OCCN: A Network-On-Chip Modeling and Simulation Framework

Marcello Coppola[1], Stephane Curaba[1], Miltos D. Grammatikakis[2, 3],
Giuseppe Maruccia[1]and Francesco Papariello[1]
[1] ST Microelectronics, AST Grenoble Lab, 12 Jules Horowitz 38019 Grenoble, France
{marcello.coppola, stephane.curaba, giuseppe.maruccia, francesco.papariello}@st.com
[2] ISD S.A., K. Varnali 22, 15233 Halandri, Greece, mdgramma@isd.gr
[3] Computer Science Group, TEI-Crete, Heraklion, Crete, Greece

ABSTRACT

The open-source On-Chip Communication Network (OCCN) defines an efficient framework for network-on-chip modeling and simulation based on an object-oriented C++ library built on top of SystemC. OCCN increases the productivity of developing communication driver models through the definition of a universal communication API. This API provides a new design pattern that enables creation and reuse of executable transaction level models (TLMs). OCCN also addresses protocol refinement, design exploration, and high-level performance modeling.

1. Introduction

Multiprocessor System-on-Chip (MPSoC) integrates on a single chip various components, such as processor cores, storage elements, embedded hardware, analog peripheral devices, MEFS [21], and MEMS [5]. The aim is to satisfy tight time-to-market constraints and provide performance and scalability for popular applications, such as network communication and multimedia. In the canonical MPSoC view, On-Chip Communication Architecture (OCCA) enables distributed computation, communication, and synchronization among system components. For complex MPSoC, OCCA design exploration must evaluate rapidly cost-effective system configurations, by examining realizability, packaging, serviceability, and programmability constraints [12, 13, 15, 20]. Currently there are two prominent types of OCCA.

- T*raditional on-chip buses*, such as AMBA [1], STBus [17, 18], and Core Connect [11]. Bus-based networks are usually synchronous and offer many variants; buses may be reconfigurable, partitionable into smaller sub-systems, might allow for read/write conflicts, e.g. CRCW PRAM models, and provide multicast, broadcast or even combining facilities.
- The next generation *network on-chip* (NoC) is able to meet application-specific requirements through a powerful communication fabric based on repeaters, buffer pools, and a complex protocol stack [2, 10, 14]. Innovative network on-chip architectures include MIT's Raw network [16], and VTT's Eclipse [8].

The proposed On-Chip Communication Network methodology (OCCN) for modeling OCCA provides a flexible, open-source, object-oriented C++-based library built on top of SystemC, together with new design methodology [8]. OCCN design methodology allows the designer to rapidly assemble, synthesize, and verify a NoC that uses pre-designed IP for each system bus. This approach dramatically reduces time-to-market, since it eliminates the need for long redesigns due to architecture optimization after RTL simulation. This methodology has enabled the design of complex on-chip networks, such as the STM STBus, a product found today in almost any digital satellite decoder [17, 18].

In Section 2, we focus on OCCN design methodology, including abstraction levels, separation of communication and computation, and inter-module communication refinement through a communication layering approach based on two SystemC-based modeling objects: the Protocol Data Unit (Pdu), and the MasterPort/SlavePort interface. We also describe the OCCN statistical library used for system-level design exploration in SystemC models. In Section 3, we illustrate OCCN modeling, communication refinement, and design exploration through a case-study. Finally, in Section 4, we provide conclusions and extensions to OCCN. We conclude this paper with a list of references.

2. OCCN Design Methodology and API

The generic features of NoC modeling involve

- *modeling at various abstraction levels*, such as functional, transactional behavioral, transactional clock accurate, RTL, and gate-level [7],
- *orthogonalization of concerns,* i.e. separation of function from architecture and communication from computation, and

1530-1591/04 $20.00 (c) 2004 IEEE

- *an OSI-like conceptual model for inter-module communication layering* [3, 6, 7], whereas each layer translates transactions requests to a lower-level protocol.

As shown in Figure 1, OCCN inter-module communication layering is based on three distinct layers.

- The *NoC communication layer* implements one or more consecutive OSI layers, starting from the physical layer, e.g. the STBus NoC communication layer abstracts the physical and data link layers.
- The *adaptation layer* maps to one or more middle layers of the OSI protocol stack. It includes both software, and hardware adaptation components. A typical software adaptation layer consists of
  - a low-level sub-layer implementing a board support package (BSP) and built in test (BIT),
  - the O.S. and driver sub-layer managing communication with external devices, and
  - the software architecture providing services to the application, such as execution control, data and message management, and exception handling.
- The top-level user-defined *application layer* translates inter-module transaction requests coming from the application API to the communication API. Thus, it maps directly to the application layer of the OSI stack.

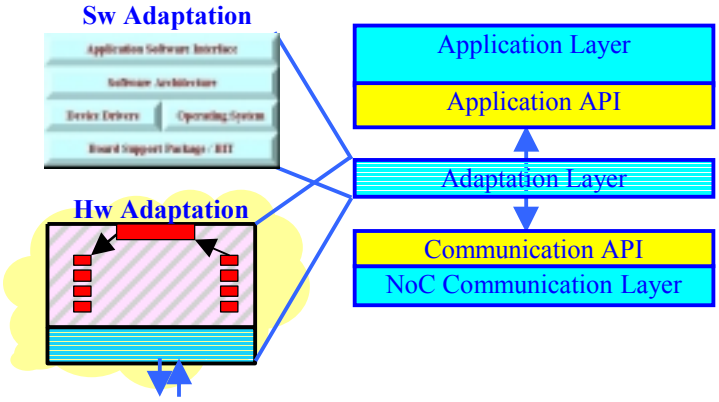

***Figure 1.* OCCN layering model with APIs**

OCCN communication layering is implemented using generic SystemC methodology, e.g. a SystemC port is seen as a service access point (SAP), with the OCCN API defining its service. We provide the following mapping.

- The *NoC communication layer* is implemented as a set of C++ classes derived from the SystemC `sc_channel` class. The layer establishes message transfer among different ports, according to the protocol stack supported by a specific NoC.
- The *communication API* is implemented as a specialization of the `sc_port` SystemC object. This API provides the required buffers for inter-module communication and synchronization and supports an extended message passing (or even shared memory) paradigm for mapping to any NoC.
- The *adaptation layer* is based on port specialization built on top of the communication API. For example, a communication driver for an application that uses variable length messages may implement segmentation, thus adapting the output of the application to the input of the channel.

The fundamental OCCN API components consist of the Protocol Data Unit (Pdu), the MasterPort and SlavePort interface, and high-level system performance modeling. These components are described next.

2.1 The Protocol Data Unit (Pdu)

A protocol data unit (or Pdu, according to OSI terminology) is a fundamental ingredient for implementing inter-module communication. It is essentially the optimized, smallest part of a message that can be independently routed through the network, i.e. a token, cell, frame, or message in a computer network, signal in a NoC, or job in a queuing network. Although basic Pdus contain only data, complex Pdus include fields, such as header, memory address, data, and CRC. OCCN groups these entities into two fields within the public Pdu class.

- The *header* field provides destination address(es), and may include source address, checksum, routing path selection (or priority), sequence number, trailer or data length for variable size Pdus, and performance- or processing-related flags. Moreover, header may provide an operation code that distinguishes: (a) request from reply Pdus, (b) read, write, or synchronization instructions, (c) synchronous and asynchronous blocking/nonblocking instructions, and (d) normal execution from setup or system test.
- The *data* field (called *payload*, or *service data unit*) is a sequence of bits that are usually meaningless to the channel. An exception is if data reduction is used, e.g. in a combining, counting, or load balancing network.

Pdus may be created using four different methods. If `HeaderType` is a user-defined C++ `struct`, `BodyUnitType` is either a basic data type, e.g. `char` and `int`, or an encapsulated Pdu, then, we can define a Pdu containing

- a body of `BodyUnitType`, as `Pdu<BodyUnitType> b`
- a body of length many elements of `BodyUnitType`, as `Pdu<BodyUnitType, length> a`

- header and body: `Pdu<HeaderType,BodyUnitType> c`
- header and body of `len` elements, as `Pdu<HeaderType, BodyUnitType, len> d`

Processes may access Pdu data and control fields using the following OCCN functions.

- The `occn_hdr(pk, field_name)` function is used to read or write the Pdu header.
- The standard operator "=" is used to
  - read or write the Pdu body,
  - copy Pdus of the same type.
- The operator s ">>" and "<<" are used to
  - send/receive Pdus from input/output streams, and
  - segmentation and re-assembly Pdus.

2.2 The MasterPort & SlavePort API

Since the same base functions are required for transmitting a Pdu for almost any OCCA, we can achieve model reuse and inter-module communication refinement through a simple transmission/reception interface. This API is implemented using specializations of `sc_port<…>` called `MasterPort<…>` and `SlavePort<…>`, defined as templates of the outgoing and incoming Pdu. In most cases, the outgoing Pdu for the Master (Slave) port is the same as the incoming Pdu for the Slave (Master) port.

The OCCN MasterPort/SlavePort API provides a message-passing interface, with send/receive primitives for point-to-point and multi-point inter-module communication. For efficiency reasons, OCCN currently implements only synchronous blocking send/receive and asynchronous blocking send, as described below.

- `void send(Pdu<…>* p, sc_time& time_out=-1, bool& sent);` This function implements synchronous blocking send. Thus, the sender will deliver the Pdu `p`, only if the channel is free, the destination process is ready to receive, and the user-defined `timeout` value has not expired. Otherwise, the sender is blocked and the Pdu is dispatched. While the channel is busy (or the destination process is not ready to receive) and the `timeout` value has not expired, waiting sender tasks compete to acquire the channel using a FIFO priority scheme. Upon function exit, the boolean flag `sent` returns false, if and only if the `timeout` has expired before sending the Pdu
- `void asend(Pdu<…>* p, sc_time& timeout=-1, bool& dispatched);` This function implements asynchronous blocking send. Thus, if the channel is free and the user-defined `timeout` value has not expired, then the sender will dispatch the Pdu `p` whether or not the destination process is ready to receive it. While the channel is busy, and the user-defined `timeout` value has not expired, waiting sender tasks compete to acquire the channel using a FIFO priority scheme. In this case, the boolean flag `dispatched` returns false, if and only if the `timeout` value has expired before sending the Pdu.
- The OCCN API implements a synchronous blocking receive using a pair of functions.
  - `Pdu<…>* receive(sc_time& time_out=-1, bool& received);` The receiver is blocked until it receives a Pdu, or until a user-defined `timeout` has expired. In the latter case, `received` is false.
  - `void reply(uint delay=0)` or `void reply(sc_time delay);` This function causes a fixed or dynamic `delay` to the receiver process, expressed as a number of bus cycles or as absolute time (`sc_time`). Return from `reply` ensures that communication is completed and that the receiver is synchronized with the sender. The following code is used for receiving a Pdu.

```
sc_time timeout = ….; bool received;
// Suppose that in is an OCCN SlavePort
Pdu<…> *msg = in.receive(timeout, received);
if (!received)
 // timeout expired: received Pdu not valid
else
 // user may perform elaboration on Pdu
 reply(); // synchronizing after 0 bus cycles
```

OCCN supports protocol inlining, that is the low-level protocol interfacing to a specific OCCA is automatically generated using the standard C++ template feature enabled by user-defined data structures. Thus, the user does not have to write low-level protocols, making instantiation and debugging easier. Savings are significant, since in today's MPSoC there are more than 20 ports, and 60 signals/port.

Using the above send/receive functions and appropriate channel setup and control functions [8], e.g. to check, enable/disable Pdu transmission or reception, or extract the exact time(s) that a particular message arrived, any kind of on-chip communication protocol can be modeled.

2.3 High-Level System Performance Modeling

High-level performance modeling is essential for MPSoC design exploration and co-design. OCCN provides a statistical package for collecting *instant and duration statistics* from monitored NoC components.

- In *time-driven simulation*, monitored objects usually have instantaneous values. During simulation, these values are recorded by calling a library-provided public member function called `stat_write`.

- In *event-driven simulation*, recorded event statistics include arrival and departure time (or duration). The interface is based on two functions.
  - First, a `stat_event_start` function call records the arrival time, and saves in a local variable the unique location of the event within the internal table of values.
  - Then, when the event's departure time is known, this time is recorded within the internal table of values at the correct location by calling the `stat_event_end` function.

The `stat_write` and `stat_event_start/end` operations may be performed by the user, or directly by the modeling library using internal object pointers. OCCN also derives specialized classes for obtaining throughput, latency, average/instant size, packet loss, and average hit ratio. In this case, an `enable_stat()`function specifies the object name, the absolute start and end time for statistics collection, the title and legends for the x and y axes, the time window, i.e. the number of consecutive points averaged in order to generate a single statistical point, and a unique object name. The data obtained can be analyzed online using visualization software, e.g. open-source `Grace`, dumped to a file for off-line processing, or combined together using joint statistical classes.

2.4 Design Exploration using OCCN

OCCN is based on experiences gained from developing OCCA for different SoC. OCCN models have been used by Academia and Industry. In order to evaluate the vast number of complex architectural and technological alternatives the architect is equipped with highly-parameterized, user-friendly, and flexible OCCN methodology.

After constructing an initial architectural solution from system requirements, this solution is refined through an iterative improvement strategy based on domain- or application-specific analytical performance models and simulation. Then, ST Microelectronics exploits a reuse-oriented design methodology. The proposed system-level configuration parameters are loaded onto the Synopsys tools `coreBuilder` and `coreConsultant`. These tools integrate a preloaded library of configurable high-level (soft) IPs, such as the `STBus` interconnect; IP integration is performed only once using `coreBuilder`. `CoreConsultant` uses a user-friendly graphical interface to parameterize IPs, and automatically generate a gate-level netlist, or a safely configured and connected RTL view together with the most appropriate synthesis strategy. Overall SoC design flow now proceeds normally with routing, placement, and optimization by interacting with various tools, such as `Physical Compiler`, `Chip Architect`, and `PrimeTime`.

3. OCCN Case-Study: Ftp-like Data Transfer

Figure 2. **Ftp-like model with generic OCCN bus**

Using layered communication, we describe a simple ftp-like inter-module transfer application. As shown in Figure 2, the application models point-to-point file transfer from a process called Transmitter in a PE (PE_1) to another process called Receiver in another PE (PE_2). Both PEs are connected using an OCCN channel. Each PE is a SystemC module containing passive C++ storage elements. Communication among OCCN modules and the channel uses a compatible MasterPort/SlavePort interface located in the modules. The application API uses a single function:

- `void ftp(int addr, MyFile& file);`

OCCN implementation uses two communication layers: cell layer corresponding to the OCCN communication API, and user-defined packet layer corresponding to the OCCN adaptation layer. The packet layer provides support services (ftp process) to the application through a `MasterPacket` (and `SlavePacket`) interface in the transmitter (resp. Receiver). The `MasterPacket` interface is derived by inheritance from `MasterPort<Pdu<...>, Pdu<...> >`. Due to space limitation only the code for transmitter is provided below. The Pdu used for inter-module communication is defined in the next code block.

inout_pdu.h

```
#include "occn.h"
struct pci_out {
    uint sequence;
    uint address; };
struct pci_in { uint ack; };
typedef Pdu<pci_out, char, 32> Request;
typedef Pdu<pci_in> Response;
typedef Pdu<char, 1000> MyFile;
```

The Transmitter module in "`transmitter.h`" and "`transmitter.cc`" defines a packet layer interface in blocks "`MasterPacket.h`" and "`MasterPacket.cc`".

transmitter.h

```
#include "occn.h" // OCCA description
#include "inout_pdu.h" // signal definitions
#include "MasterPacket.h"
// definitions for the Transmitter module
class Transmitter : public sc_module {
public:
  MasterPacket sap; // Packet layer
  SC_HAS_PROCESS(Transmitter);
  Transmitter(sc_module_name nm);// constructor
private:
  void action_tx(); // thread action
  // Internal objects and variables
  MyFile file; } // buffer for data file
```

transmitter.cc

```
#include <stdlib.h>
#include "transmitter.h"
#define NB_SEQUENCES 10
// Transmitter constructor
Transmitter::Transmitter(sc_module_name nm,
                         sc_time time_out):
          sc_module(nm), file(), sap(time_out)
{SC_THREAD(action_tx);}
// thread declaration
void Transmitter::action_tx() {
 uint addr=0;
 do { // emulate the file open
  for (int i=0; i < file.size(); i++)
    file[i] = rnd(26)+65; // random in A-Z
    sap.ftp(addr*file.size(), file);// appl.API
     // condition for ending the simulation
     if (++addr == NB_SEQUENCES) sc_stop();
  } while(1); } }
```

MasterPacket.h

```
#include "occn.h"
#include "inout_pdu.h"
// communication driver of Transmitter
class MasterPacket: public
                MasterPort<Request, Response>{
 public:  void ftp(uint addr, MyFile& buffer);
 private: sc time timeout; bool sent; }
```

MasterPacket.cc

```
#include "MasterPacket.h"
MasterPacket::MasterPacket(sc_time t):timeout(t){}
//Transmitting thread (TX_driver)
void MasterPacket::ftp(uint addr,Pdu<char>& buf){
  for (int i=0;i<1000;i++) {
    Request* msg = new Request; // create Pdu
    occn_hdr(msg, address) =addr; // set header
    occn_hdr(msg, sequence) = i; // set header
    msg = buf; // set data (body of Pdu)
    do { // retransmit if timeout expires
    send(msg, timeout, sent);
  } while(!sent); }
```

The main.cc file includes references to all modules.

main.cc

```
#include "occn.h"
#include “systemc.h”
#include "inout_pdu.h"
#include "transmitter.h"
#include "receiver.h"
#include "MasterPacket.h"
#include "SlavePacket.h"
int main(void) {
  sc_clock clock1("clock1", 10, SC_NS);
  sc_time timeout_tx(40, SC_NS); // time out
  Transmitter my_master("Trans", time_out);
  Receiver my_slave("Recv");
  StdChannel<Pdu<pci_in>,
    Pdu<pc_out,char,32> > channel("StdCh");
  my_master.clk(clock1); // bind Clock
  my_slave.clk(clock1); // bind Clock
  my_master.out(channel); // bind channel
  my_slave.in (channel); // bind channel
  sc_start(5000,SC_NS); }
```

The code blocks STBus_Master.h and STBus_Master.cc illustrates refinement for MasterPacket, (SlavePacket is omitted), if the proprietary STBus Type 1 bus is used. Observe that we do not modify Transmitter and Receiver modules, or test benches. Similar refinement can be applied to OCCN models for AMBA and VCI bus [1, 19]; for information regarding these models, refer to [4].

STBus_Master.cc

```
//Transmitter thread
#include "STBus_Master.h"
MasterPacket::MasterPacket(sc_time t):timeout(t){}
void MasterPacket::ftp(uint addr,MyFile& buffer) {
  for (int i=0; i< buffer.size(); i++) {
    Request* msg=new Request;
    occn_hdr(msg,addr) = addr; // set header
    occn_hdr(msg,source_id) = 0;
    occn_hdr(msg,tid)=0;
    occn_hdr(msg,lock)=0;
    occn_hdr(msg,be)=0xF;
    occn_hdr(msg,sequence)=i;
    switch(buffer.size()) {
      case 1: occn_hdr(opcode)= STBUS::STORE1;
            break; } // omitted rest of switch
    msg << buffer; // set body using segmentation
    send(msg, sent); } } // StBus has no loss
```

STBus_Master.h

```
#include "occn.h"
#include "STBus_ADTs.h"
class MasterPacket: public // Transmitter thread
      MasterPort<STBUS_Request, STBUS_Response> {
 public:  void ftp(uint addr, MyFile& buffer);
 private: sc time timeout; bool sent; }
```

For basic statistics, e.g. throughput or delay, appropriate `enable_stat_` calls are made from the constructor of the modeling object, i.e. rgister, FIFO, memory, or cache.

```
enable_stat_throughput("object_name", 0, 50, 1,
  "Simulation Time", "Avg Throughput for Read");
enable_stat_delay("object_name", 0, 50,
              "Arrival Time", "Departure Time");
```

The simulation efficiency of our ftp-like protocol on a Blade 1000 workstation is ~100K simulated cycles per CPU sec. Furthermore, simulation speed on customary buses, such as AMBA AHB or STBus Type 1 is almost independent of the number of initiators (transmitters) and targets (receivers), while for NoC, such as the STBus NoC, it is increasingly sensitive to the number of initiators, targets and computing nodes. This is due to the inherent complexity of the NoC architecture.

4. Conclusion and Extensions

OCCN focuses on NoC modeling by providing a flexible, state-of-the-art, C++-based framework consisting of an open-source, GNU GPL library, built on top of SystemC. OCCN design methodology offers unique features, such as

- object-oriented design concepts,
- rapid prototyping and efficient simulation through powerful C++ classes and direct linking to SystemC,
- optimized system-level modeling at various levels of abstraction, orthogonalization of concerns, refinement of communication protocols, and IP reuse principles,
- plug-and-play model integration and exchange with system-level tools supporting SystemC, and
- early design exploration for OCCA, including system performance modeling and system-level debugging and verification.

Current OCCN extensions include

- developing statistical (correlated) power estimation macro-models for NoC models,
- designing efficient algorithms for automatic design exploration and optimization of NoC systems,
- enhancing system performance modeling with
  - advanced monitoring, including generation, processing, dissemination and presentation,
  - asynchronous statistical classes supporting waves, concurrency maps, and system-level snapshots,
  - platform performance indicators focusing on monitoring system statistics, e.g. simulation speed and computation/communication load, for improving simulation performance, e.g. through automatic data partitioning or dynamic load balancing strategies.

Acknowledgments

This research has been sponsored in part by EU Medea+ project ToolIP.

References

1. Amba Bus, Arm, http://www.arm.com
2. Benini, L., and De Micheli, G. Networks on Chips: "A new SoC paradigm", *IEEE Computer*, vol. 35 (1), 2002, pp. 70—781.
3. Brunel J-Y., Kruijtzer W.M., Kenter, H.J. et al. "Cosy communication IP's". *Proc. Design Automation Conf.*, 2000, pp. 406-409
4. Caldari, M., Conti, M., Pieralisi, L., Turchetti, C., Coppola, M., and Curaba, S., "Transaction-level models for Amba bus architecture using SystemC 2.0". *Proc. Design Automation Conf.*, Munich, Germany, 2003, pp. 20026-20031.
5. Dewey, A., Ren, H., Zhang, T. "Behaviour modeling of micro-electro-mechanical systems (MEMS) with statistical performance variability reduction and sensitivity analysis". *IEEE Trans. Circuits and Systems*, 47 (2), 2002, pp. 105—113.
6. Cierto virtual component co-design (VCC), Cadence Design Systems, see http://www.cadence.com/articles/vcc.html
7. Coppola, M., Curaba, S., Grammatikakis M.D. and Maruccia, G. "IPSIM: SystemC 3.0 enhancements for refinement", *Proc. Design Automation & Test in Europe Conf.,* 2003, pp 20106-111.
8. Coppola, M., Curaba, S., Grammatikakis, M., Maruccia, G., and Papariello, F. "The OCCN user manual". Available from http://occn.sourceforge.net
9. Forsell, M. "A scalable high-performance computing solution for networks on chips", *IEEE Micro*, 22 (5), pp. 46—55, 2002.
10. Guerrier, P., and Greiner, A. "A generic architecture for on-chip packet-switched interconnections", *Proc. Design, Automation & Test in Europe Conf.*, 2000, pp. 250—256.
11. "IBM On-chip CoreConnect Bus". Available from http://www.chips.ibm.com/products/coreconnect
12. Lahiri, K., Raghunathan, A., and Dey, S. "Design space exploration for optimizing on-chip communication networks", to appear, *IEEE Trans. CAD Integr. Circuits and Syst*ems.
13. Lahiri, K., Raghunathan, A., and Dey, S. "System level performance analysis for designing on-chip communication architectures", *IEEE Trans. CAD Integr. Circuits and Syst*ems, 20 (6), 2001, pp.768-783.
14. Networks on Chip, Eds. Jantsch, A. and Tenhunen, H. .Kluwer Academic Publisher, 2003, ISBN: 1-4020-7392-5.
15. Paulin, P., Pilkington, C., and Bensoudane E., "StepNP: A system-level exploration platform for network processors", *IEEE Design and Test*, 2002, 19 (6), 17-26
16. Raw. Available from http://www.cag.lcs.mit.edu/raw
17. Scandurra A., Falconeri, G., Jego, B., "STBus communication system: concepts and definitions", internal document, STM, 2002.
18. Scandurra A., "STBus communication system: architecture specification", internal document, STM, 2002.
19. VSI Alliance, http://www.vsi.org/
20. Zivkovic, V.D., van der Wolf, P., Deprettere, E.F. et al. "Design space exploration of streaming multiprocessor architectures", IEEE Workshop Sign. Proc. Syst., San Diego, Ca, 2002.
21. Zhang, T., Chakrabarty, K., Fair, R.B. "Integrated hierarchical design of micro-electro-fluidic systems using SystemC". *Microelectronics J.*, 33, 2002, pp. 459—470.