined to be constantly σ.

Furthermore, the "next cycle" command asserts that the values computed by the C program match the values in the circuit. This is done by defining $\widehat{=}_O$ as $\bigwedge_{v \in V} v_1 = v_2$.

The C program performs input by reading the corresponding input signals of the circuit. This is enforced by defining $\widehat{=}_I$ for the input signals and variables in analogy to $\widehat{=}_O$.

4.2 Non-Cycle Accurate C Programs

The related work in [12] allows accessing the values of the signals of the circuit in arbitrary cycles by using the syntax `signal[cycle]`. Our approach does not allow this, and restricts the access to the cycle value in the current cycle only. However, one can still write a wide range of non-cycle accurate C models by adding additional program variables to "remember" previous signal values.

It is not necessary for the C program to compute the values of all latches for each cycle. Instead, only selected values may be compared by using an explicit `assert` statement. As an example, the following fragment checks that a counter (a variable imported from the circuit) increases only:

```
extern int counter;

  while(1) {
    int previous=counter;
    next_cycle();
    assert(counter>=previous);
  }
```

This C program makes no attempt to actually reproduce the computation of the circuit; it is used as a monitor only. Note that, in contrast to [12], there is no need to refer to a bound, as we perform an unbounded verification. The `while` loop is unbounded.

The assertions are implemented as follows: First, the function L_2 is defined to be σ for states that have a program counter value corresponding to an `assert` statement. Second, $\widehat{=}_O$ is defined to hold if the assertions are true. Formally, let pc_i denote the program counter of assertion i, and $cond_i(s_1, s_2)$ the condition of the assertion. Then, $\widehat{=}_O$ is defined as follows:

$$\widehat{=}_O(s_1, s_2) \quad := \quad \bigwedge_i (s_2.pc = pc_i \Longrightarrow cond(s_1, s_2))$$

The following example illustrates how inputs are synchronized: suppose the circuit performs a division $1/x$ using an iterative algorithm that is controlled by a state machine. If the signal `ready` is true, the state machine reads a new value `x`. If the signal `done` is true, the division is finished. The C program waits for the `ready` signal and copies the value of `x` from the circuit. It then waits for the `done` signal and checks the division result, which is provided by the circuit as `r`.

```
extern unsigned int x, r;
extern _Bool ready, done;

  while(1) {
    /* local variable to remember x */
    unsigned int my_x;

    /* wait for ready, then copy x */
    while(!ready) next_cycle();
    my_x=x;

    /* wait for done, then check result */
    while(!done) next_cycle();
    assert(r==1/my_x);

    next_cycle(); /* next round */
  }
```

This is implemented by simply adding constraints to $\widehat{=}_I$. The match between the variables of the C program and the signals in the circuit can be automated if there is a syntactical rule. As an example, our implementation matches variables and signals based on their names. Signals within the Verilog module hierarchy are mapped using `struct` types. Another way to implement this mapping would be a file that explicitly lists the corresponding signals and variables.

5 Experimental Results

We compare the performance of the approach presented in this paper with an implementation using Bounded Model Checking as suggested in [12]. Bounded Model Checking is used for refutation only, i.e., it cannot conclude that there is no error trace. Instead, it checks the correspondence of the program and circuit up to a given number of cycles. In contrast to that, the approach presented in this paper can conclude that both transition systems match. The experiments are performed on a 1.5 GHz AMD machine with 3 GB of memory running Linux.

The benchmarks (table 1) we use are taken from an implementation of an out-of-order RISC microprocessor with Tomasulo scheduler [26]. The processor implements a MIPS-like ISA, and features precise interrupts by means of a reorder buffer [25].

Benchmark	# latches	bug length	Run time BMC [12]					Run time abstraction
			min.	10	20	30	40	
ALU_PIPE1	163	2	1.7s	3.7s	370.7s	21.8s	8.2s	36.6s
ALU_PIPE2	163	-	-	303.7s	*	*	*	31.0s
RF1	1024	-	-	13.7s	84.8s	134.0s	356.8s	0.5s
RF2	1024	1	0.7s	7.7s	20.3s	44.4s	*	0.7s
ROB1	2963	-	-	3.8s	10.3s	21.8s	116.0s	0.2s
ROB2	2963	-	-	63.3s	*	*	*	3.8s
ROB3	2963	16	5.7s	2.5s	7.0s	10.6s	14.3s	1.8s
ROB4	2963	64	106.0s	2.5s	5.3s	9.8s	21.5s	14.1s

Table 1: Experimental Results. If no bug length is given, the program and circuit are consistent. The run time for BMC is given for various depths. The "min" column contains the run time for BMC for the shortest counterexample. A star (*) denotes that the timeout of 1000s was exceeded. The best times for refutation are in bold.

The `ALU_PIPE` circuit implements pipelined versions of arithmetic circuits. The corresponding C program observes the values that enter the pipeline and wait for the result at the end of the pipeline. They then compare the result with an internally computed result. The C program computes the result in one step. Proving the two to be consistent requires predicates that assert the correctness of the intermediate results in the pipeline. These predicates are computed automatically during the abstraction refinement phase. Note that for the satisfiable instances the time required until BMC finds a counterexample actually decreases with the bound. The unsatisfiable instance is hard for BMC.

The `RF` circuit contains the integer register file. The C program checks properties of the register values.

The `ROB` circuit is the reorder buffer of the design. It contains a large number of latches. In the `ROB1` benchmark, the C programs check properties of the control. The `ROB2` benchmark has C program which is a bit-accurate implementation of the control part. The `ROB3` benchmark uses the C program to check a (failing) property of a counter in the design.

In conclusion, BMC can outperform the abstraction based approach if there is a short counterexample. This can be justified by the fact that the abstraction based approach has to perform a simulation in order to confirm a counterexample. This simulation is as hard as a BMC instance. If the counterexample is long, the simulation step apparently benefits from the additional constraints from the abstract counterexample.

However, the abstraction based approach is superior if the property actually holds. In this case, the abstraction based approach can conclude that there is no counterexample, while BMC cannot.

6 Conclusion and Future Work

The paper presents an algorithm to check the correspondence of a C program and a circuit given in Verilog. The C program may be cycle accurate, a partial implementation, or just a monitor. The equivalence criterion is formalized and then reduced to a safety property. This property is then checked using predicate abstraction. We show the effectiveness of the algorithm using benchmarks from processor design.

In the future, we plan to implement floating point arithmetic for the C program, as C programs with floating point arithmetic are commonly used as efficient circuit model. Furthermore, we would like to investigate refinement algorithms that are specialized for this algorithm.

References

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

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

[3] 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'99)*, 1999.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Yhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. H. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[6] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Principles of Programming Languages*, January 1992.

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

[8] E. Clarke, M. Talupur, and D. Wang. SAT based predicate abstraction for hardware verification. In *Proceedings of SAT'03*, May 2003.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

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

[11] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *Formal Methods in Computer-Aided Design (FMCAD)*, number 1954 in LNCS, pages 354–372, 2000.

[12] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[13] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer Verlag, January 2003.

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

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

[16] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of C-based hardware descriptions by using symbolic simulation and program slicer. In *International Workshop on Logic and Synthesis (IWLS'03)*, 2003.

[17] K. McMillan. Applying SAT methods in unbounded symbolic model checking. In *14th Conference on Computer Aided Verification*, pages 250–264, 2002.

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, June 2001.

[20] C. Pixley. Guest Editor's Introduction: Formal Verification of Commercial Integrated Circuits. *IEEE Design & Test of Computers*, 18(4):4–5, 2001.

[21] D. Plaisted. Method for design verification of hardware and non-hardware systems, October 2000. United States Patent, 6,131,078.

[22] D. Plaisted, A. Biere, and Y. Zhu. A satisfiability tester for quantified boolean formulae. *Journal of Discrete Applied Mathematics (DAM)*, 130(2):291–328, 2003.

[23] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT) - automatic verification of a compilation process. *Int. Journal of Software Tools for Technology Transfer (STTT)*, 2(2):192–201, 1998.

[24] L. Séméria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle. RTL C-based methodology for designing and verifying a multi-threaded processor. In *Proc. of the 39th Design Automation Conference*, pages 123–128. ACM Press, 2002.

[25] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[26] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.