OCP Channel Monitors

For OCP Channel Version 2.2

Produced by:
OCP-IP SLD Working Group

Original proposal by CoWare, Inc.

1 Contents

2 Revision History

Version	Date	Author	Comments
0.1	10/01/2004	Tim Kogel (CoWare)	Original text-based proposal
0.2	11/05/2004	Tim Kogel (CoWare)	incorporate working group feedback: system monitor support for transaction cancellation and multicast
0.3	02/09/2005	Tim Kogel (CoWare)	Major revision, include documentation for 1st release
1.0	02/19/2005	Anssi Haverinen (Nokia)	Approved for release.
2.1.2	02/24/2006	Tim Kogel (CoWare)	major revision, incorporate new monitor interface for TL1, TL2, and TL3 channels
2.1.3	9/4/2006	Tim Kogel (CoWare)	update section 3.3 with new untimed and clocked trace monitors for the TL1 channel
2.2	1/12/2007	Tim Kogel (CoWare)	Reorganization to separate user guidelines and implementation details

3 Introduction

The OCP monitor package is an additional library for the OCP SystemC channel package. The monitors record the transactions going over an OCP channel for the purpose of debugging and performance analysis. This document describes the usage and the architecture of the OCP monitors.

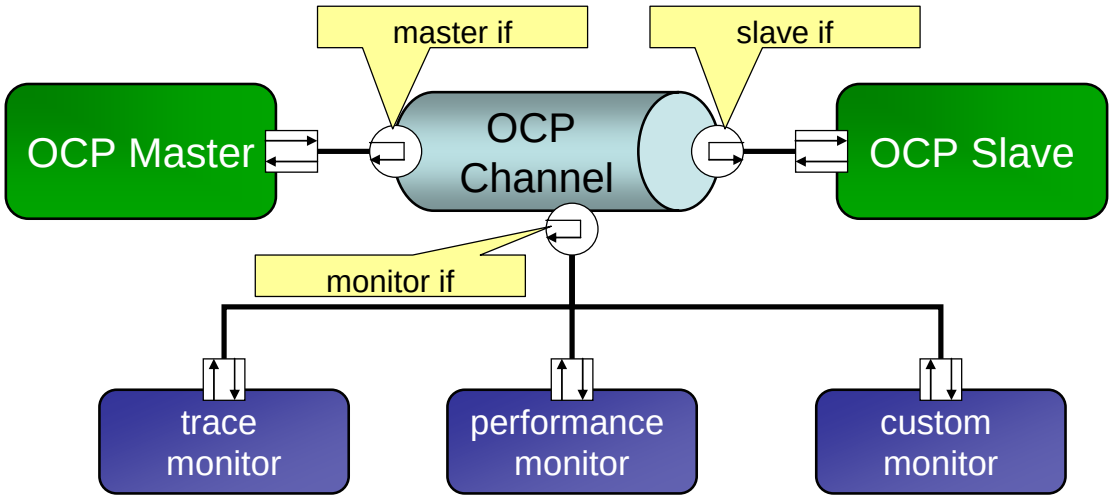

Figure 1: OCP Monitor Overview

The OCP monitors all follow the same basic architecture, which is depicted in Figure 1. In addition to the regular master and slave interfaces, all OCP channels provide a dedicated monitor interface. This monitor interface allows to observe the activity on the channel.

The OCP monitor package provides a trace monitor for the TL1 and TL2 channel as well as a performance monitor for the TL1, TL2, and TL3 channel. The trace monitor dumps an ASCII trace file with all the OCP transactions on a cycle-by-cycle bases according to the ocpdis2 format [3]. For the TL1 channel, the trace monitor comes in two flavors: the clocked trace monitor is connect to a clock signal, whereas the untimed trace monitor takes the clock period as a mandatory constructor argument. The performance monitor records transactions using the SCV transaction recording API [1].

Next to the provided trace and performance monitors, the monitor interface allows the creation of use-defined monitors. This way the user can generate specific debug and analysis views of the OCP traffic. The monitor interface allows to connect an arbitrary number and type of monitors to each individual channel.

3.1 General Use-model

The general use model of the OCP monitors is illustrated by means of a TL1 example depicted in Figure 2.

```
1     // OCP TL1 channel
2     typedef OCP_TL1_DataCl<OCPCHANNELBit32, OCPCHANNELBit32> data_type;
3     typedef OCP_TL1_Channel_Clocked< data_type >             clocked_channel_type;
4     typedef OCP_TL1_Channel_Untimed< data_type >             untimed_channel_type;
5     typedef OCP_TL1_Trace_Monitor_Clocked< data_type >       clocked_monitor_type;
6     typedef OCP_TL1_Trace_Monitor_Untimed< data_type >       untimed_monitor_type;
7     typedef OCP_Tl1_Perf_Monitor< data_type >                perf_monitor_type;
8
9     clocked_channel_type ch_clocked("ocp_clocked");
10    ch_clocked.p_clk(clk);
11    untimed_channel_type ch_untimed("ocp_untimed");
12
13    // clocked trace monitor
14    clocked_monitor_type tmon_clocked("tmon_clocked","ocp_clocked.trace");
15    tmon_clocked.p_mon(ch_clocked);
16    tmon_clocked.p_clk(clk);
17
18    // untimed trace monitor
19    untimed_monitor_type tmon_untimed("tmon_untimed",sc_time(5,SC_NS),"ocp_untimed.trace");
20    tmon_untimed.p_mon(ch_untimed);
21
22    // performance monitor
23    perf_monitor_type pmon ("pmon ",true,false);
24    pmon.p_mon(ch_clocked);
```

Figure 2: TL1 Monitor Example

The type definitions in lines 2-7 ensure, that consistent template arguments are used for the channels as well as the various monitor types. A clocked channel and an untimed channel are instantiated in lines 9-11. The remainder of the code examples shows, the instantiation of clocked trace monitor (line 14), an untimed trace monitor (line 19), and a performance monitor (line 23). Obviously the type of the trace monitor should match with the channel. The clock port of the clocked monitor should be bound to the same clock signal as the channel it is attached to (line 16).

The binding of the monitors to the channels is done using regular SystemC binding. The monitor ports are bound to the channels in the same way as OCP master and slave ports (lines 10, 16, 24). The example also illustrates, that an arbitrary number of monitors can be connected to a channel: the clocked channel has both a clocked trace monitor and a performance monitor attached.

3.2 Examples

The following examples in the channel package illustrate the usage of the new monitor interface. All these examples are installed at $(OCPROOT)/examples/supplementary

- ocp_tl1_simple illustrates the usage of the TL1 trace monitor and the performance monitor.
- ocp_tl2_simple illustrates the usage of the TL2 trace monitor and the performance monitor.
- ocp_tl3_simple illustrates the usage of the TL3 performance monitor.

Please follow the instruction in the respective README.txt to enable SCV transaction recording.

4 The Performance Monitor

4.1 Motivation

The trace monitor of the SystemC OCP channel primarily addresses the verification of OCP protocol compliance at the cycle accurate TL1 abstraction level. For this purpose the current monitor every cycle dumps the complete status vector of the OCP channel.
For several reasons this trace monitor is not appropriate for the TL2 performance channel. First, the cycle-based status dump is too low level for performance analysis purposes and requires a post-processing step to compile aggregated statistical views. Second, the OCP proprietary dump format is not compliant with any commercial tool offering, so OCP users need to create their own viewers.
The performance monitor addresses design tasks like architectural modeling and performance simulation. The actual transaction recording is based on the SystemC Verification library [1]. A specific version of the monitor for all channels in the OCP-IP SystemC package is available.
The purpose of this section is to document features, usage, and implementation of the OCP performance monitor. The next section gives a brief overview of the components in the performance monitor. Section 4.3 is focused on the user's view of the monitor and defines a set of coding guidelines. Section 7 provides more insight into the implementation of the performance monitor.

Note: the SCV library from OSCI is currently not compliant with the SystemC 2.1.v1 library. You can find a compliant library at http://www.greensocs.com/SCVDownload. Additionally you have to add -DSC_USE_SC_STRING_OLD to the compiler flags.

4.2 Overview

The OCP performance monitor enables intuitive performance analysis by means of fast transaction level recording. The analysis instrumentation is based on the SystemC Verification (SCV) standard [1].

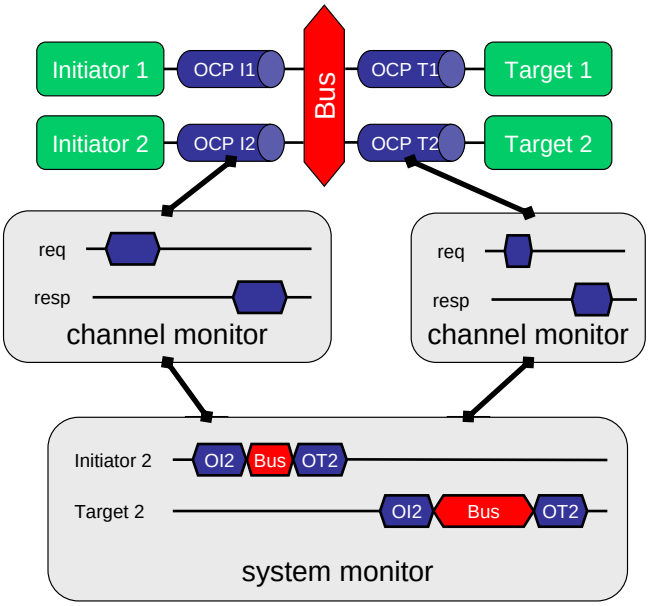

Figure 3: Performance Monitor Overview

As depicted in Figure 3, the performance monitor comprises two hierarchy levels

- The *channel monitor* is attached to a single OCP channel. It records the local transactions on this channel.
- The *system monitor* is attached to every channel monitor. It collects the individual transactions from all channels and compiles them into aggregated transactions. These aggregated transactions show the relation between the individual phases. This enables the system-level analysis of the communication in terms of latency, throughput and bottlenecks.

4.3 The Performance Monitor Module

This section is dedicated to the user view of the performance monitor. It describes the features of the monitor and the actions required by the user to instantiate and to configure the monitor. This section also defines a set of SystemC coding guidelines the user has to follow in order to obtain correct and meaningful results. We use the TL2 version of the performance monitor to discuss the features and usage.

The purpose of the SystemC monitor module is to configure the transaction recording for a specific channel. The available configuration parameters are specified as constructor arguments.

The monitor has one registration port, which has to be bound to the OCP channel.

```
25 template <class Tdata, class Taddr>
26 class OCP_TL2_Perf_Monitor: public sc_module {
27 public:
28   OCP_TL2_Monitor_Port<Tdata,Taddr> p_ocp;
29
30   OCP_TL2_Perf_Monitor (sc_module_name name,
31                       bool channel_recording  = true,
32                       bool system_recording   = true,
33                       bool master_is_node     = false,
34                       bool slave_is_node      = false,
35                       unsigned int  max_nbr_of_threads = 0,
36                       bool burst_recording    = true,
37                       bool attribute_recording= true);
38 };
```

Figure 4: Declaration of the OCP TL2 Performance Monitor

The list below describes in the constructor arguments:

- **`sc_module_name:`** mandatory constructor argument for all SystemC modules; specifies the name of the monitor module
- **`channel_recording:`** en-/disables the actual transaction recording for this particular channel
- **`system_recording:`** en-/disables the registration of this channel monitor to the system monitor
- **`master_is_node:`** specifies whether or not the SystemC module connected to the master interface of the OCP channel is a communication node. This parameter is only relevant for the system monitor and imposes some requirements on the behavior of the SystemC module (see section 4.4.2).
- **`slave_is_node:`** specifies whether or not the SystemC module connected to the slave interface of the OCP channel is a communication node. This parameter is only relevant for the system monitor and imposes some requirements on the behavior of the SystemC module (see section 4.4.2).
- **`max_nbr_of_threads:`** a value bigger than zero enables the per-thread recording feature of the channel monitor. If the value is zero, all threads are written into the same transaction stream. If the value is bigger than zero it specifies the maximum number of OCP threads. In that case the connected master and slave modules are not allowed to send OCP transactions with a thread identifier bigger than `max_nbr_of_threads-1`.
- **`burst_recording:`** By default the channel monitor records every transaction on the OCP channel. This configuration parameter enables the separate recording of complete OCP bursts.
- **`attribute_recording:`** By default the channel monitor records only the address attribute of the OCP transactions. This configuration parameter enables recording all attributes of the request and response transactions.

An illustration of the features of the performance monitor can be found in [2].

4.4 SystemC Modeling Guidelines

This section defines a set of SystemC coding guidelines the user has to follow in order to obtain correct and meaningful results.

4.4.1 Channel Monitor

Delayed Acceptance of Requests and Responses. In essence the performance monitor records the start and end of any request and response transaction on the OCP channel. The corresponding events in the new OCP TL2 performance channel are the `RequestStartEvent`, `RequestEndEvent`, `ResponseStartEvent` and `ResponseEndEvent`. The user should ensure a non-zero delay between these events in order to record meaningful transactions. Hence the user should not use automatic or immediate acceptance of OCP transactions.
The new OCP TL2 performance channel provides delayed `acceptRequest` and `acceptResponse` methods, which take a delay as a function argument. The delay can be specified as a `sc_time` object or as an integer number representing the number of clock cycles. Users of the layered OCP channels should call `wait()` between receiving and accepting a transaction.

4.4.2 System Monitor

Forwarding of the Transaction Handle Member. The system monitor essentially requires the preservation of the new `TrHandle` member in the transaction data structures throughout the end-to-end transaction.
This means any *communication node* like a bus has to forward the value of this member in the request and the response path. In case the complete transaction data-structure is copied nothing extra needs to be done, since the `TrHandle` member is copied in the copy-constructor of the request- and response-groups. A communication node is also supposed to forward all incoming transactions. In case it consumes a transaction it has to cancel this transaction through the `cancelTransaction` method of the system monitor.
The system monitor relies on the *slave modules* to forward the `TrHandle` member from the request to the response data-structure. Additionally, the response behavior has to be compliant with the 2.x version of the OCP protocol, i.e. a slave is not supposed to send a response in case of `OCP_MCMD_IDLE, OCP_MCMD_WR` and `OCP_MCMD_BCST` type of transactions.

4.4.3 Example

The OCP methodology package available on ocpip.org contains an examples which fully support the performance monitor:

- scv_ocp_tl2_slave.cpp copies the TrHandle member from the request to the response data structure
- ocp_tl2_perf_bus.cpp copies the TrHandle member from the OCP slave port to the OCP master port in the request path and vice versa for the response path
- the top-level netlist main_bus_2m_3s.cpp instantiates and binds performance monitors to all OCP channels.

4.5 Instantiation and Binding

The immediate impact on the user code is restricted to the structural SystemC code, which also instantiates the OCP channels. The complete code of a simple point-to-point system is listed below. The extra code required to use the performance monitor is highlighted by a grey background. The following enumeration discusses the required additions to the code.

```
1  // performance monitor include
2  #include "ocp_tl2_perf_monitor.h"
3
4  #include "ocp_tl2_channel.h"
5  #include "ocp_tl2_master.h"
6  #include "ocp_tl2_slave.h"
7
8  int sc_main(int, char*[])
9  {
10   scv_tr_text_init();
11   scv_tr_db db("ocp_db");
12   scv_tr_db::set_default_db(&db);
13
14   typedef unsigned int Ta;
15   typedef unsigned int Td;
16   // Creates the OCP TL2 channel
17   OCP_TL2_Channel<Ta,Td>  ch0("ch0");
18   // Set the OCP parameters for this channel
19   MapStringType  ocpParamMap;
20   readMapFromFile("ocpParams", ocpParamMap);
21   ch0.setConfiguration(ocpParamMap);
22   // specify monitor parameters
23   bool channel_recording  = true;
24   bool system_recording   = false;
25   unsigned int  max_nbr_of_threads = 4;
26   bool burst_recording    = true;
27   bool attribute_recording    = false;
28   // Creates the performance monitor
29   OCP_TL2_Perf_Monitor<Ta,Td> mon0("mon0", channel_recording,
   system_recording, false, false, max_nbr_of_threads, burst_recording,
   attribute_recording);
30
31   // bind monitor port to the channel
32   mon0.p_ocp(ch0);
33
34   // Creates masters and slaves
35   tl2_slave <Ta,Td> sl1("sl1");
36   tl2_master<Ta,Td> ms1("ms1");
37
38   // Connect masters and slaves using OCP channel
39   ms1.ocp(ch0);
40   sl1.ocp(ch0);
41   // Starts simulation
42   sc_start(2000, SC_NS);
43   return(0);
44 }
```

Figure 5: Instantiation and binding of a TL2 performance monitor to a TL2 channel

- **Include (lines 1-2):** The user must include the monitor header file.
- **SCV setup (lines 10-12):** The SCV transaction recording needs to be initialized. This example uses the text based recording of the publicly available SCV library. Additionally the recording database needs to be specified.
- **Instantiation (lines 22-29):** One performance monitor module must be instantiated for each OCP channel that is to be monitored. Naturally the template arguments of the monitor must match with the observed channel. In case of more complex systems with communication nodes special attention must be paid to the **`master_is_node`** and **`slave_is_node`** parameters (see section 4.4.1).
- **Binding (lines 31-32):** Finally the monitor port needs to be bound to the channel.

4.5.1 System Monitor

The OCP TL2 system monitor collects the transactions from all OCP channels to enable system level performance analysis. The basic idea to register all modules to be either leaf components (initiators, targets) or intermediate communication nodes (buses, bridges). System level transactions start with an initiator request and end ether when the request reaches the target (write, broadcast) or when the response reaches again the initiator. Whenever the transaction reaches a communication node, a new phase of the transaction starts.
To enable this kind of system level transaction recording, an additional 'transaction-handle' member is required in the OCP data structure to trace the streaming of transaction through the system. This transaction-handle needs to be forwarded by all communication nodes and slave modules (see section 4.4.2).

Obviously the implementation of the system monitor is more complex than the channel monitor. Basically, the system monitor maintains a map of scv_transaction_handle objects for all ongoing transactions. Every time a transaction recording method is called, the system monitor either start or ends a transaction or starts or ends a new phase of the transaction.

5 Creating TL1 Monitors

The new monitor interface supports the creation of custom monitors. This chapter is dedicated for the advanced user of the OCP channel library, who wants to modify the existing TL1 monitors or even create custom TL1 monitors. First we give a brief overview of the changes in the monitor architecture. After that the monitor interfaces are presented in detail. Any custom monitor must be implemented against these interfaces. The last sub-section explains the creation of a custom TL1 monitor by means of examples from the existing TL1 trace monitor.
This chapter does not provide additional information for the usage of the existing monitors.

5.1 History

Before version 2.1.2, the trace monitors were tightly coupled with the TL1 and TL2 channels. In fact the monitors are both members and friend classes of the channels. On the other hand, the performance monitors was using a specific monitor interface to access the current state of the channels.
These incompatible concepts made the usage of the different types of monitors not really intuitive. Additionally it was not possible to connect additional, user-defined monitors to the channel without changing the monitor code.

In the 2.1.2 release of the OCP channel library the monitor interfaces have been revised to eliminate the limitations mentioned above. The new monitor interface unifies the way monitors are connected to the channels. Additionally an arbitrary number of monitors can be connected to any channel. This enables the user of the OCP channels to develop specific monitors.
In principle the new access interface uses the observer pattern.

5.2 TL1 Monitor Interface Overview

The concept of the monitor interface is illustrated in Figure 6 by means of the TL1 channel. The channel itself implements a monitor interface, which allows to peek the current state of the monitor. Using the monitor interface external modules can access the channel without modifying

it. For this purpose the monitors have a sc_port templated with the monitor interface to connect to the channel.

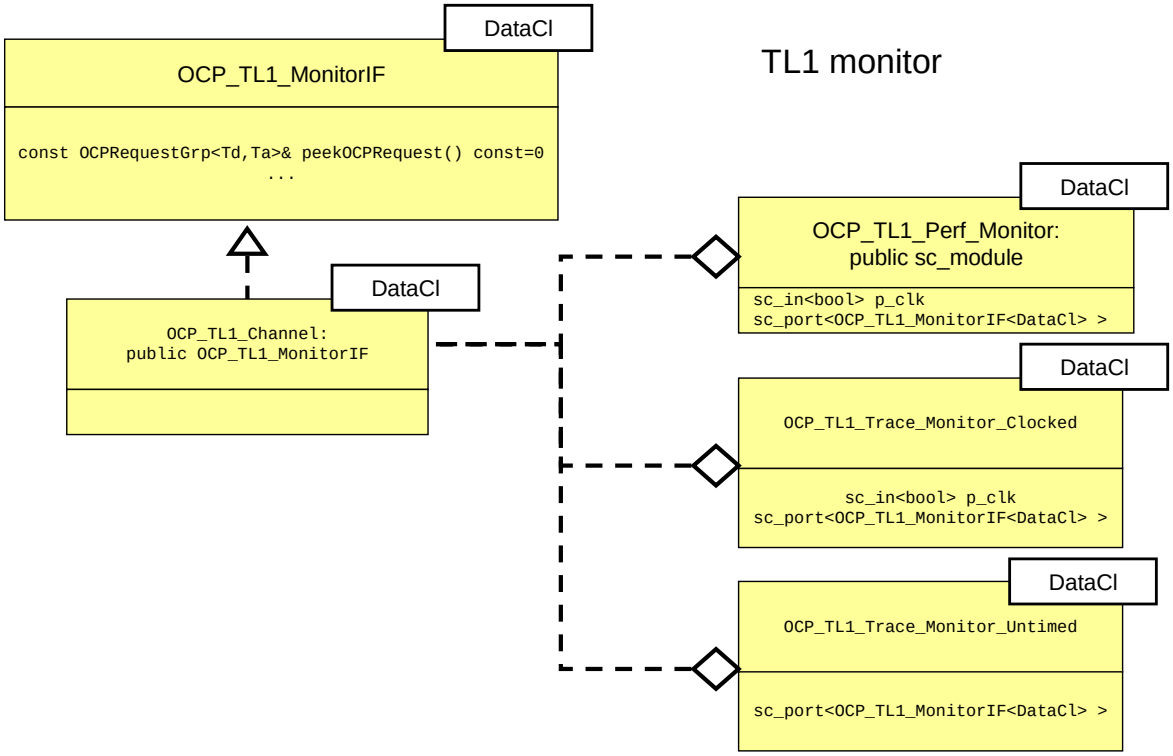

Figure 6: New TL1 Monitor Interface

Two of the TL1 monitors depicted in figure Figure 6 also have a clock input port to access the monitor in every clock cycle. Instead the OCP_TL1_Trace_Monitor_Untimed has a clock_period constructor argument to connect to the corresponding OCP_TL1_Channel_Untimed. Alternatively the monitors could have OCP master and slave ports to use event-based activation.

```
1  template <typename TdataCl>
2  class OCP_TL1_MonitorIF :  virtual public sc_interface
3  {
4  public:
5
6    typedef typename TdataCl::DataType Td;
7    typedef typename TdataCl::AddrType Ta;
8    typedef OCPRequestGrp<Td,Ta>             request_type;
9    typedef OCPDataHSGrp<Td>           datahs_type;
10   typedef OCPResponseGrp<Td>         response_type;
11   typedef ParamCl<TdataCl>           paramcl_type;
12
13   // Monitor access
14   virtual  const OCPMCmdType getMCmdTrace ()   const = 0;
15   virtual  const bool getMDataValidTrace ()    const = 0;
16   virtual  const OCPSRespType getSRespTrace () const = 0;
17
18   // port names
19   virtual const std::string          peekChannelName()   const = 0;
20   virtual const std::string          peekMasterPortName()      const = 0;
21   virtual const std::string          peekSlavePortName() const = 0;
22
23   // transactions
24   virtual const request_type& peekOCPRequest()    const = 0;
25   virtual const datahs_type&  peekDataHS()        const = 0;
26   virtual const response_type&       peekOCPResponse()   const = 0;
27
28   virtual const bool          peekRequestEnd()    const = 0;
29   virtual const bool          peekRequestStart()  const = 0;
30   virtual const bool          peekRequestEarlyEnd()     const = 0;
31
32   virtual const bool          peekResponseEnd()   const = 0;
33   virtual const bool          peekResponseStart() const = 0;
34   virtual const bool          peekResponseEarlyEnd()    const = 0;
35
36   virtual const bool          peekDataRequestEnd()      const = 0;
37   virtual const bool          peekDataRequestStart()    const = 0;
38   virtual const bool          peekDataRequestEarlyEnd() const = 0;
39
40   // thread busy
41   virtual const unsigned int  peekSThreadBusy()   const = 0;
42   virtual const unsigned int  peekSDataThreadBusy()     const = 0;
43   virtual const unsigned int  peekMThreadBusy()   const = 0;
44
45   // reset
46   virtual const bool          peekMReset_n()            const = 0;
47   virtual const bool          peekSReset_n()            const = 0;
48
49   // sideband signals
50   virtual const bool          peekMError()        const = 0;
51   virtual const unsigned int  peekMFlag()         const = 0;
52   virtual const bool          peekSError()        const = 0;
53   virtual const unsigned int  peekSFlag()         const = 0;
54   virtual const bool          peekSInterrupt()    const = 0;
55   virtual const unsigned int  peekControl()       const = 0;
56   virtual const bool          peekControlWr()           const = 0;
57   virtual const bool          peekControlBusy()   const = 0;
58   virtual const unsigned int  peekStatus()        const = 0;
59   virtual const bool          peekStatusRd()            const = 0;
60   virtual const bool          peekStatusBusy()    const = 0;
61   virtual const bool          peekExitAfterOCPMon()     const = 0;
62
63   // OCP paramertes
64   virtual paramcl_type*       GetParamCl()              = 0;
65 };
```

Figure 7: TL1 Monitor Interface

Figure 7 shows the entire TL1 monitor interface. It is implemented by the TL1 channel to enable the observation of the current state of the channel. Lines 6-7 provide some public type definitions, which should be used by the monitor. This ensures the usage of consistent types in the monitor and the channel, so compilation errors are avoided. All the remaining functions in the monitor interface allow the monitor to query the communication structures and status variables in the channel. These query methods are all const, so the monitor can only observe and cannot modify the channel. For efficiency reasons, the more heavy-lifting data structures are returned by reference (lines 24-26).

There are a few exceptions to the general rule, that the monitor functions do not affect the state of the channel. The obvious exception is the GetParamCL method in line 64, which returns a pointer to the OCP parameter class. Here we rely on the well-behavior of the monitor to not modify any of the attributes in the parameter class. The more subtle exceptions are related to the 3 trace functions in lines 14-16 and are explained in section 5.3.3.

As displayed in Figure 7, all monitors have a sc_port templated with the monitor interface. This port is bound to the ocp channel instance, which the monitor object should observe. Please refer to lines 15, 20, and 24 of Figure 2 for an example.

5.3 TL1 Monitor Modeling Guidelines

This section gives guidelines for the creation of custom TL1 monitors.

5.3.1 Sensitivity of the Monitor

The first design decision for a custom OCP monitor is the sensitivity of the monitor function.

- The TL1 monitor can be clocked like the ocp_tl1_trace_monitor_clocked. In this case the monitor needs sc_in<bool> clock input port, which should be connected to the same clock signal as the corresponding TL1 channel. The monitor then has one clock-sensitive method and samples the monitor on every rising clock edge.
- The TL1 monitor can be untimed like the ocp_tl1_trace_monitor_untimed. In this case the monitor needs a constructor parameter to specify the clock period of the corresponding channel. The monitor has a method, which is activated every cycle using wait(period) or next_trigger(period). Note that the term “untimed” might be misleading and has certainly a different meaning than in other contexts.
- Alternatively we can imagine an event driven TL1 monitor, which is activated based on the events provided in the ocp master and slave interfaces. In this case the monitor would have an ocp TL1 master port and/or an ocp TL1 slave port, which are bound to the corresponding channel. This might be overkill in case all events need to be considered, but there might be a reason for only monitoring a subset of the channel activity.

Obviously the monitors in the monitor package follow the first two options, which are anyway very similar.

5.3.2 SW Architecture of the Monitor

A second design decision is related to the SW architecture of the monitor itself. There are good reasons for keeping the actual recording separated from the monitor module. The monitor module is a SystemC module, which has the monitor port and actively queries the required information from the channel. The monitor module contains typically very little code and can be seen a wrapper to attach the actual recording mechanism to a particular channel.

The monitors provided in the monitor package are implemented along these lines:

- The trace recording class called “OCP_TL1_OCPMonGenCl” is shared between the untimed and the clocked TL1 monitor.
- The SCV based performance recording class called “OCP_Perf_Monitor_Channel_SCV” is shared between the TL1 and the TL2 performance monitor.

In both cases the recording is done separately, so the same recording class can be used for different monitor modules.

5.3.3 Call Trace Functions only once

As already indicated in section 5.2, the 3 trace functions in lines 14-16 of Figure 7 modify the state of the channel. The correct value is only returned upon the first call of the function. Subsequent calls of the same function within the same cycle return the default value, e.g. the first call of `getMCmdTrace` might return `OCP_MCMD_WR`, whereas the all subsequent calls in the same cycle return `OCP_MCMD_IDLE`.
This is unfortunately required to avoid race conditions between the monitor and the slave, so the correct value can be returned in all situations.
Note that it is no problem to connect multiple monitors to one channel. Each of the attached monitors can call the trace functions once per cycle.

5.3.4 Example Monitor

This section discusses the creation of a simple clocked TL1 monitor, which only records the current MCMD during each cycle. Due to the behavior of the trace functions (lines 14-16 of Figure 7), the recording of the MCMD requires some bookkeeping in the monitor.

```
1  template<typename TdataCl>
2  class OCP_TL1_Cmd_Monitor : public sc_module
3  {
4   public:
5    OCP_TL1_MonitorPort<TdataCl> p_mon;
6    sc_in<bool> p_clk;
7
8    SC_HAS_PROCESS(OCP_TL1_Cmd_Monitor);
9
10   OCP_TL1_Cmd_Monitor(sc_module_name mn)
11     : sc_module(mn),
12       p_mon("p_mon"),
13       p_clk("p_clk"),
14       m_mcmd(OCP_MCMD_IDLE),
15       m_scmdaccept(true)
16   {
17     SC_METHOD(OCPClockTick);
18     sensitive << p_clk.pos();
19     dont_initialize();
20   }
21
22   void end_of_elaboration(void) {
23      std::string channel_name = p_mon->peekChannelName();
24      m_cmd_recorder = new MyRecorder(channel_name);
25   }
26
27   void OCPClockTick(void) {
28     if (m_scmdaccept) {
29       OCPMCmdType l_mcmd = p_mon->getMCmdTrace();
30       if (l_mcmd != OCP_MCMD_IDLE) {
31         m_mcmd = l_mcmd;
32         m_scmdaccept = false;
33       }
34     }
35
36     m_cmd_recorder->record(m_mcmd,sc_time_stamp());
37
38     bool mon_req_accept = ((p_mon->peekRequestEnd())          ||
39                            (p_mon->peekRequestStart()    &&
40                             p_mon->peekRequestEarlyEnd()   )   );
41
42     if ( (m_mcmd != OCP_MCMD_IDLE) && (mon_req_accept) ) {
43       m_mcmd = OCP_MCMD_IDLE;
44       m_scmdaccept = true;
45     }
46  }
47
48  protected:
49   MyRecorder m_cmd_recorder;
50   OCPMCmdType m_mcmd;
51   bool m_scmdaccept;
52 };
```

Figure 8: Example TL1 Monitor

The major part of the monitor code is rather straight forward. The clocked monitor has a monitor port (line 5) and a clock input port (line 6). The only SC_METHOD called “OCPClockTick” is sensitive to the positive edge of the clock (lines 17-19). The actual recording is done in a separate class called “MyRecorder”. The recorder object is instantiated after the elaboration phase (lines 22-25). In order to give a meaningful name to the trace file, the recorder requires the

name of the corresponding channel. The name is queried from the channel in line 23 and passed as a constructor argument to the constructor of the recorder.
The implementation of the OCPClockTick method is slightly more complex. Unfortunately we need to do some bookkeeping in the channel to determine the actual command of the current request. This information cannot be retrieved directly from the channel, but must be determined in the monitor. For this we need two extra member variables: `m_cmd` stores the current command and `m_scmdaccept` stores whether the current command has been accepted.
First we check if the current command has been accepted (line 28). If that is not the case, then the situation on the channel has not changed and we record again the current command (line 36). Otherwise we retrieve the new command from the channel (line 29). In case the new command is a valid command, the two member variables are updated accordingly (lines 31, 32) and the new command is recorded (line 36).
After the calling the recorder we check if the current command has been accepted. A command is accepted if the current request ends or if a new request starts and the channel is configured to accept requests automatically (lines 38-40). In case a valid command is accepted, the two member variables are reset (lines 42-45).

The recording of the `MDataValid` and `SResp` has to follow the same pattern as the `MCmd` recording in the example above (see the implementation of the trace monitors in the monitor package). The recording of all other parameters is straight forward and does not require additional effort in the monitor.

6 Creating TL2 Monitors

6.1 TL2 Monitor Interface Overview

Ideally the TL2 monitor interface would be just as simple as the TL1 monitor interface. Instead of using clocks, the monitors could use the regular channel events to get activated in case something interesting is happening on the channel.
Unfortunately this would lead to race-conditions between the OCP master and slave components on the one hand side and the attached monitors on the other side. Since the monitors and the OCP components would use the same events, the sequence of activation would not be deterministic. However it is important, that the monitor is activated *before* the OCP components. Otherwise the OCP components could modify the state of the OCP channel before the monitor has a chance to record the respective event. For example an OCP slave could immediately accept the request. The root of the problem is that for performance reasons the TL2 channel (other than the TL1 channel) is not based on the sc_prim_channel.

Consequently the monitor interface of the TL2 channel is one level more complex (see Figure 9). In addition to the peek interface, the TL2 monitor interface allows the registration of monitor objects. For this purpose, the monitors need to implement the TL2 observer interface. Now every observer can subscribe to any set of events in the channel.

Consider the following example: The performance monitor needs to be activated on every request start event. Therefore the performance monitor subscribes itself to the channel using the RegisterRequestStart method. Additionally the performance monitor implements the

NotifyRequestStart method. The TL2 channel calls this method in all the respective subscribers (i.e. the monitors) together with the notification of the regular SystemC RequestStartEvent. This ensures that the monitors are always updated before the SystemC components are activated.

The observer interface provides a dummy implementation for all notify functions. Like this the monitors are not forced to implement all the notifications methods. In case the monitor is registered to an event it does not implement the default implementation in the observer interface will issue a warning.

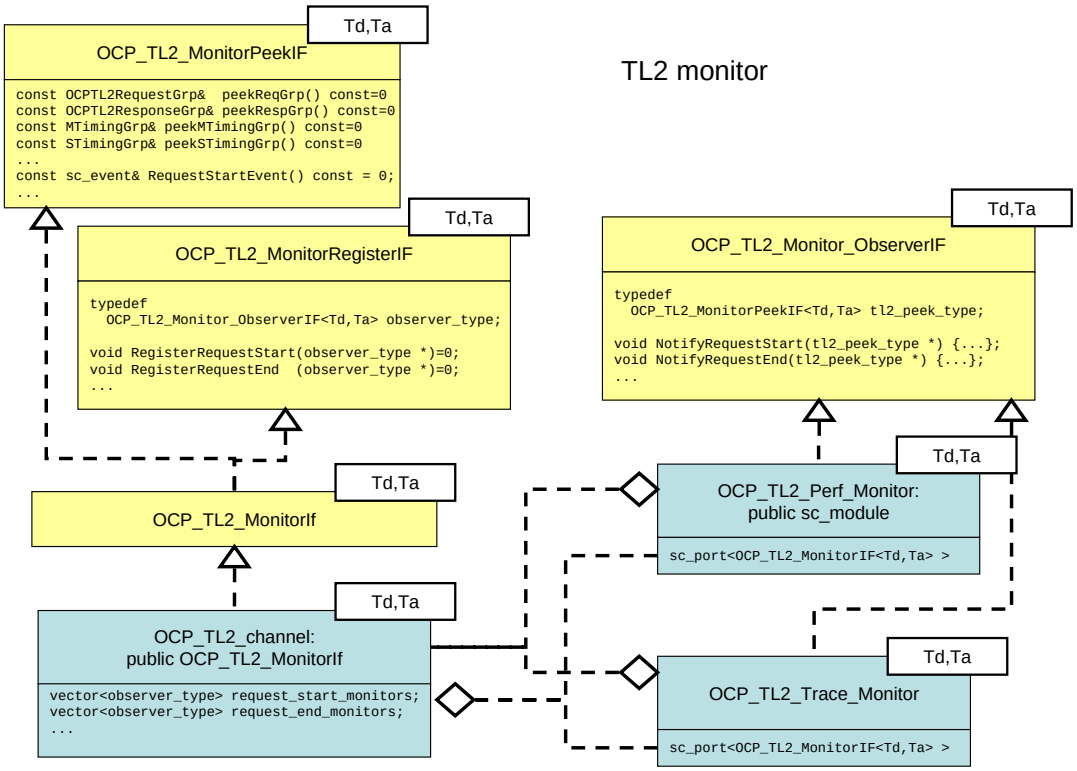

Figure 9: TL2 Monitor Interface

7 Overview of the Performance Monitor Implementation

The implementation of the performance monitor is separated into the channel monitor for local point-to-point transaction recording and the system monitor for global transaction recording. The SystemC monitor module is responsible to instantiate this channel monitor object with the correct configuration parameters and to perform the registration. In case no transaction recording is required no SystemC monitor module is bound to the TL2 channel. In this way the configuration of the transaction recording is a property of the SystemC module hierarchy.
The transaction recording itself is implemented as a set of pure C++ objects.
The channel monitor as well as the system monitor contain the SCV specific objects for transaction recording. In this way the TL2 channel itself remains completely independent from the SCV library.

The channel monitor implementation contains a pointer to the system monitor. Since only one instance of the system monitor exists, all channel monitor objects retrieve the pointer to the single system monitor from the system monitor registry.
To support cancellation of transactions in a bus node, the pointer to the system monitor object can also be retrieved by arbitrary bus nodes.

In the 2.1.2 release the implementation has been cleaned up. The separation of the performance monitor into interface and implementation has been removed. The new monitor interface already enables the implementation of arbitrary user-defined monitors. The TL1 and TL2 performance monitors share the same implementation. The TL3 performance monitor has much simpler implementation because of the templated transaction data structure.

7.1.1 Channel Monitor

The transaction recording in the performance monitor is limited to simple start and end of request and response phases. Hence the SCV implementation is rather simple and requires only a limited set of member variables:

- One scv_tr_stream object per request and response for independent recording of transactions
- One scv_tr_handle object per request and response as a handle to the currently ongoing transaction
- A number of scv_tr_generator objects to create specific entries in the database, e.g. to differentiate read and write transactions.
- Additional scv objects are instantiated in case the burst recording and/or the thread recording feature is enabled

The transaction recording can be done at the TL2 chunk level or at the OCP burst level. This is achieved by maintaining separate scv_stream objects for chunk recording and burst recording.
Note: The current implementation of the SCV library is quite limited with respect to transaction recording. For example the recording is text based and supports only a single database, i.e. all transactions are recorded in a single file. Implementation of an improved implementation of the SCV transaction recording interface is beyond the current scope of the performance monitor.

8 Future Work

The new access interface is most likely only an intermediate step towards a standardized analysis framework for verification and architecture exploration. The OSCI TLM working group is currently standardizing on the basic analysis instrumentation concepts. Once this activity is finished the OCP-IP SLD working group will most likely update the monitor interface to the new standard.

9 Related Documentation

[1] SystemC Verification Standard Specification, Version 1.0e, May 16, 2003
[2] A New SCV Compliant Transaction Recording Monitor for the SystemC OCP Channel, presentation at the OCP-IP pavilion at DATE 2005
[3] OCPIP Core Preparation Guide, Chapter 7 "CLI Tools and Methods", section ocpdis2