

Nokia Research Center  
Sami Maisniemi

6.10.2006

## はじめに

OCF は、Nokia で構築されたシステムの標準相互接続アーキテクチャとして採用されています。今のところ、相互接続を検証する手法として統一されたものはありません。その結果、次の要件を満たす新しい検証パッケージを要望する声が上がっていたことは明らかです。

- OCF 標準との相互接続の互換性を検証する
- 既存の検証環境を変更せずに最小の労力でインストールできる
- ソースコードにアクセスせずに最小の労力で構成できる
- 既存の設計フローや検証フローと互換性がある
- アサーションベースの検証またはハードウェアプロパティ言語に関する予備知識が不要である
- 追加の EDA ツールが不要である
- 保守およびサポートの点から見て費用効果が高い

アサーションベースの検証 (ABV) の目的は、時間経過に沿ったロジック動作を定義する明確なアサーションに、仕様のファンクショナルな機能を変換することです。ハードウェアプロパティ言語は、ハードウェアの記述言語や検証言語とはまったく異なります。このハードウェアプロパティ言語は、テストベンチを実装するのではなく、アサーションを実装するために使用されます。プロパティ仕様言語 (PSL) は、ベンダーに依存しないハードウェアプロパティ言語で、IEEE によって標準化されています。PSL の目的は、VHDL や Verilog のテストベンチなどの既存の検証手法を置き換えるのではなく、補完することです。また、検証の質と効率を高めることを目指しています。

## IMPLEMENTATION OF THE VERIFICATION PACKAGE

### Characteristics of OCF Interconnections

The signals of the OCF can be classified to request, response, and data handshake phases. The timing characteristics of the phases are explicitly defined by the standard. Assertions are almost similar for all signals that belong to the same phase. As a consequence, each phase is verified independently. Similar rules can be applied to any signal of the OCF excluding the sideband and the test signals.

Let us consider a simple example (figure) that contains request and response phases. The timing characteristics of the OCF can be represented as a FSM. The initialization of the FSM leads to the idle state. The request phase starts when the master command (MCmd) is unequal to idle (either write or read). Then all signals of the request phase must be asserted and held stable until the request phase is terminated. The request phase is terminated when the slave command accept (SCmdACcept) is asserted. The response phase is activated when the slave response (SResp) is unequal to null (either data valid or error). Then all signals of the response phase must be asserted and held stable until the response phase is terminated. The response phase is terminated when the master response accept (MRespAccept) is asserted.

The data handshake phase allows the de-coupling of the master address (MAddr) from the master data (MData). The data handshake phase becomes active when the master data valid (MDataValid) is asserted. The data handshake is terminated when the slave data accept (SDataAccept) is asserted. Note that the data handshaking might change behavior of other signals as well. For example, the master data info (MDataInfo) is moved from the request to the data handshake phase if the data handshaking is applied.

### Implementation of the Assertions

Nokia Research Center  
Sami Maisniemi

6.10.2006

Assertions always operate as hardware monitors i.e. assertions are passive components that do not alter the behavior of the module. The assertions of the package monitor the OCP interconnection to verify that the OCP interconnection is compatible with the OCP standard. The target of the methodology is to implement multiple simple assertions, which debugging requires only moderate efforts. In other words, multiple simple assertions are preferred to a single complicated assertion.

The PSL is very consistent and simple hardware property language. Its syntax is very declarative and structural which leads to sustainable verification environments. Let us consider the basic components of the PSL. Boolean expressions can be used to determine a state of the module. If the Boolean expression is false, a violation has occurred. Sequential expressions can be used to determine a sequence of states of the module. Sequential expressions consist of a precondition and a consequence, which are bounded by an implication operator. If the precondition is not followed by the consequence, a violation has occurred. Both illegal (never) and legal (always) states and sequences of states can be defined. Since interpretation of never statements can become very complicated, it is recommended to use never statements only in case of Boolean expressions. In most cases assertions are assumed to hold under certain conditions. For example, clocking, initialization, and test modes are usually verified separately. This kind of specific scenarios can be omitted from verification utilizing an abort statement. It is also possible to define reusable sequences that can contain both Boolean and sequential expressions. For example:

```
property EXAMPLE is
  never {BOOLEAN EXPRESSION};

property EXAMPLE is
  always ({BOOLEAN EXPRESSION or SEQUENCE} | =>
    {BOOLEAN EXPRESSION or SEQUENCE})
  abort (Reset = ACTIVE);

sequence EXAMPLE is {BOOLEAN EXPRESSION or SEQUENCE};
```

The Boolean layer of the PSL provides several built-in functions that were found very useful. For example, *stable(std logic vector)* can be used to verify stability and *isunknown(std logic vector)* can be used to verify logical states of signals. Also built-in functions *onehot(std logic vector)* and *countones(std logic vector)* were found useful. Built-in functions provide the optimal implementation in terms of simulation performance.

The OCP standard is a time unbounded protocol i.e. latencies between different phases have not been defined. For example, let us consider the following property of the OCP protocol: "The master request must be followed by the slave command accept". The latency between the master request and the slave command accept is not defined. The time bounded version of the property would be: "The master request must be followed by the slave command accept within n clock cycles". In terms of performance, the time unbounded protocols increase overall throughput. However, the time unbounded protocols also increase verification efforts remarkably, because it is difficult to define explicit timing relationships between different phases. As a consequence, several assertions of the package have been equipped with an infinite repetition operator. However, each transaction must be terminated properly and therefore each assertion is assumed to terminate before the end of simulation. As a consequence, each assertion containing an infinite repetition operator is implemented as a strong property.

Every sequential assertion was equipped with an abort statement which contained at least the asynchronous master reset. The abort statement is always applied to the entire assertion. If possible every sequential assertion was implemented as a strong assertion i.e. the assertion leads to a violation at the end of simulation if the consequence did not occur during the simulation. The exclamation mark indicates a strong property. For example:

```
property MCMD_ISUNKNOWN is
  never {isunknown(MCcmd)};

property MCMD_STABLE is
  always ({MCcmd /= "000" and SCmdAccept = '0'} | =>
    {stable(MCcmd) and SCmdAccept = '0' [*]};
```

Nokia Research Center  
Sami Maisniemi

6.10.2006

```
    Stable(MCmd) and SCmdAccept = '1'!)  
    abort(MReset_n = '0');
```

```
assert MCMD_ISUNKNOWN;  
assert MCMD_STABLE;
```

The package contains also a coverage model that provides some coverage points to guide the verification process. The purpose of the coverage points is to locate missing test cases of the verification plan. The coverage points are implemented with sequences. Since the coverage points are quite simple, only Boolean expressions are required. For example:

```
sequence MCMD_WRITE is {MCmd = "001" and SCmdAccept = '1'};  
  
cover MCMD_WRITE;
```

The capabilities of the PSL were extended with VHDL functions and processes. VHDL functions can be utilized to implement complicated mathematical expressions. VHDL processes can be utilized to store values of signals, because the PSL does not provide equivalent structures. The VHDL functions were encapsulated to a separate package. The VHDL processes were embedded to the verification units. As a consequence, the assertions remain sustainable.

### Verification of the Assertions

A verification environment that contains master, slave, and monitor components was created. The verification environment creates a golden reference model (i.e. no violations are expected) that can be utilized to verify the assertions. Each operation type (WR, RD, RDEX, RDL, WRNP, WRC, and BCST) was verified independently. Each burst mode (INCR, DFLT1, WRAP, DFLT2, XOR, STRM, and UNKN) with applicable operation types (WR, RD, WRNP, and BCST) with random and predefined data including both imprecise and precise burst sequences were verified.

During the verification also the performance penalty was measured. The CPU times were calculated as a sum of CPU times of a predefined set of test cases which consisted of 6,340 independent requests and 75 burst sequences that contained 19,020 requests. The CPU time without the PSL assertions equals to 132.83s. The CPU time with the PSL assertions equals to 143.42s. The performance penalty equals to 8.0% that can be interpreted as very moderate.

### Implementation of the Installation Script

The package was implemented so that it can be installed, configured, and integrated to the existing verification environment with minimal efforts. Since the number of assertions and coverage sequences became very high, an installation script that installs and configures the package for the existing verification environment was created (figure). The installation script that was implemented with Perl utilizes a standard OCP RTL configuration. A separate package is generated for each interconnection and the total number of interconnections is not limited. The user only implements a standard OCP RTL configuration according to the module. There is no need to access the original source code of the package. The installation script operates as follows:

- The OCP RTL configuration is processed and verified
- All VHDL processes are configured
- All assertions and coverage points are configured
- All assertions and coverage points that contain unused signals are removed
- The entity and architecture pair is set
- Packages are generated for each interconnection

### UTILIZATION OF THE VERIFICATION PACKAGE

Nokia Research Center  
Sami Maisniemi

6.10.2006

The package does not require any additional verification efforts. There is no need to modify the existing verification environment. Violations are generated automatically by the simulator. The OCP RTL configuration provides a simple and flexible method to define the interfaces of the module. For example:

```
module module_name {  
  
    # The name and bundle type of the interface  
    interface interface_name bundle ocp2 {  
  
        # The type of the interface  
        interface_type monitor  
  
        # The name of the architecture  
        proprietary 0x4E6B Nokia {  
            architecture architecture_name  
        }  
  
        # The parameters  
        param write_enable 1  
        param read_enable 1  
  
        param mreset      1  
        param addr        1 {width 32}  
        param mdata       1 {width 64}  
  
        ...  
  
        # The signal map  
        port ocp_clk      net clk  
        port ocp_MReset_n net MReset_n  
        port ocp_MCmd     net MCmd  
        port ocp_MAddr    net MAddr  
        port ocp_MData    net MData  
  
        ...  
    }  
  
    ...  
}
```

The package must be integrated to the verification environment so that all signals of the OCP interconnection can be accessed by the package (figure). The assertions operate as monitors i.e. the assertions are passive components that are not able to alter the behavior of the module. As a consequence, the assertions can access any signals regardless of the type of the signal (input, output, or buffer). However, the VHDL processes of the package are active components i.e. they are able (in theory) to alter the behavior of the module. As a consequence, the VHDL processes cannot read inputs! Therefore the package cannot be integrated to the module or the test bench directly. The package should be integrated to the top level component that connects the module to the test bench. In this case the package is able to access the signals of the OCP interconnection. If it is not possible to access the signals of the OCP interconnection directly, the user must implement auxiliary signals.

Debugging with assertions does not differ remarkably from traditional debugging. Actually assertions decrease debugging efforts. A violation is generated immediately after an error has occurred. As a consequence, the source of the error can be located with minimal efforts. In most cases hardware simulators provide excellent debugging capabilities. For example, it is possible to view the current status of the assertion and all signals that are used by the assertion.

## SUMMARY

Nokia Research Center  
Sami Maisniemi

6.10.2006

The package provides a flexible, powerful, and automatic verification method that complements the existing verification methodologies. The package improves both quality and efficiency of the functional verification by providing automation, simplifying debugging, and increasing visibility.