A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design

Sungjoo Yoo Gabriela Nicolescu Damien Lyonnard Amer Baghdadi Ahmed A. Jerraya

SLS Group, TIMA Laboratory
46 Avenue Félix Viallet, 38031 Grenoble, France
{Sungjoo.Yoo,Gabriela.Nicolescu,Damien.Lyonnard,Amer.Baghdadi,Ahmed.Jerraya}@imag.fr

Abstract

In communication refinement with multiple communication protocols and abstraction levels, the system specification is described by heterogeneous components in terms of communication protocols and abstraction levels. To adapt each heterogeneous component to the other part of system, we present a generic wrapper architecture that can adapt different protocols or different abstraction levels, or both. In this paper, we give a detailed explanation of applying the generic wrapper architecture to mixed-level cosimulation. As preliminary experiments, we applied it to mixed-level cosimulation of an IS-95 CDMA cellular phone system.

1 Introduction

In designing embedded multi-processor SoCs (systems-on-chip), communication refinement is one of crucial tasks since the communication implementation can have significant impact on system performance in terms of runtime, area, power consumption, etc. [1][2][3]. It is also a challenging task since complex functional and communication requirements of current embedded SoCs require application-specific processors (e.g. CPU's, DSP's, IP's, etc.) and high-performance/complex communication networks (e.g. giga byte-level communication bandwidth, multi-point master/slave communication, etc.).

To ease the complexity of communication refinement, most of current system design methods adopt design reuse and usage of multiple abstraction levels of communication [4][5][6][7][8][9]. In such design methods, during communication refinement, the system specification consists of **heterogeneous components** in terms of communication protocols and abstraction levels. For instance, since reused components such as IP's can have their own communication protocols that have already been fixed, the system specification, where IP's are connected with each other via a common communication resource (e.g. on-chip bus), has multiple communication protocols. System refinement with multiple abstraction levels can give an intermediate system specification that consists of sub-systems or components at different abstraction levels.

To integrate heterogeneous components within a system, wrappers have been widely used for simulation and synthesis [4][8][9][10][11][12][13][14]. In simulation, for instance, BFM (bus functional model) encapsulates a functional model with a cycle-accurate interface [10][11]. BCASH (bus-cycle accurate shell) adapts RPC (remote procedural call) and cycle-accurate communication [8][9]. In [13] and [14], interfaces of mixed-level cosimulation are presented between protocol-fixed communication and cycle-accurate communication [13] and protocol-neutral and protocol-fixed communication [14]. In system synthesis, a bus wrapper, a processor template, or a protocol transducer is used to adapt a communication protocol of reused component to that of on-chip bus [4][6][7]. In [12], COSY communication IP's use a set of specific wrappers depending on the combinations of HW-SW mapping.

Previous approaches to the usage of wrappers have limitations in that their application is limited (1) to either of simulation (e.g. BFM, BCASH, [13], and [14]) or synthesis (e.g. on-chip bus wrapper, [15], [16]), (2) to a specific pair of abstraction levels: e.g. BFM (between functional and cycle-accurate) and BCASH (between RPC and cycle-accurate), or (3) to a set of specific wrappers [12].

Compared to them, our contribution is to present a **single generic wrapper architecture** that is applicable (1) to both simulation and synthesis and (2) to various combinations of abstraction levels/ communication protocols. In this paper, we present a generic wrapper architecture and explain the details of applying the architecture to mixed-abstraction-level (in short, mixed-level) cosimulation. Its application to synthesis is presented in [17].

This paper is organized as follows. In Section 2, we introduce the generic wrapper architecture. In Section 3, we present our design flow. We explain the application of generic wrapper architecture to mixed-level cosimulation in Section 4. In Section 5, we give experimental results. We conclude this paper in Section 6.

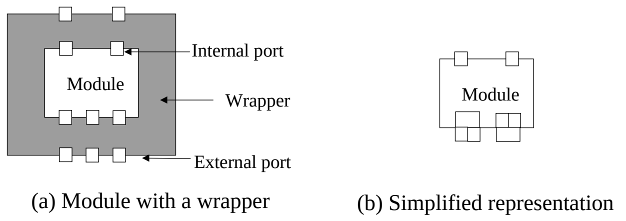
2 Generic Wrapper Architecture

In our system design flow, we represent the system with a hierarchical network of **modules**. A module consists of **behavior** and **port(s)**. Modules are connected with each other by connecting their ports via **communication channels** (in short, channels). The behavioral part of the module calls **port functions** to communicate with other modules.

2.1 Module and Wrapper

We use a wrapper to separate behavior and communication. To be specific, we use a wrapper for a module when the module is connected with a channel that has (1) a different abstraction level than that of module and/or (2) a different communication protocol than that of module. The concept of using wrappers is similar to the one used in VC (virtual component)-based design [6].

We abstract the wrapper using a concept of **internal port** and **external port**. The wrapper is composed of (1) an interface made of two sets of ports (internal ports and external ports) and (2) the behavior of wrapper. Figure 1 exemplifies a module and its wrapper. In Figure 1 (a), the external port is connected to the external



(a) Module with a wrapper (b) Simplified representation

Figure 1: Module with a wrapper.

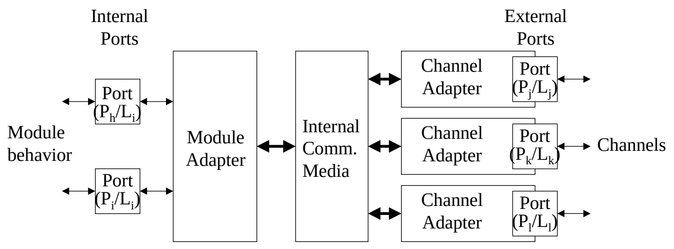


Figure 2: Generic wrapper architecture.

channel and the internal port is used by the behavioral part of module to communicate with the external channel. We use two kinds of notation to represent a module with a wrapper as shown in Figure 1 (a) and (b). The simplified representation in Figure 1 (b) is used to represent the correspondence relation between internal and external ports. As shown in Figure 1 (b), when no wrapper functionality is required between an internal port and the corresponding external port, a single port is used in the simplified representation.

2.2 Generic Wrapper Architecture

Figure 2 shows the generic wrapper architecture. It consists of internal and external ports on either side, module adapter (MA), channel adapter (CA), and an internal communication media (ICM).

For each case of adapting different protocols or different abstraction levels, an instance of the generic wrapper architecture is constructed. In the instance, ports can be given any protocols/ abstraction levels (in the figure, different protocols/abstraction levels are exemplified with P_h/L_i, P_i/L_i, P_j/L_j, P_k/L_k and P_l/L_l).

Internal communication media (ICM) is used to transfer data between module adapter and channel adapter(s). Depending on the abstraction levels of module, the ICM can be (1) a function call relation (when the module is at system or architecture level) or (2) an internal bus (when it is at RT level). Details of abstraction levels and ICM will be presented in Section 3.2 and Section 4, respectively.

2.3 Our Contribution

Compared with related work, our contribution is as follows. Comparing with optimization techniques in communication refinement [1][2][3], our work is complimentary to them since wrapper generation (for simulation and/or synthesis) is a part of architecture generation that is performed after the communication architecture and the parameters of communication (e.g. the architecture of on-chip bus, the priority of bus access, etc.) have been determined by the optimization techniques. Compared with mixed-level cosimulation techniques [18][19][20], our wrapper architecture enables many-to-many correspondence between internal and external ports in the conversion of abstraction levels/communication protocols while one-to-one correspondence is assumed in [18][19][20]. When compared with several features (interface concept and BCASH) of mixed-level cosimulation in SystemC v2.0, our work is to automatically generate interfaces of mixed-level cosimulation based on the generic architecture while SystemC gives primitives (e.g. interface concept) required for the generation of mixed-level cosimulation interfaces.

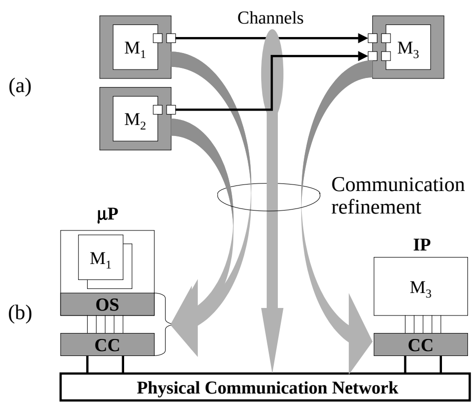


Figure 3: A simple view of commmunication refinement.

3 Communication Refinement of Heterogeneous Multi-Processor SoCs

3.1 Design Flow based on Separation between Behavior and Communication

Figure 3 shows a simple view of communication refinement. In our design flow, communication refinement means implementing wrappers and communication channels from an **abstract architecture** down to a **physical architecture**. As shown in Figure 3 (a), in an abstract architecture, modules with wrappers are connected with each other via channels. After communication refinement, a physical architecture is implemented where wrappers in the abstract architecture are implemented in the forms of hardware (communication coprocessors (CC's)) and/or software (operating systems (OS's)) depending on whether the module is mapped on a processor or on a HW component (e.g. HW IP). In the final architecture, we apply the generic architecture to the hardware part of wrapper.

In our design flow, modules can be refined (in our terms, **module refinement** can be performed) independently of their connected channels. Channels can also be refined (in our terms, **channel refinement** can be performed) independently of module refinement. The difference between (refined) module and (refined) channel, i.e. protocols and/or abstraction levels is adapted by the wrapper. Since we use hierarchical modules, the separation enables that subsystems can also be refined independently of the other part of system.

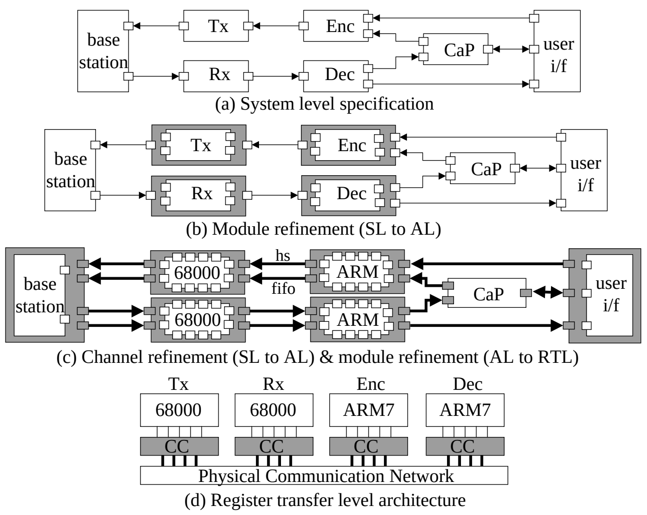
In our design flow, we have an assumption on communication refinement as follows.

Assumption 3.1 *When a module is refined, all the internal ports are refined. Thus, the abstraction levels of all the internal ports of a module are the same.*

This assumption restricts the generality of our wrapper. However, it makes it easier to automatically generate the wrappers. For instance, when adaptation is required between a module and channel(s), it is easier to manage a single module adapter to adapt them in the automatic wrapper generation than to distribute the adaptation on several adapters (in a port-by-port manner).

3.2 Abstraction Levels of Communication

To represent system communication, we use three abstraction levels of communication: system level (SL), architecture level (AL), and register transfer level (RTL).



base station
Tx
Enc
CaP
user i/f
Rx
Dec
(a) System level specification
(b) Module refinement (SL to AL)
68000
ARM
hs
fifo
(c) Channel refinement (SL to AL) & module refinement (AL to RTL)
68000
ARM7
CC
Physical Communication Network
(d) Register transfer level architecture

Figure 4: Module and channel refinement of IS-95 system.

At system level, modules communicate with each other exchanging **messages** over abstract interconnections, i.e. SL channels. There is no specific communication protocols for SL channels. Thus, SL channels provide generic functions, **send** and **receive** for the SL ports to access the SL channels. The message has no specific data type, but it has a **generic data type**.

At architecture level, each of AL channels is given its own communication protocol (e.g. FIFO, message queue, handshake, etc.) and the parameters (e.g. FIFO size). The data transferred via the AL channel have fixed data types (e.g. `int`, `float`, etc.). The AL channel provides channel functions specific to the given protocol (e.g. fifo_available, fifo_write, etc.) for the ports to access them. In terms of module, at architecture level, each module is mapped on a processor. At RT level, processors are interconnected via their communication coprocessors and physical communication network. For more details on the abstraction levels of communication, refer to [21].

3.3 Module and Channel Refinement

Figure 4 shows snapshots of module and communication refinement in the case of an IS-95 CDMA cellular phone system design [22][23]. Figure 4 (a) shows the system description at system level that consists of seven modules (seven rectangles): two vocoder modules (encoder, Enc and decoder, Dec), CDMA modem transmitter (Tx) and receiver (Rx), call processor (CaP), and two modules for the base station and the user interface of cellular phone. In the modules, small rectangles represent ports and arrows represent channels. In the figure, since all the modules and channels are at system level, no wrapper is required.

Figure 4 (b) shows an example of module refinement, where module Enc, Dec, Tx, and Rx are refined to architecture level. In this case, since the channels connected to the refined modules are still at system level, each of the four modules requires a wrapper to adapt internal AL ports and external SL channels. Figure 4 (c) shows an example of channel and module refinement from the case of Figure 4 (b). In the figure, all the channels are refined to architecture level and the four modules are refined to RT level. In this case, module base station and user i/f have wrappers to adapt internal SL ports and external AL channels. The four modules at RT level have wrappers to adapt internal RTL ports (i.e. physical pins of processors) and external AL channels. In this case, module CaP is assumed to be refined to architecture level. Thus, it does not require a wrapper.

Figure 4 (d) shows a physical architecture at RT level for the four processors. In the RTL architecture, processors have communication coprocessors to adapt their own communication protocol (e.g. AMBA of ARM processors [24]) to the protocols of channels at RT level. As previously mentioned, note that at RT level we apply the generic wrapper architecture to the hardware part of wrapper implementation, i.e. the implementation of communication coprocessors.

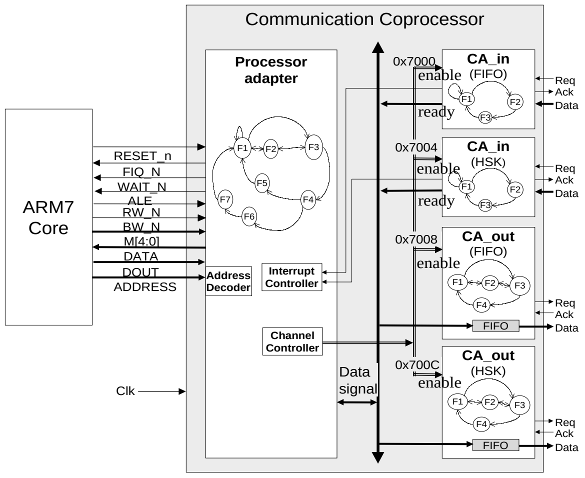


Figure 5: Communication coprocessor of an ARM7 processor in the IS-95 system.

3.4 Wrapper for Synthesis

In the RTL architecture, the communication coprocessor performs protocol conversion between the protocol of processor and those of communication channels connected with the processor. In this case, it is an instance of the generic wrapper architecture for synthesis, i.e. for the purpose of protocol conversion between processor and communication network.

Figure 5 shows an example of communication coprocessor for an ARM7 processor where module Enc is mapped in Figure 4 (d). In instantiating the communication coprocessor from the generic wrapper architecture in Figure 2, module adapter, in this case, processor adapter performs (1) address decoding (to enable channel adapters (CA's)) and (2) interrupt management (with interrupt requests coming from CA's). The CA plays a role of protocol-specific port (e.g. a port to access an on-chip bus at RT level) to communicate with an RTL channel in the physical communication network. Note that, in the communication coprocessor, we use an **internal bus** as the internal communication media (ICM) of the generic wrapper architecture. For more details of architecture and wrapper generation at RT level, refer to [17][25].

4 Wrapper for Mixed-Level Cosimulation

4.1 Functionality of Wrapper in Mixed-Level Cosimulation

To apply the generic wrapper architecture to mixed-level cosimulation, first, the functionality of wrapper should be defined. Then, each part of the functionality should be mapped to module adapter and channel adapter(s). In terms of functionality, the wrapper transforms channel access(es) via internal port(s) to channel access(es) via external port(s). To do that, from the internal port to the external port, it has the following functional chain.

- Internal port – channel interface – channel resolution – protocol/data conversion – calling external port functions – external port

From the viewpoint of internal port, channel functions (e.g. fifo_available, fifo_write, etc.) are required for the internal port to

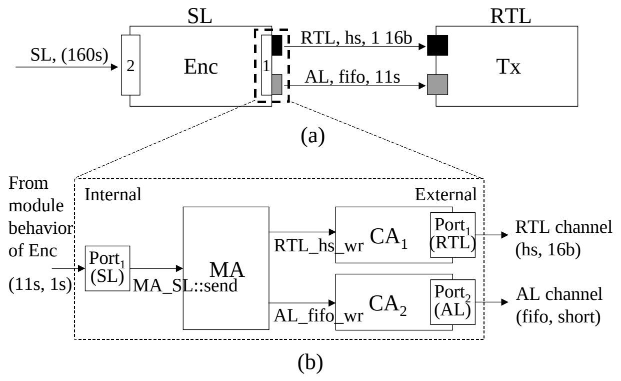


Figure 6: Case 1: Module at system level.

```
1   class MA_SL
2   : public sc_module,
3     public SL_if     // send i/f is declared.
4 { public:
5     sc_port<SL_RTL_hs_if> CA1;
6     sc_port<SL_AL_fifo_if> CA2;
7
8     // Implementation of send i/f
9     void send(struct msg) {
10        d1 = msg.rate;
11        memcpy(d2, msg.voice, 11*sizeof(int));
12
13        CA1->RTL_hs_wr(d1); // write to CA1
14        for(int i=0; i<11 ;i++) // write to CA2
15            CA2->AL_fifo_write(d2[i]);
16    }
17    ...
```

(a) Module adapter code

```
1   class CA_SL_RTL_HS // CA1
2   : public sc_module,
3     public SL_RTL_hs_if  // RTL_hs_wr i/f
4 { public:
5     void RTL_hs_wr(int data) {
6       rtl_data = (sc_bv<16>) data;
7
8       // write w/ handshake protocol at RTL
9       Port1.data = rtl_data; Port1.req = 1;
10      wait_until(Port1.ack.delayed() == 1);
11      Port1.req = 0;
12  }
13 … // in class CA_SL_AL_FIFO
14  void AL_fifo_wr(int data) {
15      al_data = (short) data;
16      // call an AL port function
17      Port2.fifo_write(al_data);
18  }
```

(b) Channel adapter code

Figure 7: Examples of module adapter and channel adapter.

exchange data. Channel interface provides the internal port with them. Channel resolution is to map the correspondence between internal and external ports. For instance, in the case that an SL channel is refined into two AL channels (e.g. fifo and handshake channels), one access (e.g. send function call) to the SL channel is mapped to two accesses (e.g. fifo_write and hs_write function calls) to the two AL channels. Protocol conversion is required when internal and external ports require different protocols. Data conversion is also required when two different abstraction levels use different data types to represent the same data. For instance, a data item of `int` type at architecture level can be represented by a data type, `logic_vector` at RT level. To exchange data via external port(s), port functions of external ports should be called by the wrapper.

4.2 Cases of Wrapper Instantiation

In this subsection, we present how to apply the generic wrapper architecture to all the possible combinations of abstraction levels in mixed-level cosimulation. In the case of three abstraction levels (system level, architecture level, and RT level), the following three cases include all the possible combinations of abstraction levels: (Case 1) module at system level, (Case 2) module at architecture level, and (Case 3) module at RT level.

4.2.1 Case 1: Module at System Level

Figure 6 shows an example of wrapper when a module is at system level and external ports at possibly different abstraction levels (with communication protocols). In Figure 6 (a), Enc module in the IS-95 system is at system level and Tx module is at RT level. There are two channels between the two modules: one (a fifo channel with 11 data items of `short` type) at architecture level and the other (a handshake channel with one data item of 16-bit `logic_vector`) at RT level.[1] Figure 6 (b) shows the wrapper instance in this case. Enc module sends a message, via internal port Port_1 at system level, to the module adapter in the wrapper. Note that, in this case, the internal communication media (ICM) in the wrapper architecture becomes a function call relation (the MA calls two functions, hs_RTL_wr and fifo_AL_wr) since the abstraction level of module EnC is system level. In this example, internal port Port_2 at system level does not require the wrapper since internal and external ports are at the same abstraction level.

Figure 7 (a) shows the behavior of module adapter (written in SystemC [8]) in this case.[2] First, to receive the message from the internal port at system level, it provides internal port Port_1 with an SL channel function called `MA_SL::send` as shown in the figure (line 9-16). The module Enc calls the function via Port_1 to send a message to the module adapter. When the function is called, the module adapter performs channel resolution by splitting the message into one `short`-type data item called `d1` and 11 `short`-type data called `d2` (line 10-11 in Figure 7 (a)). It sends the split data to two channel adapters (line 13-15). In general, depending on the correspondence between internal and external ports, the module adapter performs data splitting/merging. In this case, since one output internal port corresponds to two external ports, the module adapter performs data splitting.

The channel adapter receives the data (when its **channel adapter function** is called) and performs protocol/data conversion. Then, it writes/sends the data/message to the channel via the external port. In the case of CA_1 in Figure 6 (b), when its channel adapter function (RTL_hs_wr) is called by the module adapter (line 13 in Figure 7 (a)), the data type of received data is converted into 16-bit `sc_bv<16>` (line 6 in Figure 7 (b)), and writes it to the RTL channel according to the handshake protocol of RTL port, Port_1 (line 9-11 in Figure 7 (b)). Channel adapter function of CA_2 (AL_fifo_wr) converts the data type into `short` (line 15 in Figure 7 (b)), and writes the data to the AL channel via AL port function, `Port2.fifo_write` (line 17 in Figure 7 (b)).

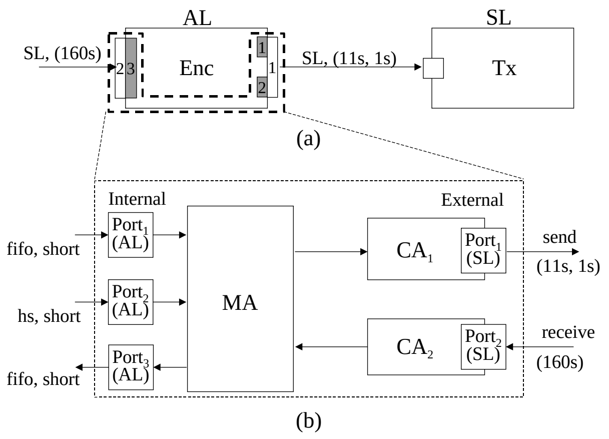


Figure 8: Case 2: Module at architecture level.

4.2.2 Case 2: Module at Architecture Level

Figure 8 (a) shows a case where Enc (Tx) module is at architecture level (system level). There is an SL channel between Enc and Tx modules. Figure 8 (b) shows the wrapper instance. In this case, note that the internal communication media (ICM) is a function

[1] In the IS-95 system, Enc performs QCELP voice coding that takes as input 160 short-type data and outputs 172 bit data and one short-type data (called **rate**). In our implementation, we packed 172 bit data into 11 short-type data (called **voice**).

[2] Note that we use interface concept supported in SystemC v2.0 to explain channel functions (of module adapter) and channel adapter functions.

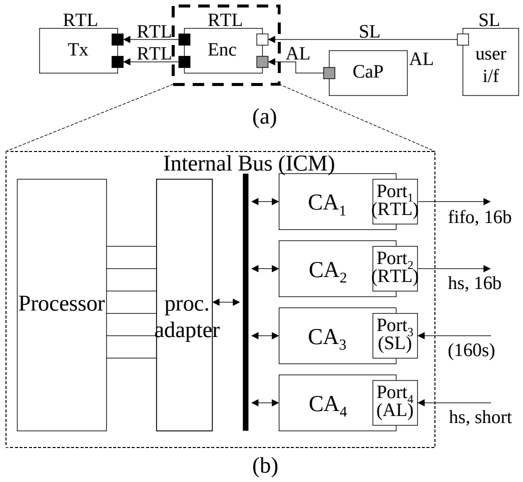


Figure 9: Case 3: Module at RT level.

call relation since the abstraction level of module Enc is architecture level. In the figure, the module adapter provides three internal ports (at architecture level) with AL channel functions for fifo and handshake protocols. Channel adapter, CA_1 receives data from the module adapter, constructs messages (structures of 11 `short`-type data and a `short`-type data) and sends them to the SL channel via external port Port_1 (at system level). In this case, the message construction corresponds to data conversion (to the data type of SL channel, i.e. the generic type of message). Since one external port (Port_1 at system level) corresponds to two internal output ports (Port_1 and Port_2 at architecture level), the module adapter performs data merging with the data received from the two internal ports.

4.2.3 Case 3: Module at RT Level

We apply the generic wrapper architecture to both simulation and synthesis. The application to both is possible since wrappers in simulation and synthesis perform a similar function: adapting the module to the external world by protocol conversion or by transforming channel accesses from (an) abstraction level(s) to (an)other abstraction level(s). Thus, when both applications of the generic wrapper architecture meet each other, to be more specific, when a module requires a wrapper for both protocol conversion and mixed-level cosimulation, we can have a wrapper where high-level simulation models and simulation models of synthesizable components are involved. Case 3 described in this section is the case. The advantage of using such a wrapper is to obtain as accurate timing accuracy as possible in mixed-level cosimulation using simulation models of synthesizable components.

Figure 9 (a) shows a case where Enc and Tx modules are at RT level, CaP module at architecture level, and user i/f module at system level. Enc module is connected with four channels (two at RT level, one at architecture level, and one at system level). At RT level, Enc module is substituted by a processor where it is mapped. Figure 9 (b) shows the processor and the wrapper instance. In the figure, the wrapper consists of processor adapter and channel adapters (with three different abstraction levels). Such a composition is possible since we apply the generic wrapper architecture both to the wrapper for synthesis, i.e. the implementation of the communication coprocessor and to the wrapper for mixed-level cosimulation. In terms of functionality, the processor adapter of the communication coprocessor (exemplified in Figure 5) is equivalent to the module adapter of the wrapper for mixed-level cosimulation since they perform channel resolution (the address decoding in the processor adapter is to select channels). As previously mentioned in Section 2.2, in this case, since the abstraction level of module Enc is RT level, we use an internal bus exemplified in Figure 5 as the internal communication media (ICM) of the wrapper architecture.

In Case 3, the processor adapter performs the functionality of channel interface for the processor. As explained in Section 3.4, for instance, if the processor is an ARM7 processor, the processor adapter is connected with the processor with AMBA protocol [24]. It can also perform interrupt management depending on whether the processor uses interrupt or not. For the functionality of channel resolution, the processor adapter performs address decoding of memory accesses from the processor to enable channel adapters. In the case of Figure 9 (b), for the two RTL output ports, Port_1 and Port_2, the processor adapter receives data (located at a memory area) from the processor and sends them to the corresponding channel adapters (CA_1 and/or CA_2). The channel adapters write them to RTL channels. CA_3 receives messages from the SL channel via Port_3 at system level and sends them (structures of 160 `short`-type data located a memory area) to the processor adapter. From Port_4 at architecture level (via CA_4), the processor adapter receives `short`-type data. When data arrive at external port(s) (e.g. when a `short`-type data item arrives at Port_4), the processor adapter can trigger an interrupt to the processor to notify the data arrival after receiving **ready** signal(s) from the channel adapter(s) corresponding to the external port(s) [17].

4.3 Flow of Generating Cosimulation Models

We are developing an automatic tool (called **wrapper generator**) to generate wrappers for both mixed-level cosimulation and synthesis. To construct the wrapper automatically, we use a library called **wrapper library** which consists of two sets of components: one for module adapters and the other for channel adapters. Each module has at least one module adapter at each of the abstraction levels. Depending on the possibility of module refinement (e.g. possibility of channel splitting or merging), it can have more than one module adapter at an abstraction level. In the case that the wrapper functionality is not required between internal and external ports (e.g. Port_2 of module Enc shown in the case of Figure 6), corresponding internal ports are ignored in the interconnection with the module adapter. Each communication protocol has also one channel adapter for each pair of the abstraction level of module and that of its external port. Note again that the internal communication media (ICM) in the wrapper architecture can be (1) a function call relation or (2) an internal bus depending on the abstraction level of module. This scheme allows to build any wrappers starting from only a few number of basic components, i.e. module adapters and channel adapters.

Figure 10 shows the flow of our cosimulation tool to generate (mixed-level) cosimulation models along with communication refinement from the initial abstract architecture, intermediate architecture, to the physical architecture. In the flow, we use two simulation libraries: one (called **cosimulation library**) for high-level simulation models of modules and channels (at system level and architecture level) and the other (called **synthesizable code library**) for simulation models of synthesizable components (at RT level). Our cosimulation tool selects, for each module/channel in abstract/intermediate/physical architectures, an appropriate simulation model from the libraries according to the abstraction level of module/channel. Then, it generates cosimulation codes with the selected simulation models in possibly different languages (e.g. SystemC, SDL, VHDL, etc.).

With the initial abstract architecture, high-level simulation models are used in cosimulation. During incremental communication refinement, we obtain intermediate architectures where synthesizable components and high-level simulation models coexist, simulation models of both libraries are used in mixed-level cosimulation. In Figure 10, shaded regions represent simulation models of

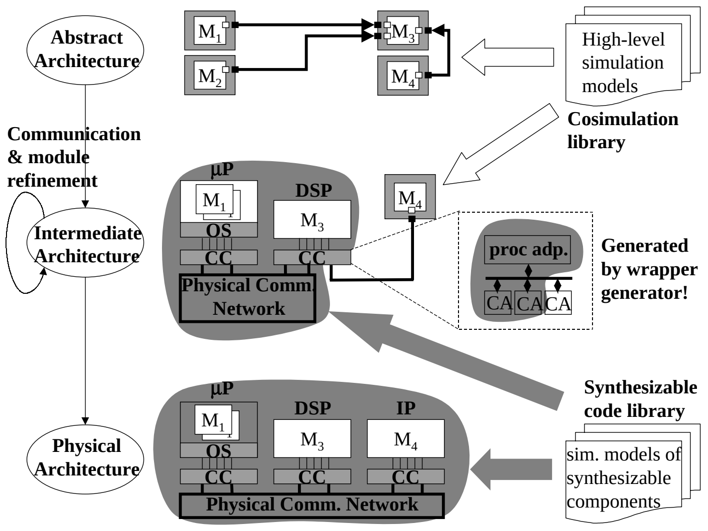

Abstract Architecture
M1
M2
M3
M4
High-level simulation models
Cosimulation library
Communication & module refinement
μP
M1
OS
CC
DSP
M3
CC
Physical Comm. Network
M4
proc adp.
CA
CA
CA
Generated by wrapper generator!
Intermediate Architecture
Synthesizable code library
μP
M1
OS
CC
DSP
M3
CC
IP
M4
CC
Physical Comm. Network
sim. models of synthesizable components
Physical Architecture

Figure 10: Flow of generating cosimulation models.

synthesizable components. To generate wrappers for mixed-level cosimulation, we use a wrapper generator for a module that requires a wrapper as exemplified in Figure 10. Note that we also use the wrapper generator to generate the communication coprocessors (explained in Section 3.4) in the part of physical architecture (in the intermediate architecture). After communication refinement, only simulation models of synthesizable components are used to simulate the physical architecture at RT level.

5 Experiments

As preliminary implementations, we have implemented wrappers for two cases of mixed-level cosimulation (SL-AL cosimulation and AL-RTL cosimulation) and applied them to the mixed-level cosimulation of the IS-95 system. In cosimulation between system level and architecture level, we implemented SL models in SDL [26] since SDL provides the functionality of SL channel: infinite fifo channels and send/receive functions. We implemented AL models in SystemC [8] with its channel models that provide finite fifo channels. In this case, we implemented wrappers for Case 1 and 2 since modules are at either system level or architecture level.

In cosimulation between architecture level and RT level, we use SystemC channels for both AL channels and AL modules, and instruction set simulators (ISS's) of processors as RTL simulation models. In this case, we implemented wrappers for Case 3 since all the channels are at architecture level while modules can be at RT level or architecture level. In the case of mixed-level cosimulation including RTL models, simulation speed varies significantly depending on how many RTL modules are simulated. Thus, the designer can have trade-off between simulation runtime and simulation accuracy in mixed-level cosimulation. In our RTL implementations of the IS-95 system, we mapped four modules, Tx, Rx, Enc, and Dec on four processors, two 68000's and two ARM7's, respectively. By changing the abstraction levels of four modules between architecture level (SystemC model) and RT level (ISS of 68000 or ARM7), we have obtained the variation of simulation speed from 380KHz (all SystemC simulation at architecture level), $\sim$1KHz (single ISS and all the other models at architecture level), $\sim$0.5KHz (two ISS's), to $\sim$0.25KHz (four ISS's). Further details of the experiments are given in [13]

6 Conclusion

To adapt heterogeneous components to the other part of system, we present a generic wrapper architecture that can adapt different communication protocols or different abstraction levels, or both. In this paper, we explained the application of the generic wrapper architecture to mixed-level cosimulation with three abstraction levels of communication: system level, architecture level, and register transfer level. We reported also on its application to mixed-level cosimulation of an IS-95 CDMA cellular phone system.

References

[1] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey, "Communication Architecture Tuners: A Methodology for the Design of High-Performance Communication Architectures for System-on-Chips", *Proc. Design Automation Conf.*, pp. 513–518, June 2000.

[2] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient Exploration of the SoC Communication Architecture Design Space", *Proc. Int'l Conf. on Computer Aided Design*, pp. 424 – 430, Nov. 2000.

[3] M. Drinic, D. Kirovski, S. Meguerdichian, and M. Potkonjak, "Latency-Guided On-Chip Bus Network Design", *Proc. Int'l Conf. on Computer Aided Design*, pp. 420 – 423, Nov. 2000.

[4] S. Vercauteren, B. Lin, and H. De Man, "Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications", *Proc. Design Automation Conf.*, June 1996.

[5] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-Based Design", *Proc. Design Automation Conf.*, pp. 178 – 183, 1997.

[6] C. K. Lennard, P. Schaumont, G. de Jong, A. Haverinen, and P. Hardee, "Standards for System-Level Design: Practical Reality or Solution in Search of a Question?", *Proc. Design Automation and Test in Europe*, pp. 576–585, Mar. 2000.

[7] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers., 2000.

[8] Synopsys, Inc., "SystemC, Version 2.0", available at http://www.systemc.org/.

[9] Coware, Inc., "N2C", available at http://www.coware.com/cowareN2C.html.

[10] J. A. Rawson, "Hardware/Software Co-Simulation", *Proc. Design Automation Conf.*, pp. 439–440, 1994.

[11] L. Séméria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++", *Proc. Asia South Pacific Design Automation Conference*, Jan. 2000.

[12] J-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, and L. Pasquier, "COSY Communication IP's", *Proc. Design Automation Conf.*, pp. 406–409, June 2000.

[13] P. Gerin, S. Yoo, G. Nicolescu, and A. A. Jerraya, "Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures", *Proc. Asia South Pacific Design Automation Conference*, 2001.

[14] G. Nicolescu, S. Yoo, and A. A. Jerraya, "Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design", *Proc. Design Automation and Test in Europe*, 2001.

[15] R. Lysecky, F. Vahid, and T. Givargis, "Techniques for Reducing Read Latency of Core Bus Wrappers", *Proc. Design Automation and Test in Europe*, pp. 84 – 91, Mar. 2000.

[16] Sonics, Inc., "Silicon Backplane μNetwork", available at http://www.sonicsinc.com/Pages/Networks.html.

[17] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip", *to appear in Proc. Design Automation Conf.*, June 2001.

[18] K. Hines and G. Borriello, "Optimizing Communication in Embedded System Co-simulation", *Proc. Int'l Workshop on Hardware-Software Codesign*, pp. 121–125, Mar. 1997.

[19] K. Hines and G. Borriello, "Dynamic Communication Models in Embedded System Co-Simulation", *Proc. Design Automation Conf.*, pp. 395–400, June 1997.

[20] K. Hines and G. Borriello, "A Geographically Distributed Framework for Embedded System Design and Validation", *Proc. Design Automation Conf.*, pp. 140–145, June 1998.

[21] W. O. Cesario, L. Gauthier, D. Lyonnard, G. Nicolescu, and A. A. Jerraya, "An XML-based Meta-model for the Design of Multiprocessor Embedded Systems", *VHDL International User's Forum (VIUF) Fall Workshop*, Oct. 2000.

[22] TIA/EIA-95A, "Mobile Station-Base Station Compatibility Standard for Dual-Mode Wideband Spread Spectrum Cellular Systems", 1995.

[23] S. Yoo, J. Lee, J. Jung, K. Rha, Y. Cho, and K. Choi, "Fast Prototyping of an IS-95 CDMA Cellular Phone: a Case Study", *Proc. the 6th Conference of Asia Pacific Chip Design Languages*, pp. 61–66, Oct. 1999.

[24] ARM Ltd., "ARM7 Data Sheet", *available at http://www.arm.com/ Documentation/UserMans/PDF/ARM7vC.pdf.*

[25] A. Baghdadi, D. Lyonnard, N-E. Zergainoh, and A. A. Jerraya, "An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC", *Proc. Design Automation and Test in Europe*, Mar. 2001.

[26] F. Belina, D. Hogrefe, and A. Sarma, *SDL with APPLICATIONS from PROTOCOL SPECIFICATION*, Carl Hanser Verlag and Prentice Hall International (UK) Ltd., 1991.