What lies between Design Intent Coverage and Model Checking?

Sayantan Das Prasenjit Basu Pallab Dasgupta P.P. Chakrabarti
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur, India.

Abstract

Practitioners of formal property verification often work around the capacity limitations of formal verification tools by breaking down properties into smaller properties that can be checked on the sub-modules of the parent module. To support this methodology, we have developed a formal methodology for verifying whether the decomposition is indeed sound and complete, that is, whether verifying the smaller properties on the submodules actually guarantees the original property on the parent module. In practice, however designers do not write properties for all modules and thereby our previous methodology was applicable to selected cases only. In this paper we present new formal methods that allow us to handle RTL blocks in the analysis. We believe that the new approach will significantly widen the scope of the methodology, thereby enabling the validation engineer to handle much larger designs than admitted by existing formal verification tools.

1. Introduction

Most leading chip design companies are today seriously exploring the role of *formal property verification* (FPV) within their existing pre-silicon validation flows. Past experience shows that FPV works very well at the unit level with modules of modest size, but runs into serious capacity issues when presented with designs of moderate size. The theoretical lowerbounds on the complexity of model checking techniques indicate that the situation is unlikely to improve significantly, even with the numerous advances that are taking place with the engineering of the FPV tools. FPV is unlikely to present us with a push-button solution for property verification on large designs.

Practitioners of FPV in the industry often find ways out of the capacity barrier by human ingenuity. If the FPV tool runs into capacity issues while attempting to check a property P, on a module M, they often attempt to *prove* P on M by checking local properties on the component submodules of M. In other words, if M consists of submodules $M_1, \ldots, M_k$, they try to figure out a set of properties, P_i for each module M_i, such that proving P_i on M_i for each M_i is sufficient to guarantee P on M. Experience shows that this methodology usually works well, and enables the validation engineer to handle larger designs than admitted by the FPV tools.

The main gap in this methodology is the lack of a formal proof that the refinement of the specification P into $P_1, \ldots, P_k$ is sound and complete. If the validation engineer's conjecture is incorrect, then a bug can hide between P and $P_1, \ldots, P_k$. In other words it may be the case that each individual submodule M_i satisfies its specification P_i, but M does not satisfy P. Therefore, in order to support this methodology we need a formal tool that compares $P_1, \ldots, P_k$ with P, verifies whether $P_1, \ldots, P_k$ *covers* all behaviors relevant to P, and more importantly shows the gap between the specifications in a meaningful form.

In an earlier work [3], we presented a methodology called *design intent coverage*, for comparing the architectural properties of a design module with the sets of RTL properties of its submodules. Our tool checks for the existence of a gap between these specifications and presents the validation engineer with a set of *missing* properties that are sufficient to cover the gap.

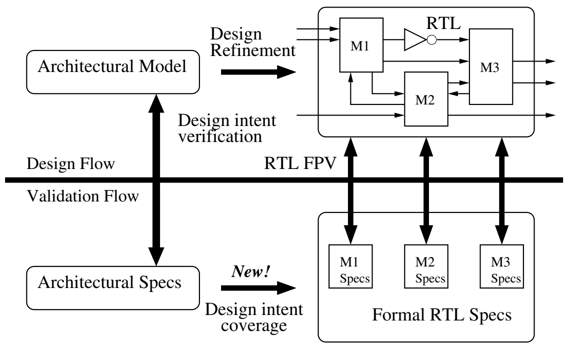


Figure 1. Design Intent Coverage

While the *design intent coverage* paradigm has been well received, there is a serious limitation of the previ-

3-9810801-0-6/DATE06 © 2006 EDAA

ous approach. In practice, the sub-modules of a module are stitched together (interconnected) using some simple logic (referred to as *glue logic*). The previous version of our tool was capable of comparing only temporal logic specifications (say P and $P_1, \ldots, P_k$). Since it did not handle the glue logic between submodules, it was unable to establish the proof of coverage in many cases where the validation engineer was actually right. Our goal in this paper is to present a methodology for handling such RTL logic in the coverage analysis.

In practice designs will always have some simple blocks for which validation engineers will not normally write any formal properties. Typical examples include pre-verified custom cells. In the proposed approach we will be able to extract the logic of these blocks into our analysis. The glue logic between submodules may also be viewed as special RTL blocks.

The fundamental problem addressed in this paper is therefore as follows. We are given a property P over the interface of a module M. The FPV tool is unable to verify P on M due to capacity limitations. The module M has submodules $M_1, \ldots, M_k$. The validation engineer has given us sets of local properties over *some* of these modules (say, $M_1, \ldots, M_j$) and the RTL of the remaining modules (say, $M_{j+1}, \ldots, M_k$). Our task is to verify whether checking the local properties on $M_1, \ldots, M_j$ is sufficient to guarantee that the module M (consisting of $M_1, \ldots, M_k$) satisfies the property P, and if not, then to present the gap in terms of properties that demonstrate the gap.

The intention here is to allow small RTL blocks to be considered, while we preserve the computational advantage of comparing two formal specifications over the model checking task of comparing a specification with an implementation. At one extreme we require the functionality of $M_1, \ldots, M_k$ to be expressed solely in terms of properties (which is our intent coverage problem), while at the other extreme we allow $M_1, \ldots, M_k$ to be presented as RTL (which is the model checking problem). The work presented in this paper is inclined more towards the former, since that is where the computational advantage lies – albeit at the expense of the human intervention in writing properties for $M_1, \ldots, M_j$.

The paper is organized as follows. In Section 2 we formalize the problem and present the basis for coverage analysis. Section 3 presents the notion of a *coverage gap* and Section 4 presents the algorithms for presenting the coverage gap in a legible form. Section 5 presents the runtimes of our tool on several industrial designs.

2. The new Intent Coverage Problem

In order to clearly differentiate between the properties of a module and the specification (properties or RTL) of its submodules, we will refer to the former as the *architectural specs* and the latter as the *RTL specs*. Therefore the *architectural specs* of a module, M, consists of a set, $\mathcal{A}$, of properties that we wish to prove on that module, but are unable to do so, due to capacity limitations of the FPV tool. Let $\mathcal{AP}_{\mathcal{A}}$ be the set of signals over which the properties in $\mathcal{A}$ are defined.

In the original version of the design intent coverage problem, the *RTL specs* consisted solely of properties over the submodules, $M_1, \ldots, M_k$ of M. In the new version of the problem, the RTL specs has two parts, namely a set of properties, $\mathcal{R}$ over some of the submodules and the RTL of the remaining modules. We shall refer to these remaining modules as *concrete modules*. Let $\mathcal{AP}_{\mathcal{R}}$ be the set of signals over which the properties in $\mathcal{R}$ are defined.

Assumption 1 *Throughout this paper we assume that $\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$.*

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.

We define a *state* as a valuation of the signals at a given time. A *run* is an infinite sequence of states over time.

Definition 1 [Coverage Definition:]
The RTL specification covers 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 specification and is consistent with the concrete modules. $\square$

Our coverage problem is as follows:

- To determine whether the RTL specification covers the architectural intent, and
- If the answer to the previous question is *no*, then to determine a set of additional temporal properties that represent the coverage gap (that is, these properties together with the RTL specification succeed in covering the architectural intent).

The following theorem answers the first question.

Theorem 1 The RTL specification consisting of the properties $\mathcal{R}$ and concrete modules $\mathcal{M}$, covers the architectural intent $\mathcal{A}$, iff the temporal property $\neg\mathcal{A} \wedge \mathcal{R}$ is false in $\mathcal{M}$.

Proof: *The property $\neg\mathcal{A} \wedge \mathcal{R}$ represents the set of runs which refutes the architectural intent but are passed by the RTL properties. If this property is false in $\mathcal{M}$ then these runs are not present in the complete RTL specification. Hence all runs passed by $\mathcal{R}$ and $\mathcal{M}$ are present in $\mathcal{A}$ and thus the RTL specification covers the architectural intent. On the other hand if $\neg\mathcal{A} \wedge \mathcal{R}$ is true in $\mathcal{M}$ then there exists a run which is passed by the RTL specification but will be refuted by the* *architectural specs and hence the RTL does not cover the architectural intent.*

The theorem shows that the primary coverage question can be answered by model checking the property $\neg\mathcal{A} \wedge \mathcal{R}$ in $\mathcal{M}$. This is feasible when $\mathcal{M}$ is a set of small modules. The following example demonstrates the essence of the coverage problem.

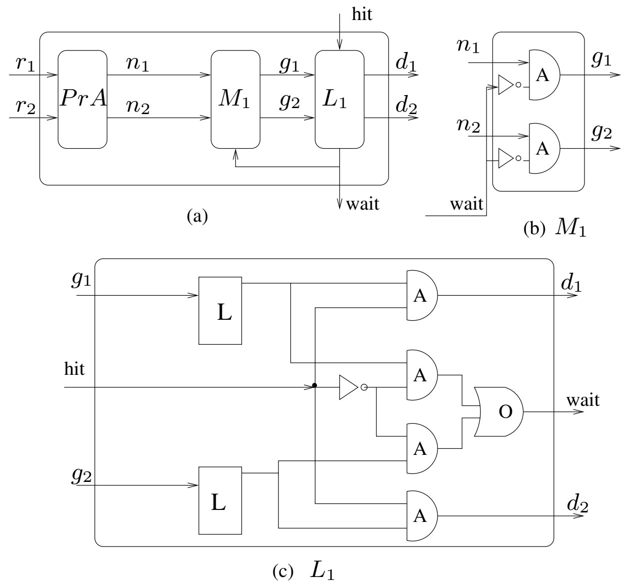


Figure 2. Memory Arbitration Logic(MAL)

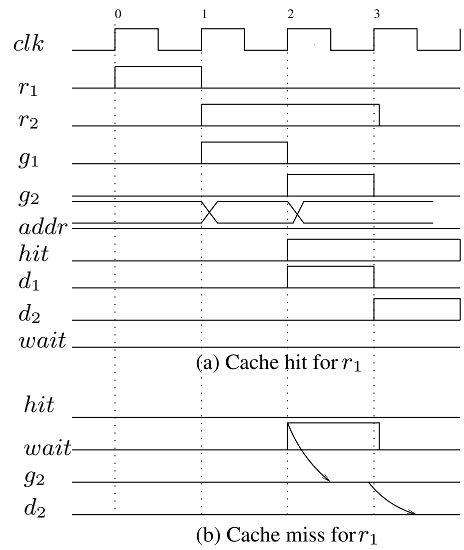


Figure 3. Timing Diagram

Example 1 Fig 2 shows the architecture of a simple *Memory Arbitration Logic*(MAL) in the presence of a cache. There are two request inputs, r_1 and r_2, for two independent on-chip requesting modules. The priority arbiter PrA arbitrates between r_1 and r_2 and asserts either n_1 or n_2 in the next cycle. The module L_1 is a cache access logic. The input, hit, to this logic indicates a cache hit. In case of a cache miss, L_1 asserts the wait signal which masks the arbitration decision through the logic M_1. The outputs d_1 and d_2 are inputs to the requesting devices respectively. When the page becomes available in the cache, d_1 or d_2 is asserted accordingly. In the figure 'A' represents an AND gate, 'O' an OR gate and 'L' a latch.

The architectural intent requires that r_1 has higher priority than r_2. This means that if r_1 comes before r_2 then it is never the case that r_2 has it's page available before r_1. This intent can be expressed by the following LTL property:

$$\mathcal{A} = G(\neg\, wait \wedge r_1 \wedge X(r_1\, U\, r_2) \rightarrow X(\neg d_2 U\, d_1))$$

Suppose we are unable to verify $\mathcal{A}$ on the whole design.[1] We must therefore refine the specification. Let us assume that we are given the RTL for M_1 and L_1 and the following properties for PrA.

$$R_1 = \mathrm{G}(r_1 \rightarrow X\, n_1) \qquad R_2 = \mathrm{G}(\neg r_1 \wedge r_2 \rightarrow X\, n_2)$$

Our primary coverage problem is to determine whether the architectural intent $\mathcal{A}$ is covered by the RTL modules and the properties of PrA. In this case the answer is positive. Consider the scenario as shown in the timing diagram in Fig 3. Here r_1 is asserted in time 0 and de-asserted in time 1. The input r_2 is asserted in time 1. Also consider the case where the $wait$ signal is initially low. Now n_1 will be asserted in time 1. Here there can be two different scenarios depending on whether there is a hit or miss. If hit occurs, d_1 will be asserted in the next cycle and hence the architectural intent is not violated. If there is a miss (as shown in the Fig 3(b)) then $wait$ will be high which would prevent g_2 to be asserted in time 2. The $wait$ signal would remain high until the data comes to the cache and hit is asserted which would assert d_1 in the same cycle, thus preventing $\mathcal{A}$ to be violated.

Formally our tool answers this primary coverage question by checking the truth of the property $NU = (R_1 \wedge R_2) \wedge \neg(A)$ in the model consisting of M_1 and L_1. The model checker returns a negative answer, and therefore the answer to the coverage question here is *positive.* $\Box$

3. Computing the Coverage Gap

In this section we address the more complex problem of computing and representing the coverage gap. One way

1 This is a toy example which is unlikely to run into capacity issues, but we use this assumption to demonstrate our approach in simple terms

to demonstrate that a coverage gap exists is to produce a counter-example run, that is, a run that satisfies the RTL specification but refutes the architectural intent. However, this only reflects a fraction of the coverage gap. On the other hand, our aim is to find the set of missing temporal properties in the RTL specification, which when included in the RTL specification closes the coverage gap.

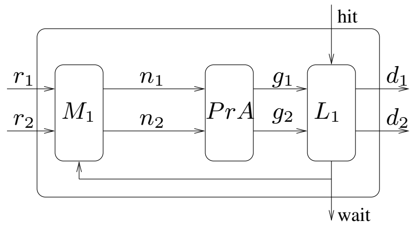


Figure 4. Mem Arbitration Logic(With GAP)

Example 2 Let us consider a slight variant of the MAL described in Ex 1 as shown in Fig 4. Now the request lines r_1 and r_2 are connected to M_1 and the outputs n_1 and n_2 of M_1 are used to drive PrA. The outputs of PrA is connected to the grant inputs g_1 and g_2 of L_1. The new RTL properties of PrA would be:

$$R'_1 = \mathrm{G}(n_1 \rightarrow X\ g_1) \qquad R'_2 = \mathrm{G}(\neg n_1 \wedge n_2 \rightarrow X\ g_2)$$

In this scenario the architectural property A is not covered by the RTL specification. For example whenever we have the scenario where r_1 is asserted for one cycle and r_2 asserted in the next cycle, and if there is a miss for r_1 but a hit for r_2, then d_2 will be asserted before d_1. Thus the architectural intent is not guaranteed by the RTL specification. Specifically the coverage gap lies only on those scenarios where the data for a later r_2 is in the cache while the data of a previous r_1 is not. In other words, the coverage gap can be accurately represented by the following property that considers exactly the above scenarios:

$$U = G(\neg wait \wedge r_1 \wedge X(r_1 U(r_2 \wedge X \neg hit)) \rightarrow X(\neg d_2 U d_1))$$

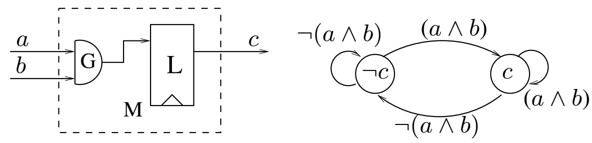
We have $(R_1 \wedge R_2 \wedge U) \wedge \neg(A)$ is false in L_1 and hence closes the coverage gap. In general, our aim will be to determine the *weakest* set of temporal properties that close the coverage gap between the RTL specification and the architectural intent. This intent is formally expressed below. $\Box$

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 $\mathcal{F}_1$. $\Box$

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 $(\mathcal{R} \wedge \mathcal{R}_H) \wedge \neg\mathcal{A}$ is false *in $\mathcal{M}$, 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 $(\mathcal{R} \wedge \mathcal{R}'_H) \wedge \mathcal{A}$ is also* false *in $\mathcal{M}$. In other words, we find the weakest property that suffices to close the coverage hole. Adding the weakest property strengthens the RTL specification in a minimal way.* $\Box$

In order to determine the coverage hole we generate the temporal formula which exactly represents a RTL model $\mathcal{M}$. We do this as follows: Given a RTL model $\mathcal{M}$ we extract the Finite State Machine (FSM) $\mathcal{S}_M$ modeling it. $\mathcal{S}_M$ is a 6 tuple $\langle I, O, S, S_0, L, T\rangle$ where, I is the set of inputs, O is the set of outputs, S is the set of states, S_0 is the initial state, $L(s)$ is a boolean function over the variables in s, where $s \in S$, $T(s, i, s')$ is an LTL property corresponding to the transition (s, i, s') in S_M. Specifically for a transition (s, i, s') from state s to s' on input i the transition property is $L(s) \wedge i \wedge X(L(s'))$. The transition function T is the collection of all these properties.

Definition 4 LTL formula T_M for FSM M
For a FSM M = $\langle I, O, S, S_0, L, T\rangle$ we define an LTL formula $T_M = L(S_0) \wedge G(\vee_{(\langle s,i,s'\rangle \in T)} L(s) \wedge i \wedge X(L(s')))$. T_M exactly represents all the runs which are present in M.



(a) Simple Model (b) The extracted FSM

Figure 5. A simple Model(Example 3)

The following example illustrates this construction.
Example 3 Consider the concrete module M as shown in Fig 5(a). M has 'a' and 'b' as inputs and 'c' as the output. Fig 5(b) shows the extracted FSM of this model. Let c' be the next state variable corresponding to the output variable c. The initial state of the model is $c = 0$, hence L(s_0) = $\neg c$. After minimization T_M will be as,

$$T_M = (\neg c) \wedge G(\neg c.a.b.c' \ \vee \neg c.\neg(a.b).\neg c' \\ \vee c.a.b.c' \ \vee \ c.\neg(a.b).\neg c')$$

It must be noted that the RTL model $\mathcal{M}$ may consist of multiple concurrent models say M_1,M_2,...,M_k. In such case we generate k temporal formulas T_{M_1},T_{M_2},...,T_{M_k} for each model. T_M is then generated by taking conjunction of all these k properties $\Box$.

The following theorem characterizes the coverage hole.

Theorem 2 The coverage hole in the RTL specification is unique and is given by $\mathcal{A} \vee \neg(\mathcal{R} \wedge T_M)$.
Proof: Let $\mathcal{R}_H = \mathcal{A} \vee \neg(\mathcal{R} \wedge T_M)$. It is easy to see that $((\mathcal{R} \wedge T_M) \wedge \mathcal{R}_H) \Rightarrow \mathcal{A}$, and therefore $\mathcal{R}_H$ closes the coverage hole.

Let $\mathcal{R}'_H$ be a property such that $\mathcal{R}'_H$ is weaker than $\mathcal{R}_H$ and $(\mathcal{R} \wedge \mathcal{R}'_H \wedge T_M) \Rightarrow \mathcal{A}$. Since $\mathcal{R}'_H \not\Rightarrow \mathcal{R}_H$, there exists a run, π, that satisfies $\mathcal{R}'_H$ but not $\mathcal{R}_H$.

Suppose π satisfies $\mathcal{R} \wedge T_M$. Then by the definition of $\mathcal{R}'_H$, π satisfies $\mathcal{A}$. But if π satisfies $\mathcal{A}$, then π must satisfy $\mathcal{R}_H$ (by the definition of $\mathcal{R}_H$). This is a contradiction.

Otherwise, suppose π does not satisfy $\mathcal{R} \wedge T_M$. Therefore π satisfies $\neg(\mathcal{R} \wedge T_M)$, and again π must satisfy $\mathcal{R}_H$ (by the definition of $\mathcal{R}_H$). Again we have a contradiction. Therefore $\mathcal{R}_H$ is the unique weakest property that closes the coverage gap. $\square$

We now consider the problem of computing the *uncovered architectural intent* as defined below.

Definition 5 [Uncovered architectural intent:] *An uncovered architectural intent is a property $\mathcal{A}_H$ over $\mathcal{AP}_\mathcal{A}$, such that $(\mathcal{R} \wedge T_M \wedge \mathcal{A}_H) \Rightarrow \mathcal{A}$, and there exists no property $\mathcal{A}'_H$ over $\mathcal{AP}_\mathcal{A}$ such that $\mathcal{A}_H \Rightarrow \mathcal{A}'_H$ and $(\mathcal{R} \wedge T_M \wedge \mathcal{A}'_H) \Rightarrow \mathcal{A}$. In other words, we find the weakest property over $\mathcal{AP}_\mathcal{A}$ that closes the coverage hole.* $\square$

4. Representing the Coverage Hole

Theorem 2 gives us a formalism for computing the coverage hole, but does not convey the missing properties in a meaningful way. Our aim is to present the coverage hole and the uncovered architectural intent to the designer in a form that is syntactically close to the architectural intent and is thereby amenable to visual comparison with the architectural intent. The following example highlights this intent.

Example 4 We consider the coverage of A by R'_1,R'_2 and the concrete modules M_1 and L_1 as given in Ex 2. By Theorem 2, the coverage gap between A and R'_1,R'_2,M_1 and L_1 is given by the property:

$$\varphi = A \vee \neg(R'_1 \wedge R'_2 \wedge T_{M_1} \wedge T_{L_1})$$

which does not convey a meaningful information to the designer. On the other hand, consider the property U of Ex 2:

$$U = G(\neg wait \wedge r_1 \wedge X(r_1 U(r_2 \wedge X \neg hit)) \rightarrow X(\neg d_2 U d_1))$$

U is stronger than φ, but represent the coverage gap more effectively than φ because, the designer can visually compare U with A and see what remains to be covered. $\square$

Our tool is based on two key algorithms. The first algorithm computes the bounded terms in the coverage gap and then *pushes* them into the syntactic structure of the architectural properties to obtain the uncovered part. The second algorithm takes architectural properties having unbounded temporal operators and systematically weakens them into structure preserving decompositions and checks the components that remains to be covered.

4.1. Coverage Algorithm

The core idea behind our algorithm is to present a structure preserving form of the coverage gap. Our algorithm takes each formula $\mathcal{F}_\mathcal{A}$ from the architectural intent $\mathcal{A}$ and finds the coverage gap, $\mathcal{G}$, for $\mathcal{F}_\mathcal{A}$, with respect to the RTL properties $\mathcal{R}$ and the concrete Models $\mathcal{M}$. Since $\mathcal{R}$ and $\mathcal{M}$ are required to cover every property in $\mathcal{A}$, we use this natural decomposition of the problem. The algorithm below implements this idea. Here we use $\mathcal{U}$ to represent the RTL coverage hole $\mathcal{R}_H$ and $\mathcal{M}$ to represent the concrete module in the RTL specification.

Algorithm 1 Coverage Algorithm

Find_Coverage_Gap($\mathcal{F}_\mathcal{A}$,$\mathcal{R}$,$\mathcal{M}$)

1. Generate T_M from the concrete module $\mathcal{M}$ and compute $\mathcal{U} = \mathcal{F}_\mathcal{A} \vee \neg(\mathcal{R} \wedge T_M)$
2. If $\neg(\mathcal{U})$ is not false in $\mathcal{M}$ then
   (a) Unfold $\mathcal{U}$ to create a set of uncovered terms, $\mathcal{U}_M$, that approximates the coverage gap;
   (b) Use universal quantification to eliminate signals belonging to $\mathcal{AP}_\mathcal{R} - \mathcal{AP}_\mathcal{A}$,
   (c) Push the terms of $\mathcal{U}_M$ into $\mathcal{F}_\mathcal{A}$ to obtain $\mathcal{F}_\mathcal{U}$.
   (d) Weaken $\mathcal{F}_\mathcal{U}$ to obtain the final uncovered formula $\mathcal{G}$.
3. Return $\mathcal{G}$;

The details of Algorithm 1 were presented in [3]. Here we explain it's operation with the help of the design in Ex 2.

In the design described in Ex 2, $\mathcal{A} = G(\neg wait \wedge r_1 \wedge X(r_1\ U\ r_2) \rightarrow X(\neg d_2 U\ d_1))$, R'_1 and R'_2 are the RTL properties of PrA. L_1 and M_1 constitute the concrete modules M. The first step of the algorithm generates the temporal properties T_{L1} and T_{M1} corresponding to L_1 and M_1 respectively. M_1 is a combinational block and thus T_{M1} is generated by nesting a global operator G above the Boolean function it implements.

$$T_{L1} = G((r_1 \wedge wait \leftrightarrow g_1) \wedge (r_2 \wedge wait \leftrightarrow g_2))$$

For generating T_{M1} our algorithm first generates the FSM for M_1 and then generates T_{M1} from it.

$$T_{M1} = (\neg g_1 \wedge \neg g_2 \wedge \neg wait) \wedge G[(g_1 \wedge hit' \wedge d'_1) \vee (g_1 \wedge hit' \wedge d'_2) \vee (\neg(g_1 \wedge hit') \wedge \neg d'_1) \vee (\neg(g_1 \wedge hit') \wedge \neg d'_1) \vee (g_1 \wedge \neg hit' \wedge wait') \vee (g_1 \wedge \neg hit' \wedge wait') \vee (\neg(g_2 \wedge \neg hit') \wedge \neg d'_1) \vee (\neg(g_2 \wedge \neg hit') \wedge \neg d'_1)]$$

The first step of Algorithm 1 generates $\mathcal{U} = A \wedge R \wedge T_M$ where $T_M = T_{M1} \wedge T_{L1}$. Since $\neg\mathcal{U}$ is false in M, in steps 2(a) $\mathcal{U}$ is unfolded upto it's fixpoint [2]. After unfolding and abstracting out the local RTL variable d we obtain $\mathcal{U}_\mathcal{M}$ as follows:

$$\mathcal{U}_M = \{\neg r_1 \wedge X r_2 \wedge XX\neg hit \wedge X d_1, \neg r_1 \wedge X r_2 \wedge XX\neg hit \wedge X\neg d_2 \wedge XX d_1\}$$

The distribution of the above terms into the parse tree of A is done in the step 2(c) of the algorithm as shown in Fig 6. This step determines that the gaps lie inside the unbounded operator until(U).

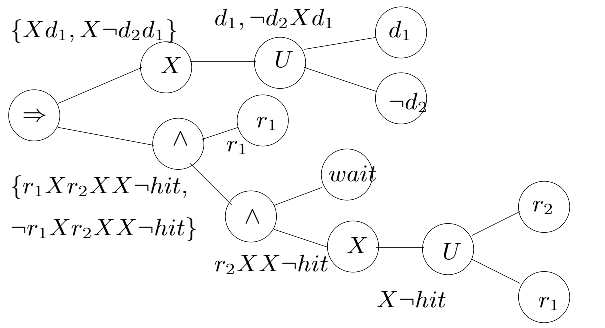


Figure 6. Pushing the terms in $\mathcal{U}_M$

The step 2(d) of Algorithm 1 uses heuristics to decompose the property into weaker fragments and then return those fragments that are not covered by the RTL specification. This step is useful when the coverage gap lies in properties having the unbounded temporal operators, like G, F and U. We explain the method with the following property:

φ: $G(\,(\,a\,U\,b\,)\,\Rightarrow(\,c\,U\,d\,)\,)$

Suppose we want to weaken the property by augmenting a new literal $\neg e$ with the variable instance c. The choice of the 'e' is guided by the variable which reaches the temporal operator during the execution of step 2(c). Here we have to weaken the variable instance c for weakening of φ. So we need to replace the variable instance with the disjunction of the variable and the new literal. The resulting weakened property may be any one of the following:

φ': $G(\,(\,a\,U\,b\,)\,\Rightarrow(\,(c\,\vee\,\neg e)\,U\,d\,)\,)$
φ'': $G(\,(\,a\,U\,b\,)\,\Rightarrow(\,(c\,\vee\,e)\,U\,d\,)\,)$

Here $\varphi = \varphi' \wedge \varphi''$ and it may be the case that RTL covers φ' but not φ'' in which case we report φ'' as the coverage gap.

Returning to our example the until operator to the left of the implication operator is weakened using $X\neg hit$ and we obtained the following gap property:

$$U = G(\neg wait \wedge r_1 \wedge X(r_1 U(r_2 \wedge X\neg hit)) \rightarrow X(\neg d_2 U d_1))$$

U closed the the gap between the Aspec and the RTL.

5. Results on SpecMatcher

SpecMatcher is our tool for verifying design intent coverage. The original tool used to accept only LTL specifications. With the new methods, the tool now also accepts RTL modules. Table 1 shows the runtimes of our tool on several designs. For each design, we selected an architectural property which requires contributions from multiple submodules. For example, ARM AMBA AHB is bus protocol involving master, slave and arbiter devices. The exact arbitration policy is not defined in the protocol, we therefore targeted a system level property with the RTL of the arbiter and set of properties over the master and slave. The tool accepted all 29 RTL properties and the RTL of the arbiter and produced the coverage gap in less than a minute.

Circuit	No. of RTL properties	Time (sec)		
		Primary Coverage Question	T_M building Time	Gap Finding Time
Memory Arb. Logic	26	4.7	2.3	26.1
Intel Design	12	8.2	0.9	15.2
ARM AMBA AHB	29	12.07	9.8	22.5
Paper Ex. (Fig 1)	2	0.18	0.06	1.2

Table 1. Runtimes of SpecMatcher

The first two designs have non-trivial complexity as indicated by the number of properties. The last design is the toy example demonstrated in this paper. The time break-ups show the time spent (on a 2GHz P4) by the tool in each of the major steps of the coverage algorithm.

If we bring in larger RTL blocks into the picture, we will have state explosion in two of the steps. Firstly, the primary coverage question requires model checking on the RTL blocks. Secondly, the building time for T_M will go up. Therefore the proposed method should not be viewed as a new way to do model checking. The value of the original methodology lies in providing the validation engineer with a formal proof that the decomposition of the specification is correct, or with a clear representation of the coverage gap. The new methodology aids the process by allowing simple modules (such as glue logic) to be accepted as is, but is still totally inclined towards the original goal.

Acknowledgments

Pallab Dasgupta and P.P. Chakrabarti acknowledge the Department of Science & Technology, Govt. of India for the partial support of this work.

References

[1] *ARM AMBA Specification Rev 2.0*, http://www.arm.com

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

[3] Das, S., Basu, P., Banerjee, A., Dasgupta, P., Chakrabarti, P.P., Mohan, C.R., Fix, L., Armoni, R., Formal Verification Coverage: Computing the Coverage Gap between Temporal Specifications, In *Proc. of ICCAD*, San Jose, 198-203, 2004.