OCP Profiles and Transactions

**OCP uVC – Verification Component**

Mark Litterick

Sept 2008
Contents

1. Overview ............................................................................................................................................................ 1

2. OCP Profiles .................................................................................................................................................. 1
   2.1 Profile Specification ....................................................................................................................................... 2
   2.2 Profile Usage ............................................................................................................................................. 4

3. OCP Transactions ......................................................................................................................................... 5
   3.1 Transaction Encapsulation ............................................................................................................................ 5
   3.2 Transaction Generation ................................................................................................................................ 6
   3.3 Transaction Monitoring ................................................................................................................................ 7

4. Summary ........................................................................................................................................................ 9

5. Information .................................................................................................................................................... 9
1. Overview

Architecting effective verification components for something as flexible as the Open Core Protocol requires a detailed understanding not only of the corresponding specification, but also the environments and methodology in which the verification component will be used. This article demonstrates how two key aspects of OCP — profiles and transactions — were adopted as fundamental building blocks for the architecture of a verification component targeted at constrained-random validation of OCP components and systems.

The article uses the Verilab OCP uVC as an example. The Verilab OCP uVC is a mixed language OCP compliant verification component that supports a major subset of Open Core Protocol Specification 2.2. The OCP uVC is implemented using SystemVerilog and e verification languages and is compliant with both the Open Verification Methodology (OVM) and e Reuse Methodology (eRM). The verification component can be used in SystemVerilog only applications without the Specman layer (or license), or it can be used in Specman-based verification environments as a regular eVC.

2. OCP Profiles

One of the key strengths of the OCP protocol is its configurability. It is this configurability that allows it to be efficiently applied to a wide variety of SoC interface requirements such as register programming, streaming data applications or block-data burst transfers. The configurability is achieved in OCP by means of many configuration parameters which affect all aspects of bus behavior, from signal width through to communication protocol. In order to ensure that this flexibility does not lead to interoperability challenges, OCP introduces the concept of profiles into the specification guidelines. A profile is a predefined set of OCP configuration parameter values that constitute a coherent subset of OCP interface behavior with well-defined communication characteristics. Figure 1 illustrates a simplified example, showing read transfers for three profiles defined to characterize register transfer (REG), streaming data (STREAM) and single-request/multiple-data burst (SRMD_BURST) interfaces:
Profiles have been adopted as a cornerstone for the OCP uVC architecture. All aspects of the verification component behavior are profile aware including constraints, coverage, checks, signal connections and of course functional behavior. The profile infrastructure is engineered to be simple, single-sourced and maintainable, ensuring vertical and parallel reuse throughout many projects and design derivatives.

2.1 Profile Specification

The greatest benefits in applying a profile-based approach to OCP based projects are realized by generating a limited library of profiles with minimal user extendibility. This library is then reused for all OCP development within the team, with additional profiles only being added when new interface characteristics are required. Typically there are 3 or 4 profiles per project and 10 to 12 profiles in total, depending on the application domain.
User extendibility should be limited to signal widths, in-band or side-band signal inclusion and features which do not fundamentally affect the OCP protocol communication flow for the specific interface instance.

In the uVC, example the profile library and user extensions are all derived from a single base-class containing all of the OCP configuration parameters. This base-class can be in e, if a Specman layer is present, or in SystemVerilog otherwise. Figure 2 illustrates the simplicity of the base-class definition, specification of profile library, and some user extensions for the eVC use-case (the SystemVerilog solution is similar). Profile definitions can also be generated directly from OCP Core RTL Configuration Files (and vice-versa).

```verilab
// base-class - verilab code
struct profile {
  env_name : env_name_t;
  profile_name : profile_t;
  // declare <80 parameters
  // no constraints in base-class
  addr : bit;
  addr_width : uint;
  burstlength : bit;
  data_width : uint;
  reqinfo : bit;
  // ...
};

// library profiles - client central code
extend profile_t : [REG];
extend REG profile {
  // constrain all parameters
  keep addr == 1;
  keep soft addr_width == 32;
  keep burstlength == 0;
  keep soft data_width == 256;
  keep reqinfo == 0;
  keep tags == 0;
  // ...
};

extend profile_t : [STREAM];
extend STREAM profile {
  // constrain all parameters
  keep addr == 0;
  keep addr_width == 0;
  keep burstlength == 0;
  keep soft data_width == 256;
  keep reqinfo == 0;
  keep tags == 0;
  // ...
};

extend profile_t : [SRMD];
extend SRMD profile {
  // constrain all parameters
  keep addr == 1;
  keep soft addr_width == 32;
  keep burstlength == 1;
  keep soft data_width == 256;
  keep reqinfo == 0;
  keep tags == 256;
  // ...
};

// user profiles - project specific code
extend REG profile {
  // constrain soft parameters
  keep addr_width == 13;
  keep data_width == 32;
};

extend STREAM profile {
  // constrain soft parameters
  keep addr_width == 128;
};

extend SRMD profile {
  // constrain soft parameters
  keep addr_width == 32;
  keep data_width == 16;
  keep tags == 64;
};
```

Figure 2 - Profile Library Hierarchy in e
2.2 Profile Usage

Each instance of the OCP uVC is individually bound to a specific profile during construction phase of the verification environment. For the eVC use-case the generated e profile is automatically copied to the SystemVerilog side and the profiles are validated using the OCP Configuration Compliance Checks. The propagation of the profile information throughout the mixed language OCP uVC architecture is illustrated in Figure 3.

Profiles encapsulate the vast majority of the configuration information for the verification component instance. The only additional requirements are the operating mode (master/slave/monitor, system/core), and optional model behavior outside scope of OCP (e.g. random or inactive values when not in use, automatic reset generation, forced response ordering, pipeline depth emulation, etc). The benefits of encapsulating and propagating the configuration information in a profile object are clear when you consider that all of the following aspects of OCP uVC behavior are profile aware and the constraining of the profile is performed in a single statement (e.g. `keep profile_name == strm_profile;`):
3. OCP Transactions

In Chapter 4 of the OCP specification, the protocol semantics are discussed in some detail – in particular, a transaction consists of one or more transfers, and a transfer consists of one or more phases – for example the definition of a burst is:

“**A burst is a set of transfers that are linked together into a transaction having a defined address sequence and number of transfers.**”

Getting the transaction object encapsulation right is a key factor in the efficiency and usability of a sequence-driven constrained-random verification environment. By choosing multi-transfer transactions as the basic object for stimulus generation and protocol monitoring, instead of single-transfers, we have ensured that the user interface for the OCP uVC is both natural and powerful.

3.1 Transaction Encapsulation

This transaction encapsulation is achieved primarily by the use of *lists* for all fields that can be affected by having multiple transfers in the same transaction object. Of course a transaction with a single transfer (for example a non-burst register read), simply has a corresponding list length of 1 for the appropriate fields. An extract from the basic transaction object definition is shown in Figure 4:
3.2 Transaction Generation

The real power of encapsulating the transactions in this manner becomes more apparent when the user interface is considered. For instance, for a proactive master instance all of the relevant master field values (e.g. MCmd, MBurstLength, MData for write), and delays (e.g. request latency, response-accept latency), for the entire burst transaction are generated in a single step and can be constrained together. Likewise, for a reactive slave instance, the entire response transaction is generated in a single step and contains all of the relevant slave field values (e.g. SResp, SData for read), and delays (e.g. command-accept latency, response latency), for every beat of the transfer. Figure 5 illustrates how straightforward it is to constrain the response latency for the first beat of the burst to be slower than that for all subsequent beats for a slave instance of the OCP uVC:

![Figure 4 - Transaction Object Definition in e](image)

<table>
<thead>
<tr>
<th>vlab_ocp_transaction</th>
<th>name</th>
<th>vlab_ocp_env_name_t;</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_ocp_profile</td>
<td></td>
<td>vlab_ocp_profile;</td>
</tr>
<tr>
<td>%m_id</td>
<td></td>
<td>uint;</td>
</tr>
<tr>
<td>%m_cmd</td>
<td></td>
<td>vlab_ocp_command_t;</td>
</tr>
<tr>
<td>%m_tag_id</td>
<td></td>
<td>uint(bits:TAG_WIDTH);</td>
</tr>
<tr>
<td>%m_data</td>
<td>list of</td>
<td>uint(bits:DATA_WIDTH);</td>
</tr>
<tr>
<td>%m_resp</td>
<td>list of</td>
<td>vlab_ocp_response_t;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>%m_req_accept_latency</td>
<td>list of</td>
<td>int;</td>
</tr>
<tr>
<td>%m_resp_latency</td>
<td>list of</td>
<td>int;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Figure 5 - Constraining Burst Response Latency

Note that while the field values are absolute, the transaction delays are affected by observed traffic on the OCP interface and interaction with the DUT. So, for instance, the response latency delays are not decremented until the previous response is accepted by the master in the above diagram. Also, zero-latency is supported for all delay parameters.

3.3 Transaction Monitoring

The same transaction object is also used in the passive monitor. In this case the relevant fields are updated for every phase of each transfer that makes up the transaction – in other words the observed transaction grows until it is complete. This process is illustrated in Figure 6, which shows overlapped read and write SRMD burst transactions. In each case the request information is complete at the end of the request phase, but the data and response lists grow with each beat of the burst. Therefore, it is easy to see where you are in a burst using m_data.size() – the transaction fields are all complete when the last (or in the case of write, only) response phase ends. A simple API allows the user to get_request(), get_response(), get_datahs() or get_transaction() in response to appropriate trigger events (req_phase_end, datahs_phase_end, resp_phase_end and transaction_end).
Figure 6 - Monitor Events and API
4. Summary

This article illustrates that by careful adoption of OCP profile strategy, and using a natural implementation for transaction encapsulation in the architecture of the Verilab OCP uVC, we have engineered a user-friendly and powerful verification component that enables efficient constrained-random verification techniques to be applied to OCP component and system validation.

It shows that profile-based configuration makes it easier to integrate OCP verification components, as the control can be directly mapped onto OCP profiles. This type of configuration allows for profile libraries to be developed for the verification environment, that correlate to the design profiles and are easier to maintain. Since stimulus, checking, monitoring, functional coverage and connectivity are profile aware in the OCP uVC, the profile configuration is a single point of control for configuration related behaviour.

We demonstrate the OCP transactions can be naturally encapsulated in the verification environment, allowing intuitive and powerful control of transaction and response generation, and logical interface monitoring.

5. Information

For additional information on the Verilab OCP uVC please contact ocp.info@verilab.com