Predicate Abstraction and Refinement for Model Checking VHDL State Machines

Mustapha Bourahla

Computer Science Department, University of Biskra
BP 145 RP, Biskra, Algeria, 07000
Email: mbourahla@hotmail.com

Mohamed Benmohamed

Computer Science Department, University of Constantine
Constantine, Algeria, 25000

Abstract

In this paper we present an automatic combination of abstraction-refinement by which we translate a VHDL model describing a state system to an initial equivalent abstract system described by SMV to explore its state space to verify CTL proper-
ties. We present the method implemented to compute automatically abstractions using decision procedures. This method can handle different kinds of infinite state systems including systems composed of concurrent components and it can be ex-
tended for more complex VHDL concepts. Abstract models may admit spurious counterexamples (false negative results) which are executions at the abstract level with no corresponding executions at the concrete level. We devise a new algorithm which analyzes such counterexamples and refine the abstract model correspondingly by eliminating gradually the false negative results. We illustrate our approach on an example and we confirm its effectiveness on a large design.

1 Introduction

The main idea of abstract interpretation of digital systems, is to interpret the behavior of a system in a different abstracted (and therefore simplified) system with fewer states for handling the state explosion problem in applying model checking to large industrial designs. An abstraction can be seen as a relation between two systems. On one hand, the original system has the complete description of its behavior, whereas its abstraction preserves some of that behavior and abstracts the rest. The verification task is then performed in the abstracted system. There are two types of abstractions: exact abstractions are those where the result of the verification in the abstract system implies an
equivalent result in the concrete system. In the case of conservative abstractions, on the other hand, only certain results in the verification of the abstract system can be implied in the original system.

Verification by abstraction appears to be promising for reasoning about control intensive designs in which control is finite but the data part is infinite or very large [9] [10]. Abstract models are usually provided manually, and theorem proving is used to check that the provided abstract mapping preserves the properties. Recently, novel techniques based on abstract interpretation have been proposed in the context of the verification of temporal properties where theorem proving is used to compute automatically finite abstractions [2] [7] [8]. These techniques are quite effective, but require heavy use of theorem proving and decision procedures. There are methods/tools that compute an abstract system from the text of a finite state program and an abstraction relation [6]. It should be realized that it is important to avoid the construction of the concrete model which represents the semantics of the considered program before generating the abstract system. Otherwise, one would have to store the concrete system which might be too large. The produced abstract system is usually smaller than the concrete one, and hence is much simpler to model-check.

Verification by abstraction can also be applied to infinite state systems as shown in [12]. However, in all these approaches the verifier has to provide the abstract system and an important amount of user intervention is required to prove that the abstract system simulates the concrete one. What is needed is a method to automatically compute an abstract system for a given infinite state system and an abstraction relation. A method that achieves this for a restricted form of abstraction functions, namely those induced by a set of predicates on the concrete states, is given in [14]. This method has, however, the drawback that it generates an abstract graph rather than the text of an abstract program with the consequence that one can neither apply further abstractions nor techniques for avoiding the state explosion problem as, for example, partial-order techniques. There is another method [3] based on elimination during the construction of abstract systems. Then, to construct an abstract transition of a concrete transition starting from the universal relation, which relates every abstract state to every abstract state, this method eliminates pairs of abstract states such that after elimination of a pair the obtained transition is still an abstraction of the concrete transition. This method is too complex because the number of transitions of the universal relation is exponential in the number of variables. This method was combined with other techniques based on partitioning the set of abstract variables, using substitutions but this partitioning leads to a more non-deterministic abstract system and then more spurious counterexamples.

The drawback of using abstraction followed by model checking as a verification and analysis technology consists in the fact that abstractions are approximations of the original systems that induce false negative results. For
instance, a model checker may exhibit an error trace that corresponds to an execution of the abstract program that violates the desired properties. However, this error trace may not correspond to an execution trace in the concrete program. This situation indicates that the abstraction is too coarse, and that the results of model checking the abstract system are not conclusive. That is too many details were abstracted and the abstraction needs to be refined.

We propose a method for the automatic construction of predicate abstractions extracted from VHDL models to abstract infinite transition systems such that the abstract model by construction simulates the concrete system. These systems can be composed of concurrent components. But the process of constructing the abstract system does not depend on whether the computational model is synchronous or asynchronous, i.e., interleaving based. In general, our technique computes an upper approximation of the original system. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction. Our method differs from that in [3], so that only the last behavior which caused the spurious counterexample is eliminated. Clarke [6] has presented other technique in another framework of abstraction based on abstraction from a concrete model, but when a spurious counterexample is present, to get a refined abstract model his method eliminates all the non reachable states in the spurious counterexample by using comparison with the behavior of the concrete model. This also costs time and it is not necessary.

The VHDL models are written using a subset of the language [19] and a certain modeling style taken from the most synthesis tools. The VHDL model states are represented symbolically and the abstract state is a conjunction of one of these states and truth assignment to the abstract Boolean variables. The false negative results will be gradually eliminated by an automatic process called refinement which uses information obtained from spurious counterexamples. The verification methodology is based on abstraction followed by model checking and refinement. If there is no possible refinement, the system will report counterexamples by mapping each step in the trace to the concrete domain. We have used an example to explain our method which is implemented to automatically construct the abstract systems.

This paper is organized as follows: Section 2 presents modeling style of transition systems with VHDL. In Section 3, we present the framework of predicate abstraction used by our algorithm of abstraction presented in Section 4. Section 5 presents the algorithm of refinement. An overview of the tool and analysis of results are presented in Section 6. At the end a conclusion is given.
2 Modeling Transition Systems with VHDL

Hardware Description Languages (HDLs), most notably VHDL, have gained considerable popularity in the specification of hardware designs. VHDL supports process level parallelism. It employs constructs with complicated semantics to achieve concurrency, communication and synchronization among the processes [19]. VHDL constructs such as signal assignment statements and wait statements facilitate deterministic inter process communication and coordination. One can exploit these features of VHDL to write succinct behavioral descriptions.

Definition 2.1 (transition system). A transition system $M$ is a tuple $M = (S,V,T,I)$, where

- $S$ is a set of system states
- $V$ is a set of system variables of any type
- $T$ is a set of system transitions, each transition is associated with a guard expression and a set of action expressions over the set $V$
- $I$ is a set of initial states

VHDL is a language particularly adapted to the description of transition systems because of its high level syntax (instructions if ... then ... else, case ...) which allows direct translations of traditional graphic representations like graphs and diagrams. With the VHDL syntax, we can name the states, the signals, etc. This gives us a clear and readable descriptions. For abstraction we have chosen a subset of VHDL to describe transition systems. A transition system can be composed of one behavioral component or many concurrent components. Each component is described by one process. The variables of the transition system can be of any type: Boolean, bit, integer, real, etc., and the states are represented symbolically. A directive is introduced to write the CTL formulas to be checked over the VHDL model. We illustrate our verification approach on the well known algorithm that computes the GCD (Great Common Divider) of two natural numbers $x$ and $y$. The transitions are of the form condition/action, with the meaning that the transition takes place if condition is true, and then action is executed. The VHDL model of GCD is shown in Figure 1 ("<=" is the VHDL assignment operator).

```vhdl
entity GCD is port(clk : in bit; x, y : in integer;
                   start : inout bit; z : out integer);
end entity GCD;
architecture Behavior of GCD is
  Type State is (S0, S1, S2);
  Signal S : State := S0; Signal xp, yp : natural;
begin
  process begin wait until Clk = '1';
    case S is
      when S0 => xp := x; yp := y;
      when S1 => if xp < yp then S := S0; xp := xp - yp; else S := S1; yp := yp - xp end if;
      when S2 => if xp = 0 then S := S0; z := yp end if;
    end case;
  end process;
end Behavior;
```
when S0 =>
    if start = '0' then S <= S0 end if;
    if start = '1' then xp <= x; yp <= y; S <= S1; end if;
when S1 =>
    if xp < yp then yp <= yp - xp; S <= S1; end if;
    if xp > yp then xp <= xp - yp; S <= S1; end if;
    if xp = yp then Z <= xp; S <= S2; end if;
when S2 => Start <= '0'; S <= S0
end case;
end process;
-- $ AG(Start = '1' => AF(xp = yp))
end behavior;

Fig. 1. VHDL model of GCD and property specification

3 Framework of Predicate Abstractions

Predicate abstraction consists of using predicates over concrete variables as Boolean abstract variables [14]. It can be defined in the framework of abstract interpretation using Galois connections.

Definition 3.1 (Abstraction by Galois Connection). Let $S_c$ and $S_a$ represent the concrete and abstract state domains respectively. A Galois connection [2] from $S_c$ to $S_a$ is a pair of functions $\alpha : 2^{S_c} \rightarrow 2^{S_a}$ and $\gamma : 2^{S_a} \rightarrow 2^{S_c}$ such that:

- $\alpha$ and $\gamma$ are total and monotonic.
- $\forall X \in 2^{S_c}, \gamma \circ \alpha (X) \supseteq X$, and
- $\forall X \in 2^{S_a}, \alpha \circ \gamma (X) \supseteq X$.

Theorem 3.2 (Relation between connection and simulation). Let $R_c$ and $R_a$ represent the transition relations of $M^c$ and $M^a$ respectively. If $(\alpha, \gamma)$ is a connection between $S_c$ and $S_a$ and $\forall S' \in 2^{S_a}$, then $\alpha(Pre(R_c, \gamma(S'))) \subseteq Pre(R_a, S')$ then $M^c \preceq M^a$, where $Pre$ is the pre image function.

If $P$ is a predicate over concrete variables, a predicate abstraction can be expressed as a Galois connection [14] as follows:

$$\alpha(P) = \bigwedge \{B^a | P \Rightarrow \gamma(B_a)\} = P^a,$$

where $B^a$ is any Boolean expression over the set $\{B_1, ..., B_k\}$ which is the set of abstract variables corresponding to the set of concrete predicates $\{\phi_1, ..., \phi_k\}$. $\gamma$ is defined as a substitution function, that is, $\gamma(P^a) = P^a[\phi_1/B_1, ..., \phi_k/B_k]$, where each Boolean variable $B_i$ is substituted by its corresponding correct predicate $\phi_i$. Thus, the abstraction of a concrete set of states represented by a predicate $P$ over concrete variables is defined as the smallest Boolean formula $P^a$ over the abstract variables $B_i$, that is, an over approximation of $P$. For computing the most precise Boolean abstraction with respect to a set
of predicates, for systems where the transition relation is given as a relational predicate, an efficient enumeration of all Boolean combinations \( B^a \) to test the assertion \( P \Rightarrow \gamma(B^a) \) should be specified. This will abstract systems where the transition relation is given as a predicate. Each implication \( P \Rightarrow \gamma(B^a) \) is submitted to the decision procedure to test its validity. Notice that any approximation of \( P^a \) is a valid abstraction of \( P \).

Thus, in order to compute for a concrete system \( M \), an abstract system \( M^a \), it is sufficient to abstract the initial state \( I \) by computing \( \alpha(I) \), and to abstract each transition \( t \in T \) as follows:

\[
t^a = \alpha(t) = \alpha(action_t(V, V')) = \bigwedge \{ (B^a, B^a') | \vdash post[t](\gamma(B^a)) \Rightarrow \gamma(B^a') \},
\]

that is, the pair \((B^a, B^a')\) characterizing the abstraction of the set of possible predecessors by \( t \) and the abstraction of the set of possible successors by \( t \), where \( post \) expresses the strongest post condition by a transition \( t \) of a predicate \( P \) over the state variables of \( V \), it is defined as follows:

\[
post[t](P) = \exists V'. action_t(V', V) \land P(V'),
\]

where \( action_t(V', V) \) is defined as the relation between the current state and next state, that is the expression:

\[
(s = s_i) \land guard \land \bigwedge_{i=1}^{l} (v_i' \leftarrow e_i) \land (next(s) = s_j)
\]

The preservation of properties expressed in temporal logic is established via equivalences and preorders between the concrete and abstract models.

**Theorem 3.3** (weak preservation). Let \( M \) be a concrete system, and let \( M^a \) be a predicate abstraction of \( M \) using any set of predicates. We have \( M^a \models \alpha(\phi) \Rightarrow M \models \phi \), for each temporal formula \( \phi \).

**Proof.** All the executions of \( M \) are executions of \( M^a \), then if a property holds along all execution paths of \( M^a \), it holds in all execution paths of \( M \). This means that \( M^a \) simulates \( M \), because the following holds for each transition \( t \) of \( M \):

\[
\forall P. post[t](P) \Rightarrow \gamma(post[\alpha(t)](\alpha(P))). \quad \square
\]

This theorem indicates that when a property is established in the abstract system, its corresponding concrete property holds in the concrete system. However, nothing can be concluded when the property does not hold in the abstract system. Strong preservation results can be applied in this case under some conditions.

**Theorem 3.4** (strong preservation). Let \( M \) be a concrete system, and let \( M^a \) be a predicate abstraction of \( M \) using any set of predicates that includes all the literals appearing in the guards of \( M \) and in the property \( \phi \). If \( M^a \) is deterministic, we have \( M^a \models \alpha(\phi) \Leftrightarrow M \models \phi \), \( M^a \) and \( M \) are equivalent.

You can find its proof in [14]. The strong preservation result allows us
to avoid false negative results by mapping abstract error traces to concrete executions violating the property. However, the condition for strong preservation requires that $M^a$ be deterministic. This is usually not the case. Each abstract state is then a conjunction of a subset of the set of Boolean variables which are the codes of the finite abstract domain. The concretization of an abstract state is a set of concrete states that can be represented as a predicate. We have used these notions of predicate abstractions to automatically abstract transition systems described with VHDL. The next section presents the algorithm and illustration on the example of GCD.

4 Automatic Construction of Predicate Abstractions

The algorithm uses decision procedures for the automatic construction of a predicate abstraction of a concrete, infinite state system described as a transition system with VHDL. The abstraction of a concrete system $M = (S, V, T = \{t_1, ..., t_n\}, I)$ is an abstract system $M^a = (S, V^a, T^a = \{t^a_1, ..., t^a_n\}, I^a)$ such that:

- $V^a$ is the set $\{B_1, ..., B_k\}$
- $T^a$ is a set of abstract transitions.
- $I^a$ is the abstract initial state computed as $\alpha(I)$.

The abstraction algorithm consists in computing $I^a$ and for each concrete transition $t$ defined as $(s = s_i) \land \text{guard} \land \text{action} \land (\text{next}(s) = s_j)$ a corresponding abstract transition $t^a$ defined as $(s = s_i) \land \text{guard}^a \land \text{action}^a \land (\text{next}(s) = s_j)$

**Algorithm 1 Abstraction**

**Step 1:** Define the abstraction function $\alpha$ using the predicates in the transition guards and the CTL formula. The function $\gamma$ is the corresponding substitution function.

**Step 2:** For each guard, the abstract guard $(\text{guard}^a)$ is computed as $\alpha(\text{guard})$. When using the literals of the guards as abstract Boolean variables, $\alpha(\text{guard})$ is an exact abstraction, where each literal of guard is substituted syntactically by its corresponding abstract Boolean variable.

**Step 3:** Construction of a list $L$ of all the Boolean expressions $B^a$ of the form $\bigwedge(B_i \lor \neg B_i)$ using the abstract variables.

**Step 4:** The action assignments of each transition will be abstracted to a Boolean expression composed of maximum number of abstract variables and it should validate the implication:

$$\text{post}[t](\text{true}) \Rightarrow \gamma(\text{the abstract Boolean expression}).$$

The "abstract Boolean expression" is a conjunction of all the Boolean expressions $B^a$ taken from the list $L$, where the implication $\text{post}[t](\text{true}) \Rightarrow \gamma(B^a)$ is valid. This means that for each abstract variable $B_i$ in this expression, the strongest post condition by $t$ of any arbitrary state is in $\gamma(B_i)$ or in $\neg \gamma(B_i)$, that is, in $\phi_i$ or in $\neg \phi_i$.

If the abstract variable is not in the expression this means that is not deterministic.

**Step 5:** The variable $S$ is not abstracted since it is of finite type.
4.1 Illustration on the Example

We use predicates over concrete variables which are extracted from the VHDL model, as Boolean abstract variables. The transition table generated after the parse of the VHDL model (Figure 1), is shown on Table 1.

<table>
<thead>
<tr>
<th>N</th>
<th>Present State</th>
<th>Guard</th>
<th>Action</th>
<th>Next State</th>
<th>Guard(a)</th>
<th>Action(^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S0</td>
<td>Start = 0</td>
<td>Empty</td>
<td>S0</td>
<td>B1</td>
<td>Empty</td>
</tr>
<tr>
<td>2</td>
<td>S0</td>
<td>Start = 1</td>
<td>xp := x; yp := y</td>
<td>S1</td>
<td>\neg B1</td>
<td>Empty</td>
</tr>
<tr>
<td>3</td>
<td>S1</td>
<td>xp &lt; yp</td>
<td>yp := yp - xp</td>
<td>S1</td>
<td>B2</td>
<td>Empty</td>
</tr>
<tr>
<td>4</td>
<td>S1</td>
<td>xp &gt; yp</td>
<td>xp := xp + yp</td>
<td>S1</td>
<td>\neg B3</td>
<td>Empty</td>
</tr>
<tr>
<td>5</td>
<td>S1</td>
<td>xp = yp</td>
<td>z := xp</td>
<td>S2</td>
<td>\neg B2 \lor \neg B3</td>
<td>Empty</td>
</tr>
<tr>
<td>6</td>
<td>S2</td>
<td>True</td>
<td>Start := 0</td>
<td>S0</td>
<td>true</td>
<td>B1 := true</td>
</tr>
</tbody>
</table>

Table 1

Transition table of the GCD

The columns Guard\(a\) and Action\(^a\) are filled in after the abstraction. First, we compute the abstract initial state. The VHDL model contains one initialization statement (\(S := S0\)), this will not be abstracted. Then, the abstract initial states are any state verifying the formula (\(S = S0\)). Second, we compute the abstract guards of all the transitions along with the specification predicates. The set of predicates presented in the column Guard with the set of predicates generated from the CTL formulas presented by the directive \(--\$\ \) (in this case, the set is \(\{\text{Start} = '1', xp = yp\}\)), will be the entry to the abstraction algorithm for producing the set of abstract Boolean variables and the equivalent abstract predicate of each concrete predicate using a decision procedure. The algorithm will take the predicates in the Guard column one by one and it will try to express them with the already constructed abstract variables. If it is not possible, it decomposes the predicate to simpler predicates (by simpler, we mean removing the Boolean connectors) and then, it associates a new abstract variable to one of them which is not already associated and then it will retry the process until the predicate is completely expressed with the constructed abstract variables. We will repeat the process until all the predicates can be expressed with abstract variables.

If the set of abstract variables for our example is \(\{B1 \text{ for } (\text{start} = 0), B2 \text{ for } (xp < yp), \text{ and } B3 \text{ for } (xp > yp)\}\), this means that the abstraction function \(\alpha\) is defined by the predicate \((B1 \leftrightarrow (\text{start} = 0)) \land (B2 \leftrightarrow (xp < yp)) \land (B3 \leftrightarrow (xp > yp))\), then we need to represent all the guard predicates with the minimum of abstract variables using calls to a decision procedure. The abstract guard of transition number 3 (see Table 1), for instance, is \(\neg B2 \lor \neg B3\) because the implication \((xp = yp) \Rightarrow \gamma(\neg B2 \land \neg B3)\) is checked to be valid.

Third, we compute the abstraction for each assignment in the action of each transition. The assignments are in the Action column. We need to realize a conjunction of the maximum number of abstract variables (or their
negations) to abstract these assignments such that, the following implication
"assignment of transition action \( \Rightarrow \gamma \) (conjunction of the maximum number
of Boolean abstract variables)" should be valid. These implications will be
checked by calls to a decision procedure. The abstraction of an action is the
conjunction of all the abstractions of its assignments. After the construction
of all the abstract predicates, a translation program will generate the equiv-
alent SMV module (Figure 2). The SMV system [20] is a tool for checking
finite state systems against specifications in the temporal logic CTL.

Module main
VAR
    B1 : boolean; B2 : boolean; B3 : boolean;
    S : {S0, S1, S2};
INIT
    S = S0
TRANS
    (S = S0 & B1 & next(S) = S0) | (S = S0 & !B1 & next(S) = S1) |
    (S = S1 & B2 & next(S) = S1) | (S = S1 & B3 & next(S) = S1) |
    (S = S1 & !B2 & !B3 & next(S) = S2) |
    (S = S2 & next(B1) & next(S) = S0)
INVAR
    (B2 & !B3) | (!B2 & B3) | (!B2 & !B3)
SPEC
    AG(!B1 -> AF(!B2 & !B3))

Fig. 2. SMV Abstract model

4.2 Invariant Generation

In the SMV module (Figure 2), there is an invariant. The invariant is a formula
representing a set of states and each state reachable in the system, is in this set.
The invariant formula is to make consistence between the abstract Boolean
variables already generated so that to not get a state in which there will not
be a formula making no sense, and then avoiding the system to reach useless
states. By example there will not be any concrete state verifying the formula
\( B2 \land B3 \) (its equivalent in the concrete domain is "\((xp > yp) \land (yp > xp)\)"").
The idea of the following algorithm is to check if not \( (B2 \land B3) \) is a tautology.
If yes, this combination will be removed from the invariant formula.

Algorithm 2 Invariant Generation

Consider \( B \) is the set of abstract Boolean variables
Consider \( P \) is the set of the equivalent predicates
Consider \( P^S \) is the set of subsets from \( P \), where predicates of each
element from \( P^S \), are using the same subset of concrete variables
Consider \( B^S \) is a set of subsets from \( B \), where each \( B_j \in B \) is an abstract of
one of \( P_j^S \) from \( P^S \)
The algorithm first, searches the abstract Boolean variables that are abstractions of concrete predicates using the same concrete variables and grouping them in clusters of variables. Then the algorithm will try to check all the conjunctions composed of these clusters of variables. If the negation of the equivalent conjunction in the concrete domain, is checked to be a tautology, it will not be inserted in the invariant formula. We should remark that the invariant can be true (empty).

4.3 Model Checking the Abstract Model

Once an abstract system is constructed, the SMV model checking system is used to explore its state-space. The advantage of model checking over other verification techniques is its ability to generate counterexamples when a property is violated. The error trace is a sequence of states and transitions starting from the initial state of the system leading to a state violating the property. Error traces of an abstract system can be mapped to executions of a concrete system since each abstract transition corresponds to a single concrete one. Figure 3 shows an error trace which is a spurious loop counter example violating the property specified.

The simulation of the error trace on the concrete system indicates that it does not correspond to an execution of the concrete system. However, this does not rule out the possibility that the property is violated. In the next section, we present an algorithm to show how model checking can guide the automatic refinement of an abstract system until the property is verified or a counterexample corresponding to a concrete execution violating the property is generated.

5 Automatic Refinement of Abstractions

The specification above is not satisfied by the initial abstract system already constructed. The system SMV produced an error trace indicating the violation
of this specification (Figure 3). By analysis of this error trace, we understand that the abstract system is executing a trace which cannot be executed in the concrete system. The abstract system is too abstract and it needs to be refined. Effectively, the transition number 4 (see Table 1) is defined by the formula \( S = S_1 \land B_3 \land \text{next}(S) = S_1 \). The Boolean variable \( B_3 \) can get the next value true or false which is not deterministic, but in the concrete system the next value of \((xp > yp)\) will eventually get the value false. Figure 4 shows the different counterexamples (A, B and C) generated after each step in the refinement process of this module until the satisfaction of the specified liveness property.

\[
\neg B_1 \land \neg B_2 \land \neg B_3 \land (S = S_0) \\
\neg B_1 \land \neg B_2 \land B_3 \land (S = S_1) \\
\neg B_1 \land B_2 \land \neg B_3 \land (S = S_1) \\
\neg B_1 \land \neg B_2 \land B_3 \land (S = S_0) \\
B_1 \land \neg B_2 \land \neg B_3 \land (S = S_0) \\
B_1 \land \neg B_2 \land B_3 \land (S = S_1) \\
B_1 \land \neg B_2 \land B_3 \land (S = S_2) \\
B_1 \land \neg B_2 \land B_3 \land (S = S_0) \\
\neg B_1 \land \neg B_2 \land \neg B_3 \land (S = S_1) \\
B_1 \land \neg B_2 \land B_3 \land (S = S_0)
\]

Fig. 4. Error traces generated by model checking different levels of refinement

Thus, we have model checked four abstract models produced gradually by refinement from the initial abstract model (Figure 2), until the property is verified. In the following and before presenting our algorithm of refinement, we will explain our method of refinement on this example. We take the formula of the last transition \( t^e \) in the error trace (see Figure 3)

\[
t^e \equiv (S = S_1) \land \neg B_1 \land \neg B_2 \land B_3 \land \neg \text{next}(B_1) \land \neg \text{next}(B_2) \land \text{next}(B_3) \land (\text{next}(S) = S_1)
\]

By mapping to the abstract transition system already we have (the initial abstract model), the equivalent abstract transition \( t^a \), is (see Figure 2)

\[
t^a \equiv (S = S_1) \land B_3 \land (\text{next}(S) = S_1)
\]

This transition formula should verify the equality \( t^e \land t^a = t^e \) to be considered. In our approach we take only the abstract variables that are used in the abstraction of predicates composed of concrete variables used by the action of the transition \( t^e \), which has \( t^a \) as its abstract transition (in this case they are \( B_2 \) and \( B_3 \)). Then, we take one abstract variable (for example, \( B_3 \)). The predicate \( \text{next}(B_3) \) is in \( t^e \) (because, \( t^e \land \text{next}(B_3) = t^e \)) and it is not in \( t^a \), because \( t^a \land \text{next}(B_3) \neq t^a \). Then we should check the validity of
action \Rightarrow \gamma(B3)$. In other words the following implication should be valid $(xp = xp - yp) \Rightarrow (xp > yp)$. But, the decision procedure does not valid this, and because it causes an error in the model, we may use its negation to avoid it and the transition will be written like the following for the new refined abstract model.

$$(S = S_1 \land B3 \land \neg next(B3) \land next(S) = S_1)$$

When we model check this modified abstract module, we get another error trace (Figure 4 A) and the formula of the last transition $t^e$ in the new error trace is

$$t^e \equiv (S = S_1 \land B1 \land \neg B2 \land next(B2) \land \neg next(B3) \land next(S) = S_1)$$

The equivalent transition in the current abstract model $t^a$, is

$$t^a \equiv (S = S_1 \land B3 \land \neg next(B3) \land next(S) = S_1)$$

This is the abstract transition already modified. Now we apply the same rule as above and the abstract variable $B2$ will be taken. The predicate $next(B2)$ is not occurring in the current transition so we have to validate $(xp = xp - yp) \Rightarrow (yp > xp)$ which is also invalid and it can be eliminated from the abstract system. The transition of the second refined abstract model should be written now like this

$$(S = S_1 \land B3 \land \neg next(B2) \land \neg next(B3) \land next(S) = S_1).$$

There still an error (Figure 4 B), and its trace gives the formula of the last transition

$$t^e \equiv (S = S_1 \land B2 \land \neg B3 \land next(B2) \land \neg next(B3) \land next(S) = S_1)$$

This formula modifies by the same way, the transition $t^a$ to be

$$(S = S_1 \land B2 \land \neg next(B2) \land next(S) = S_1)$$

Now the error trace (Figure 4 C) gives the formula

$$t^e \equiv (S = S_0 \land B1 \land \neg B2 \land next(B1) \land \neg next(B2) \land next(B3) \land S = S_0)$$

The equivalent transition in the current abstract system is

$$t^a \equiv (S = S_0 \land B1 \land S = S_0).$$

There is no action for this transition, so we can take one of the abstract variables $B1$, $B2$, or $B3$. For example, we take $B1$ and the new transition of the refined abstract model is (the implication $true \Rightarrow \gamma(B1)$ is not valid)

$$(S = S_0 \land B1 \land \neg next(B1) \land S = S_0).$$

Therefore, this last refined abstract model verifies the specification.
5.1 Refinement Algorithm

After this execution of refinement process on the example, we present our detailed algorithm. This algorithm will be called every time we get an error trace path after model checking an abstract model. The algorithm does not introduce new predicates. It refines the transitions without adding states, so this method of refinement is for liveness and reachability properties.

Algorithm 3 Refinement

Let \( P \) to be the path of the abstract error trace

While \( P \) is not empty do

Let \( t^e \) to be the formula of the last transition in the abstract error trace and \( t^a \) is the corresponding transition in the abstract model, which is verifying the equality \( t^a \land t^e = t^a \)

Let \( A \) to be the set of abstract variables of concrete predicates using concrete variables occurring in the action of the equivalent concrete transition \( t^c \) of \( t^a \)

while \( A \) is not empty do

take \( v \) from \( A \)

if \((\neg next(v) \land t^e = t^c)\) and \((\neg next(v) \land t^a \neq t^a)\) and not Valid(action\_t \Rightarrow \gamma(v)) then

Refine the abstract model by changing \( t^a \) to be \( t^a \land \neg next(v) \)
and return

elseif \((\neg next(v) \land t^e = t^c)\) and \((\neg next(v) \land t^a \neq t^a)\) and not Valid(action\_t \Rightarrow \neg\gamma(v)) then

Refine the abstract model by changing \( t^a \) to be \( t^a \land next(v) \)
and return

end if

end while

\( P := P - \text{last abstract state} \)

end while

There is no possible refinement then, output the concrete counterexample after mapping the abstract error trace.

Thus, the main idea of this algorithm is to search the non deterministic abstract variables in the last abstract transition in the error trace. Then it takes these variables one by one to negate their next value which gives a refined abstract model (it is an under approximation). If such variables don’t exist it will go backward until the first transition. At the end if there are no deterministic variables, the concrete counterexample will be produced using the function \( \gamma \).

6 Overview and Experiments

Figure 5 shows an overview of the tool implementing our methodology based on abstraction - model checking - refinement which is dedicated to the verification of infinite state systems.
We have implemented this tool under the operating system Windows and we have used the decision procedure of the system SVC (Stanford Validity Checker) [1] [16] to prove theorems. After the execution of the abstraction process we call the system SMV to model check the produced abstract model. If there is a counterexample we call the refinement process as explained above to produce a refined new abstract model in the case if the counterexample is spurious.

In addition to the GCD example, we have used this system to verify the mutual exclusion property (this is a safety property but no refinement was needed) in the Bakery protocol, and by which we have tested the inter-process communication with VHDL and its equivalent in the system SMV. The Bakery protocol is composed of many parallel components each one is represented by a VHDL process. We have also verified the ATM (Asynchronous Transfer Mode) switch [5]. This is relatively a large design and it uses many components. The table below shows our experiments with the three designs. The table shows the number of abstract variables used. It shows the number of implications generated and proved for each abstraction. Also the number of refinements, and the global time of verification.

<table>
<thead>
<tr>
<th>Case</th>
<th># of abstract variables</th>
<th># of calls to decision procedure</th>
<th># of refinements</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCD</td>
<td>3</td>
<td>18</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Bakery</td>
<td>3</td>
<td>33</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td>ATM</td>
<td>17</td>
<td>254</td>
<td>8</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 2
Experiment results
7 Conclusion

We have presented a novel abstraction refinement methodology for symbolic model checking VHDL models describing state machines, which can be infinite transition systems. The methodology which is implemented by a completely automatic tool, consists of an algorithm for the automatic construction of predicate abstraction by which we translate a VHDL model to an equivalent abstract SMV model, and an efficient algorithm for automatically refining a coarse abstraction when model checking the abstract system fails. This refinement algorithm eliminates gradually the spurious paths in the error trace. The construction of the initial abstract system and the refinement process use many calls to the decision procedure. The choice of non deterministic abstract variables in the refinement algorithm has big effect on its performance and good heuristics for their selection will approve it.

References


