Discovering the Input Assumptions in Specification Refinement Coverage

Prasenjit Basu Sayantan Das Pallab Dasgupta[*] P.P. Chakrabarti[*]

Department of Computer Science & Engineering,
Indian Institute of Technology, Kharagpur, INDIA 721302
{pbasu,sayantan,pallab,ppchak}@cse.iitkgp.ernet.in

Abstract— The design of a large chip is typically hierarchical – large modules are recursively expanded into a collection of submodules. Each expansion refines the design due to the addition of level specific details. We believe that a similar approach is necessary to scale the capacity of formal property verification technology – as the design gets refined from one level to another, the formal specification must also be refined to reflect the level specific design decisions. At the heart of this approach we propose a checker that identifies the input assumptions under which the refined specification "covers" the original specification. This enables the validation engineer to focus the verification effort on the remaining input scenarios thereby reducing the number of target coverage points for simulation.

I. Introduction

In recent times, most leading chip design companies are seriously investigating the possibility of integrating *formal property verification* (FPV) [1] into their pre-silicon validation flows. It is widely accepted that the major challenge in existing FPV technology is in capacity. Unfortunately the theoretical lower bounds on the complexity of model checking problems imply that the technology is unlikely to scale beyond a point, in spite of the advances in the engineering of formal tools. Therefore a new methodology is required to extend the frontiers of FPV technology and enable us to formally verify properties over large designs.

How do we succeed in developing the RTL of large and complex designs? We do so by recursively decomposing the design functionality into the functionality of smaller modules, and by adding more details as we go down the module hierarchy. We continue this process of refinement until the modules are simple enough to be treated as unit level modules. We believe that the future of formal property verification lies in adopting a similar approach for formal property specifications. Whenever, we refine the design functionality into that of its component modules, we need to refine the formal specification of the design into that of its component modules. We refer to this paradigm as *formal specification refinement*.

The key formal method required for enabling specification refinement is a prover that checks whether the refined specification consisting of the specifications of the component modules *covers* the original specification of the design. In other words we need to verify that *all behaviors that refute the original specification also refute the refined specification*. For example, let $\mathcal{A}$ denote the set of architectural properties of a design. Suppose we implement the design in terms of a set of RTL modules, $M_1 \ldots M_k$. Capacity limitations of existing FPV tools do not allow us to verify $\mathcal{A}$ over the whole design. Today validation engineers write properties, $\mathcal{R}_i$ over each RTL module, M_i and verify them, but this does not guarantee that they together satisfy $\mathcal{A}$. This guarantee can be obtained by a specification refinement proof which verifies that $\mathcal{R}_1, \ldots, \mathcal{R}_k$ together cover $\mathcal{A}$. Since the proof compares the specifications only (and not specification versus implementation – as in existing FPV techniques), we do not have major capacity limitations. In an earlier work, we developed the foundations of specification refinement proofs [2]. In that work we presented the methodology for comparing two temporal logic specifications at different levels of abstraction, and finding whether one covers the other.

What should we do if the refinement checker returns a negative answer? One option is to add more RTL properties to plug the coverage gap. In practice however, it is not always possible to cover system level properties in this way. Therefore the remaining properties in $\mathcal{A}$ must be checked dynamically during simulation.

This brings us to an interesting problem. Typically the coverage gap is a function of the input scenarios. In other words, we often find that the RTL properties succeed in covering the original specification $\mathcal{A}$ under some input constraints and fail for the rest. If we are able to compute the input constraints under which the refinement checker succeeds, then we can target the simulation on the remaining scenarios. This is the main goal of this paper.

In this paper we define a methodology for constraining the coverage gap between temporal specifications to determine the input constraints under which the gap is closed. The negation of these constraints cover the scenarios that must be checked through simulation. We believe that this enhancement to the basic refinement checker is of significant practical value in design validation through specification refinement.

The paper is organized as follows. In Section II, we present the theoretical background of the specification refinement problem. The formalism for computing the input constraint that covers the gap is presented in Section III. Section IV presents the algorithms for finding a legible representation of the input constraint. Section V presents the results of

[*]Pallab Dasgupta and P.P.Chakrabarti acknowledge the partial support of the Dept. of Sci & Tech, Govt. of India.

our prototype tool on several test cases. We have presented a case study on a realistic example in the Appendix.

II. Specification Refinement Coverage

In this paper we will focus on covering the gap between the formal architectural specification and the combined formal properties of the RTL components through specification refinement.

1. The key architectural properties are specified using a formal property specification language.[1] This forms the *architectural intent*, $\mathcal{A}$.
2. In the first phase of implementation, the design is planned in terms of a set of RTL blocks, and the functionality of each of these RTL blocks are defined. At this stage, formal properties are written to express some of the correctness requirements of each RTL block. The properties of the RTL blocks taken together is called the *RTL specs*, $\mathcal{R}$.
3. Our specification refinement checker checks whether the RTL specs, $\mathcal{R}$, cover the architectural intent, $\mathcal{A}$. We refer to this check as the *primary coverage question*.

The following definition formally describes the notion of coverage of the architectural intent by the RTL specs.

Definition 1 [Coverage Definition:]
The RTL specs cover the architectural intent iff there exists no run that refutes one or more properties of the architectural intent but does not refute any property of the RTL specs. □

In [2], it was shown that *the RTL specs $\mathcal{R}$, cover the architectural intent $\mathcal{A}$, iff the temporal property $\mathcal{R} \Rightarrow \mathcal{A}$ is valid.*

Most model checking tools for LTL [3] and its derivatives already have the capability of performing validity (or satisfiability) checks on temporal specifications, and can therefore be used to check whether $\mathcal{R} \Rightarrow \mathcal{A}$ is valid.

An arsenal of heuristic algorithms had been proposed in [2], for representing the coverage gap as a set of temporal properties when the answer to the primary coverage question is negative. While this feedback is useful in deciding whether and where more properties need to be added, it does not help the validation engineer to find the input scenarios that cover the behaviors in the gap.

III. Covering Input Assumptions

Specification refinement is not always feasible. It relies on the design architect's ability to add properties over individual RTL modules in a way to cover the architectural properties of the whole design. The refinement checker verifies the correctness of the decomposition and shows the gaps, but it does not automate the decomposition of the specification.

It is therefore natural to expect that in spite of adding more RTL properties on component modules, some architectural behaviors will remain uncovered. These behaviors must be targeted by simulation and dynamic property verification techniques.

[1]We shall use Linear Temporal Logic (LTL) in this paper

Our objective in this paper is to partition the architectural behaviors in terms of the input scenarios that trigger those behaviors. We use the specification refinement approach to show that the RTL specs cover the architectural specs under some of the input assumptions (expressed as properties over input variables). Since these scenarios are covered by specification refinement FPV, the remaining scenarios (also expressed as properties over inputs) represent the cases that must be verified through simulation. We believe that this is a very useful feedback to the validation engineer.

The following example demonstrates our intent. This is a toy example, used to demonstrate the main idea intuitively. In the appendix, we present a real example.

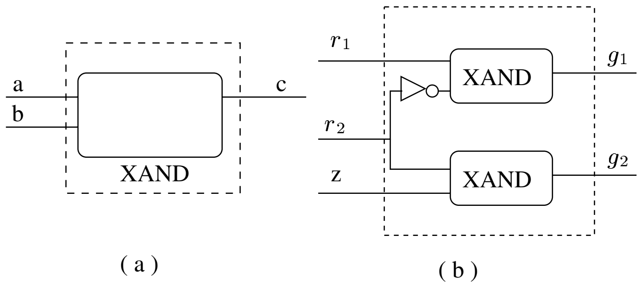

Fig. 1. A sample arbiter

Example 1 Let us consider the design of an arbiter that arbitrates between two request lines r_1 and r_2 from two master devices. Let the corresponding grant lines to the master devices be g_1 and g_2. The arbiter also receives an input z from a slave device, that remains high as long as the slave device is *ready*.

The arbiter specification requires us to treat r_2 as a high-priority request. Whenever r_2 is asserted and the slave is ready (that is, z is high), the arbiter must give the grant, g_2 in the next cycle, and continue to assert g_2 as long as r_2 remains asserted. When r_2 is not high, the arbiter parks the grant on g_1 regardless of whether r_1 is asserted. We are further given, that the request r_2 is fair in the sense that it is de-asserted infinitely often (enabling g_1 to be asserted infinitely often).

The above architectural intent may be expressed in LTL as follows:

$$\begin{array}{ll} A_1: & G\,F(\,\neg r_2\,) \\ A_2: & G(\,(\,r_2 \wedge z\,) \Rightarrow X(\,g_2\,U\,\neg r_2\,)\,) \\ A_3: & G(\,(\,\neg r_2\,) \Rightarrow X\,g_1\,) \end{array}$$

Let us now consider an implementation of the arbiter using a component called XAND, as shown in Fig 1. The specification of the module XAND is as follows:

$$R_1{}': \quad G(\,(\,a \wedge b\,) \Rightarrow X\,c\,)\,)$$

It may be noted that we do not require the internal implementation of the RTL module XAND. Property $R_1{}'$ is part of the RTL specification for XAND. Substituting the signal names of the instances of XAND in Fig 1(b) with r_1, r_2, g_1, g_2 and z, and adding the fairness property on r_2, we have the RTL specification as:

R_1: $G\,F(\,\neg r_2\,)$
R_2: $G(\,(\,r_1 \;\wedge\; \neg r_2\,) \;\Rightarrow\; X\;g_1\,)$
R_3: $G(\,(\,r_2 \;\wedge\; z\,) \;\Rightarrow\; X\;g_2\,)$

The first property is the same fairness constraint as in the architectural intent. The second property says if r_1 is asserted and r_2 is de-asserted then g_1 is asserted in the next cycle. The third property states that whenever r_2 and z are asserted together, then g_2 is asserted in the next cycle.

The primary coverage problem is to determine whether $(R_1 \wedge R_2 \wedge R_3) \Rightarrow (A_1 \wedge A_2 \wedge A_3)$ is valid. In this case the answer is negative. For example whenever we have a scenario where both r_1 and r_2 are low, the architectural intent requires g_1 to be asserted, but the RTL specification does not have this requirement, which shows that A_3 is not covered. The coverage gap can be accurately represented by the following property:

U_1: $G(\,(\,\neg r_1 \;\wedge\; \neg r_2\,) \;\Rightarrow\; X\;g_1\,)$

Our previous specification refinement checker will produce the property U_1 as the coverage gap. In the new approach we will produce the input constraint:

$$I_C\,:\,G(\,r_1\;\vee\;r_2\,)$$

Under assumption I_C, we have $(R_1 \wedge R_2 \wedge R_3) \Rightarrow A_3$. Therefore RTL FPV succeeds in covering A_3 under these cases. Simulation should target input sequences satisfying $\neg I_C$, which is:

$$I_U\,:\,F(\,\neg r_1\;\wedge\;\neg r_2\,)$$

This is indeed what we want, since the coverage gap U_1 represents the cases where r_1 and r_2 are low together. □

The inputs to our problem are:

1. The *architectural intent* $\mathcal{A}$, as a set of LTL properties over a set $\mathcal{AP}_\mathcal{A}$, of Boolean signals, and
2. The *RTL specification* $\mathcal{R}$, as another set of LTL properties over a set, $\mathcal{AP}_\mathcal{R}$, of Boolean signals,
3. Additionally, we have the interface information of the architectural block; namely the set of inputs $\mathbb{I}_\mathcal{A}$, and the set of outputs $\mathbb{O}_\mathcal{A}$, of the architectural block, where $\mathcal{AP}_\mathcal{A} = \mathbb{I}_\mathcal{A} \cup \mathbb{O}_\mathcal{A}$.

We shall also use $\mathcal{A}$ to denote the conjunction of the properties in the architectural intent, and $\mathcal{R}$ to denote the conjunction of the properties in the RTL specification.

Throughout this paper we assume that $\mathcal{AP}_\mathcal{A} \subseteq \mathcal{AP}_\mathcal{R}$. This assumption essentially means that the RTL specification has the same names for their signals as the corresponding ones in the architectural intent. The RTL specification can have other signals in addition to these. Typically this is not a restrictive assumption within the design hierarchy, since it is generally considered a good practice for designers at a lower level of the design hierarchy to inherit the interface signal names from the previous level of hierarchy.

Definition 2 [Strong and weak properties:]
A property $\mathcal{F}_1$ is stronger than a property $\mathcal{F}_2$ iff $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ and $\mathcal{F}_2 \not\Rightarrow \mathcal{F}_1$. We also say that $\mathcal{F}_2$ is weaker than $\mathcal{F}_1$. □

Definition 3 [Coverage Hole in RTL Spec:]
A coverage hole in the RTL specification is a property $\mathcal{R}_H$ over $\mathcal{AP}_\mathcal{R}$, such that $(\mathcal{R} \wedge \mathcal{R}_H) \Rightarrow \mathcal{A}$, and there exists no property, $\mathcal{R}'_H$, over $\mathcal{AP}_\mathcal{R}$ such that $\mathcal{R}'_H$ is weaker than $\mathcal{R}_H$ and $(\mathcal{R} \wedge \mathcal{R}'_H) \Rightarrow \mathcal{A}$. In other words, this is the weakest property that suffices to close the coverage hole. Adding the weakest property strengthens the RTL specification in a minimal way. □

Since $\mathcal{AP}_\mathcal{A} \subseteq \mathcal{AP}_\mathcal{R}$, each property of the architectural intent is a valid property over $\mathcal{AP}_\mathcal{R}$. In [2], we have shown that the coverage hole in the RTL specification is unique and is given by $\mathcal{A} \vee \neg\mathcal{R}$.

Definition 4 [Covering Input Constraint:]
A covering input constraint is a property $\mathcal{I}_\mathcal{C}$ over $\mathbb{I}_\mathcal{A}$, such that under the assumption $\mathcal{I}_\mathcal{C}$, the RTL specification covers the architectural intent i.e. $\mathcal{I}_\mathcal{C} \Rightarrow (\mathcal{R} \Rightarrow \mathcal{A})$, and there exists no property $\mathcal{I}'_\mathcal{C}$ over $\mathbb{I}_\mathcal{A}$ such that $\mathcal{I}'_\mathcal{C}$ is weaker than $\mathcal{I}_\mathcal{C}$ and $\mathcal{I}'_\mathcal{C} \Rightarrow (\mathcal{R} \Rightarrow \mathcal{A})$. In other words, we find the weakest assumption over $\mathbb{I}_\mathcal{A}$ that suffices to close the coverage gap. □

According to the definition, the covering input constraint $\mathcal{I}_\mathcal{C}$ is a property over $\mathbb{I}_\mathcal{A}$, such that $\mathcal{I}_\mathcal{C} \Rightarrow (\mathcal{A} \vee \neg\,\mathcal{R})$. That is, it is stronger than the coverage hole in the RTL specification and no other property over $\mathbb{I}_\mathcal{A}$ which is weaker than $\mathcal{I}_\mathcal{C}$ has the same characteristic.

Definition 5 [Uncovered Input Space:]
An uncovered input space is a property $\mathcal{I}_\mathcal{U}$ over $\mathbb{I}_\mathcal{A}$, such that every run that refutes one or more properties of the architectural intent but does not refute any property of the RTL specification, implies $\mathcal{I}_\mathcal{U}$. Intuitively, these are the input scenarios for which the required guarantees have not been specified in the RTL specs. □

Clearly the input scenarios which fall outside the covering input constraint, are the uncovered input scenarios. Thus the uncovered input space is $\neg\,\mathcal{I}_\mathcal{C}$.

IV. Computing the *Uncovered Input Space*

In this section, we present the algorithms for determining the uncovered input space as defined in Definition 5.

A. Coverage Algorithm

Our algorithm takes each formula $\mathcal{F}_\mathcal{A}$ from the architectural intent $\mathcal{A}$ and the interface definition of the architectural block and finds the uncovered input space, $\mathcal{I}_\mathcal{U}$, for $\mathcal{F}_\mathcal{A}$, with respect to the RTL specification $\mathcal{R}$. Since $\mathcal{R}$ is required to cover every property in $\mathcal{A}$ (by definition), we use this natural decomposition of the problem.

Algorithm 1 Coverage Algorithm

```
Find_Uncovered_Input_Space($\mathcal{F}_\mathcal{A}$, $\mathcal{R}$, $\mathbb{I}_\mathcal{A}$, $\mathbb{O}_\mathcal{A}$)
  1. Compute $\mathcal{U} = \mathcal{F}_\mathcal{A} \vee \neg\,\mathcal{R}$
```

2. If $\mathcal{U}$ is not valid then
   (a) Unfold $\mathcal{U}$ up to its fixpoint to create two sets of uncovered terms; one is the disjunction of the terms before the fixpoint, ${\mathcal{U}_M}^{BF}$ and the other is the disjunction of the terms at the fixpoint, ${\mathcal{U}_M}^{AF}$;
   (b) Use universal quantification to eliminate signals belonging to $\mathcal{AP}_\mathcal{R} - \mathcal{AP}_\mathcal{A}$ from both the sets;
   (c) $\mathcal{I}_{BF}$ = Call Find_ISpace_BeforeF(${\mathcal{U}_M}^{BF}$, $\mathbb{I}_\mathcal{A}$, $\mathbb{O}_\mathcal{A}$);
   (d) $\mathcal{I}_{AF}$ = Call Find_ISpace_AtF($\mathcal{F}_\mathcal{A}$, ${\mathcal{U}_M}^{AF}$, $\mathbb{I}_\mathcal{A}$, $\mathbb{O}_\mathcal{A}$);
   (e) $\mathcal{I}_\mathcal{C}$ = Call Combine_Both_Parts($\mathcal{I}_{BF}$, $\mathcal{I}_{AF}$, k); [where k is the no. of unfolding required for $\mathcal{U}$ to reach the fixpoint]
   (f) $\mathcal{I}_\mathcal{U} = \neg\, \mathcal{I}_\mathcal{C}$;
3. Return $\mathcal{I}_\mathcal{U}$;

The first step computes the coverage gap $\mathcal{U}$. If $\mathcal{U}$ is valid then $\mathcal{F}_\mathcal{A}$ is covered. Otherwise we determine the uncovered input space. The second step of the algorithm performs this task. This step is further divided into six steps. The following subsections describe each of these steps in details.

B. Step 2(a): Unfolding of $\mathcal{U}$

In this step we recursively unfold $\mathcal{U}$ and generate two sets of disjunction of terms, namely ${\mathcal{U}_M}^{BF}$ and ${\mathcal{U}_M}^{AF}$, that contain only Boolean subformulas and Boolean subformulas guarded by a finite number of X (next) operators.

Definition 6 [X-depth, X-pushed, X-guarded:]
A formula is said to be X-pushed if all the X operators in the formula are pushed as far as possible to the left. A formula is said to be X-guarded if the corresponding X-pushed formula starts with an X operator whose scope covers the whole formula. The X-depth of an operator within a formula is the number of X operators whose scope covers the operator in the X-pushed form. □

Any LTL property can be recursively unfolded over time steps to create an equivalent properties over Boolean formulas and X-guarded LTL formulas. It is known that for every temporal property such a decomposition begins to produce similar X-guarded subformulas after a well defined number of unfolding. During the unfolding process we check whether such a fixpoint has been reached. Once we reach the fixpoint (say after k steps of unfolding), then we take the formula produced after $(k - 1)$ levels of unfolding and abstract out the temporal operators other than X as in [4]. We rewrite it in the form of disjunction of terms to generate ${\mathcal{U}_M}^{BF}$. In case $k = 1$, we consider ${\mathcal{U}_M}^{BF}$ as $True$. ${\mathcal{U}_M}^{AF}$ is obtained by dropping the terms that contain any temporal operator other than X from the one step decomposed form of the fixpoint formula.

Example 2 Let us consider the architectural property A and the RTL properties $R1$ and $R2$ as given below where r_1, r_2, r_3 are inputs to the architectural block and g_1 is the output:

$$A:\ G(\,r_1 \ \Rightarrow\ X\ g_1)$$

$$R1:\ (\,r_1 \ \wedge\ \neg\, r_3\,)\ \Rightarrow\ X\ g_1$$

$$R2:\ G(\,(\,r_1 \ \wedge\ r_2\,)\ \Rightarrow\ X\ g_1$$

We compute the disjunction of the uncovered terms before the fixpoint, ${\mathcal{U}_M}^{BF}$ and obtain the following formula:

$${\mathcal{U}_M}^{BF}:\ \neg\, r_1\ \vee\ X(\,g_1\,)\ \vee\ (\,r_1\ \wedge\ \neg\, r_3\ \wedge\ \neg\, X(\,g_1\,)\,) \vee\\ (\,r_1\ \wedge\ r_2\ \wedge\ \neg\, X(\,g_1\,)\,)$$

Again unfolding the fixpoint part and abstracting out the unbounded temporal operators, we yield the following disjunction of terms as ${\mathcal{U}_M}^{AF}$.

$${\mathcal{U}_M}^{AF}:\ \neg\, r_1\ \vee\ X(\,g_1\,)\ \vee\ (\,r_1\ \wedge\ r_2\ \wedge\ \neg\, X(\,g_1\,)\,)\quad \Box$$

C. Step 2(b): Abstraction

In this step we universally eliminate the variables in $\mathcal{AP}_\mathcal{R} - \mathcal{AP}_\mathcal{A}$ from the properties ${\mathcal{U}_M}^{BF}$ and ${\mathcal{U}_M}^{AF}$.

D. Step 2(c): Finding covering input constraint before fixpoint

We treat each different X-guarded Booleans (different in X-guarded Boolean variables and/or in the number of X operators guarding the variables) in ${\mathcal{U}_M}^{BF}$ as separate Boolean variables and characterize them as inputs or outputs depending on whether the X-guarded Boolean variables are inputs or outputs respectively. Now we universally abstract out the outputs from ${\mathcal{U}_M}^{BF}$ and obtain the covering input constraint before fixpoint, namely $\mathcal{I}_{BF}$.

Example 3 Let us again consider the architectural property A and the RTL properties $R1$ and $R2$ as given in Example 2. We consider $X\ g_1$ as an output variable since g_1 is an output of the architectural block. We universally abstract out the output $X\ g_1$ from ${\mathcal{U}_M}^{BF}$ to get $\mathcal{I}_{BF}$ as below:

$$\mathcal{I}_{BF}:\ \neg\, r_1\ \vee\ (\,r_1\ \wedge\ \neg\, r_3\,)\ \vee\ (\,r_1\ \wedge\ r_2\,)\qquad \Box$$

E. Step 2(d): Finding covering input constraint at fixpoint

In this step we first use the algorithm described in [2] to find out the uncovered architectural intent (say $\mathcal{G}$) using ${\mathcal{U}_M}^{AF}$ as the uncovered minterms. We have developed an approximate strategy for automatically identifying the input scenarios that *trigger* non-vacuous interpretations of a temporal property. Given a property, φ, defined over input variables $\mathcal{I}$, and output variables $\mathcal{O}$, we generate a formula S_φ that is stronger than φ and is defined only over $\mathcal{I}$. Since S_φ is stronger than φ and is free from output variables, it follows that S_φ describes input scenarios that make φ vacuously true. Therefore, we restrict the input space by $\neg S_\varphi$, which covers all non-vacuous runs.

Example 4 Consider the property, $\varphi : G[\,r_1 \Rightarrow X\ g_1\,]$. Then $S_\varphi = G(\neg r_1)$ and the constraint that restricts the input space to non-vacuous runs is $\neg S_\varphi = F\ r_1$, where r_1 as an input and g_1 is an output. □

To find the required property over the inputs we define two operators, namely $\mathcal{S}$ and $\mathcal{W}$ which correspond to strengthening and weakening operations respectively. We present the rules for pushing the operators $\mathcal{S}$ and $\mathcal{W}$ downto the variables.

- **The rules for $\mathcal{S}$ operator:**
  - $\mathcal{S}(\theta\ f) \equiv \theta(\mathcal{S}(f))$ where θ is from $\{X, G, F\}$
  - $\mathcal{S}(f\ \eta\ g) \equiv \mathcal{S}(f)\ \eta\ \mathcal{S}(g)$ where η is from $\{\vee, \wedge, U\}$
  - $\mathcal{S}(\ f\ \Rightarrow\ g\ )\ \equiv\ \mathcal{W}(\ f\ )\ \Rightarrow\ \mathcal{S}(\ g\ )$
  - $\mathcal{S}(\ \neg\ f\ )\ \equiv\ \neg\ \mathcal{W}(\ f\ )$
- **The rules for $\mathcal{W}$ operator:**
  - $\mathcal{W}(\theta\ f) \equiv \theta(\mathcal{W}(f))$ where θ is from $\{X, G, F\}$
  - $\mathcal{W}(f\ \eta\ g) \equiv \mathcal{W}(f)\ \eta\ \mathcal{W}(g)$ where η is from $\{\vee, \wedge, U\}$
  - $\mathcal{W}(\ f\ \Rightarrow\ g\ )\ \equiv\ \mathcal{S}(\ f\ )\ \Rightarrow\ \mathcal{W}(\ g\ )$
  - $\mathcal{W}(\ \neg\ f\ )\ \equiv\ \neg\ \mathcal{S}(\ f\ )$

After applying the above rules on $\mathcal{S}(\ \mathcal{G}\ )$, we yield a form where only the variable instances are preceded by an $\mathcal{S}$ or $\mathcal{W}$ operator. The substitution rule for a variable instance v is to substitute v by False for $\mathcal{S}(\ v\ )$, and by True for $\mathcal{W}(\ v\ )$.

We substitute the output variable instances of $\mathcal{G}$ according to the actions derived from $\mathcal{S}(\ \mathcal{G}\ )$ and keep the input variables intact, thus obtaining a formula which is stronger than $\mathcal{G}$ and does not have any output variables. We call this formula, the covering input constraint at fixpoint, namely $\mathcal{I}_{AF}$.

Example 5 Let us consider the same architectural property A and RTL properties $R1$ and $R2$ as given in Example 2. After applying the coverage algorithm of [2] we yield the formula $\mathcal{G}$ as follows:

$$\mathcal{G}\ :\ G(\ (\ r_1\ \wedge\ \neg\ r_2\ )\ \Rightarrow\ X\ g_1\ )$$

Now we replace the output variable instances by True/False so as to strengthen $\mathcal{G}$. In this case we substitute g_1 by False. After reducing the substituted formula we get $\mathcal{I}_{AF}$ as shown below:

$$\mathcal{I}_{AF}\ :\ G(\ \neg\ (\ r_1\ \wedge\ \neg\ r_2\ )\ ) \qquad \Box$$

F. Step 2(e): Combining two parts

Here we combine the input constraints $\mathcal{I}_{BF}$ and $\mathcal{I}_{AF}$ to produce the covering input constraint $\mathcal{I}_{\mathcal{C}}$. It may be noted that intuitively $\mathcal{I}_{BF}$ tells about the input constraint before the fixpoint and $\mathcal{I}_{AF}$ tells about the input constraint from the fixpoint under which the RTL properties cover the architectural intent. Hence we augment $\mathcal{I}_{AF}$ with $(k-1)$ X operators in the prefix where k is the fixpoint depth of $\mathcal{U}$ and take conjunction between $\mathcal{I}_{BF}$ and the X-augmented $\mathcal{I}_{AF}$ to yield $\mathcal{I}_{\mathcal{C}}$.

Example 6 Let us again consider the properties of Example 2. The fixpoint depth in this example is 2. After augmenting one X operator in the prefix of $\mathcal{I}_{AF}$ we get the following formula:

$$\mathcal{I}'_{AF}\ :\ X\ G(\ \neg\ (\ r_1\ \wedge\ \neg\ r_2\ )\ )$$

We produce $\mathcal{I}_{\mathcal{C}}$ by taking the conjunction of $\mathcal{I}_{BF}$ and $\mathcal{I}'_{AF}$.

$$\mathcal{I}_{\mathcal{C}}\ :\ (\ \neg\ r_1\ \vee\ (\ r_1\ \wedge\ \neg\ r_3\ )\ \vee\ (\ r_1\ \wedge\ r_2\ )\ )\ \wedge \\ X\ G(\ \neg\ (\ r_1\ \wedge\ \neg\ r_2\ )\ ) \qquad \Box$$

G. Step 2(f): Final step

Finally, we present $\mathcal{I}_{\mathcal{U}}$ as the negation of $\mathcal{I}_{\mathcal{C}}$.

Example 7 Continuing with the same example we can see that the uncovered input space in this case is as follows:

$$\mathcal{I}_{\mathcal{U}}\ :\ (\ r_1\ \wedge\ \neg r_2\ \wedge\ r_3\ )\ \vee\ X\ F(\ r_1\ \wedge\ \neg\ r_2) \qquad \Box$$

V. Results

We have implemented a tool for computing the uncovered input scenarios. In Table I, we give the run times of our tool for some example circuits, including ARM AMBA AHB [5] and two Intel test cases, on a 2.8 GHz Intel Xeon processor with 4GB RAM. The second and third columns of the table present the no. of architectural signals and RTL signals respectively, for the circuit under test (mentioned in the first column). In the fourth and fifth columns, we produce the time (in seconds) taken by our tool to solve the primary coverage question and to compute the uncovered input space respectively. The results are quite promising. In all of the cases, the tool produced the uncovered input space, wherever necessary, in reasonable time.

TABLE I
RUNTIMES OF OUR TOOL

Circuit	$\lvert\mathcal{AP}_{\mathcal{A}}\rvert$	$\lvert\mathcal{AP}_{\mathcal{R}}\rvert$	Time(s)	
			Primary Coverage Question	Uncov. Input Space
Memory Arb. Logic	17	26	1.66	26.01
Intel Test Case1	8	16	0.005	0.02
Intel Test Case2	8	16	0.01	0.11
AMBA AHB Arb.	12	12	0.005	0.07
Paper Example	5	5	0.005	0.02

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*, MIT Press, 2000.

[2] S. Das, P. Basu, A. Banerjee, P. Dasgupta, P. P. Chakrabarti, C.R. Mohan, L. Fix, R. Armoni, “Formal verification coverage: computing the coverage gap between temporal specifications,” In *Proc. of ICCAD*, 198-203, 2004.

[3] A. Pnueli, “The temporal logics of programs,” In *Proc. of FOCS*, 46-57, 1977.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, “Symbolic model checking using SAT procedures instead of BDDs,” In *Proc. of DAC*, 1999.

[5] *ARM AMBA specification rev 2.0*, http://www.arm.com/

A. Case Study: Memory Arbitration Logic

Fig. 2 shows the architecture of a memory arbitration logic in the presence of a L1 cache. There are four request inputs, $r1, \ldots, r4$, for four independent on-chip requesting modules. Each of these four modules also assert write-enable signals $w1, \ldots, w4$ respectively. When wi is high, it indicates that the

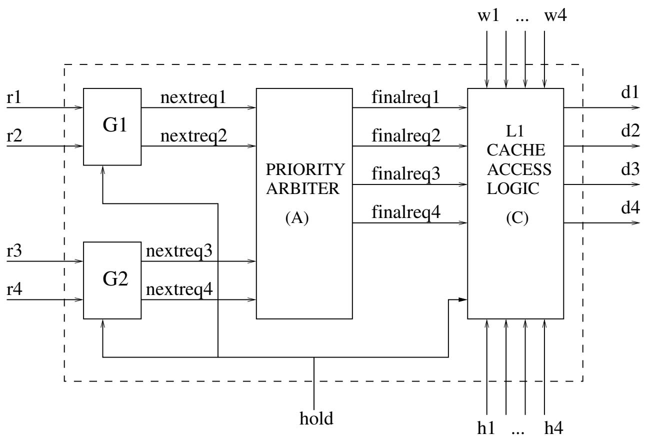

Fig. 2. Memory arbitration with L1 cache

corresponding requesting device intends to perform a write operation, where as the read operation is the default (when wi is low). The signals $h1, \ldots, h4$ are inputs from the cache that indicates whether the page requested by $r1, \ldots, r4$ respectively are present in the cache. The input signal *hold* signifies that when it is asserted the memory arbitration logic stops accepting any request. The signals $d1, \ldots, d4$ indicates whether the page requested by $r1, \ldots, r4$ respectively is ready.

The architectural intent requires that $r1$ and $r2$ have higher priority than $r3$ and $r4$. Specifically this is translated into the architectural property that-

> If two devices with different priorities make requests for the cache control unit with the higher priority device making the request before the lower priority one, the device with higher priority will always have its page ready at the output, before the device with lower priority.

This architectural level requirement can be expressed by four properties for the four different ways in which a high priority request can come with a low priority one. For example, when we consider $r1$ (high priority) with $r3$ (low priority) we have the property-

A_1: $G(((\,r1 \wedge \neg r2 \wedge \neg r3\,) \wedge X(\,r1\, U\, r3\,)\,) \Rightarrow (\,\neg d3\, U\, d1\,))$

We have the following fairness conditions on the inputs and outputs:

Q_1: $G(\,r1 \Rightarrow (\,r1\, U\, d1\,)\,)$
Q_2: $G(\,r3 \Rightarrow (\,r3\, U\, d3\,)\,)$

which require the requesting device to hold the request line high until the page becomes ready. In addition we have the following two fairness restrictions:

Q_3: $G(\,\neg r1 \Rightarrow (\,\neg d1 \wedge X(\,\neg d1\,)\,)\,)$
Q_4: $G(\,\neg r3 \Rightarrow (\,\neg d3 \wedge X(\,\neg d3\,)\,)\,)$

which states whenever a device is idle (i.e. not requesting), its page ready signal will be low in that cycle and its next cycle. Fig. 2 shows the architecture of this logic in terms of four modules. G1 and G2 are round-robin arbiters. Whenever *hold* is asserted they ignore the request lines. The variable *lastreq* represents the state of G1 indicating which device was granted in the last round. Below we present the RTL properties for G1. Those for G2 are similar.

R_{G1_1}: $G((r1 \wedge \neg r2 \wedge \neg hold) \Rightarrow (nextreq1 \wedge X(\neg lastreq)))$
R_{G1_2}: $G((\neg r1 \wedge r2 \wedge \neg hold) \Rightarrow (nextreq2 \wedge X(lastreq)))$
R_{G1_3}: $G(\,(\,r1 \wedge r2 \wedge \neg hold \wedge lastreq\,)$
$\Rightarrow (\,nextreq1 \wedge X(\,\neg lastreq\,)\,)\,)$
R_{G1_4}: $G(\,(\,r1 \wedge r2 \wedge \neg hold \wedge \neg lastreq\,)$
$\Rightarrow (\,nextreq2 \wedge X(\,lastreq\,)\,)\,)$
R_{G1_5}: $G(\,hold \Rightarrow (\,\neg nextreq1 \wedge \neg nextreq2\,)\,)$
R_{G1_6}: $mutex(\,nextreq1,\, nextreq2\,)$

The priority arbiter in Fig 2 selects requests in the priority order, namely G1-highest and G2-lowest. The specification for this module is given below:

R_{A_1}: $G(\,nextreq1 \Rightarrow X\, finalreq1\,)$
R_{A_2}: $G(\,nextreq2 \Rightarrow X\, finalreq2\,)$
R_{A_3}: $G((\neg nextreq1 \wedge \neg nextreq2 \wedge nextreq3) \Rightarrow X\, finalreq3)$
R_{A_4}: $G((\neg nextreq1 \wedge \neg nextreq2 \wedge nextreq4) \Rightarrow X\, finalreq4)$
R_{A_5}: $mutex(\,finalreq1, \ldots, finalreq4\,)$

The cache access logic in Fig 2 directly interacts with the cache and performs according to the transfer type and whether the transfer is a cache hit or a cache miss. Whenever a cache miss is occurred in a read transfer the cache logic keeps the corresponding request pending in a wait buffer. *bfull* is an internal signal which notifies the wait buffer full condition. The following three properties are given for device 1. For other devices we have the similar properties.

R_{C_1}: $G(\,(\,finalreq1 \wedge h1\,) \Rightarrow X\, d1\,)$
R_{C_2}: $G(\,(\,finalreq1 \wedge w1 \wedge \neg h1\,) \Rightarrow X\, d1\,)$
R_{C_3}: $G(\,(\,finalreq1 \wedge \neg w1 \wedge \neg h1 \wedge \neg bfull\,) \Rightarrow X\, F\, d1\,)$

Also for the buffer full condition, the following two properties are required.

R_{C_4}: $G(\,\neg hold \Rightarrow \neg bfull\,)$
R_{C_5}: $G(\,bfull \Rightarrow X\, F(\,\neg bfull\,)\,)$

Finally we have the mutual exclusion property for the ready signals.

R_{C_6}: $mutex(\,d1,\, d2,\, d3,\, d4\,)$

The answer to our primary coverage problem for the architectural property A_1 is negative i.e. A_1 is not implied by the RTL specification. On a closer look it can be observed that in the scenario when the higher priority request is a read operation that results in a cache miss, A_1 is not guaranteed by the RTL properties. A counter example scenario that demonstrates this gap is as follows:

> *Device 1 has requested for a page read in the present cycle but device 2 and device 3 haven't. In the next cycle device 3 has made a request and device 1's request results in a cache miss. So in the following cycle device 1 does not get the corresponding page ready signal but device 3's request results in a cache hit and hence in the succeeding cycle device 3 has its page ready signal available at the output.*

Our tool returns the following property as the uncovered input space:

A_H : $F(\,(\,(\,r1 \wedge \neg r2 \wedge \neg\, r3\,) \wedge (\,(\,X(\,\neg w1\,) \wedge$
$X(\,\neg h1\,)\,) \vee hold\,) \wedge X(\,r1\, U\, r3\,)\,)\,)$

The property represents the set of scenarios that are not covered by the RTL specification.