DesignCon 2005

AMBA Compliance Checking Using Static Functional Verification

Adrian J. Isles, Averant, Inc.
aisles@averant.com

Jeremy Sonander, Saros Technology UK
jeremy@saros.co.uk

Mike Turpin, ARM UK
Mike.Turpin@arm.co.uk

Abstract
We show how designs that implement the AMBA protocol specification can be verified automatically using Static Functional Verification (SFV) technology. SFV is an approach for verifying the correctness of RTL designs by using formal analysis to prove that a property holds for a design under all inputs, sequences of inputs and states. This provides a level of confidence in the correctness of a design that is unachievable with simulation. We show how different AMBA protocol rules can be described using the Property Specification Language (PSL). A methodology is also provided that allows these rules to be reused across various designs that implement the AMBA protocol specification.

Author(s) Biography
Adrian J. Isles
Adrian Isles is one of the architects and chief developers of Solidify. He obtained his B.S. in Electrical Engineering from Howard University in 1993 and his M.S. and Ph.D. in Electrical Engineering and Computer Science from the University of California, Berkeley in 1997 and 2000, respectively. His research focus is in areas related to computer-aided design and formal verification of integrated circuits. Since 1990, he has had several work experiences with Intel Corporation and Massachusetts Institute of Technology, Lincoln Laboratory. He has been with Averant since 1999.

Jeremy Sonander
Jeremy Sonander is an application engineer at Saros Technology UK who supports the company's range of formal verification products. He has written the user interface for SolidAHB, a formal tool which will establish a design's compliance with the AHB bus protocol. Prior to joining Saros 9 years ago, he worked for Gould Electronics where he designed a number of FPGAs and ASICs and wrote the embedded software that controls a thermal printer.

Mike Turpin
Mike Turpin is a Principal Verification Engineer at ARM UK, specializing in formal verification (equivalence checking and property checking) and assertion methodology. Since joining ARM 4 years ago, Mike has applied assertions and formal verification to CPU cores, L1 memory controllers, and AMBA interfaces. Prior to ARM, Mike spent 5 years working for an Avionics company and then 7 years working as an application engineer for EDA companies (selling formal verification tools). Mike published a paper at Boston SNUG 2003, entitled "The Dangers of Living with an X" [1], which won the Technical Committee Award.

Introduction

The ARM range of 16/32 bit microprocessors lead the embedded microprocessor market and the associated AMBA bus protocols have become a popular choice of companies designing complex, state of the art, systems on a chip (SoC). One of the key verification tasks in the design of these systems is to ensure that each of the design units in the system obeys the interconnecting bus protocol. Failure to achieve this can result in poor product quality, ASIC re-spins, and delayed entry into the market with corresponding loss of revenues. Existing protocol checking methods are typically simulation based, require the creation of large numbers of test vectors, cannot expose *all* corner cases (i.e. it is non-exhaustive), and typically take man months of effort and a large investment in EDA tools.

The purpose of this paper is to show how Static Functional Verification (SFV) technology [2] can be used to verify design units that implement the AMBA protocol [3]. SFV is a technology that allows for verification to be performed exhaustively without the need for test-benches or test vectors.

Two traditional problems that keep SFV from entering the main stream are:

- Designers have to write properties (or assertions) to describe expected behavior.
- Proving properties can be difficult (tool capacity limitations, false negative verification results, etc.).

These problems can be addressed by changes in design methodology, as well as educating the design community about how to write assertions and use formal tools. SFV tools themselves are improving, becoming easier to use and giving better results that increase the exhaustive pass rate of assertions.

There are some constrained tasks where these traditional problems can be overcome. Bus protocol compliance checking is particularly well suited to the SFV approach since:

- The behavior to be verified is well defined, allowing a set of standard protocol rules to be implemented as assertions.
- The behavior of the environment is also well defined (avoids proofs failing due to interface false negatives).
- The signals in question will appear on the interface of a block in a format that is known in advance.
- It can typically be checked at the block level and, as a result, does not suffer from some of the capacity limitations found in checking properties at the chip level.

All that is required is that the designers understand what portion of the protocol is implemented in the design and a mapping between the signal names in the design and signal names given in the protocol specification.

We show how a set of AMBA protocol rules, can be translated from English into a property language, such as PSL [4]. We then demonstrate how they can be formally verified and added to the design verification methodology. Typically, a static tool can

either prove a property exhaustively (i.e. shows that it is always true), disprove a property (i.e. produce an error trace from reset to show how a design violates a property), or returns an inconclusive result (cannot prove or disprove the property). For inconclusive properties, we show how the protocol rules can be added to the simulation environment to validate whether or not the property passes in simulation.

A more detailed introduction to formal methods can be found in [5,6] and [7] for assertion-based verification. Previous work on writing assertions for the AMBA AHB protocol can also be found in [8]. Note that the work presented that paper, however, occurs at the transaction layer, tends to be more global in nature and therefore less amenable to efficient verification using SFV technology.

Static Functional Verification

Static Functional Verification (SFV) technology is an analysis technique that eliminates the need for simulation vectors and can provide a guarantee that certain types of behaviors always hold for a design. This is a qualitative difference from what one can get in simulation. As a practical matter, verification using simulation is incomplete: the only way to show that a design is free of a bug is to exhaustively simulate all possible input sequences that may occur after reset. This, of course, is not feasible for even relatively small designs: much less the complex, state of the art, mulit-million gate System-on-a-Chip (SoC) designs that are being developed today.

SFV works by taking synthesizable Verilog or VHDL RTL descriptions, along with a set of mathematical properties that describe, in a unambiguous way, how the design is suppose to behave. For each property, an SFV tool performs an analysis of the design to determine if the property always holds. Compared to simulation, verification is typically fast and most properties can be verified in the order of minutes or hours.

The design specification of the behaviors to be verified can be provided in terms of property languages/libraries, such as the Property Specification Language (PSL) [4], Open Verification Library (OVL) [9], System Verilog Assertions (SVA) [10] or the Hardware Specification Language (HPL) [2]. The properties in this paper will be specified in terms of PSL. Many commercial simulators now have built in support for PSL, and thus can be used for both static verification and dynamic verification (simulation). PSL properties can be both embedded directly in the RTL description or kept in a separate property file. An example PSL property is given below:

```
assert always ( req  -> (next[2]( ack || nack )));
```

In this example, `req`, `ack` and `nack` are signals in the design. This property specifies the behavior that if the `req` signal is asserted, then in two clock cycles, the `ack` signal or the `nack` signal should also be asserted.

THE AMBA Bus Protocol Specification

The AMBA bus protocol specification is an on-chip bus protocol standard that facilitates the interconnection of functional blocks and is particularly suitable for implementation of SoC designs. The AMBA-2.0 protocol exists in several flavors: the Advanced High-performance Bus (AHB), the Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB). In this paper, we will be focusing on AHB, but the methodology applies to all three. AHB is the high performance version of AMBA that is useful for connecting processors to on-chip memories, high performance peripherals, and secondary processors. ASB is an alternative to AHB, but is not high performance and APB is optimized specifically for low power applications. Finally, the AHB-Lite protocol is a subset of the AHB protocol and is used in design that contain only a single bus master.

The protocol rules are a specification which unambiguously describe the correct AMBA protocol as described in the AMBA 2.0 specification document [3]. An example of one of the rules is given in Figure 1.

```
                  no_change_when_busy
BUSY indicates a pause in the transfer. During BUSY
the address and control signals must reflect the next
transfer in the burst.
```

Figure 1. Description of BUSY (AHB 2.0, Section 3.5)

Each AHB design can contain both master and slave units that can participate in bus operations. The master is able to initiate read and write operations on the bus and the slave responds to these operations. The properties in this paper will address verification of the master. However the methodology applies to both master and slave units. In addition, AHB supports various data bus width configurations, including 32, 64 and 128-bit bus widths. The property sets developed can easily be parameterized so that the property set developed for a smaller width can easily be extended to a larger one. Note that the AMBA protocol also requires a bus arbiter unit to ensure that only one bus master can use the bus for transfer operations. Since each implementation may have a different arbitration algorithm, each AMBA arbiter implementation may require a unique set of properties to be written. Verifying AMBA arbiters is beyond the scope of this paper, although arbiters are an ideal application of SFV (can exhaustively prove some standard properties e.g. only one requester is granted at a time).

From the perspective of verification, we classify the rules into five types. The protocol specification will typically provide compulsory rules, recommended rules and optional rules. These are the direct property equivalents of the English language specification as given in the example in Figure 1. To ensure that the correct subset of these is applied to the design we also need configuration check rules and converse rules. The configuration checks are used to demonstrate which optional parts of the AMBA specification are implemented in the design. Finally, the converse rules are used to prove which optional

parts of the specification are not implemented in the design. The number and classification of these rules for AHB and AHB-Lite is shown in Table 1 below.

Protocol	# of protocol rules	# of configuration checks	# of converse rules
AHB Master	50	20	30
AHB Slave	13	4	4
AHB-Lite Master	33	20	30
AHB-Lite Slave	8	2	2

Table 1: Number of properties for each subset of the AHB

Methodology

Our methodology consists of 5 parts: choosing which block to verify, choosing the subset of the protocol rules which should be used for verification, mapping design signals to protocol signals, verifying the design, and performing simulation (for failing or inconclusive properties). Each is explained in the following sub-sections.

Choosing the Appropriate Block For Verification

The first step is determining which block should be used for verification.

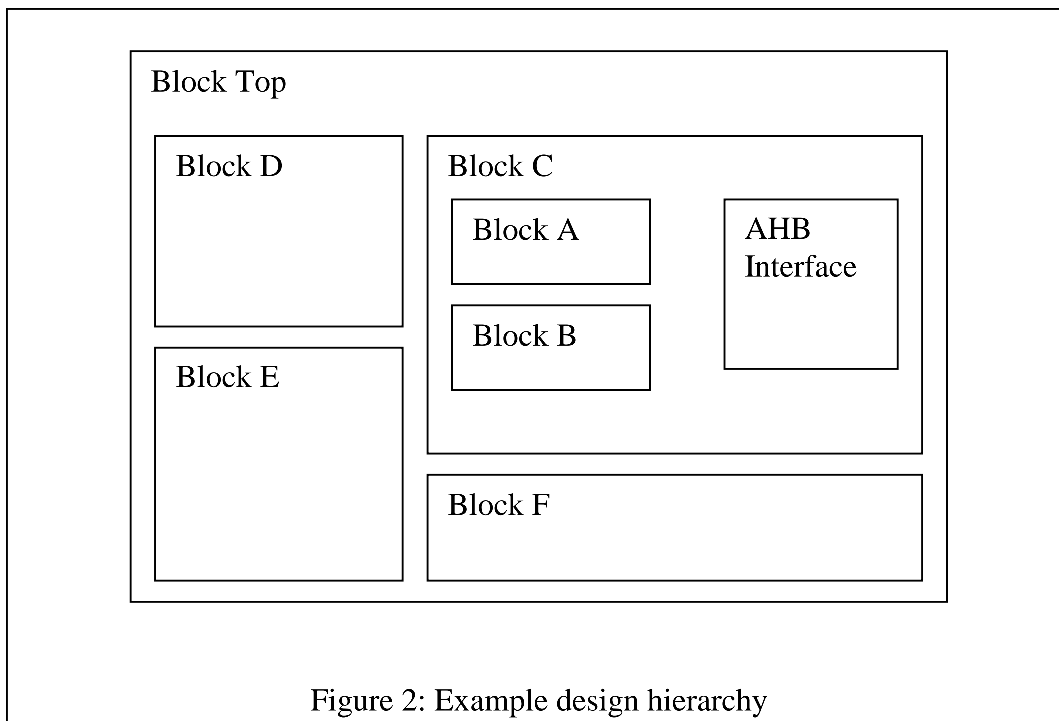


Figure 2: Example design hierarchy

Choosing just the sub-block which implements the bus interface unit can be good because it allows for much faster verification run times. For the example design in Figure 2, just the AHB interface block could be given to the tool. However this can be problematic because the verification tool will then consider all input combinations/sequences – including some that may never occur in the environment in which the block is instantiated. In the example, the inputs to the AHB interface from Block A and Block B are now unrestricted. So although verification will be fast, interface false negatives may occur (due to the inputs taking values that are not possible in the context of the complete design). Interface false negative problems can be mitigated by specifying assumptions on the inputs. These input assumptions (which can be expressed in PSL using the `assume` construct) specify a sub-set of the legal values which can appear at the design inputs. Of course, using these assumptions creates a proof obligation at a higher level of the hierarchy (i.e. you must eventually prove that the specified input assumptions are correct).

Going further up the hierarchy reduces the number of false negatives and the need to restrict the AHB interface block's inputs, since more of the driving logic is now seen by the tool. For example, choosing to verify at the level of Block C will reduce the illegal inputs the AHB interface receives from Block A and Block B (since they are now included). However, inputs of Block A and B are now unrestricted - which may result in these blocks generating output patterns which would not occur had Blocks D and E been visible. So although the false negatives are reduced, they are not eliminated.

By choosing to run at the top-level of the design, false negatives are minimized but the run time is significantly increased (and additional complexities from multiple clock domains can occur). The optimum point of attack is design dependant, typically smaller designs are best run at the top-level and large designs are best run at the block-level.

At the top-level, all input assumptions should be found in the documentation for the device. Entering the assumptions from the documentation is a great way of checking both the design and its documentation in one go.

Choosing The Protocol Subset

The AMBA specification does not require all features of the protocol to be implemented, so some portions of the protocol rule set do not have to be verified for a particular design. For example, if the master does not initiate a particular class of data transfers, then the part of the specification that is involved in that type of transfer does not have to be implemented. When protocol rules are omitted because the relevant behavior is believed to be absent, it is important to verify that the untested behavior can never happen. A well-designed protocol verification tool will track these dependencies and automatically try to prove the absence of untested functionality. For example, the AHB protocol specification allows for BUSY cycles: that is, it allows for a master that has initiated a transfer to insert idle time in the middle of the transfer. There are actually 7 protocol rules associated with how BUSY cycles are handled. These rules, however, can all be disabled if it can be proven that the particular master (under verification) never generates BUSY cycles.

Performing the Signal Mapping

Once the AMBA protocol rules have been encoded into a set of PSL properties, they can then be used to verify different designs that implement the AMBA protocol specification. The AMBA protocol specification does not require the names of the signals appearing in the design to be named in any particular way, so in order to apply the rules to different designs a mapping has to be provided between the signals appearing in the design and the signals appearing in the AMBA protocol specification. There are actually 15 signals that need to be mapped for an AHB Master. To make the portability of the property set easier from design to design, the PSL properties can be written as named properties that can be parameterized (where the formal parameters correspond to the protocol signal names). The properties can then be instantiated using the design specific signals. The complete list of protocol signals for each AHB bus master is given in Table 2, along with a brief description for each.

Protocol Signal Name	**I/O**	**Description**
`HCLK`	input	Bus clock
`HGRANT`	input	Bus grant
`HRDATA`	input	Read data bus
`HREADY`	input	Indicates transfer completed from Slave
`HRESETn`	input	Bus reset (active low)
`HRESP`	input	Transfer response from Slave. Can be: `OKAY`, `ERROR`, `RETRY`, or `SPLIT`.
`HADDR`	output	Address bus
`HBURST`	output	Burst type
`HBUSREQ`	output	Bus request signal
`HLOCK`	output	Indicates locked bus access
`HPROT`	output	Protection control
`HSIZE`	output	Transfer size
`HTRANS`	output	Transfer type. Can be: `NONSEQ`, `SEQ`, `IDLE`, or `BUSY`.
`HWDATA`	output	Write data bus
`HWRITE`	output	Indicates a write transfer

Table 2: AMBA AHB Protocol Signal List

In addition to the signal mapping, the user also needs to provide the SVF tool with an expression that denotes how the design is reset. If only the sub-block that implements the AHB protocol is verified in isolation, then that expression is easy: the protocol specification only requires that `HRESETn` be asserted low. However, if the protocol is verified at a higher level of the hierarchy, then it may be necessary to specify a more

complex reset sequence. Finally, legal input conditions to the block may also need to be specified. This, of course, will be useful for reducing potential false negative problems.

Verifying The Design

Once the appropriate subset of the protocol rules has been chosen, the next step is to actually run the properties in the SVF tool. Results fall into one of three classes:

- **Exhaustive**: the property is true for all legal input sequences. Even though verification is exhaustive, a proof can pass in seconds/minutes rather than hours/days.
- **Failed**: the property failed from a reachable design state (i.e., `N` clock cycles from reset). An SVF tool should provide a counter example that shows how the design can violate the property (such as an input sequence in a VCD file).
- **Inconclusive**: the property cannot be shown to exhaustively pass, or fail from a reachable design state. Properties in this class might be partially proven e.g. no failure for all possible input sequences within `N` cycles from reset.

Verification Fails

When verification of one of the PSL properties fails, the part of the design being verified may violate one or more of the protocol rules. The violation is illustrated with sequence of input vectors that reset the design and then go on to make it violate a protocol rule. The failure could be due to an actual bug in the design or an interface false negative that occurred because some illegal input pattern has been used.

Most interface false negatives occurring in AHB master/slave protocols can be systematically eliminated. This is because most AHB inputs to a master will come from a slave, and must comply with the AHB protocol rules for a slave. Similarly most inputs to a slave will come from a master, and will comply with the AHB protocol rules for the master. Hence having developed a set of master rules, a straightforward translation will produce a set of slave input assumptions, and visa versa for the master input assumptions. With the extra restriction in place, the formal tool may be able to exhaustively prove that the design now always works, or may simply return another way of breaking it. Either case yields useful information.

Inconclusive Verification

Inconclusive verification results occur when the SVF tool is unable to prove or disprove the property, due to either:

- proof is too complex and the tool has reached some resource limit (CPU time/memory).
- a counter example (starting from reset) cannot be found.

Our experience over a wide range of designs has been that about 5% of AHB protocol rules fall into this category. Partial proof results (e.g. could prove a property holds for the first 25 cycles from reset) is useful, but not conclusive, information.

Simulation Integration

For properties that are inconclusive, dynamic (rather than static) verification can be employed to gain confidence in design correctness. RTL simulation is typically used to partially verify the property from the top level. Indeed, the advantage of writing the protocol rules using PSL is that the properties that are written for SFV can be reused in a simulation environment (or vice-versa, i.e. bug-hunt with simulation prior to SFV in order to remove low-hanging fruit). While simulation can never guarantee the absence of a bug, it is good at bug-hunting and passing tests will give some confidence that the design is AHB compliant.

Protocol Rule Types and Examples

Using formal techniques we can find out more about a design than simply a yes/no on the question of compliance. As described earlier, there are 5 distinct classes of rules that can be usefully applied to a design, with each class revealing a different type of information:

- **Compulsory** rules to establish compliance.
- **Recommended** rules to establish good design practice.
- **Optional** rules to verify non-essential functionality.
- **Converse** rules to confirm the protocol compliance checker is correctly set up for the design being tested and allow confidence in the results.
- **Configuration** rules to establish the capabilities of the design being tested.

We will now look at each of these types in more detail and consider an example of each.

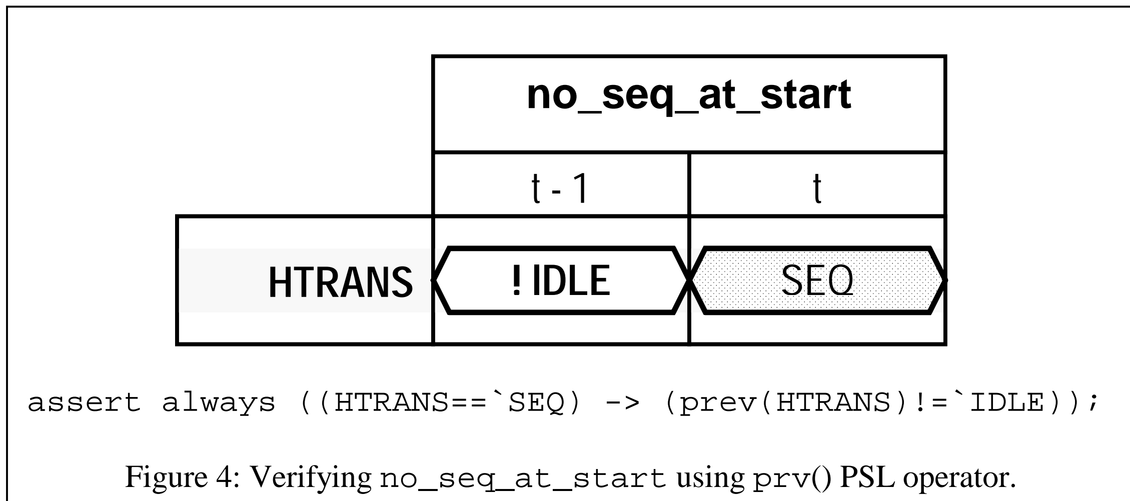
Compulsory

These rules are the first things to be considered when creating a formal compliance test. The goal here is to prove the design cannot violate all the required behavior defined by the protocol. All the compulsory rules must be run. If any of these rules fail verification, then one must assume that the design contains a serious flaw and is likely to either fail itself or cause other devices in the system to fail. An example of such a rule is given in Figure 3.

```
                    no_seq_at_start
Sequential transfers cannot follow IDLE, since the
first transfer of any burst must be non-sequential.
```

Figure 3: No Sequential at start of transfer (see AHB 2.0, Section 3.5)

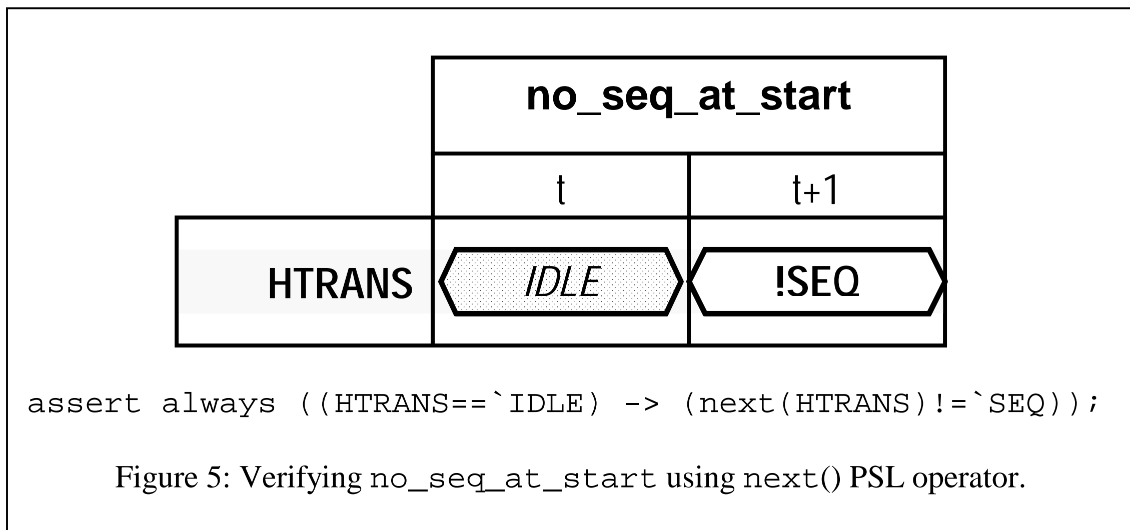
We can prove this rule in one of two ways. First, if the current cycle (time `t`) is `SEQ`, then previous cycle (time `t-1`) cannot be `IDLE`. The PSL property for this rule is given in Figure 4. The temporal operators for the PSL property in the figure – as well as throughout the paper - are all with respect to the positive edge of `HCLK`. Note that the figure also contains a timing diagram to illustrate the property: *if* the conditions (in italics + dotted) occurs *then* the requirements (in bold) must be proven.




```
assert always ((HTRANS==`SEQ) -> (prev(HTRANS)!=`IDLE));
```

Figure 4: Verifying `no_seq_at_start` using `prv()` PSL operator.

Another equally valid property would be: if the current cycle is idle, the next cycle cannot be sequential. This is shown in Figure 5.



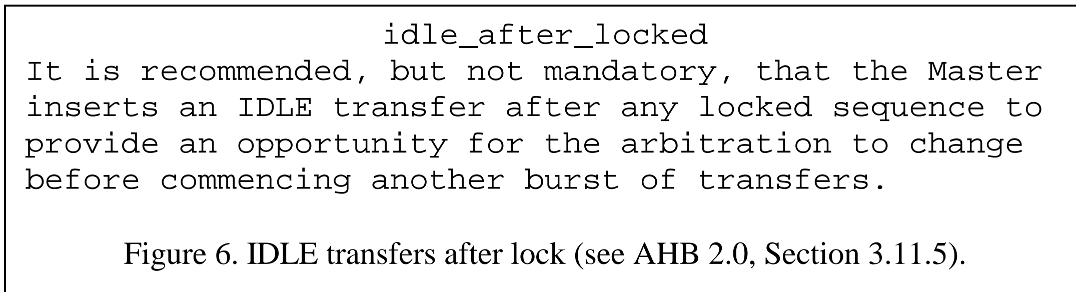

```
assert always ((HTRANS==`IDLE) -> (next(HTRANS)!=`SEQ));
```

Figure 5: Verifying `no_seq_at_start` using `next()` PSL operator.

Designs violating this rule lose any possible claim of AHB compliance.

Recommended

These rules can establish the design follows accepted good practice. Such recommendations might be concerned with minimizing power consumption, improving performance, or improving tolerance of the design to erroneous behavior from other devices. A failed recommendation means the design could be improved, but it is not a breach of the protocol and the design is still compliant: it can still be used and it can be expected to work. An example of such a rule is given in Figure 6.

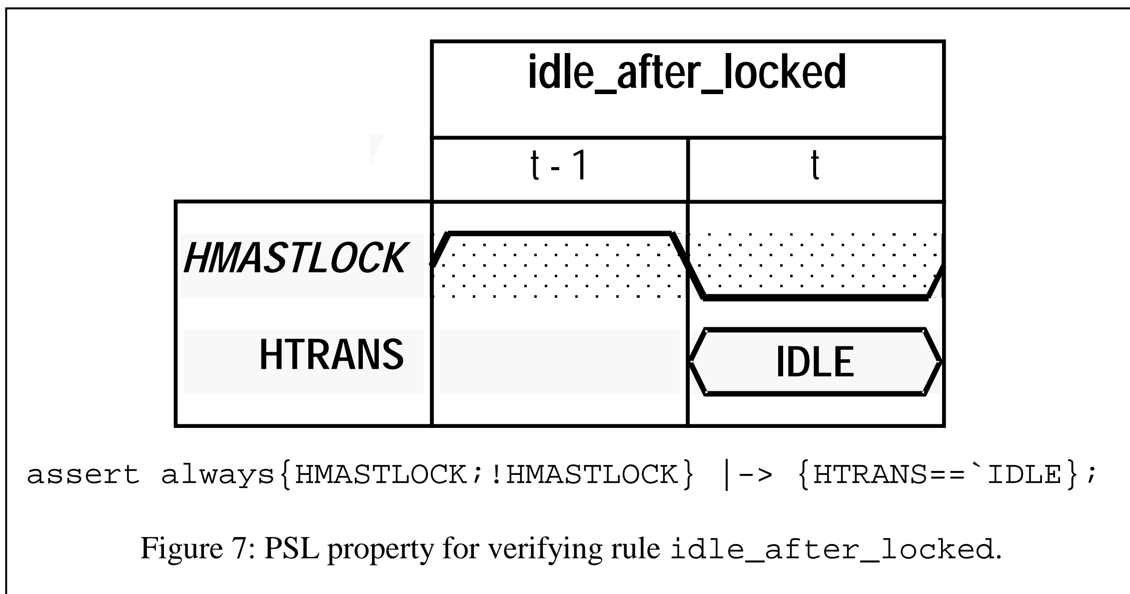


`idle_after_locked`

`It is recommended, but not mandatory, that the Master inserts an IDLE transfer after any locked sequence to provide an opportunity for the arbitration to change before commencing another burst of transfers.`

Figure 6. IDLE transfers after lock (see AHB 2.0, Section 3.11.5).

For AHB-Lite, we aim to prove this relationship using the property given in Figure 7.




```
assert always{HMASTLOCK;!HMASTLOCK} |-> {HTRANS==`IDLE};
```

Figure 7: PSL property for verifying rule `idle_after_locked`.

Optional

Most protocols contain some sort of optional functionality. This is most common for bus masters, where a number of different types of transfer could potentially be used to achieve the same result. The designer of a bus master may choose to omit some functionality corresponding to optional parts of the protocol he or she does not intend to use without loss of compliance. An example of such a rule is given in Figure 8.

```
align_2byte
Transfers of 2 byte words must be to even addresses.
```

Figure 8: Two byte address alignment (see AHB 2.0, Section 3.6).

This rule can be proved using the PSL property given in Figure 9.

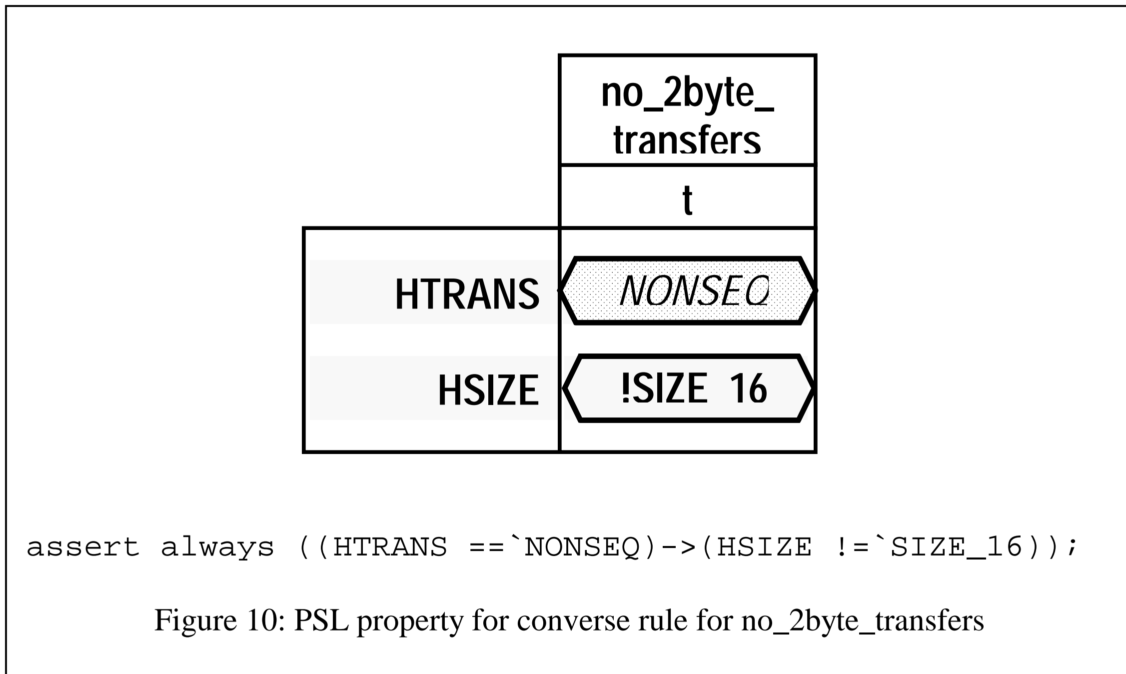
align_2byte
t
HTRANS !IDLE
HSIZE 2
HADDR[1:0] 0

```
assert always (((HTRANS!=`IDLE) && (HSIZE==SIZE_16))
                    -> (HADDR[0]==1'b0));
```

Figure 9: PSL property for `align_2byte`

Converse Rules

Now consider an AHB master that is only intended to perform 32-bit transfers. In this case, the rule given in Figure 9 would be redundant and could be disabled. The existence of optional rules, however, opens up a hazard with formal compliance checking. Suppose, for example, that the `align_2byte` rule is turned off during compliance checking, since the design is only intended to be used for 32-bit transfers. If, however, in some unusual circumstance, 16-bit transfers are actually produced by the design, then that feature would be used even though it was never verified. Simply switching off the test for byte alignment is not enough as it would allow the possibility of a broken design passing. To avoid this possibility we need a converse rule for each optional rule. The purpose of a converse rule is to prove the optional functionality, whose check has been disabled, is genuinely absent. An example the converse of `align_2byte` is `no_2byte_transfers`, which is shown in Figure 10.



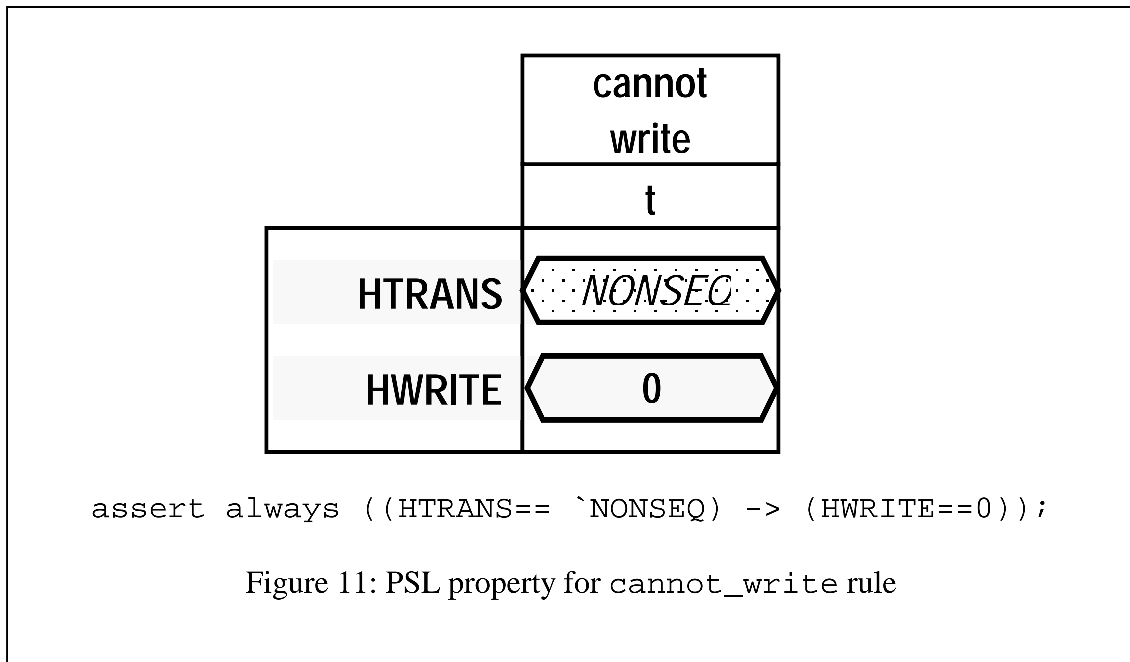

```
assert always ((HTRANS ==`NONSEQ)->(HSIZE !=`SIZE_16));
```

Figure 10: PSL property for converse rule for no_2byte_transfers

Configuration Checks

The final class of rules is not intended to be part of the compliance test at all, although they can help with checking the correct configuration of the compliance checker when dealing with optional functionality. Configuration checks exist to confirm the engineer's understanding of the type of design being verified and can be used to help reduce the possibility that a given design is instantiated in the wrong environment.

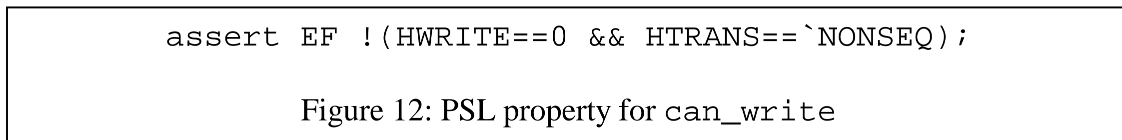
As an example, suppose a data retrieval device is being developed which should only ever read data from a system. It would be reassuring to the designer if it could be confirmed that the device will never initiate a write transaction. Such a reassurance would be good because it provides a guarantee that the device's presence could not interfere with the integrity of the data in any way. The rules given in Figures 11 and 12 could be used to clarify such a situation for the designer. For any design, one should pass and one should fail.




```
assert always ((HTRANS== `NONSEQ) -> (HWRITE==0));
```

Figure 11: PSL property for `cannot_write` rule

Note that the property in Figure 12 uses a CTL operator to check that there exists a path where a write occurs. Currently, most simulators do not support the optional branching extensions of PSL.



```
assert EF !(HWRITE==0 && HTRANS==`NONSEQ);
```

Figure 12: PSL property for `can_write`

Conclusions

The goal of this paper is to introduce the reader to SFV (Static Functional Verification) technology and show how it can be used to verify functional blocks that implement bus protocols such as AMBA. The primary advantage of SFV is that the design's functionality can be 100% exhaustively proven against a given specification (i.e. it can show that a design is bug-free for *all* legal input combinations/sequences). Dynamic verification cannot give such a guarantee: it can only ever claim “no known bugs”.

Most SoC designs today are still only verified dynamically, mainly via RTL simulation of the design in a test-bench. Changing to SFV requires a change in methodology on the part of the designer and/or verification engineer. This is particularly daunting to engineers who have been trained to think in terms of a simulation mindset and have little experience in writing properties or assertions. Verifying bus protocols is one verification task that can be easily performed using SFV technology - requiring very little change in the methodology and no experience with formal verification.

References

[1] Mike Turpin, “The Dangers of Living with an X (bugs hidden in your Verilog),” Boston SNUG, 2003.
[2] Averant Solidify Development Team, *Functional Static Verification of RTL Designs*, www.averant.com, June 10th, 2002.
[3] “AMBA™ Specification Rev 2.0”, ARM Limited, 1999.
[4] “Property Specification Language Reference Manual Version 1.1,” Accellera, June 9, 2004.
[5] Edmund Clarke, Orna Grumberg, and Doran A. Peled, *Model Checking*, The MIT Press, 2000.
[6] T. Kropf, *Introduction to Formal Hardware Verification*, Springer-Verlag, 2000.
[7] Harry Foster, Adam Krolnik and David Lacey, *Assertion-Based Design*, Kluwer Academic Publishers, 2003.
[8] Erich Marschner, Bernard Deadman, and Grant Martin, “IP Reuse Hardening via Embedded Sugar Assertions,” International Workshop on IP SoC Design, October 30, 2002.
[9] “Open Verification Library Users Reference Manual,” Accellera, 2003.
[10] ”SystemVerilog 3.1,” Accellera, April 2003