# COSY Communication IP's

J-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Pétrot*, L. Pasquier**,
E.A. de Kock, W.J.M. Smits

Philips Research Laboratories Eindhoven, Paris** Université Pierre et Marie Curie, Paris*
E-mail: brunel@natlab.research.philips.com

## ABSTRACT

The Esprit/OMI-COSY project defines transaction-levels to set-up the exchange of IP's in separating function from architecture and body-behavior from proprietary interfaces. These transaction-levels are supported by the "COSY COMMUNICATION IPs" that are presented in this paper. They implement onto Systems-On-Chip the extended Kahn Process Network that is defined in COSY for modeling signal processing applications. We present a generic implementation and performance model of these system-level communications and we illustrate specific implementations. They set system communications across software and hardware boundaries, and achieve bus independence through the Virtual Component Interface of the VSI Alliance. Finally, we describe the COSY-VCC flow that supports communication refinement from specification, to performance estimation, to implementation.

**Keywords:** System Design; IP; Communication Interface.

## 1. INTRODUCTION

Mixing and matching software & hardware modules (IPs) is essential for designing new generation electronic systems. This relies on defining interface standards. Emerging standards, e.g. from the VSI Alliance [3], will first allow for connecting hardware "Virtual Components" to different buses. We propose higher interfaces for VCs that also target alternative software and hardware implementations[2] (See Figure 1).

Application-level transactions (APP) are used for programming a network of functions that specifies what the system is supposed to do [5]. Functions communicate through directed point-to-point first-in-first-out loss-less channels, using *read* and *write* on variable-length vectors for data streaming and *select* for reacting to non-deterministic control events (e.g. user). Application-level transactions are refined into system transactions (SYS) when choosing implementation of functions to SW or HW components. Performance and cost are then the primary issues. Hence, SYS transactions provide for parameters to trade-off throughput against memory-size, e.g. FIFO-depths. Finally, system transactions that operate on abstract data-types and high-level I/O semantics, are unraveled into more detailed interfaces. For hardware we use VCI as generic interface to 'any' physical bus specific protocol (PHY).

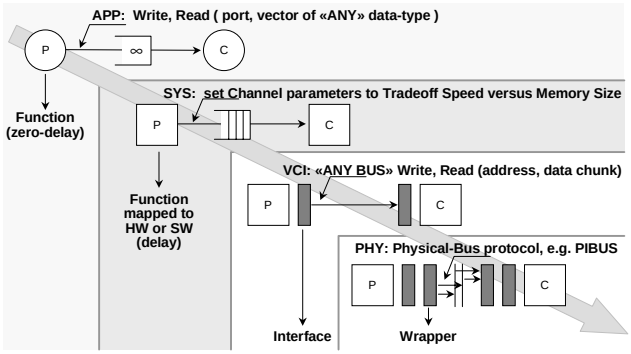


**Figure 1. Interface levels**

These transaction levels are used for defining a "COSY IP". It has *one functional model* that complies with the APP interface, and one or more *software and hardware implementations* that comply with the SYS interface. Figure 2 shows how this is used for system integration. Above the (APP) line, functional models are used for creating a process network that is executed to verify that it meets the functional requirements. Next, the downward arrows crossing the (APP) line refer to a mapping diagram. The designer selects IPs implementations. In the example, processes 1 and 2 are allocated to separate software tasks, while processes 3 and 4 are allocated to coprocessors. The SYS communications, across SW and/or HW boundaries are implemented by "COSY COMMUNICATION IPs" [1]. They provide channel controllers, drivers and interfaces, on top of the operating system (RTOS) for software and of the generic bus interface (VCI) for hardware with dedicated wrappers to the physical bus. The COSY COMMUNICATION IPs also provide for delay models that allow to assess the impact of essential communication parameters on the performance, e.g. fifo depths, choices of physical buses and memories, etc.

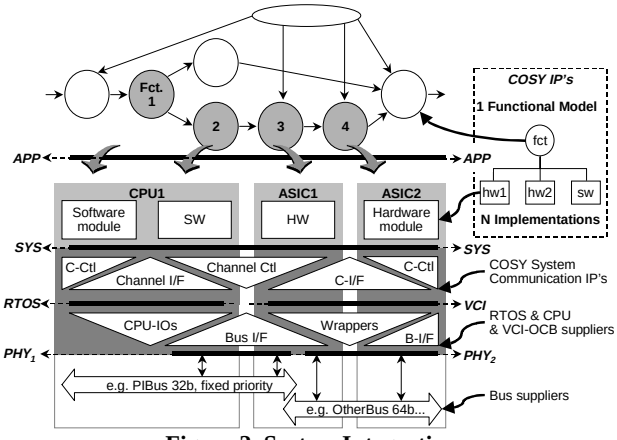


**Figure 2. System Integration**

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

The remainder of this paper is organized as follows. The generic implementation and performance models of the system-level transactions are presented in Section 2. In Section 3 we detail specific implementations that are supported by the COSY Communication IPs. Support for selecting and predicting the performance of the communications in the COSY-VCC flow is addressed in Section 4 before the conclusion.

## 2. GENERIC COMMUNICATION MODEL

We present the generic implementation and performance model of the system-level communications that was developed in COSY. On top of Figure 3 we illustrate a SYS channel between producer P and consumer C, where FIFO F holds at most D data-items of type T. We note $u_m$ (resp. $v_p$) the sequence of variable length data-vectors that are written by P (resp. read by C) during an interval $[t_0, t_1]$. Each *write* blocks the producer until the last item is copied to F, symmetrically for *read* until the last item is retrieved from F. This is implemented using a 'threshold' protocol to minimize synchronization overhead while preventing deadlock whatever are the vector sizes and the FIFO depth [2].

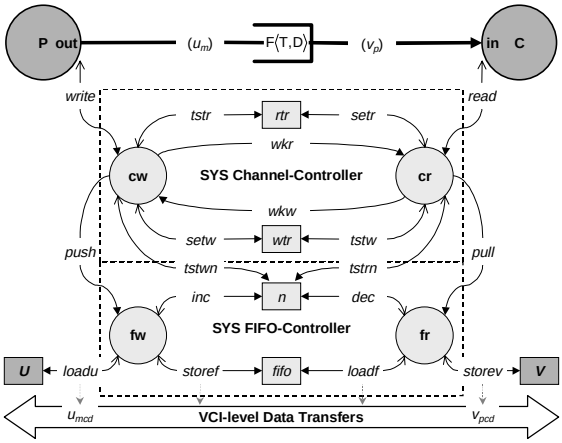


**Figure 3. Generic Implementation Model**

The generic implementation model is detailed on the transformation graph in Figure 3. Circles represent control processes and rectangles represent shared memory elements. Bi-directional arrows indicate synchronous control events: the initiator stalls until completion of the task by the target node (bold arrow). Simple arrows indicate asynchronous control events: the initiator proceeds after generating the event. The Channel-Controller synchronizes write and read operations, while the FIFO-Controller performs actual data transfers. The Channel-Writer CW decomposes a *write* request into *push* actions of data-chunks in FIFO depending on available places. When blocked, CW uses *setw* for requesting a *wkw*-callback from the Channel-Reader CR when there are at least *wtr* empty places in FIFO. The FIFO-Writer FW decomposes *push* request into *loadu* and *storef* operations that copy data from source location U to FIFO, and issues *inc* for incrementing the number of items *n* in FIFO. The Channel-Reader CR and FIFO-Reader FR play a symmetrical role on *read*.

The associated performance model defines delay equations for the latency of *read* and *write* operations (Figure 4). We distinguish between delay constants that are characterized for each specific implementation prior to execution, and variables that can only be known at execution or simulation time.

Delay constants, in bold, account for *propagation delays* ($\mathbf{dc_{sig}}$ , lines 1 & 6), *memory access times* ($\mathbf{dm_{sig}}$, lines 3 & 8), *process wake-up times* ($\mathbf{dw_{sig}}$ , lines 2 & 7) and *execution delays* ($\mathbf{de_{fct}}$, lines 4 & 9). They are statically set from e.g. propagation delay of physical bus transfer, memory access time of SRAM; wake-up time for initializing a function, execution delay of a software basic block.

Run-time variables, in italic, first account for the number of *data-vectors* (*m*), *data-chunks* (*c*) and *data-items* (*d*) on write, as well as the branch (*Bcw*, line 4) that is taken in CW at each invocation. Run-time variables also account for *waiting times* ($dt_{wkw}$, line 5) of CW on call-backs, and for *arbitration delays* ($da_{sig}$, line 5 & 10) on shared resources, e.g. bus, cpu, memory…

$$L_{write} = m\times\mathbf{dc_{write}} + c\times(\mathbf{dc_{tstwn}}+\mathbf{dc_{tstr}}+\mathbf{dc_{setw}}) \quad (1)$$
$$+ m\times\mathbf{dw_{write}} \quad (2)$$
$$+ c\times(\mathbf{dm_{tstwn}}+\mathbf{dm_{rtr}}+\mathbf{dm_{wtr}}) \quad (3)$$
$$+ c\times\mathbf{de_{cw0}} + \sum_c(B_{cw1}\times\mathbf{de_{cw1}} + \ldots B_{cw4}\times\mathbf{de_{cw4}}) \quad (4)$$
$$+ \sum_m da_{write} + \sum_c(da_{tstwn} + da_{tstr} + da_{wkw} + dt_{wkw}) \quad (5)$$
$$+ c\times\mathbf{dc_{push}} + d\times(\mathbf{dc_{loadu}}+\mathbf{dc_{storef}}+\mathbf{dc_{inc}}) \quad (6)$$
$$+ c\times\mathbf{dw_{push}} \quad (7)$$
$$+ c\times\mathbf{dm_{inc}} + d\times(\mathbf{dm_{loadu}}+\mathbf{dm_{storef}}) \quad (8)$$
$$+ c\times\mathbf{de_{fw0}} + d\times\mathbf{de_{fw1}} \quad (9)$$
$$+ \sum_c(da_{push}+da_{inc}) + \sum_d(da_{loadu}+da_{storef}) \;; \quad (10)$$

**Figure 4. Write Latency**

A procedure, called YAPITI, is used for calibrating the delay constants for a specific implementation. First, delay constants are estimated by analyzing the actual code. For instance, the delay for propagating the *write* request ($\mathbf{dc_{write}}$) between the producer and the Channel-Writer will account for a simple function call (e.g. 10 cycles), if both are executed in the same software task, or for a RTOS queue-event (e.g. 900 cycles) if they are executed in separate tasks. Next, write latencies are measured on the implementation using a cycle-true simulator. Finally, both estimated and measured results are compared, and delay constants are refined. This calibration distinguishes between "actual" and "intrinsic" delays. The latter nullify arbitration delays (**da**) and call-back waiting times (**dt**) such that the calibrated performance model can be re-used for any application on COSY architecture. Results of the calibration procedure are illustrated in Figure 5 for a specific implementation.

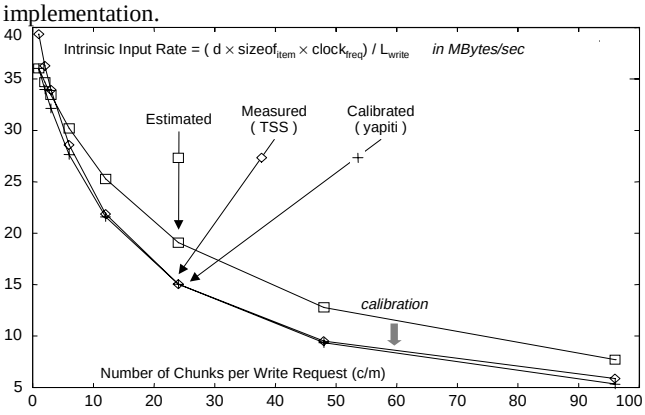


**Figure 5. Write Input Rate - Calibration**

The read and write latency equations are implemented in performance models in the COSY-VCC flow. The contribution of

the arbitration and call-back delays are added at simulation time, such that the results account for the contention on shared resources when exploring alternative system implementations.

## 3. IMPLEMENTATION SCHEMES

We developed several implementation schemes of the generic system-level communication model. Examples are detailed below.

**YSH1 – Software To Hardware scheme 1.** Figure 6 illustrates the YSH1 scheme between a Producer in software and a Consumer in hardware. FIFO buffer and status variables (*rtr*, *wtr*, *n*) are mapped to shared memory. Write-components (CW; FW) are executed in the Producer's task, using *memcpy* for data transfers (*loadu*, *storef*). Read-components (CR; FR) are executed in a separate task, triggered on *read* by an interrupt from the hardware interface that includes a DMA engine for the *loadf* transfers. The call-back signals (*wkr*, *wkw*) are implemented by pSOS queue system calls *q_send* and *q_receive*.

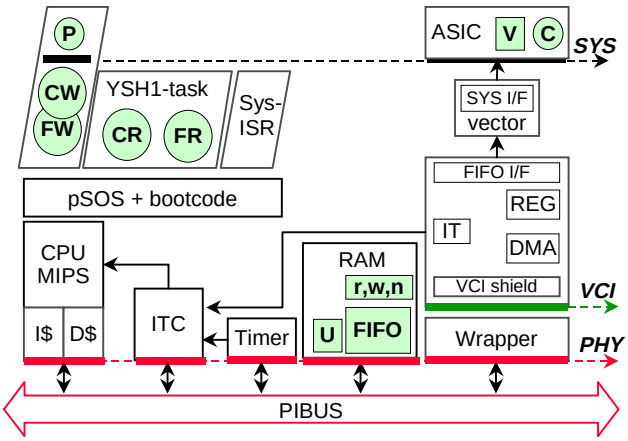


**Figure 6. YSH1 scheme**

**YHS1 – Hardware To Software scheme 1.** YHS1 scheme applies for a Producer in hardware and a Consumer in software. FIFO is implemented in the hardware interface as well as CW & FW. Read-components (CR, FR) are executed in the Consumer's task and retrieve fifo-items from a slave memory-location in the hardware interface. The status variables (*rtr* , *n*) are memory-mapped registers in the interface. The hardware interface automatically updates *n* and callbacks the reader (*wkr*) by means of an interrupt and an ISR that re-triggers the Consumer's task via a *q_send* system call. The call-back mechanism on write is not implemented as the Producer in hardware cannot suspend.

**YSS1 – Software To Software scheme 1.** This is the simplest scheme between a Consumer and a Producer in software. Both use their own run-time task, in which respective parts of the protocol run. FIFO is mapped in shared memory, and accessed by *memcpy*. RTOS queues are used to implement the *wkw* and *wkr* signals.

**YHH1 – Hardware To Hardware scheme 1.** This scheme applies for a Producer and a Consumer in hardware. The Producer uses DMA for pushing items in the FIFO that is implemented in the Consumer's interface. No software support is required, expect for the address configuration phase.

Software components are implemented by a pSOS C library (SYSLIB). From a user point of view, it allows for executing a YAPI process as a pSOS task without changing the body function. The user must provide an interface function which can be automatically derived from the corresponding YAPI class constructor.

Hardware interfaces, from UPMC, support the COSY communication schemes by keeping fifo-status information and by generating interrupts at a read/write coprocessor request, or on threshold conditions. They also allow a coprocessor to communicate with "any" physical bus by using the VCI-OCB standard.

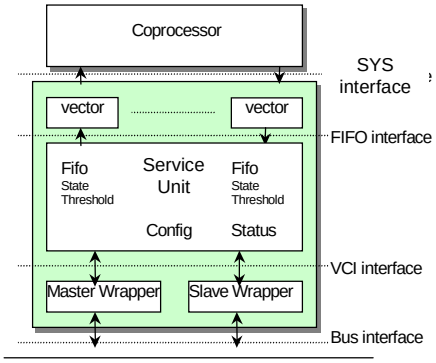


**Figure 7. Generic UPMC interface module**

Figure 7 shows the three levels that translate SYS read and write requests from the coprocessor to bit and cycle level protocol of the physical bus. The *vector unit*, on top, translates the SYS vector primitive into a simple fifo protocol consuming or producing the data vector item per item. It may optionally issue an interrupt on each request. The *service unit*, below, translates the fifo protocol into VCI protocol. It provides a fifo service, and configuration and status registers. Each fifo has a state register indicating the number of free slots, and a threshold register that triggers an interrupt when the fifo state reach this number. A master fifo has in addition a run-time configurable address generator. The last level is the *bus wrapper*; in COSY a VCI-PIBUS wrapper has been realized. The interface module is written in generic VHDL, with parameters for number and type of fifos, width and depth, and for number of status and configuration registers. Implementation with 1 slave input and 1 slave output fifo, each 32 bits wide and 8 slots deep, contains 5884 gates and supports a 100 MHz clock.

```
void boot_code(void)
{
 1: COSY_process_idT p,f;
 2: COSY_channel_idT p2f;

 3: COSY_syslink_init();
 4: p = COSY_create_process("prod1","producer");
 5: f = COSY_reate_process("filter1","filter");
  ...
 6: p2f = COSY_create_channel(p,"out",f,"in",
                              sizeof(pixel),DEPTH);
  ...
 7: COSY_map_process(p,SW);COSY_map_process(f,HW);
  ...
 8: COSY_map_channel(p2f,YSH1);
  ...
 9: COSY_create_upmc_interface(f,V_PRIO,BASEADR,
                               "in","out",NULL);
  ...
 10:COSY_create_task(p,PRODUCER_PRIO);
 11:COSY_start_syslink();
}
```

**Figure 8. Boot code example**

Finally, the SYSLIB library also provides an API for generating the boot-code of the system. This is illustrated in Figure 8 for a 3-processes network. It starts by building up the "logical" process

network (lines 4, 5, 6) after which processes and channels are mapped to their respective implementations (lines 7 and 8). Next, the required UPMC interfaces are instantiated (line 9) which includes setting the interrupt priority and memory-base address. Finally, run-time tasks are created and the application is started.

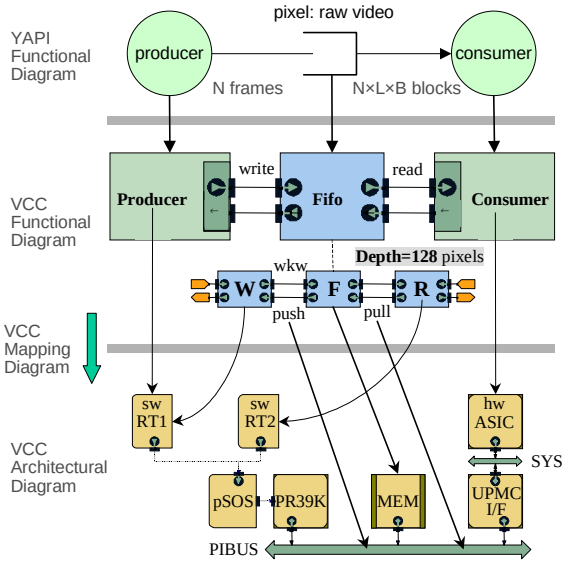


**Figure 9. COSY-VCC design entries**

## 4. COSY-VCC FLOW

Philips and Cadence Design Systems collaborate in developing the COSY-VCC flow. Design entries are illustrated on Figure 9 with a simple Producer-Consumer system. Once captured and verified with YAPI, the design is automatically imported in VCC. Functional components are then mapped to architectural components. The refined 'WFR' channel is used for selecting a COSY communication scheme: we select here a YSH1 128-deep channel. Performance simulation use the WFR models for the communication delays. For instance, *wkw* call-backs are generated by F and received by W. In YSH1, W is mapped to software hence the RTOS model accounts for the task arbitration and resume time. Once started, W generates e.g. a *push* event that is delayed by a delay model that is generated by YAPITI as exposed in Section 2:

```
viewport integer d, bcw; /*dynamic variables*/
delay_model() {
   input ( wkw ); /* from F */
   run(); /* run W functional model */
   . . .
   if ( bcw == BCW1 )
        delay ( 4.75 + d * 24.35 ); /* push */
   output ( push ); /* to F */
} /*generated for YSH1 (mips,psos,sram,pibus)*/
```

The generated *push* event triggers the bus delay model, before reaching F that triggers memory delays and generates new events.

Once architectural choices are assessed in VCC, the design can be automatically exported to TSS cycle-true co-verification environment (Figure 10). This generates the boot-code and all communication sw & hw components. Implementation modules of YAPI functions are used if readily available, else we encapsulate these functions into real-time task for software, and we use a co-simulation link for hardware. Finally, we generate a detailed netlist, using CPU's cores, buses, memories, etc., and we export probes and monitors from VCC to TSS level for calibrating the performance models of new IPs.

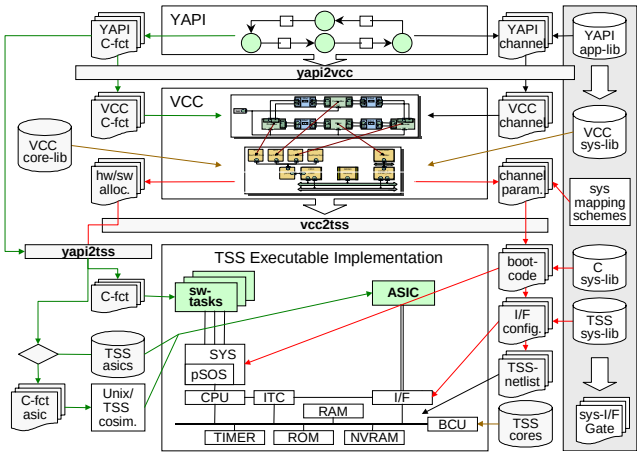


**Figure 10. COSY-VCC flow**

## 5. SUMMARY AND FUTURE WORK

COSY Communication IPs are developed for defining IPs having a function that can target SW or HW implementation. The communication mechanisms reflects several abstraction levels that are used in the COSY-VCC system integration flow for: functional verification, performance estimation and link to implementation. Preliminary results on a Digital Video Broadcast system quantify simulation speed ratios: one second of video takes 3 minutes at YAPI level, 6 at VCC level and 3 to 4 hours at TSS level. Work continues with the validation of the performance models, and with a final assessment in the context of the Nexperia-Digital Video Platform for Philips Semiconductors[6]. The assessment results will be used for contributing to the definition of IP interfaces standards and commercial IP integration flow that are urgently needed for designing new generation electronic systems.

**Acknowledgements:** This work is supported by the European Commission under ESPRIT COSY EP25443. It evolves with the contributions from Cadence Design Systems, Université Pierre et Marie Curie in Paris and Politecnico di Torino.

## 6. REFERENCES

[1] J.Y Brunel et al., «COSY: a methodology for system design based on reusable hardware & software IP's,» in: J-Y. Roger (ed.), Technologies for the Information Society, IOS Press, 709-716, 1998

[2] J.-Y. Brunel et al., «Communication Refinement in Video Systems on Chip,» *CODES'99*, Rome, 1999, pp. 142-146

[3] D. Fairbank et al., «The VSI Alliance: journey from vision to production,» *Electronic Design*, vol. 46, no. 1, pp. 86-92, Jan 12 1998.

[4] G. Martin et al., «Methodology and technology for design of communications and multimedia products via system-level IP integration,» *DAC*, 1998

[5] E.A.de Kock et al., «YAPI: Application Modeling for Signal Processing Systems,» *Submitted to DAC2000*, Los Angeles, 2000

[6] Peter Clarke, «Philips extends TriMedia reuse into Nexperia cores» EE Times, Aug 30,1999