Design Intent Coverage Revisited

ARNAB SINHA, PALLAB DASGUPTA, BHASKAR PAL, SAYANTAN DAS,
PRASENJIT BASU, and P.P. CHAKRABARTI
Indian Institute of Technology, Kharagpur

Design intent coverage is a formal methodology for analyzing the gap between a formal architectural specification of a design and the formal functional specifications of the component RTL blocks of the design. In this article we extend the design intent coverage methodology to hybrid specifications containing both state-machines and formal properties. We demonstrate the benefits of this extension in two domains of considerable recent interest, namely (a) the use of auxiliary state-machines in formal specifications, and (b) the use of modest sized RTL blocks in the design intent coverage analysis.

Categories and Subject Descriptors: category [**Integrated Circuits**]: Design Aids—*Verification*

General Terms: Verification

Additional Key Words and Phrases: Design Intent Coverage

ACM Reference Format:

Sinha, A., Dasgupta, P., Pal, B., Das, S., Basu, P., and Chakrabarti, P. P. 2009. Design intent coverage revisited. ACM Trans. Des. Autom. Electron. Syst., 14, 1, Article 9 (January 2009), 32 pages, DOI = 10.1145/1455229.1455238 http://doi.acm.org/10.1145/1455229.1455238

1. INTRODUCTION

The design of complex digital circuits is typically done top-down. The design cycle begins with the development of the microarchitectural specifications, which captures the design's architectural intent. The design team systematically attempts to implement the design intent by choosing a set of functional blocks that must work together to achieve the design intent. The larger blocks are hierarchically implemented in terms of smaller blocks and so on. Each step of this cycle refines the design intent, since the implementation typically has more details than the specification from which it is built. In each step, either the components are chosen from existing legacy designs, or the required component specifications (in English) are developed. Design entry (RTL) for a new

Author's address: B. Pal, Indian Institute of Technology, Kharagpur, India 731302; email: bhaskarpal@gmail.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1084-4309/2009/01-ART9 $5.00 DOI 10.1145/1455229.1455238 http://doi.acm.org/10.1145/1455229.1455238

component starts when the component specification is simple enough to be implemented as a unit module.

Verification, on the other hand, starts bottom-up. The unit modules are verified against the unit-level specification, before they are integrated into respective blocks. The blocks are then verified against the block-level specifications. This process continues until all the components are integrated into the main design and only then we are able to verify the integrated design against the architectural specifications. Finding design intent violations so late in the design flow can be very expensive. Moreover, there is a serious gap in available verification technology at this level: formal tools do not scale to this level, and achieving meaningful simulation coverage over the integrated design is rapidly becoming infeasible in practice.

In a previous work [Basu et al. 2006; Dasgupta 2006] we have shown that if key properties of the architectural intent are formally captured early in the design flow, then they can be formally carried through the top-down design flow. The proposed methodology is shown in Figure 1. The design team works with the verification team while developing the component block specifications. While the design team expects that the component block specifications developed by them are explicit enough to capture the design's architectural intent, the verification team formally checks the validity of this expectation. This is done by selectively defining formal properties for the component blocks and then checking whether we can prove the architectural properties of the integrated design from the properties of the components and the way they are interconnected (that is, the top-level schematic). The development of component properties is manual, but the proof methodology is automatic. When the blocks are small enough, the block properties can be formally checked on the RTL for the blocks using standard model checking tools.

There are at least three benefits in this approach, namely:

(1) We compose the formal specifications of the components while proving a global property over the integrated design. This is significantly more scalable than composing the components themselves, since the properties are much smaller than the state machines representing the components.
(2) We can perform the verification early in the design flow, even before the components are coded. Logical errors due to incorrect decomposition of the design intent can be detected early. Component specifications will be more accurate and correct-by-construction.
(3) The acceptability of third-party design modules can be verified early if they come with known properties. We can use the known properties along with the properties of other components to check whether the design intent properties are satisfied.

The key component of the new approach, which has been presented in Basu et al. [2006], is a method for proving the global properties of a design using the properties of its components and the way they are interconnected. This is not an equivalence checking problem, since the components properties will typically have more functional details and more variables than the global properties.

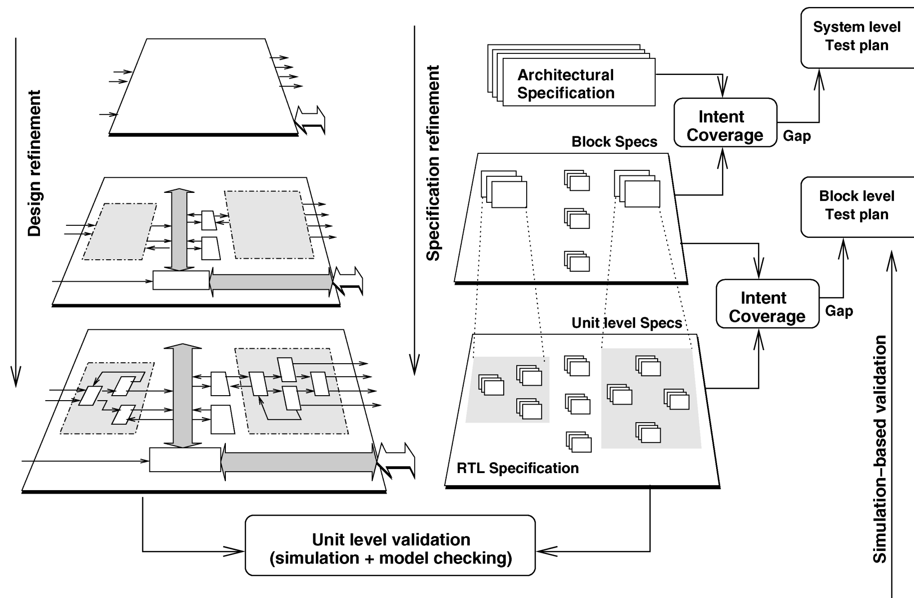

Fig. 1. Specification refinement. (Figure source: *A Roadmap for Formal Property Verification*, Pallab Dasgupta, Springer, 2006).

This is a coverage problem, where the component specifications together should catch all bugs that are caught by the global specification. Formally, any trace that is invalid with respect to the global properties should refute one or more of the component properties. The invalid traces of the component specifications should *cover* (or be a superset of) the invalid traces of the global properties. We refer to this coverage as *design intent coverage*.

This is a totally different approach for coverage analysis of formal specifications as compared to the existing body of literature on formal coverage analysis, which primarily test the completeness of specifications by mutating a given implementation and then testing whether the properties fail as a consequence of the mutation [Chockler et al. 2001; Chockler et al. 2001; Chockler et al. 2003; Hoskote et al. 1999; Katz et al. 1999]. A relatively recent survey on formal coverage analysis can be found in Chapter 5 of Dasgupta [2006].

Though this coverage is always theoretically possible (since the components actually constitute the design implementation), it is not practical to assume that the verification team will always be able to achieve this closure by developing the formal specifications of the components. Therefore, we also need a methodology for computing the coverage gap between the global properties and the collection of component properties. The coverage gap is represented by another property that captures exactly those scenarios under which a bug detected by the global specification is not caught by the component properties. Finding this gap is important, since verification should target these scenarios specifically when the integrated design is available.

The core algorithm presented in Basu et al. [2006] compares two specifications, each consisting of a set of temporal logic properties. The first specification,

$\mathcal{A}$, consists of the global properties of the integrated design—we shall refer to this specification as the *architectural specification*. The second specification, $\mathcal{R}$, consists of the collection of local properties of the components—we shall refer to this as the *RTL specification*. This naming convention is only to indicate that $\mathcal{A}$ is a higher level specification as compared to $\mathcal{R}$. We assume that each signal in $\mathcal{A}$ is also present in $\mathcal{R}$, but not vice versa. The algorithm finds a property $\mathcal{P}$ that represents the gap between $\mathcal{A}$ and $\mathcal{R}$, that is, $\mathcal{P}$ fails in all those traces where $\mathcal{A}$ fails but $\mathcal{R}$ passes.

Design intent coverage [Basu et al. 2006] and compositional model checking [Sistla and Clarke 1985] can be seen as two extremes of a component based design verification methodology. In the traditional model checking problem, the components are given in RTL, and the goal is to check whether the components together imply (or model) the architectural specification. The gap between these represent the counter-examples. In the design intent coverage problem, the components are given as specifications, and the goal is to check whether the component specifications together imply the architectural specification. The design intent coverage approach has computational advantage, since model checking methods typically run into capacity issues while attempting to compose the state machines of the components.

In this article we explore the space in between these two extremes. Specifically, we examine those cases where we are given formal properties for some of the components and the RTL for the rest of the components. This enables us to handle glue logic between modules (which may be modeled as small RTL blocks), and extends our approach to cases where legacy designs are used for some of the components (for which no properties are available). A preliminary version of this work was presented in Das et al. [2006].

In this article, we extend the design intent coverage methodology [Basu et al. 2006] to specifications that consist of auxiliary state machines annotated with formal properties. This style of development of formal specifications has received considerable recent attention for several reasons as explained in Section 5.

The theoretical basis for both the extensions presented in this article is similar in the sense that in both problems we compare specifications consisting of formal properties and state machines. Moreover, the problems of model checking and design intent coverage are fundamentally similar when properties are specified in Linear Temporal Logic (LTL) [Pnueli 1977], because LTL satisfiability (which is the formal basis for design intent coverage algorithms) coincides with LTL model checking [Sistla and Clarke 1985]. In Section 2 we demonstrate this similarity and methods for reducing one to the other—this reduction forms the foundation of the methods presented in this article.

The article is organized as follows. Section 2 presents the relation between LTL satisfiability and LTL model checking—these results are not new, but they develop the necessary background for the methods presented in this article. Section 3 presents our original algorithms for design intent coverage. Section 4 introduces the role of auxiliary state machines (ASM) in specification development and then extends the design intent coverage methodology to specifications containing ASMs. Section 5 extends the design intent coverage methodology to

```
module gray_counter (reset, x1, x2, clk);
input reset, clk;
output x1, x2;

always@(posedge clk)
begin
  if(!reset) begin
    x1 <= x2;
    x2 <= ~x1;
  end
  else begin
    x1 <= 1'b0;
    x2 <= 1'b0;
  end
end

endmodule;
```

Fig. 2. Verilog implementation of a 2-bit gray counter.

cases where we are given formal properties for some of the components and the RTL for the rest.

2. LTL: SATISfiABILITY VERSUS MODEL CHECKING

The formal methods presented in this article are developed for specifications in *Linear Temporal Logic* (LTL) [Pnueli 1977], and will also work for language standards such as PSL [PSL] and SVA [SVA], which are based on LTL. For the sake of completeness, the syntax and semantics of LTL is presented in Appendix A.

This section demonstrates the fundamental similarities between LTL model checking and LTL satisfiability. This relation is germane to the problems addressed in the other sections.

Example 1. Consider a two bit Gray counter. A gray counter has the following property: *the next value of the counter differs from the present value of the counter in exactly one bit*. Let us represent the two bits as x_1 and x_2. We consider a high active input `reset`, which resets the counter in the following cycle, that is:

$$G(\texttt{ reset} \Rightarrow X(\neg x_1 \wedge \neg x_2)).$$

Suppose, we wish to verify the following property: *if the counter is not reset, then the next value of the counter differs from the present value by exactly one bit*. In LTL this property may be expressed as follows:

$$\varphi : \qquad \mathrm{G}(\neg\texttt{reset} \Rightarrow (x_1 \oplus X x_1) \oplus (x_2 \oplus X x_2)).$$

Figure 2 shows an implementation of the counter (in Verilog). We wish to verify whether this implementation satisfies φ. The traditional approach for LTL model checking checks whether any accepting run for $\neg\varphi$ belongs to the implementation. This check is done by extracting a state-machine $\mathcal{T}$ representing

the implementation and verifying whether the product of $\mathcal{T}$ with any acceptor of $\neg\varphi$ is empty.

Any state machine can be represented by a collection of Boolean state transition functions. In other words for each state bit s_i, we have a function, f_i that accepts the present values of the state bits and inputs, and returns the value of s_i in the next cycle. It is easy to express these transition functions in LTL as:

$$G(X s_i \;\Leftrightarrow\; f_i(\text{present-state bits, present-input bits})).$$

For example, the transition relation, $\mathcal{T}$, corresponding to the Gray counter implementation of Figure 2 can be expressed in LTL as follows:

$$P_1 \;:\; G\,((\neg\texttt{reset} \;\wedge\; x_2) \;\Leftrightarrow\; X(x_1)) \tag{1}$$

$$P_2 \;:\; G\,((\neg\texttt{reset} \;\wedge\; \neg x_1) \;\Leftrightarrow\; X(x_2)) \tag{2}$$

All formal verification tools extract the transition systems from each component prior to model checking. The scalability problem is not in this extraction, but in computing the product of the transition systems. Therefore the tasks of extracting the transition system for individual components of modest size and then translating the component transition systems into LTL are straight forward for formal property verification tools. Model checking tools will not do this normally, but this translation will be an important link in the scheme of things when we deal with design intent coverage problems involving state machines.

There can be two approaches of verifying the property φ.

(1) Model-check the property φ on the model extracted from the 2-bit gray counter module.

(2) The other approach is to check the validity of the following LTL property:

$$(P_1 \wedge P_2) \Rightarrow \varphi.$$

Obviously a model checking tool will not take the second approach, but it is interesting to see that the second approach is fundamentally equivalent to the first. The second approach is equivalent to checking whether the logical representation of $\mathcal{T}$ implies φ, that is, we check whether $\mathcal{T} \Rightarrow \varphi$ is valid. This follows from the fact that $\mathcal{T} \Rightarrow \varphi$ is valid iff $\mathcal{T} \wedge \neg\varphi$ is unsatisfiable, which is exactly what model checking techniques verify.

The previous example indicates that for LTL, the model checking problem and the satisfiability problem are fundamentally similar. Not surprisingly, the computational complexity of both problems are the same (both are PSPACE-complete) [Sistla and Clarke 1985]. Therefore, theoretically, we can always transform the LTL model checking problem into a LTL satisfiability problem. We do not explicitly take this approach in practice, because verification tools perform a lot of reductions (pruning) on the state-machine before the actual model checking step, and these reductions are easier to apply on state-machines rather than on formal properties.

In this article, we use the notion of transforming state-machines into LTL properties to unify the problems of design intent coverage and model checking, and solve the unified problem using LTL satisfiability. Moreover, we define new

algorithms for refining the specification under such transformations to reflect the coverage gap between the two specifications in a readable way.

3. BACKGROUND

This section presents a summary of our previous work on design intent coverage for the sake of completeness of this article.[1] The original theorems are stated without the proofs.

In our original problem formulation, design intent coverage essentially compares a high level specification, $\mathcal{A}$, with a low level specification, $\mathcal{R}$, and determines whether $\mathcal{R}$ covers $\mathcal{A}$, that is, whether every invalid scenario for $\mathcal{A}$ is also an invalid scenario for $\mathcal{R}$, so that no bug detected by $\mathcal{A}$ is missed by $\mathcal{R}$.

To distinguish between the high level specification, $\mathcal{A}$, and the low level specification, $\mathcal{R}$, we shall refer to $\mathcal{A}$ as the *architectural intent*, and $\mathcal{R}$ as the *RTL specification*. The inputs to the original design intent coverage methodology 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.

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.

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

This assumption essentially means that the low level specification has the same names for their signals as the corresponding ones in the high level specification. 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.

Given a specification, we define a *state* as a valuation of the signals used in the specification. A *run* is an infinite sequence of states.

Definition 1. (*Coverage Definition*)
The RTL specification covers the architectural intent if and only if there exists no run that refutes one or more properties of the architectural intent but does not refute any property of the RTL specification.

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

[1] Parts of this section have been abstracted from the book: *A Roadmap for Formal Property Verification* by Pallab Dasgupta, Springer 2006.

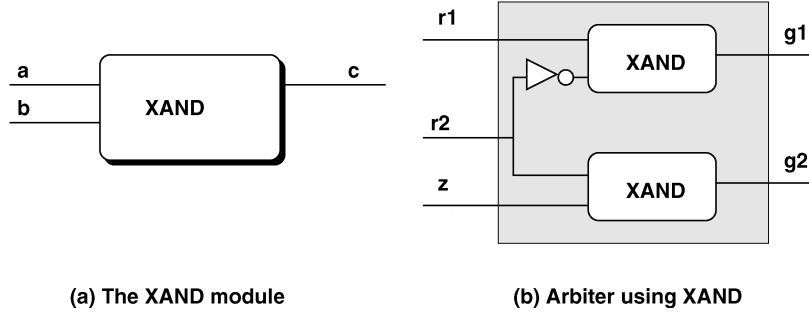

Fig. 3. A toy arbiter.

properties together with the RTL specification succeed in covering the architectural intent).

The following theorem shows us a way to answer the first question.

THEOREM 1. *The RTL specification, $\mathcal{R}$, covers the architectural intent $\mathcal{A}$,* iff *the temporal property $\mathcal{R} \Rightarrow \mathcal{A}$ is valid.*

Example 2. 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).
This architectural intent may be expressed in LTL as follows:

A_1: $G\ F(\neg r_2)$
A_2: $G((r_2 \wedge z) \Rightarrow X(g_2\ U\ \neg r_2))$
A_3: $G((\neg r_2) \Rightarrow X\ g_1)$.

Let us now consider an implementation of the arbiter using a component called XAND, as shown in Figure 3. The specification of the module XAND is as follows:

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

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 Figure 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.

Our 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. It is clear that A_1 is implied by the RTL specification, but we can see that neither A_2 nor A_3 is covered by the properties in the RTL specification.

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. This shows that A_3 is not covered.

Also, consider those scenarios where r_2 and z are asserted together, but z de-asserts before r_2 (that is, the slave becomes unavailable before the transfer completes). In these scenarios, the architectural intent requires g_2 to remain high as long as r_2 remains high (property A_2), but the RTL specification does not guarantee this.

It is not hard to compute the coverage gap between two temporal specifications and specify a property that theoretically represents the coverage gap. The main challenge is in presenting the new property in a form that is syntactically similar and visually comparable with the original specification, so that the validation engineer is able to visually examine the new property and realize the set of architectural behaviors that have not been covered by the RTL specification. Let us first see how the coverage gap can be computed.

Example 3. Let us consider the coverage of the property, A_3 of Example 2 by the RTL specification. We have already established that A_3 is not covered. However this information does not accurately point out the coverage gap between A_3 and the RTL specification. Specifically, the coverage gap lies only for those scenarios where r_1 and r_2 are simultaneously low at some point of time. In other words, the coverage gap can be accurately represented by the following property that considers exactly the above scenarios:

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

We have $R_2 \wedge U_1 \Rightarrow A_3$, and therefore U_1 closes the coverage gap between R_2 and A_3. In general, our aim is 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 as follows.

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 $(\mathcal{R} \wedge \mathcal{R}_H) \Rightarrow \mathcal{A}$ is valid, 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 $(\mathcal{R} \wedge \mathcal{R}'_H) \Rightarrow \mathcal{A}$ is valid. 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.

Since $\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$, each property of the architectural intent is a valid property over $\mathcal{AP}_{\mathcal{R}}$. The following theorem characterizes the coverage hole.

THEOREM 2. *The coverage hole in the RTL specification is unique and is given by* $\mathcal{A} \vee \neg\mathcal{R}$.

There is an intuitive explanation of the coverage hole as defined by Theorem 2. The goal of the design intent coverage analysis is to find those behaviors that refute $\mathcal{A}$ but satisfy $\mathcal{R}$, that is, those behaviors that satisfy:

$$\varphi \;=\; \neg\mathcal{A} \;\wedge\; \mathcal{R}.$$

The property representing the coverage hole must reject exactly these behaviors, hence the property is $\mathcal{A} \vee \neg\mathcal{R}$ which is $\neg\varphi$. The following example demonstrates the notion of a *coverage hole* in our formulation.

Example 4. Let us again consider the arbiter of Example 2. We had seen that the coverage gap lies in A_2 and A_3. By Theorem 2 we have the coverage hole in the RTL specification as:

$$R_H: \quad ((A_2 \wedge A_3) \vee \neg(R_1 \wedge R_2 \wedge R_3)).$$

We can also write the same coverage hole as the conjunction of the following two properties:

$$R_H{}': \quad (A_2 \vee \neg(R_1 \wedge R_2 \wedge R_3))$$
$$R_H{}'': \quad (A_3 \vee \neg(R_1 \wedge R_2 \wedge R_3)).$$

In other words, we can examine the coverage of each architectural property separately and produce a set of properties representing the coverage hole.

Typically, the coverage hole, $\mathcal{A} \vee \neg\mathcal{R}$, will contain signals belonging to $\mathcal{AP}_{\mathcal{R}} - \mathcal{AP}_{\mathcal{A}}$. To demonstrate the part of the architectural intent that is not covered by the RTL specification, we need a further level of abstraction. The definition of the *uncovered architectural intent* is as follows.

Definition 4. (*Uncovered architectural intent*). The uncovered architectural intent is a property $\mathcal{A}_H$ over $\mathcal{AP}_{\mathcal{A}}$, such that $(\mathcal{R} \wedge \mathcal{A}_H) \Rightarrow \mathcal{A}$ is valid, and there exists no property $\mathcal{A}'_H$ over $\mathcal{AP}_{\mathcal{A}}$ such that $\mathcal{A}'_H$ is weaker than $\mathcal{A}_H$ and $(\mathcal{R} \wedge \mathcal{A}'_H) \Rightarrow \mathcal{A}$ is valid. In other words, we find the weakest property over $\mathcal{AP}_{\mathcal{A}}$ that suffices to close the coverage hole.

Theorem 2 gives us a formalism for computing the coverage hole, but does not present the missing properties in a meaningful way. Our aim is to present the coverage hole and the uncovered architectural intent before 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 expressibility of the logic used for specification does not always permit a succinct representation of the coverage hole. In such cases, we prefer to present the coverage hole as a succinct set of properties that closes the coverage gap,

but may be marginally stronger than the actual coverage gap. The following example highlights this intent.

Example 5. We consider the coverage of A_3 by R_2 in the specifications given in Example 2. By Theorem 2, the coverage gap between A_3 and R_2 is given by the property:

$$\varphi = A_3 \vee \neg R_2.$$

The knowledge that this property is satisfiable does not convey any meaningful information to the designer. On the other hand, consider the property U_1 of Example 3:

$$U_1: \quad G((\neg r_1 \wedge \neg r_2) \Rightarrow X\ g_1).$$

U_1 is stronger than φ, but is able to represent the coverage gap more effectively than φ. This is because, the designer can visually compare U_1 with A_3 and see what remains to be covered.

It is also important to be able to preserve structural similarity with the architectural intent when we present the coverage hole. For example, the property U_1 can also be written as:

$$G(r_1 \vee r_2 \vee X\ g_1)$$

or as:

$$G((\neg X\ g_1 \wedge \neg r_1) \Rightarrow r_2).$$

These representations are logically equivalent to U_1, but are not visually similar to A_3. Preserving structural similarity is a very important issue in presenting the gaps between formal property specifications.

Our original design intent coverage algorithm preserves syntactic similarity with the architectural properties by extracting terms from the coverage gap and then *pushing* these terms into the syntactic structure of the architectural properties to obtain the uncovered part.

3.1 The Intent Coverage Algorithm

The intent coverage 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 specification $\mathcal{R}$.

The first step determines the coverage gap formula $\mathcal{U}$ in terms of RTL variables. If $\mathcal{U}$ is valid then $\mathcal{F}_\mathcal{A}$ is covered. Otherwise, we need to find an abstraction of $\mathcal{U}$ over the architectural variables that is syntactically close to $\mathcal{F}_\mathcal{A}$. The second step of the algorithm performs this task.

In Step 2(a), we recursively unfold $\mathcal{U}$ and generate a disjunction of terms, $\mathcal{U}_M$, that contain only Boolean subformulas and Boolean subformulas guarded by a finite number of X (next) operators. We guarantee that $\mathcal{U}_M$ is as strong as the coverage gap, $\mathcal{U}$. By this approximation, we eliminate all unbounded temporal operators from $\mathcal{U}$, which helps us to push the terms in $\mathcal{U}_M$ into the syntactic structure of $\mathcal{F}_\mathcal{A}$. Before we describe the unfolding step we present

Algorithm 1. Find_Coverage_Gap($\mathcal{F}_{\mathcal{A}}$,$\mathcal{R}$)

(1) Compute $\mathcal{U} = \mathcal{F}_{\mathcal{A}} \vee \neg\mathcal{R}$
(2) If $\mathcal{U}$ is not valid 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) $\mathcal{F}_{\mathcal{U}}$ = Call Push_Term($\mathcal{F}_{\mathcal{A}}$, $\mathcal{U}_M$, 1);
(3) Return $\mathcal{F}_{\mathcal{U}}$;

the definitions of an *X-pushed formula*, *X-guarded formula* and *X-depth of an operator* within a formula.

Definition 5. (*X-pushed formula*). 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.

Definition 6. (*X-guarded formula*). 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.

Definition 7. (*X-depth of an operator*). 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.

Example 6. Let us consider the temporal property:

$$\mathcal{P} \;=\; ((X\;\; p)\, U\; (X\;\; X\;\; q)) \;\wedge\; (X\;\; F\;\; r).$$

The X-pushed form of $\mathcal{P}$ is:

$$\mathcal{P}_X \;=\; X\; ((p\; U\; (X\;\; q)) \;\wedge\; (F\; r)).$$

Now $\mathcal{P}$ is an X-guarded formula because the corresponding X-pushed formula $\mathcal{P}_X$ starts with an X operator whose scope covers the whole formula. $\mathcal{P}$ contains two unbounded temporal operators, U and F. The X-depth of both U and F is 1.

Our methodology for decomposing $\mathcal{U}$ into a disjunction $\mathcal{U}_M$ of terms is as follows. It is known that any LTL property can be recursively unfolded over time steps to create an equivalent property over Boolean formulas and X-guarded LTL formulas. For example, the property $p\; U\; q$ may be rewritten as:

$$q \;\vee\; [p \;\wedge\; X(p\; U\; q)]$$

after one level of unfolding, and as:

$$q \;\vee\; [p \;\wedge\; X(q \;\vee\; (p \;\wedge\; X(p\; U\; q)))]$$

after two levels of unfolding. After k-level unfolding, we can distribute the X operators over the Boolean operators and the $\wedge$ operator over the $\vee$ operator to obtain a disjunction of terms, where each term consists of a conjunction of

Boolean literals, X-guarded Boolean literals, and X-guarded temporal formulas. For example, $p\ U\ q$ can be rewritten as follows after two levels of unfolding:

$$(q) \vee (p \wedge X\ q) \vee (p \wedge X\ p \wedge X\ X(p\ U\ q)).$$

Since a temporal formula has a finite number of members in its closure [Clarke E. M., Grumberg O., and Peled D. A. 1999], it follows that for every temporal property such a decomposition begins to produce similar X-guarded subformulas after a well defined number of unfolding steps. During the unfolding process we check whether such a fixpoint has been reached.

Once we have the disjunction of the terms, we drop the terms that contain any temporal operator other than X and call the remaining formula as $\mathcal{U}_M$. Dropping terms from the disjunction ensures that $\mathcal{U}_M$ is at least as strong as $\mathcal{U}$. $\mathcal{U}_M$ contains only Booleans and X-guarded Booleans, which is appropriate for Step 2(b) and Step 2(c).

Example 7. Consider the property $\mathcal{U} = (p\ U\ q) \vee X\ F(\neg p)$. After one step of unfolding the property looks like:

$$\mathcal{U}^1 = q \vee (p \wedge X(p\ U\ q)) \vee X\ F(\neg p).$$

Dropping the present state variables and removing an X from the remaining sub formulas of $\mathcal{U}^1$ generates:

$$\mathcal{U}' = (p\ U\ q) \vee F(\neg p).$$

Since $\mathcal{U}'$ is not equivalent to $\mathcal{U}$, we have not yet reached the fixpoint. After the next step of unfolding of $\mathcal{U}'$ and then dropping the present state variables and removing an X from the remaining sub formulas yields:

$$\mathcal{U}'' = (p\ U\ q) \vee F(\neg p),$$

which is equivalent to $\mathcal{U}'$. Since any further decomposition will generate the same formula, this is the fixpoint. After two steps of unfolding the property $\mathcal{U}$ becomes:

$$\begin{aligned}\mathcal{U}^2 = \quad & q \vee (p \wedge X(q \vee p \wedge X(p\ U\ q))) \\ & \vee X(\neg p \vee X\ F(\neg p)).\end{aligned}$$

It can be rewritten in the form of disjunction of the terms as follows:

$$\begin{aligned}&(q) \vee (p \wedge X\ q) \vee (p \wedge X\ p \wedge X\ X(p\ U\ q)) \\ &\quad \vee (X(\neg p)) \vee (X\ X\ F(\neg p)).\end{aligned}$$

Now, dropping the terms containing any temporal operators other than X yields the following set of terms as $\mathcal{U}_M$:

$$\mathcal{U}_M = \{q,\ p \wedge X(q),\ X(\neg p)\}.$$

It may be noted that $\mathcal{U}_M$ may contain variables belonging to $\mathcal{AP}_\mathcal{R} - \mathcal{AP}_\mathcal{A}$. In order to obtain the uncovered architectural intent, we need to eliminate these variables. This is done in Step 2(b).

THEOREM 3. *The property represented by the set of terms $\mathcal{U}_M$ closes the coverage hole for $\mathcal{F}_\mathcal{A}$.*

In Step 2(b) of the coverage algorithm, we universally eliminate the variables in $\mathcal{A}P_{\mathcal{R}} - \mathcal{A}P_{\mathcal{A}}$ from the property $\mathcal{U}_M$.

Theorem 4. *The property represented by the set of terms $\mathcal{U}_M$ after universal abstraction closes the coverage hole for $\mathcal{F}_{\mathcal{A}}$.*

Our target is to represent this coverage gap as a set of properties that are structurally similar to $\mathcal{F}_{\mathcal{A}}$. We achieve this objective by distributing the terms in $\mathcal{U}_M$ into the structure of $\mathcal{F}_{\mathcal{A}}$. The following theorem shows that this approach is theoretically sound.

Theorem 5. *The property $\mathcal{U}_M \vee \mathcal{F}_{\mathcal{A}}$ is at least as weak as $\mathcal{U}_M$ and closes the coverage gap for $\mathcal{F}_{\mathcal{A}}$.*

The remainder of this section presents the methodology for distributing the terms in $\mathcal{U}_M$ into the structure of $\mathcal{F}_{\mathcal{A}}$. The intuitive idea is to push the terms to the sub formulas having similar variables. However, $\mathcal{U}_M$ may also contain some terms that contain variables from $\mathcal{AP}_{\mathcal{A}}$ other than those in $\mathcal{F}_{\mathcal{A}}$. Let us denote these variables by $\mathcal{EV}$ (for *entering variables*).

The Function *Push_Term($\mathcal{F}$, $\mathcal{U}_M$, θ)* pushes the terms in $\mathcal{U}_M$ into the syntactic structure of property, $\mathcal{F}$. To intuitively explain the working of this function, consider a case where $\mathcal{F}$ is of the form $f \Rightarrow g$. Let $Var(f)$ and $Var(g)$ denote the set of variables in f and g, respectively. We compute the universal abstraction of $\mathcal{U}_M$ with respect to $Var(g)$ and recursively push the restricted terms (containing only variables in $Var(g)$) to g. We then compute the universal abstraction of $\mathcal{U}_M$ with respect to $Var(f) \cup \mathcal{EV}$ and recursively push the restricted terms to f. The decision to push terms containing entering variables to the left of the implication is heuristic (but correct, since we could push them either way). Pushing the entering variables in the other side may give us another form of the uncovered architectural intent (and we may like to present both forms).

In case $\mathcal{F}$ is of the form $f \wedge g$, we push each term of $\mathcal{U}_M$ to both f and g.

In case $\mathcal{F}$ is of the form *fUg* or *Ff* or *Gf*, we maintain the list of variables of $\mathcal{U}_M$ with $\mathcal{F}$ for later use by the *variable weakening algorithm* (Step 2(d)).

The third argument, θ of the function *Push_Term* specifies whether $\mathcal{U}_M$ should be considered in disjunction with $\mathcal{F}$ (in which case $\theta = 1$) or in conjunction with $\mathcal{F}$ (denoted by $\theta = 0$). At the root level, we always have $\theta = 1$ (since we compute $\mathcal{F}_{\mathcal{A}} \vee \mathcal{U}_M$). However the semantics of negation sometimes require us to recursively call *Push_Term()* with $\theta = 0$.
The functions *UABS*() and *XABS*() are as follows:

UABS(φ, $\mathcal{SV}$): This function takes a set of terms, φ, and a set of variables $\mathcal{SV}$ as input and universally eliminates the set of variables given by $\mathcal{AP}_{\mathcal{A}} - \mathcal{SV}$ from φ. It returns the property given by the union of the abstracted set of terms.

XABS(φ): This function takes a set of terms, φ, extracts those terms that are within the scope of one or more X operators, and returns these terms after dropping the most significant X operator.

Algorithm 2. Push_Term($\mathcal{F}, \mathcal{U}_M, \theta$)

case $(\mathcal{F} \equiv (f \Rightarrow g))$:

    if $(\theta = 1)$ {   Push_Term($f, \neg UABS(\mathcal{U}_M, Var(f) \cup \mathcal{EV}), 0$);
                 Push_Term($g, UABS(\mathcal{U}_M, Var(g)), 1$); }

    if $(\theta = 0)$ {   Push_Term($f, \neg\mathcal{U}_M, 1$);
                 Push_Term($g, \mathcal{U}_M, 0$) ; }

case $(\mathcal{F} \equiv (f \vee g))$:

    if $(\theta = 1)$ {   Push_Term($f, UABS(\mathcal{U}_M, Var(f) \cup \mathcal{EV}), 1$);
                 Push_Term($g, UABS(\mathcal{U}_M, Var(g)), 1$); }

    if $(\theta = 0)$ {   Push_Term($f, \mathcal{U}_M, 0$);
                 Push_Term($g, \mathcal{U}_M, 0$); }

case $(\mathcal{F} \equiv (f \wedge g))$: {   Push_Term($f, \mathcal{U}_M, \theta$);
                 Push_Term($g, \mathcal{U}_M, \theta$); }

case $(\mathcal{F} \equiv (\neg f))$:   Push_Term($f, \neg\mathcal{U}_M, \neg\theta$);

case $(\mathcal{F} \equiv (X\ f))$: { $\mathcal{U}_{M_{-X}} = XABS(\mathcal{U}_M)$;
                 Push_Term($f, \mathcal{U}_{M_{-X}}, \theta$); }

case $(\mathcal{F} \equiv (G\ f)$ or $(F\ f)$ or $(f\ U\ g))$:
    Maintain the list of variables of $\mathcal{U}_M$ with $\mathcal{F}$ for later
    use by variable weakening algorithm (Step 2(d));

case $(\mathcal{F} \equiv p \in \mathcal{AP}_T)$:
    if$(\theta = 1)$ Replace p by $p \vee \mathcal{U}_M$;
    if$(\theta = 0)$ Replace p by $p \wedge \mathcal{U}_M$;

THEOREM 6. *Push_Term($\mathcal{F}_\mathcal{A}, \mathcal{U}_M$, 1) returns a property $\mathcal{F}_\mathcal{U}$ that closes the coverage hole.*

Example 8. Let us consider the architectural property A and the RTL property R, given as follows, where $r_1, r_2, g_1, g_2 \in \mathcal{AP}_\mathcal{A}$.

$$A: \quad r_1 \Rightarrow X(g_1\ U\ g_2)$$
$$R: \quad (r_1 \wedge r_2) \Rightarrow X(g_1\ U\ g_2).$$

After step 2(b) of Algorithm 1 we have the following set of terms for $\mathcal{U}_M$:

$$\mathcal{U}_M = \{\neg r_1,\ X(g_2),\ r_1 \wedge r_2 \wedge \neg X(g_2)\}$$

The distribution of the above terms into the parse tree of A by Algorithm 2 is shown in the Figure 4.

After the execution of the Push_Term algorithm $\mathcal{F}_\mathcal{U}$ is represented in the following form:

$$\mathcal{F}_\mathcal{U}: (r_1 \wedge \neg r_2) \Rightarrow X(g_1\ U\ g_2).$$

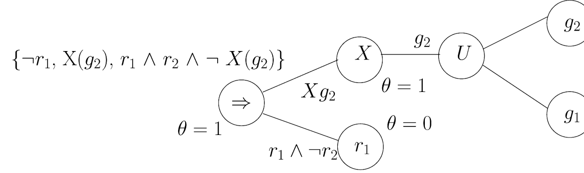

Fig. 4. Working of algorithm Push_Term.

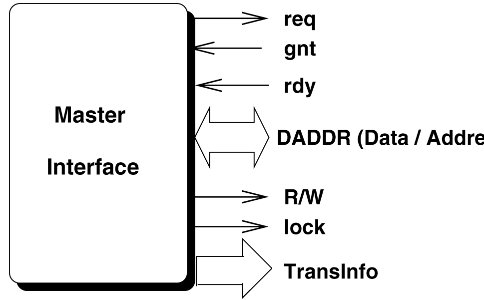

Fig. 5. Master interface of MyBus protocol.

We also have the sole element g_2 in the variable list corresponding to the node U. This list is to be used in Step 2(d) of the intent coverage algorithm.

4. INTENT COVERAGE WITH AUXILIARY STATE MACHINES

In this section, we present the new design intent verification problem for specifications containing auxiliary state machines and formal properties. We start with a demonstration of the use of auxiliary state-machines in developing formal specifications. In Section 4.5 we discuss the generic extensions of the problem.

4.1 Auxiliary State-Machines in Formal Property Specifications

Consider a Bus protocol (MyBus protocol) that supports multiple master devices, multiple memory-mapped slave devices, and a single arbiter. The Bus has 64-bit multiplexed address and data lines. During a transfer, address and data are time multiplexed on these 64 lines, with address and data appearing in alternate cycles. We focus on the behavior of a specific master device, $\mathcal{M}$, which is the *highest priority* device on the Bus. The interface of $\mathcal{M}$ is shown in Figure 5.

The master interface is IDLE when $\mathcal{M}$ does not intend to perform a transfer. When the master intends to start a transfer, it raises its request line, `req`. On receiving the `gnt`, $\mathcal{M}$ decides whether it wants to start a locked transfer. In case of a locked transfer, it asserts the *lock* signal in the next cycle (INIT). Irrespective of the transfer types, $\mathcal{M}$ floats a valid transfer with the address in the Bus and waits (from the next cycle onwards) for the `rdy` signal from the slave device. We refer this phase of the transfer as the ADDRESS cycle.

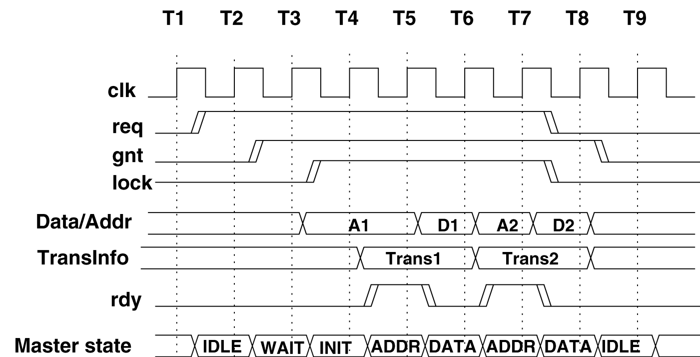

Fig. 6. A sample transfer.

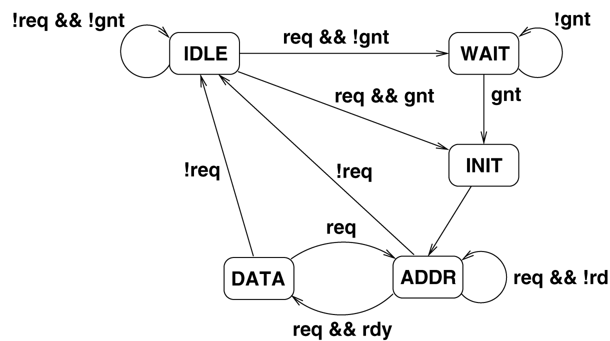

Fig. 7. State-machine for a MyBus transfer.

The `rdy` signal from the slave indicates that the slave is ready for the transfer. On receiving this signal, the master enters the DATA cycle and does the following:

(1) In the case of a write, it floats the data on the Bus.
(2) In the case of a read, it expects the slave to produce the data on the Bus.

The intent to read/write is indicated by a R/W signal—high indicates write, low indicates read. After each data cycle, the master may start another address cycle by floating the next address on the Bus. At any point in time, the master can return to the IDLE state by lowering the `req` line, which signals the end of the transfer to the arbiter. In case of any locked transfer, $\mathcal{M}$ also deasserts the *lock* signal. A sample transfer is shown in Figure 6. Other transfer related information (e.g., width, length of the transfer) are conveyed by a vector `TransInfo`.

Figure 7 show the abstract state-machine for the MyBus master device. The state-machine contains only sufficient information that carries it through the major phases of the protocol. For example, the status of the `gnt` line are not indicated in the DATA and ADDR cycles. We encode this state machine in terms of a 3-bit variable, *fsm_state*.

Next, we present one sample architectural property based on the auxiliary state-machine. We also assume that $\mathcal{M}$ always starts locked transfer after receiving the grant.

— A_1: If the master is forced to continue in the ADDRESS cycle by the slave (*rdy* is de-asserted), it should not change the address floated in the Bus until it receives the *rdy* signal from the slave. This property (modeled in LTL) is as follows.

$$G((fsm_state = ADDR \ \wedge \ \neg rdy) \ \Rightarrow X(DADDR == Prev_ADDR)).$$

$Prev_ADDR$ is a vector that stores the address value of the previous cycle.

— A_2: If the master is in the INIT cycle, it should assert its *lock* signal and drive a valid transfer information in the next cycle. The valid transfer information is encoded by a proposition called *ValidTrans*. This property is as follows.

$$G((fsm_state = INIT) \ \Rightarrow (lock \wedge X(ValidTrans))).$$

The MyBus protocol will be the running example in our discussion. We will modify the state-machine to show different aspects of the problem, as and when required.

4.2 Formal Characterization

Auxiliary state-machine based design intent verification essentially checks whether the RTL specs ($\mathcal{R}$) covers the architectural properties ($\mathcal{A}$), where $\mathcal{A}$ is specified using an auxiliary state-machine $\mathcal{T}$. The *states* of T are encoded with a set ($\mathcal{AP}_{\mathcal{T}}$) of auxiliary state-bit variables. In the example presented in the previous subsection, `IDLE`, `WAIT`, `INIT`, `DATA`, and `ADDR` are the states. The transitions are enabled by the different conditions over the input/output variables.

The inputs to the problem are as follows.

(1) The *architectural specification* $\mathcal{A}$ is a set of LTL properties over $\mathcal{AP}_{\mathcal{A}} \cup \mathcal{AP}_{\mathcal{T}}$, where $\mathcal{AP}_{\mathcal{A}}$ and $\mathcal{AP}_{\mathcal{T}}$ are the sets of architectural signals and the auxiliary state-bits, respectively.
(2) The *RTL specification* $\mathcal{R}$ is a set of LTL properties over $\mathcal{AP}_{\mathcal{R}}$, where $\mathcal{AP}_{\mathcal{R}}$ is the set of the RTL signals.
(3) The auxiliary state-machine $\mathcal{T}$, where the states are different valuations of $\mathcal{AP}_{\mathcal{T}}$ and the transitions are enabled by boolean constraints over $\mathcal{AP}_{\mathcal{A}}$.

In our discussion, we assume that the formal properties are defined over a single auxiliary state-machine $\mathcal{T}$. The approach can easily be generalized to specifications having multiple auxiliary state-machines by defining $\mathcal{T}$ to be the product of the individual state-machines. Since the auxiliary state-machines are user defined and small, the product computation is not expensive.

The verification problem is as follows:

— To determine whether the RTL specification covers the architectural specification, consisting of architectural properties defined over auxiliary state-machines.

—If the answer to the previous question is no, then to determine a set of additional properties that represent the coverage gap (that is, these properties together with the RTL specification succeed in covering the architectural specification).

The following example will illustrate the gap in a formal property.

Example 9. Consider the following architectural and RTL properties for the auxiliary state-machine.

Arch. spec ($\mathcal{A}$): We consider architectural property A_2 given as follows.

$$G((fsm_state = INIT) \Rightarrow (lock \wedge X(ValidTrans)).$$

RTL spec ($\mathcal{R}$): Whenever, $\mathcal{M}$ receives the grant, it drives valid transfer information after two cycles.

$$G((\neg gnt \wedge X(gnt)) \Rightarrow (XXX(ValidTrans))).$$

In this case, the governing equation is,

$$\mathcal{R} \Rightarrow \left(\left(\bigwedge_j \mathcal{T}_{A_j}\right) \Rightarrow \mathcal{A}\right).$$

It is easy to appreciate that the RTL property covers the following path `WAIT` $\rightarrow$ `INIT` $\rightarrow$ `ADDR` partially. The property has not covered the lock status requirement in the state `INIT`. Therefore, the expected gap will be, *whenever the current and the next state of the auxiliary state-machine is* `INIT`*, in the same cycle* `lock` *will be asserted*. Formally,

$$G((fsm_state = INIT) \Rightarrow (lock)).$$

An auxiliary state-machine $\mathcal{T}$ can be transformed to a set of LTL properties. For example, the auxiliary state-machine of Figure 7 can be represented as follow:

$$\begin{aligned}
P1 &: G(X(IDLE) \Leftrightarrow (IDLE \wedge \neg req \wedge \neg gnt) \vee (ADDR \wedge \neg req) \\
&\quad \vee (DATA \wedge \neg req)) \\
P2 &: G(X(DATA) \Leftrightarrow (ADDR \wedge req \wedge rdy)) \\
P3 &: G(X(ADDR) \Leftrightarrow (ADDR \wedge req \wedge \neg rdy) \vee (INIT) \\
&\quad \vee (DATA \wedge req)) \\
P4 &: G(X(INIT) \Leftrightarrow (WAIT \wedge gnt) \vee (IDLE \wedge req \wedge gnt)) \\
P5 &: G(X(WAIT) \Leftrightarrow (WAIT \wedge \neg gnt) \vee (IDLE \wedge req \wedge \neg gnt)).
\end{aligned}$$

The initial state can be explicitly specified as a Boolean (such as `IDLE`):

$$\mathcal{T} = IDLE \wedge P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5.$$

The following theorem shows us an intuitive way to answer the first question.

THEOREM 7. *The RTL specification, $\mathcal{R}$, covers the architectural specification $\mathcal{A}$, iff the temporal property $\mathcal{R} \wedge \neg(\mathcal{T} \Rightarrow \mathcal{A})$ is unsatisfiable.*

Algorithm 3. Coverage Algorithm

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

(1) compute $\mathcal{U} = \mathcal{F}_\mathcal{A} \vee \neg(\mathcal{R} \wedge T)$
(2) If $\neg(\mathcal{U})$ is not false 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 $\mathcal{A}P_\mathcal{R} - \mathcal{A}P_\mathcal{A}$,
   (c) Push the terms of $\mathcal{U}_M$ into $\mathcal{F}_\mathcal{A}$ to obtain $\mathcal{F}_\mathcal{U}$.
(3) Return $\mathcal{F}_\mathcal{U}$;

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

Theorem 7 answers the primary coverage question by checking the validity of $\mathcal{R} \Rightarrow (\mathcal{T} \Rightarrow \mathcal{A})$.

$$\begin{aligned} & \mathcal{R} \Rightarrow (\mathcal{T} \Rightarrow \mathcal{A}) \\ \Rightarrow\ & (\mathcal{R} \wedge \mathcal{T}) \Rightarrow \mathcal{A}. \end{aligned} \tag{3}$$

If the answer to the primary coverage question is negative, then we need to deduce the coverage gap. This is performed by extending our original coverage algorithm.

4.3 Coverage Algorithm

The goal of 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 consisting of $\mathcal{A}$ and $\mathcal{T}$, and finds the coverage gap, $\mathcal{G}$, for $\mathcal{F}_\mathcal{A}$, with respect to the RTL properties $\mathcal{R}$. Since $\mathcal{R}$ is required to cover every property in $\mathcal{A}$, we target each property of $\mathcal{A}$ individually. The following algorithm uses $\mathcal{U}$ to represent the RTL coverage hole between $\mathcal{R}$ and a given architectural property $\mathcal{F}_\mathcal{A}$.

Algorithm 3 is similar in philosophy to the one presented in Section 3, except that we need to handle Step 2(c) in a totally different way. If we directly use the **Push_Term** algorithm presented in Section 3, in our case (where the architectural specs is defined over auxiliary state-machines), the gap is typically produced in a nonintuitive form. We apply a *state-bit renaming heuristic* to solve the problem, as described in the next subsection.

4.4 The State-bit Renaming Heuristic

The following example demonstrates the problem with generalizing our original **Push_Term** algorithm to specifications with auxiliary state-machines.

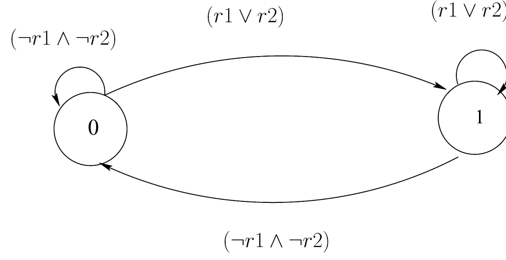

Fig. 8. The architectural state-machine.

Example 10. Let the architectural and RTL properties be given as follows,

$$\mathcal{A}: (r_1 \wedge \neg s) \Rightarrow X(g_1 \cup g_2) \wedge X(s) \tag{4}$$

$$\mathcal{R}: (r_1 \wedge r_2) \Rightarrow X(g_1 \cup g_2), \tag{5}$$

where, r_1, r_2, g_1, $g_2 \in \mathcal{AP}_\mathcal{A}$ and s is the only state-bit for the auxiliary state-machine shown in Figure 8.

The auxiliary state-machine is transformed into the following LTL properties.

$$T1: \quad G(X(s) \Leftrightarrow (s \wedge (r_1 \vee r_2)) \vee (\neg s \wedge (r_1 \vee r_2)) \tag{6}$$

$$T2: \quad G(X(\neg s) \Leftrightarrow (s \wedge \neg r_1 \wedge \neg r_2) \vee (\neg s \wedge \neg r_1 \wedge \neg r_2)). \tag{7}$$

Now, we test the validity of the following equation:

$$\mathcal{R} \wedge (T1 \wedge T2) \Rightarrow \mathcal{A}.$$

The architectural property is silent regarding the atomic proposition r_2, and hence it is easy to appreciate that there exists a gap between the specifications (since the RTL specifies nothing when $(r_1 \wedge \neg r_2)$ holds true). Applying left-insertion heuristic (as described in Basu et al. [2006]), SpecMatcher gives the following coverage gap:

$$\mathcal{F}_\mathcal{U}: G((r_1 \wedge \neg r_2 \wedge \neg s \wedge X(s)) \Rightarrow X((\neg g_1 \wedge \neg s) \vee (g_1 \cup g_2))). \tag{8}$$

Clearly, this representation of the coverage gap is incomprehensible. How to represent the gap more succinctly and meaningfully? Before addressing this question, we analyze the steps in the original **Push_Term** algorithm.

Analysis of Push_Term Algorithm. Without loss of generality, we assume left-insertion strategy throughout the discussion. The algorithm defines *Var*(f) (the variables in formula f) and the *entering variables* ($\mathcal{EV}$), (variables belonging to $(\mathcal{AP}_\mathcal{A} - \mathcal{F}_\mathcal{A})$, where $\mathcal{F}_\mathcal{A}$ is the set of variables in the architectural formula). In **case**($\mathcal{F} \equiv (f \Rightarrow g)$), the variables either in *Var*(f) or in $\mathcal{EV}$ enter the left-subtree of the formula (shown in Figure 9).

This insight helps us in understanding the genesis of such a complicated gap. The antecedent of the implication in equation 8 contains both the terms s and $X(s)$, making the equation more complex, because the presence of the current and the future values of the state-bit on the same side of the implication

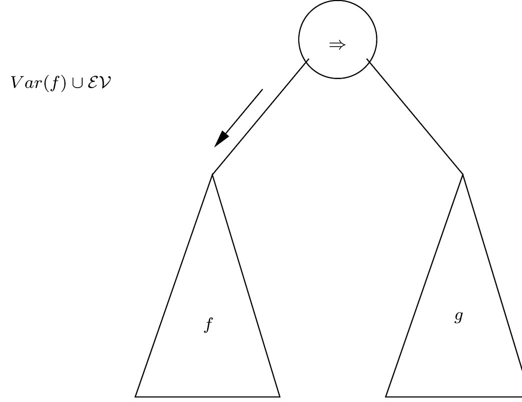

Fig. 9. The entering variables enter the left subtree in **case:** $(\mathcal{F} \equiv (f \Rightarrow g))$.

bears no physical significance. In order to avoid these situations, we propose the following: *In all the specifications and the state-transitions, replace the X-guarded terms of the state-bits with new variable names.* For example, $\mathcal{A}$ in Example 10 becomes,

$$\mathcal{A}' : (r_1 \wedge \neg s) \Rightarrow X(g_1 \vee g_2) \wedge next_s. \tag{9}$$

We have replaced $X(s)$ (in $\mathcal{A}$) with $next_s$ (in $\mathcal{A}'$). Now, since $next_s \notin (Var(f) \cup \mathcal{EV})$, it will not enter the same subtree as s does. Applying this change in our algorithm, we obtain the following gap which is more meaningful to the verification engineer.

$$\mathcal{F}_{\mathcal{U}}' : G(\neg s \wedge (r_1 \wedge \neg r_2) \Rightarrow X(g_1 \cup g_2)). \tag{10}$$

Handling the Gap. Consider the properties in Example 9. on the perspective of the factored state machine (refer Figure 7).

$$\mathcal{A} : \quad G((fsm_state = INIT) \Rightarrow (lock \wedge X(ValidTrans))$$
$$\mathcal{R} : \quad G((\neg gnt \wedge X(gnt)) \Rightarrow (XXX(ValidTrans))).$$

In this case, the governing equation is,

$$\mathcal{R} \Rightarrow \left(\left(\bigwedge_j \mathcal{T}_{A_j}\right) \Rightarrow \mathcal{A}\right).$$

Next, we discuss the adopted approach in order to meaningfully present the gap. The steps for presenting the gaps are elaborated below.

(1) Rename the X-guarded architectural state-bits in $\mathcal{A}$ according to *State-bit Renaming Heuristic*.
(2) Execute the regular **Push_Term** algorithm and perform *universal abstraction* of the state bits in the obtained complex gap. Those bits disappeared, and hence, a weaker gap is obtained. The final gap, in terms of the variable *fsm_state* and interface signals, is presented as follows:

$$G((fsm_state = INIT) \Rightarrow (lock)).$$

It is easy to appreciate this gap, since the given RTL property indicates the path from state `WAIT` to `ADDR` partially in the architectural auxiliary state machine.

4.5 Generic Extensions

We start with the simplest LTL property based intent coverage problem and gradually broaden the scope of the problem.

—*Case 1*: Both the architectural specs ($\mathcal{A}$) and the RTL specs ($\mathcal{R}$) are LTL properties, with $\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$. We check the validity of

$$\mathcal{R} \Rightarrow \mathcal{A}, \tag{11}$$

and the gap (if present) is represented following the **Push_Term** algorithm.

—*Case 2*: The architectural state-machine ($\mathcal{T}_{\mathcal{A}}$) and the LTL architectural specs based on it ($\mathcal{A}$) are given. The RTL specs ($\mathcal{R}$) are also given in LTL. The architectural specs ($\mathcal{A}$) are on ($\mathcal{AP}_{\mathcal{A}} \cup \mathcal{AP}_{\mathcal{T}_{\mathcal{A}}}$). Moreover, ($\mathcal{AP}_{\mathcal{A}} \subseteq \mathcal{AP}_{\mathcal{R}}$) and ($\mathcal{AP}_{\mathcal{T}_{\mathcal{A}}} \cap \mathcal{AP}_{\mathcal{R}} = \phi$). In this case, we check the validity of,

$$\mathcal{R} \Rightarrow (\mathcal{T}_{\mathcal{A}} \Rightarrow \mathcal{A}) \tag{12}$$

and the gap (if present) is represented following the *State-bit Renaming Heuristic* described in Subsection 4.4.

—*Case 3*: Next, suppose there is an underlying state-machine for the RTL also. Let it be $\mathcal{T}_{\mathcal{R}}$. The RTL specs are on ($\mathcal{AP}_{\mathcal{R}} \cup \mathcal{AP}_{\mathcal{T}_{\mathcal{R}}}$). Besides, ($\mathcal{AP}_{\mathcal{T}_{\mathcal{R}}} \cap \mathcal{AP}_{\mathcal{A}} = \phi$). In this case, we check the validity of:

$$(\mathcal{T}_{\mathcal{R}} \Rightarrow \mathcal{R}) \Rightarrow (\mathcal{T}_{\mathcal{A}}. \Rightarrow \mathcal{A}). \tag{13}$$

—*Case 4*: We further assume that there are n RTL modules. For each module, suppose there exists a state-machine ($\mathcal{T}_{R_i} \forall i = 1$ to n). Let the RTL specs be $\mathcal{R}_1, \ldots, \mathcal{R}_n$. In order to verify the design intent, we check the validity of,

$$\bigwedge_i (\mathcal{T}_{R_i} \Rightarrow \mathcal{R}_i) \Rightarrow (\mathcal{T}_{\mathcal{A}} \Rightarrow \mathcal{A}). \tag{14}$$

—*Case 5*: Lastly, suppose the architectural state-machine is *factored* in m state-machines (e.g. $\mathcal{T}_{A_j}, j = 1$ to m). In that case, we check the validity of the following:

$$\bigwedge_i (\mathcal{T}_{R_i} \Rightarrow \mathcal{R}_i) \Rightarrow \left(\left(\bigwedge_j \mathcal{T}_{A_j}\right) \Rightarrow \mathcal{A}\right). \tag{15}$$

5. DESIGN INTENT VERIfiCATION WITH CONCRETE MODULES

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 problem considered in this section, 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*. Here, we define a *state* as a valuation of the signals at a given time. A *run* is an infinite sequence of states over time.

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 8. *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 that 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.

Example 11. Figure 10 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 its 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[2]. We must therefore refine the specification. Let us assume that we are given the RTL for M_1 and

[2]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.

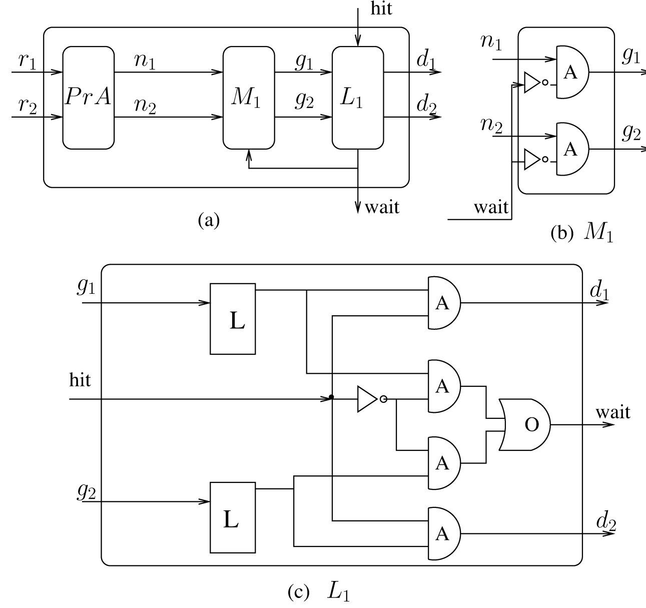

Fig. 10. Memory Arbitration Logic (MAL).

L_1 and the following properties for PrA:

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

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 Figure 11. 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 Figure 11(b)) then *wait* will be high, which would prevent g_2 being 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 $\mathcal{A}$ being 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.* □

5.1 Computing the Coverage Gap

In this section, we address the more complex problem of computing and representing the coverage gap. One way to demonstrate that a coverage gap exists is to produce a counter-example run, that is, a run that satisfies the RTL

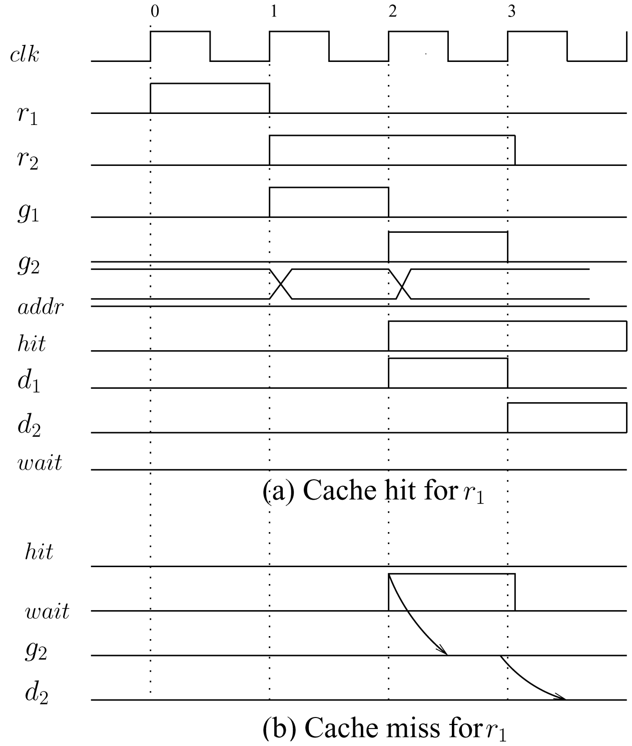

Fig. 11. Timing diagram.

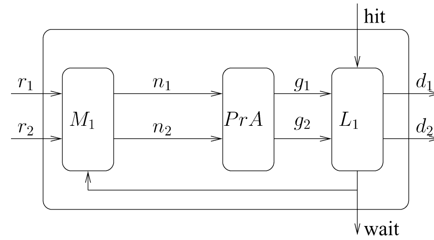

Fig. 12. Mem Arbitration Logic (With GAP).

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.

Example 12. Let us consider a slight variant of the MAL described in Example 1 as shown in Figure 12. 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 are connected to the grant inputs g_1 and g_2 of L_1. The new RTL properties of PrA would be:

$$R'_1 = \mathrm{G}(n_1 \rightarrow Xg_1) \qquad R'_2 = \mathrm{G}(\neg n_1 \wedge n_2 \rightarrow Xg_2).$$

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.

In order to determine the coverage hole, we generate the temporal formula, which exactly represents the RTL concrete module $\mathcal{M}$ (as demonstrated in Example 2).

The following theorem characterizes the coverage hole.

THEOREM 9. 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. □

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

Definition 8. (*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. □

5.2 Representing the Coverage Hole

Theorem 9 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 13. We consider the coverage of A by R_1',R_2' and the concrete modules M_1 and L_1 as given in Example 12. By Theorem 9, 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 meaningful information to the designer. On the other hand, consider the property U of Example 12:

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

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

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 remain to be covered.

5.3 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 Coverage Algorithm presented in Das et al. [2006] implements this idea. There, we have used $\mathcal{U}$ to represent the RTL coverage hole $\mathcal{R}_H$ and $\mathcal{M}$ to represent the concrete module in the RTL specification. In fact, in that algorithm we compute

$$\mathcal{U} = \mathcal{F}_\mathcal{A} \vee \neg(\mathcal{R} \wedge T_M)$$

before applying the **Push_Term** heuristic. Here, we explain its operation with the help of the design in Example 12.

In the design described in Example 12, $\mathcal{A} = G(\neg wait \wedge r_1 \wedge X(r_1 U r_2) \rightarrow X(\neg d_2 U d_1))$, 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:

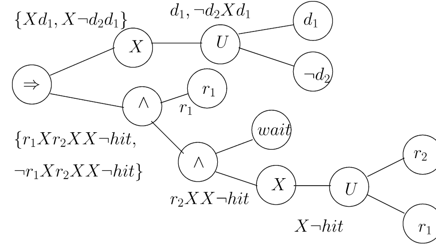

Fig. 13. Pushing the terms in $\mathcal{U}_M$.

$$
\begin{aligned}
T_{M1} = {} & (\neg g_1 \wedge \neg g_2 \wedge \neg wait) \wedge G[(g_1 \wedge hit' \wedge d_1') \\
& \vee (g_1 \wedge hit' \wedge d_2') \vee (\neg(g_1 \wedge hit') \wedge \neg d_1') \vee (\neg(g_1 \wedge hit') \wedge \neg d_1') \\
& \vee (g_1 \wedge \neg hit' \wedge wait') \vee (g_1 \wedge \neg hit' \wedge wait') \vee (\neg(g_2 \wedge \neg hit') \\
& \wedge \neg d_1') \vee (\neg(g_2 \wedge \neg hit') \wedge \neg d_1')].
\end{aligned}
$$

The first step of Algorithm 3 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 its fixpoint [Clarke et al. 1999]. After unfolding and abstracting out the local RTL variable d, we obtain $\mathcal{U}_M$ as follows:

$$
\begin{aligned}
\mathcal{U}_M \; = \; & \{\neg r_1 \wedge X r_2 \wedge XX\neg hit \wedge X d_1, \\
& \neg r_1 \wedge X r_2 \wedge XX\neg hit \wedge X\neg d_2 \wedge XX d_1\}.
\end{aligned}
$$

The distribution of these terms into the parse tree of A is done in the step 2(c) of the algorithm, as shown in Figure 13. This step determines that the gaps lie inside the unbounded operator until(U).

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: $\quad G((a \; U \; b) \Rightarrow (c \; U \; d))$.

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 that 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': $\quad G((a \; U \; b) \Rightarrow ((c \vee \neg e) \; U \; d))$
φ'': $\quad G((a \; U \; b) \Rightarrow ((c \vee e) \; U \; d))$.

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

Table I. Runtimes of SpecMatcher for Concrete Modules

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. (Figure 10)	2	0.18	0.06	1.2

Table II. Runtimes of SpecMatcher for Auxiliary state machines

Circuit	No. of RTL Properties	Time (sec)		
		Primary	No. of states	Gap Finding Time
ARM AMBA AHB	29	14.17	13	25.7
MyBus Example	2	0.18	5	1.2

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 architectural specification and the RTL.

6. RESULTS ON SPECMATCHER

SpecMatcher is our tool for verifying design intent coverage. Table I and II show the runtime of our tool on several designs with concrete modules and with auxiliary state machines respectively. For each design, we selected architectural properties, which requires contributions from multiple submodules. For example, ARM AMBA AHB is a bus protocol involving master, slave and arbiter devices. 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.

7. CONCLUSION

The design intent coverage approach is a novel attempt toward extending the frontiers of FPV (formal property verification) by enabling the coverage of system level properties by the collection of module level properties. In this work, we have extended this methodology to work for other hybrid specifications where the module level specification can contain RTL descriptions and/or the module/system level properties can be annotated with auxiliary statemachines. This enhancement certainly enables the design intent methodology to work with wider range of specification models. We have also shown the applicability of our proposal on several bus protocols.

APPENDIX

A: SYNTAX AND SEMANTICS OF LTL

The formal syntax and semantics of LTL [Clarke E. M., Grumberg O., and Peled D. A. 1999] is defined over a Kripke structure. Formally, we define a Kripke structure as a tuple, $K = \langle \mathcal{AP}, S, \tau, s_0, \mathcal{F} \rangle$, where:

—S is a finite set of states,

—$\mathcal{A}P$ is a set of atomic propositions labeling S,

—$\tau \subseteq S \times S$ is the transition relation, which must be total (for all states $s_i \in S$, there exists a state $s_j \in S$ such that $(s_i, s_j) \in \tau$),

—$s_0 \subseteq S$ is the set of start states,

—$\mathcal{F} : S \rightarrow 2^{\mathcal{A}P}$ is a labeling of states with atomic propositions true in that state.

A *path*, π, in the Kripke structure is an infinite sequence of states, $s_0, s_1, \ldots$, such that for all i, $s_i \in S$, and $(s_i, s_{i+1}) \in \tau$. s_0 is called the starting state of π. We use π_j to denote the suffix of π starting from s_j.
The formal syntax of LTL is as follows:

—Each atomic proposition in $\mathcal{A}P$ is a LTL formula.

—If f and g are LTL formulas, then so are $\neg f$, $f \wedge g$, $X\ f$, $f\ U\ g$.

The formal semantics of LTL is as follows (f and g are LTL formulas; p is an atomic proposition; $\pi = s_0, s_1, \ldots$ is a path in K):

—$\pi \models p$ iff $p \in \mathcal{F}(s_0)$

—$\pi \models \neg f$ iff $\pi \not\models f$

—$\pi \models f \wedge g$ iff $\pi \models f$ and $\pi \models g$

—$\pi \models X\ f$ iff $\pi_1 \models f$

—$\pi \models f\ U\ g$ iff $\exists j$, such that $\pi_j \models g$ and $\forall i, i < j$ we have $\pi_i \models f$

ACKNOWLEDGMENTS

The authors would like to thank all the reviewers for critically judging our work and substantially improving the value and the quality of this article.

REFERENCES

Basu, P., Das, S., Banerjee, A., P. Dasgupta, P. P. C., Mohan, C., Fix, L., and Armoni, R. 2006. Design intent coverage—a new paradigm for formal property verification. *Comput.-Aid. Des. Int. Circ. Syst. 25*, 10, 1922–1934.

Chockler, H., Kupferman, O., Kurshan, R. P., and Vardi, Y. M. 2001. A practical approach to coverage in model checking. In *Proceedings of the Conference on Computer Aided Verification*. 66–78.

Chockler, H., Kupferman, O., and Vardi, M. Y. 2001. Coverage metrics for temporal logic model checking. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 528–542.

Chockler, H., Kupferman, O., and Vardi, Y. M. 2003. Coverage metrics for formal verification. In *Proceedings of the* 12th *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Lecture Notes in Computer Science, vol. 2860, Springer, 111–125.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking*. MIT Press.

Das, S., Basu, P., Dasgupta, P., and Chakrabarti, P. P. 2006. What lies between design intent coverage and model checking? In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, Munich, Germany, 1217–1222.

Dasgupta, P. 2006. *A Roadmap for Formal Property Verification*. Springer.

Hoskote, Y., Kam, T., Ho, P. H., and Zhao, X. 1999. Coverage estimation for symbolic model checking. In *Proceedings of the Design Automation Conference*. 300–305.

KATZ, S., GRUMBERG, O., AND GEIST, D. 1999. Have i written enough properties?"—a method of comparison between specification and implementation. In *Proceedings of the Conference on Correct Hardware Design and Verification Methods*. 280–297.

PNUELI, A. 1977. The complexity of propositional linear temporal logics. In *Proceedings of the Foundations of Computer Science*. 46–57.

PSL. *Property Specification Language (PSL)*. www.eda.org/vfv/docs/PSL-v1.1.pdf. PSL.

SISTLA, A. P. AND CLARKE, E. M. 1985. The complexity of propositional linear temporal logics. *J. ACM 32*, 3, 733–749.

SVA. *SystemVerilog 3.1a Language Reference Manual*. www.eda.org/sv/SystemVerilog_3.1a.pdf. SVA.

Received July 2007; revised February 2008; accepted August 2008