Center for Embedded Computer Systems
University of California, Irvine

System-on-Chip Transaction-Level Modeling Style Guide

Dongwan Shin
Lukai Cai
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-24
July, 2004

System-on-Chip Transaction-Level Modeling Style Guide

Dongwan Shin
Lukai Cai
Andreas Gerstlauer
Rainer Dömer
Daniel D. Gajski

Technical Report CECS-TR-04-24
July, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
+1 (949) 824-8919

http://www.cecs.uci.edu

Abstract

Within SoC Design Environment (SCE), starting from an initial system specification, an implementation of the system is created through a series of interactive and automated steps by gradually synthesizing and assembling a system design using components taken out of a set of databases.

SCE uses four models to reflect design decisions during system-level synthesis: specication mode, carchitecture model, network model and communication model. The communication model can be pin-accurate or transaction-level model. This report defines and describes transaction-level communication model (TLM) required for system-on-chip (SoC) design.

Generally, TLMs need to represent processing elements (PEs), memories, communication elements (CEs) and protocol channels connecting components. In this report we aim to provide an exhaustive list of requirements for transaction-level modeling in an automated SoC design flow using the example of concrete models. Specifically, the communication model in this report is used successfully in SCE.

Contents

1 **Introduction** **1**

2 **Overview of Transaction-Level Model** **2**
- 2.1 Layered Structure of Transaction-level Model 6
  - 2.1.1 Layers of Protocol Stack 6
  - 2.1.2 Layered Shells of PEs 7

3 **Processing Elements** **8**
- 3.1 Channel Adapters 9
  - 3.1.1 Protocol Stack Adapters 9
  - 3.1.2 Memory Adapters 9
  - 3.1.3 Bridge Adapters 9
- 3.2 Programmable PEs 11
- 3.3 Hardware PEs 12
  - 3.3.1 Hardware PE with a local memory 12

4 **Memories** **18**

5 **Communication Elements** **19**
- 5.1 Bridges 19
- 5.2 Transducers 21

6 **Synchronization Channels** **21**

7 **Arbitration Channels** **21**

8 **Protocol Channels** **23**

9 **Example** **23**

References **28**

List of Figures

1 SCE design flow. 1
2 Top-level structure of a communication model. 3
3 Top-level code of a communication model. 4
4 Design unit in transaction-level model. 5
5 An example of memory adapter channel. 10
6 An example of bridge adapter channel. 10
7 An example of programmable PE in transaction-level model. 11
8 An example of programmable PE in SpecC (application layer shell). 13
9 An example of programmable PE in SpecC (operating system layer shell). 14
10 An example of programmable PE in SpecC (HAL shell). 15
11 An example of hardware PE in transaction-level model. 15
12 An example of hardware PE with a local memory. 16
13 An example of a hardware PE with a local memory in SpecC (local memory). 16
14 An example of a hardware PE with a local memory in SpecC (hardware PE). 17
15 An example of shared memory in the transaction-level model. 18
16 An example of shared memory in SpecC. 19
17 An example of a bridge in transaction-level model. 20
18 An example of bridge in SpecC. 20
19 An example of a transducer in transaction-level model. 21
20 An example of transducer in SpecC. 22
21 Examples of synchronization channels. 23
22 An example of a protocol channel (I). 24
23 An example of a protocol channel (II). 25
24 An example of transaction level model. 26

System-on-Chip Transaction-Level Modeling Style Guide

D. Shin, L. Cai, A. Gerstlauer, R. Dömer, D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

July, 2004

1 Introduction

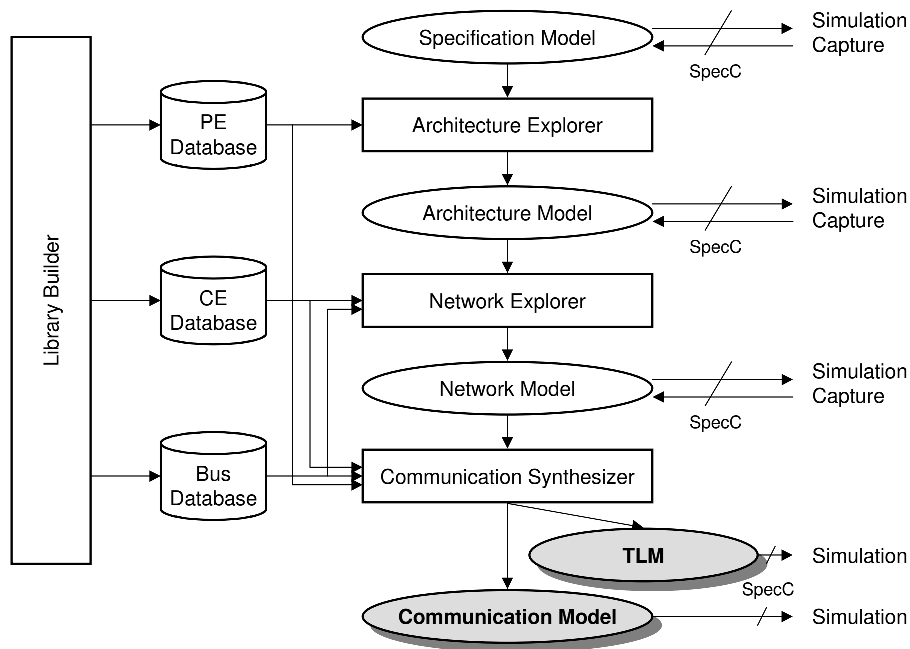

Figure 1: SCE design flow.

SoC Design Environment (SCE) [4] is an environment for capturing the architecture and platform specification of embedded computer systems. It supports the design of such systems from the specification down to the communication model. It does so by capturing of design decisions and automatic generation of new models as shown in Figure 1. SCE follows a **Specify-Explore-Refine** methodology. The design flow starts with a model representing the design functionality (**Specify**). At each following design step, SCE users first explore the design space (**Explore**) and then make design decisions. Integrating those decisions, SCE then automatically generates a new model at lower abstraction level (**Refine**).

In the SCE design flow (Figure 1), four models are used for the representation of a design at different levels of abstraction. Each design model is executable; it can be simulated to verify the correctness of the design and obtain design performance metrics at each design step.

The specification model [5] is the most abstract model, which serves the as an input to SCE tools. It is a purely functional model that captures the functionality of the designed design and should not imply any implementation details.

The architecture model [6] reflects the allocation of system components and the mapping of the specified functionality onto the allocated components. The communication between those components is still described very abstract by message passing channels.

The network model [7] reflects the communication network of the design. It represents the allocation and selection of network stations and logical links between them. While the communication between components in the architecture models is captured end-to-end, in the network model this communication is refined down to point-to-point.

Finally, the communication model incorporates bus protocols into the model. The communication model can be pin-accurate [8] or a transaction-level model. The transaction level model abstracts away the pin-accurate protocol details and thereby gains higher simulation speeds.

All models are captured in SpecC [1], therefore they do have to adhere to the syntax and semantics of the SpecC language. It is recommended that the designer starts with the specification model and later uses the SCE tools to automatically generate lower level models. However, the SCE tools also support manually written low level models as long as they obey certain modeling rules. This report defines the modeling style required for the two SCE communication models (pin-accurate communication model and transaction level model), which are highlighted in the Figure 1.

This report can be used for two purposes. First, it can help user to interpret the code of the communication model, which is automatically generated by the communication synthesizer. Second, it gives the user guidelines to manually write a valid communication model that is acceptable to the SCE tools.

The rest of the report is organized as follows. Section 2 presents the overall structure of a transaction-level model. The major elements of a transaction-level model are described one by one in detail. Section 3 describes the guidelines to model processing elements shown in the transaction-level model. Section 4 describes the modeling of shared memories in the transaction-level model. Section 5 describes the modeling of communication elements, such as bridges and transducers in the transaction-level model. Section 7 describes the modeling of arbitration. Section 6 describes the modeling of interrupt handling for synchronization. In Section 8, the protocol channels are described. Finally, Section 9 describes a example of transaction-level model in SCE.

2 Overview of Transaction-Level Model

The transaction level model abstracts away pin-accurate protocol details and models the protocols as function calls of channels. At the top-level of the design unit, the design consists of concurrent PEs, memories and CEs which communicate through protocol channels. The SCE tools require that the transaction-level model follow certain rules, which will be described below.

Figure 2 and Figure 3 show a template of a valid communication model. A communication model has to be an executable SpecC model, therefore it has to define a `Main` behavior. A communication model is composed of three parts: a stimuli generator, a monitor and a actual design unit as shown in Figure 2. The stimuli generator (`Stimulus`) supplies test vector to the input ports of the design. The output produced by the design unit is observed and validated by the monitor (`Monitor`). The design unit (`Design`) is the target of the design space exploration in SCE environment. SCE tools require the design unit to follow certain modeling rules and restrictions. Note that the modeling rules and restrictions defined in this report only apply

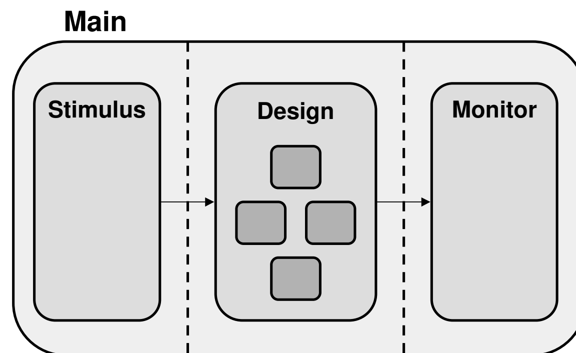

Figure 2: Top-level structure of a communication model.

to the design unit, since the stimuli generator and monitor will not be considered and touched by SCE tools. Therefore, the stimuli generator and monitor can be freely described using any valid SpecC code.

In general, it is hard for SCE tools to determine which behaviors are part of the design unit. Thus the user has to specify the behaviors comprising the design unit. In practice, this is realized by attaching a pre-defined annotation to the communication model.

Rule 1 *A communication model has an annotation `_SCE_TOP_LEVEL`, which contains the name of the top-level behavior of the design unit.*

For example in Figure 3, the annotation in the line 39, specifies the design unit. Once the top-level behavior of the design unit is specified, the SCE tools can conveniently figure out all other behaviors that belong to the design unit.

The design unit of the communication model must obey the following rules.

Rule 2 *Design unit has exactly the same set of ports as the corresponding behavior in the specification model.*

Leaving the interface of the design unit unchanged, allows connecting it to the testbench behaviors (stimulus and monitor) without changing the latter. Thus it allows simulation of the communication model.

Rule 3 *Design unit has exactly one method, the `main()` method, which contains exactly one statement that is a `par` statement.*

Note that by the definition of a hierarchical behavior, each sub-behavior instance inside the design unit can be called at most once in the `par` statement. For example, having two `PE.main()` calls in the `par` statement is not allowed.

Rule 4 *A design unit has a set of sub-behavior instances and a set of channel instances in the transaction-level model.*

The set of sub-behaviors of the design unit can be PE behaviors, memory behaviors, or communication elements behaviors - they will be defined later in more detail. In the transaction-level model, channels represent bus protocols connecting PEs, memories and CEs.

Rule 5 *A design unit has a set of instances of processing elements, memories, communication elements, synchronization channels, arbitration channels and protocol channels.*

```
1  import "c_double_handshake";
2
3  behavior Stimulus(i_sender input) {            // Stimuli creator
4    void main(void) {
5      // while(...) { ... ; input.send(...) ; ... }
6    }
7  };
8
9  behavior Monitor(i_receiver output) {             // Output monitor
10   void main(void) {
11     // while(...) { ... ; output.receive(...) ; ... }
12   }
13 };
14
15 behavior Design(i_receiver input, i_sender output) {  // System design
16   // ...
17
18   void main(void) {
19     // fsm { ... }
20   }
21 };
22
23 behavior Main() {                         // Top level
24   c_double_handshake input, output;
25
26   Stimulus stimulus(input);
27   Design   design(input, output);
28   Monitor  monitor(output);
29
30   int main(void) {
31     par {
32       stimulus.main();
33       design.main();
34       monitor.main();
35     }
36   }
37 };
38
39 note _SER_TOP_LEVEL = "Design";
```

Figure 3: Top-level code of a communication model.

The design unit consists of processing elements (PEs), memories and communication elements (CEs). They are communicating with each other through protocol channels as shown in Figure 4. The synchronization channels are used to support interrupt handling. The arbitration channels resolve multiple concurrent accesses of protocol channels if necessary.

```
1  behavior Design(void)
2  {
3      // Arbitration channels
4      c_mutex cpuAccess;
5      c_mutex slaveAccess;
6
7      // Bus protocol channels
8      Toshiba_GBus CPUBus;
9      Toshiba_GBus SlaveBus;
10     Samsung_KM684002A_Bus SRAMBus;
11
12     // Programmable PE
13     Toshiba_TX49H2_TLM CPU(CPUBus, cpuAccess);
14
15     // Synchronization channels
16     IntrA intrA(CPU);
17     IntrB intrB(CPU);
18
19     // Hardware PEs
20     HW_Standard_DMA_TLM DMA(CPUBus, CPUBus, cpuAccess, intrA);
21     HW_Standard_HW_TLM HW(SlaveBus, intrB);
22
23     // Memory
24     SRAM_TLM SRAM(SRAMBus);
25
26     // Memory controller
27     SRAMCtrl_TLM SRAMCtrl(CPUBus, SRAMBus);
28
29     // Bridge
30     Bridge_TLM Bridge(CPUBus, SlaveBus, slaveAccess);
31
32     void main(void) {
33         par {
34             CPU.main();
35             HW.main();
36             DMA.main();
37             Bridge.main();
38             SRAM.main();
39             SRAMCtrl.main();
40         }
41     }
42 };
```

Figure 4: Design unit in transaction-level model.

The design unit usually contains finer model elements (system components). Those finer model elements capture both the computation architecture and the communication network. The system components are:

- Processing element behaviors model the processing elements (PEs) allocated to perform the desired computation.

- Memory behaviors model the shared memories allocated to store data shared by PEs.
- Communication element behaviors model the bridge and transducer interfacing between different communication protocols.
- Protocol channels or bus wires model the connection between PEs, memories and CEs.

The model elements are defined one by one in the following sections.

2.1 Layered Structure of Transaction-level Model

In order to separate the concerns of modeling different aspects of a functionality in terms of communication and computation, we take a layered approach, which is well known in the network community for describing protocols.

2.1.1 Layers of Protocol Stack

For the communication functionality, we will implement several layers of protocol stack based on OSI reference model: *application layer*, *presentation layer*, *session layer*, *transport layer*, *network layer*, *link layer*, *media access layer*, and *protocol layer*. The upper part of the protocol stack (from application layer to network layer) is implemented during the network exploration. The remaining part of the protocol stack (from link layer to protocol layer) will be inserted by communication synthesizer. In the following, we will describe and define each layer in more detail.

The application layer corresponds to the computation functionality of the system, which defines the behavior of the application implemented by the system design. The application layers describe the processing of data in the system components that exchange data by passing messages over channels.

The presentation layer is used to describe the formatting between typed data in the application and typeless byte-stream transferred through the network. The presentation layer performs the type conversion. If the data in the application is already untyped, the adapter channels may be omitted.

The session layer establishes a connection between components and is responsible for end-to-end synchronization. The session layer marks the interface between application and operating system. In the session layer, different streams originating from different sources may be combined, hence messages of different channels need to be multiplexed on shared streams.

The transport layer is needed to break up the byte stream into smaller packets that will be routed over the network if a transducer participates in the data transfer. The transport layer is modeled as a hierarchical behavior. It contains an instantiation of the transport layer behavior. It may also contain a set of adapter channels that perform the packetization.

The network layer determines routing of data packets from sender to receiver. Assuming reliable stations and links, routing in SoCs is usually done statically, i.e. all packets of a channel take the same fixed, pre-determined path through the system. In a standard bus-based communication, a dedicated logical link is established between two stations for each channel routed through them, assuming the underlying layers support a large enough number of simultaneous logical links between all pairs of stations. The network layer may also contain a set of adapter channels that perform the routing.

The link layer provides services to establish logical links between adjacent stations and to exchange data packets those. The link layer is the highest layer of drivers for external interfaces and peripherals in the operating system. The link layer defines the type of a station (e.g. master/slave) for each of its incoming and outgoing links. As a result, it implements any necessary synchronization between stations by interrupt or acknowledgment. As part of link layer, polling might be required for synchronization, for example, in case of interrupt sharing.

The media access layer is responsible for slicing blocks of bytes into transfer units available at the physical interface. In the process, its implementation has to guarantee that the rates of successive transfers within a block match for all communication partners. Furthermore, the media access layer determines who is allowed to access a shared medium at a given point in time. In other words, it resolves the simultaneous access of bus masters by means of arbitration. Depending on the chosen arbitration scheme, additional arbitration stations are introduced into the system as part of the media access layer.

The protocol layer implements the transfer protocols over a medium in the hardware of component's interface. It is responsible for driving and sampling the external pins according to the protocol timing diagrams and thereby matching the transmission timing on the sender and receiver side. As part of the protocol layer, protocol converters are introduced into the system. Protocol converter connects two busses with different protocols by translating different protocols.

Due to characteristics of standard bus-based SoC communication, layers have been tailored specifically to these requirements. For example, in a reliable bus-based communication architecture, error correction, flow control, buffering or dynamic routing are not required. Therefore, the transport layer is empty and the network layer is largely simplified.

2.1.2 Layered Shells of PEs

In order to separate the concerns of modeling different aspects of a programmable PE, we also take a layered approach [9]. From inside out, five layers have to be followed to model a PE: *application layer shell*, *operating system layer shell. hardware abstraction layer shell. hardware layer shell*, and *bus-functional layer shell*.

The inner-most application layer shell encapsulates the computation required by the application that is executed on the PE. In general, the application layer shell is hierarchically composed of smaller behaviors, each contains a piece of the computation assigned to the PE. For inter-behavior communication inside the application layer, both channels and variables can be connected to the behavior ports. The modeling styles inside the application layer can be found in SpecC specification model reference manual. However, the application layer shell can only have interface type ports and no variable ports. Furthermore, only certain interface types are allowed for the ports.

Rule 6 *An application layer shell has only interface ports and no variable ports. It may contain a set of channel adapter instances such as application and presentation layer. The interface types allowed are as follows:*

`i_sender` *(un-typed)*

`i_receiver` *(un-typed)*

`i_tranceiver` *(un-typed)*

`i_send`

`i_receive`

memory interfaces

The operating system layer shell encapsulates the communication functionality related to operating system on the PE.

Rule 7 *The operating system layer shell contains exactly one behavior instance, of the type application layer shell. It may contain a set of channel adapter instances such session, transport, and network layer.*

Same as the application layer shell, the operating system layer shell only has interface ports and obviously, only untyped interface types are allowed.

Rule 8 *An operating system layer shell has only interface ports and no variable ports. The interface types allowed are as follows:*

`i_sender` *(un-typed)*

`i_receiver` *(un-typed)*

`i_tranceiver` *(un-typed)*

`i_send`

`i_receive`

memory interfaces

Rule 9 *An operating system layer shell needs to have interrupt handling tasks, which invoke the applications that are waiting on the corresponding interrupt.*

The rest of the shells, such as hardware abstraction layer, hardware layer, and bus-functional layer shell, are defined in PE database [9]. The refinement tools will insert the shells into the model and connected them properly.

3 Processing Elements

In the transaction-level model, each processing element is represented by a PE behavior, which implements a part of computation of the specification. It specifies the functionality that has to be implemented on the PE rather than the internal structure of the PE.

Rule 10 *A PE has to be represented by a hierarchical SpecC behavior.*

PEs are represented by SpecC behaviors and should be shown in top-level of behavior hierarchy. The definition of hierarchical behavior can be found in specification model report [5]. he composition type of the PE behavior can be either `seq`, `par`, or `fsm` as defined in LRM.

Rule 11 *A PE has to be annotated by* `_AR_MAPPED_TO`.

The PE allocation table contains the names of the allocated PEs. The PE is assigned to one of the allocated PEs by `_AR_MAPPED_TO`. This annotation is attached during architecture exploration.

Rule 12 *Each PE must have only interface ports and no variable ports.*

The port type of PE behaviors can be the interfaces out of the protocol channels for each bus. The interfaces can have master/slave interfaces. The port type will be either master or slave.

The basic functionality of a PE is data processing or the computation on typed data, such as `integer`, `float` and other data types. However, the communication media can not recognize the type of data when the data is transferred between PEs. Therefore, the PE behaviors in the transaction-level model must provide conversion of typed message to the corresponding untyped message at their interface. Also the synchronization between the PEs for safe data transfers needs to be implemented, e.g. interrupt handling.

In the transaction-level model, communication element (CE) may be needed to interface between two busses with incompatible protocols. Basically the CE receives the data from one PE, buffers it and then send it out to the other PE. Usually a CE has a finite amount of internal buffer, which limits the size of data exchanged between the transducer and the PEs in one transaction. In this case, PEs have to break data into smaller *packets* to avoid over-run of the buffer in the transducer. Therefore PE behaviors in the transaction-level models must also model the packetization process if necessary.

Rule 13 *Each PE has the implementation of a protocol stack at its interfaces.*

Inside each processing element, a protocol stack is implemented. Some layers in the protocol stack are automatically generated by refinement tools while others are taken out of bus database.

3.1 Channel Adapters

During network exploration and communication synthesis, the protocol stack implementations are inlined into the components. The components that implement interfaces need to be inlined into the other components connected by inserting `adapters`. Three types of adapter are available: the protocol adapter, the memory adapter and the bridge adapter.

3.1.1 Protocol Stack Adapters

In order to separate the concerns of modeling different aspects of communication in a PE, a layered approach is taken. From inside out, several layers have to be followed to model a communication functionality of a PE: *application layer*, *presentation layer*, *session layer*, *transport layer*, *network layer*, *link layer* and *media access layer*.

Rule 14 *If a component is connected to other components, the protocol adapters for the protocol stack is instantiated at its interface of the component.*

In the transaction-level model, the implementation of protocol adapters such as presentation, session, transport, network, link and MAC layer are inlined into the component.

3.1.2 Memory Adapters

Rule 15 *If a component is connected to shared memories and hardware components with local memories, then it needs adapters to connect them in transaction-level model.*

A memory adapter is necessary if components are connected to a shared memory. Figure 5 shows an example of memory adapter channel. The memory adapter implements the interface of the memory (`I_AR_MEM_SRAM`), which is defined in the corresponding network model.

3.1.3 Bridge Adapters

Rule 16 *If a component is connected to bridges, it needs to have adapters to make them connected in transaction-level model.*

The bridge adapters are necessary if components are connected to a bridge. Figure 6 shows an example of a bridge adapter channel. The bridge adapter implements the interface of the bridge (`I_Bridge`), which is defined in the corresponding network model.

```
1  channel c_mem_adapter(i_mem_link m) implements I_AR_MEM_SRAM
2  {
3      unsigned long int read_v1(int i) {
4          long int val;
5          m.read(OFS_v1 + i, &val, sizeof(val));
6          return val;
7      }
8      void write_v1(int i, unsigned long int data) {
9          long int val;
10         val = data;
11         m.write(OFS_v1 + i, &val, sizeof(val));
12     }
13 };
```

Figure 5: An example of memory adapter channel.

```
1  channel c_bridge_adapter_TLM(
2      i_tranceiver hw) implements I_Bridge
3  {
4      void receive(void* data, unsigned long len)
5      {
6          hw.receive(data, len);
7      }
8  };
```

Figure 6: An example of bridge adapter channel.

3.2 Programmable PEs

For PEs with fixed, pre-defined interfaces and communication functionality, the transaction-level implementation and bus-functional implementation of the PE are stored in the PE database. The transaction-level implementation of the PE is the abstraction of its pin-level, bus-functional implementation as shown in Figure 7.

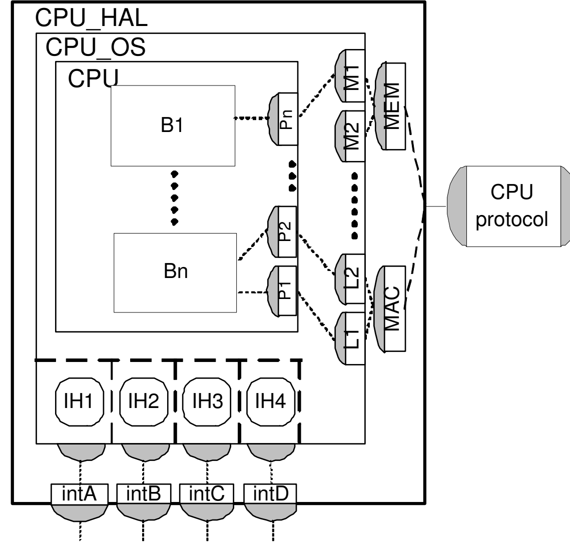

Figure 7: An example of programmable PE in transaction-level model.

Rule 17 *A programmable PE must implement an interface for interrupt handling if it has interrupt pins.*

The interrupt handling is accomplished in HAL shell of the programmable PE. The HAL shell must provide interrupt service routines. The methods in the implementation interface of a PE will be called by the synchronization channels that will be explain later in Section 6.

Rule 18 *A programmable PE must have `_PE_BF_MODEL` and `_PE_BF_BUS` annotation.*

The `_PE_BF_MODEL` indicates the bus function implementation of the PE in the PE database. The transaction level implementation of the PE contains interface ports at its interface. The interface ports will be connected to the associated protocol channels. The associated bus protocol is represented by the `_PE_BF_BUS` annotation.

Rule 19 *The bus functional implementation of a programmable PE must have a `_PE_HAL_MODEL` annotation.*

As described in database manual [9], a transaction-level programmable PE model in the database can be thought of as an additional communication layer that wraps around the PE behavioral model, which in turn comes from the architecture model. A transaction-level model of a PE can consist of several layers of behaviors that create a hierarchy or a tree. A top-level transaction-level layer has to exist that provides a protocol channel interface of the PE.

Rule 20 *The HAL shell of the programmable PE should contain implementations of the MAC layers of the associated bus interface protocols. The MAC layer implementations are stored in PE database.*

The HAL shell of the programmable PE is defined in the PE database. The communication refinement tool takes the HAL shell from the PE database and instantiates OS layer shell in it. The implementation of MAC layers will be connected to the interface ports of the programmable PE.

Rule 21 *The operating system layer shell of the programmable PE can contain the implementations of transport, network and link layer.*

The operating system layer shell is inserted with implementations of transport and network layer during the network exploration. The link layer is implemented during communication synthesis.

Rule 22 *The operating system layer shell can have only one behavior instance, which is the application layer shell of the programmable PE.*

As shown in Figure 8 and Figure 9, the operating system layer shell contains the application layer shell of the programmable PE.

Rule 23 *The HAL shell of the programmable PE can contain only one behavior instance, which is its operating system layer shell.*

As shown in Figure 10, the operating system layer shell is instantiated in HAL shell of the programmable PE.

3.3 Hardware PEs

Programmable PEs are general purpose programmable processors while hardware PEs are application specific hardware devices that need to be synthesized. Figure 11 shows a typical hardware PE in the transaction-level model. An implementation of each protocol stack is instantiated and connected at the interface of the hardware PE.

3.3.1 Hardware PE with a local memory

One common scheme for communication between processors and hardware devices is to use memory mapped-IO. Basically, the hardware unit provides a set of registers that can be accessed by the processor like memory. Note that memory-mapped IO modeling is not applicable to the programmable PEs.

In this case, the hardware PEs in the transaction-level model is described as a combination of a PE behavior and a memory behavior. More specifically, a memory behavior is instantiated inside the application layer behavior. Note a typed memory interface is implemented by the memory behavior as shown in Figure 12, so that behaviors inside the application layer of the hardware PC can access the memory.

Rule 24 *A hardware PE with a local memory must have memory sub-behaviors running concurrently with the rest of sub-behaviors.*

The local memory sub-behavior must run concurrently with other sub-behaviors in the HW PE, because they need to be accessed by the PEs without the help of any synchronization.

Rule 25 *The local memory sub-behavior in a HW PE must implement the read/write interfaces which can be used by the sub-behaviors in the HW PE and can be accessed by other PEs.*

A memory behavior inside a hardware PE provides read/write interfaces, so that the PE can access the local memory by interface method calls.

Figure 13 and Figure 14 show SpecC code of a hardware PE with a local memory.

```
1  // application layer shell of processor
2  behavior Toshiba_TX49H2(
3      I_Bridge br,
4      i_tranceiver L2,
5      i_mem_link ml)
6  {
7      // presentation layer adapters
8      c_cpu_dma_double_handshake c2(L2);
9      c_cpu_dma_double_handshake c3(L2);
10
11     // memory adapter
12     c_mem_adapter mem(ml);
13
14     // bridge adapter
15     c_bridge_adapter c1(br);
16
17     // original behavior
18     B2 b2(c1, c2, c3, mem);
19
20     void main(void) {
21         b2.main();
22     }
23 };
24
25 // interrupt service routines
26 interface IToshiba_TX49H2IntServices
27 {
28     void DMAHandler(void);
29     void HWHandler(void);
30 };
31
32 // interrupt vectore table
33 interface IToshiba_TX49H2IntVectors
34 {
35     void int0handler(void);
36     void int1handler(void);
37 };
```

Figure 8: An example of programmable PE in SpecC (application layer shell).

```
39 // operating system layer shell of processor
40 behavior Toshiba_TX49H2_OS(
41     IToshibaGBusLinkAccess mac,
42     IToshibaGBusMasterMemAccess mem) implements IToshiba_TX49H2IntServices
43 {
44     c_handshake intDMA, intHW;
45
46     // link layer adapters
47     ToshibaGBusMasterLink dma(mac, intDMA, ADDR_DMA);
48     ToshibaGBusMasterLink hw(mac, intHW, ADDR_HW);
49     ToshibaGBusMasterMem shm(mem, ADDR_MEM);
50
51     // bridge adapter
52     c_bridge_adapter_TLM br(hw);
53
54     // application layer shell
55     Toshiba_TX49H2 cpu(br, dma, shm);
56
57     // interrupt service routines
58     void DMAHandler(void) {
59         intDMA.send();
60     }
61     void HWHandler(void) {
62         intHW.send();
63     }
64
65     void main(void) {
66         cpu.main();
67     }
68 };
```

Figure 9: An example of programmable PE in SpecC (operating system layer shell).

```
70 behavior Toshiba_TX49H2_TLM(
71     IToshibaGBusMaster protocol,
72     i_semaphore access) implements IToshiba_TX49H2IntVectors
73 {
74     // MAC layer adapters
75     ToshibaGBusMAC mac(protocol, access);
76     ToshibaGBusMasterLinkAccess link(mac);
77     ToshibaGBusMasterMemAccess mem(mac);
78
79     // operating system layer shell
80     Toshiba_TX49H2_OS cpu_os(link, mem);
81
82     // interrupt vector table
83     void int0handler(void) {
84         cpu_os.DMAHandler();
85     }
86     void int1handler(void) {
87         cpu_os.HWHandler();
88     }
89
90     void main(void) {
91         cpu_os.main();
92     }
93 };
```

Figure 10: An example of programmable PE in SpecC (HAL shell).

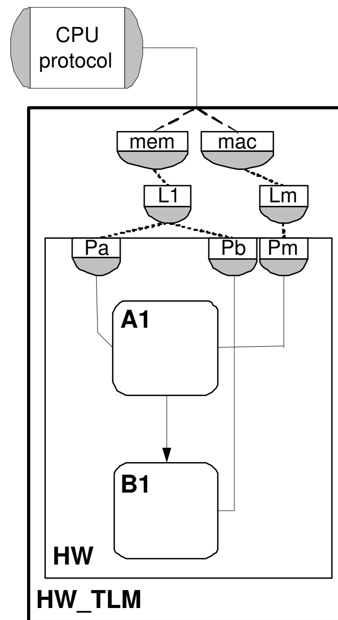

Figure 11: An example of hardware PE in transaction-level model.

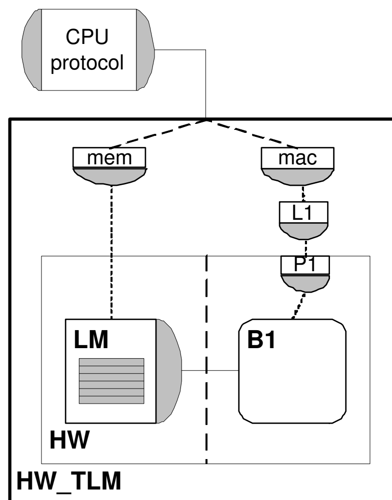

Figure 12: An example of hardware PE with a local memory.

```
1  behavior Local_Mem_TLM(
2      IToshibaGBusSlaveMemAccess shm)
3      implements I_AR_HW_Standard
4  {
5      char mem[MEM_SIZE]; // array or struct type variable
6
7      // memory adapter
8      long int read_v1(void) {
9          long int *ptr;
10         ptr = (long int*) mem+OFS_v1;
11         return *ptr;
12     }
13     void write_v1(long int data) {
14         long int *ptr;
15         ptr = (long int*) mem+OFS_v1;
16         *ptr = data;
17     }
18
19     void main(void) {
20         while (true) {
21             shm.serve(ADDR_MEM, mem, MEM_SIZE);
22         }
23     }
24 };
```

Figure 13: An example of a hardware PE with a local memory in SpecC (local memory).

```
26 behavior HW_Standard(
27     i_tranceiver L1,
28     IToshibaGBusSlaveMemAccess shm)
29 {
30     // link layer adapter
31     c_hw_cpu_double_handshake c1(L1);
32 
33     // local memory
34     Local_Mem_TLM LM(shm);
35 
36     // original behavior
37     B2 b2(c1, LM);
38 
39     void main(void) {
40         par {
41             b2.main();
42             LM.main();
43         }
44     }
45 };
46 
47 behavior HW_Standard_TLM(
48     IToshibaGBusSlave protocol,
49     i_send intrA)
50 {
51     // CPUBus slave MAC layer adapter for general data access
52     ToshibaGBusSlaveLinkAccess access(protocol);
53 
54     // CPUBus slave link layer adapter for general data access
55     ToshibaGBusSlaveLink L1(access, intrA, ADDR_HW);
56 
57     // CPUBus MAC layer adapter for memory access
58     ToshibaGBusSlaveMemAccess shm(protocol);
59 
60     HW_Standard HW(L1, shm);
61 
62     void main(void) {
63         HW.main();
64     }
65 };
```

Figure 14: An example of a hardware PE with a local memory in SpecC (hardware PE).

4 Memories

Memory is represented by a SpecC behavior and should be shown in the top-level of the behavior hierarchy. Each memory, corresponding to a transaction-level implementation needs to exist in the PE database. Since the transaction-level model captures type-less data communication, the memory model also should provide byte-oriented access instead of type-specific. Figure 15 shows an example of a shared memory in the transaction-level model.

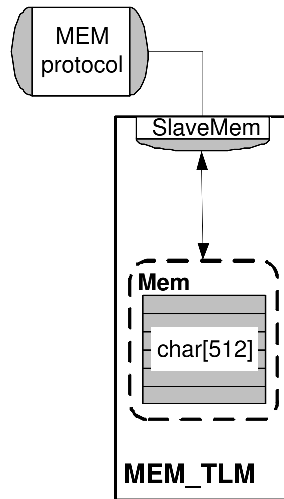

Figure 15: An example of shared memory in the transaction-level model.

The SCE tools require that memories follow certain rules, as presented below.

Rule 26 *A memory is represented by a SpecC behavior, which has only interface ports and no variable ports.*

A memory is represented by a SpecC behaviors. Its ports are of a interface type defined in the protocol channel for each bus. The interface must be a slave interface of the associated memory interface protocol.

Rule 27 *A memory behavior generally has one variable, which is a C `char` type array. This array contains all variables stored in the memory as its members.*

The size of memory is the same as the size of the `char` type array variable. The memory behavior provides methods to read/write the `char` type array variable.

Rule 28 *A memory behavior contains an instance of the MAC layer implementation from the associated memory interface protocol.*

A slave memory interface protocol from the bus database is instantiated in the MAC layer of a memory behavior.

Rule 29 *A memory behavior must be bus slave. Thus, the memory behavior continuously calls a `serve` method in its `main` method.*

The `serve` method is defined in the bus database and provides access to the `char` type array variable in the memory behavior. The `serve` method takes the base address of the memory, the `char` type array variable, and the size of the memory as arguments.

Figure 16 shows the SpecC code of a shared memory. In the example, all variables that are stored in the memory are packed into a `char` type array variable `mem`. As required the `serve` method is invoked in the main body.

```
1  behavior SRAM_TLM(
2      IKM684002ASlave protocol)
3  {
4      char mem[MEM_SIZE];
5
6      // SRAMBus MAC layer adapter
7      KM684002AMemServe shm(protocol);
8
9      void main(void) {
10         while (true) {
11             shm.serve(ADDR_MEM, mem, MEM_SIZE);
12         }
13     }
14 };
```

Figure 16: An example of shared memory in SpecC.

5 Communication Elements

elements such as transducers and bridges are represented by SpecC behaviors and should be instantiated in the top-level behavior. A CE behavior may have sub-behaviors. In order to use a behavior as a communication element it has to be annotated by `_CR_MAPPED_TO`.

Rule 30 *A communication element is represented by a hierarchical SpecC behavior.*

Rule 31 *A CE may only have interface ports and no variable ports.*

The port type of the CE is of the interface type defined in the according bus protocol channel. The interfaces can be either master or slave interfaces.

5.1 Bridges

In the network model a bridge is captured on an abstract level. During the communication synthesis this abstract model is replaced by a more concrete version out of the CE database as shown in Figure 17.

Rule 32 *A bridge must have `_CE_BF_MODEL` and `_CE_BF_BUS` annotation.*

The `_CE_BF_MODEL` indicates the bus functional implementation of the bridge in the CE database and `_CE_BF_BUS` contains a list of the two busses to be connected.

Rule 33 *Bridges do implement only the MAC layer of a protocol stack.*

The corresponding SpecC code is shown in Figure 18.

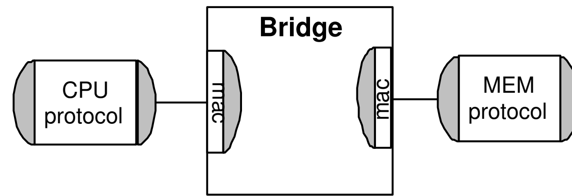

Figure 17: An example of a bridge in transaction-level model.

```
1  behavior Bridge_TLM(
2      IToshibaGBusSlave cpuBus,
3      IToshibaGBusMaster slaveBus,
4      i_semaphore slaveAccess)
5  {
6      // CPUBus slave MAC layer adapter
7      ToshibaGBusMAC mac(slaveBus, slaveAccess);
8
9      void main(void) {
10         bit[35:0] addr;
11         bit[63:0] d;
12         unsigned bit[10] t;
13         while (true) {
14             addr = 0x00000000;
15             t = cpuBus.Listen(&addr, 0x00000000);
16             if (((unsigned int)addr) == ADDR_HW) {
17                 switch((unsigned int)(t[1:0])) {
18                   case 1:   // cpu read
19                     d = mac.LoadCycle(addr, t[9:2]);
20                     cpuBus.WriteCycle(d);
21                     break;
22                   case 2:   // cpu write
23                     d = cpuBus.ReadCycle();
24                     mac.StoreCycle(addr, d, t[9:2]);
25                     break;
26                 }
27             }
28         }
29     }
30 };
```

Figure 18: An example of bridge in SpecC.

5.2 Transducers

During the communication synthesis, transducers are treated like PEs. The protocol stack at the interface of the transducers needs to be implemented. In the transaction-level model, a transducer implements the network layer, link layer and the MAC layer of the selected protocol stacks. A PE on the other hand implements all layers except protocol layer. An example is shown in Figure 19.

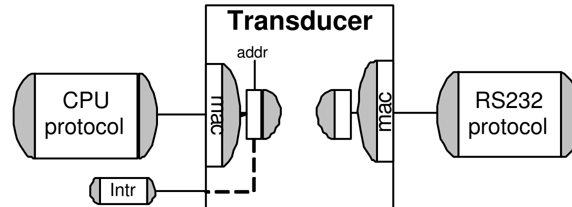

Figure 19: An example of a transducer in transaction-level model.

Rule 34 *Transducers have to implement network, link and MAC layer of the chosen protocol stacks.*

Rule 35 *Transducers need to implement an interrupt based synchronization scheme.*

The synchronization by interrupt is necessary in the transducer as part of the link layer implementation.

6 Synchronization Channels

The synchronization between programmable PEs and other system components is accomplished by interrupts. A synchronization channel is represented by a SpecC channel and should be instantiated in top-level behavior.

Rule 36 *A synchronization channel must have one port of the interface type defined in the HAL shell of the programmable PE that provides the interrupt handling.*

Each synchronization channel implements an interrupt pin of the programmable PE where it is attached. It implements a `i_send` interface to assert interrupt to the programmable PE. The programmable PE can be mapped to the port of the synchronization channel.

Rule 37 *A synchronization channel must invoke in its* `send` *implementation the interrupt handler method (as defined in the HAL shell) of the programmable PE.*

The synchronization channel must have the `send` method in the `i_send` interface, that invokes the interrupt service routine of the programmable PE.

Figure 21 shows examples of synchronization channels for interrupt handling. In the example, `IntrA` has only one port, `IToshiba_TX49H2IntVectors`, implemented by the programmable PE. In the `send` interface method of the `IntrA`, the interrupt handler for `IntrA` interrupt line is invoked.

7 Arbitration Channels

If multiple masters are connected to the same bus, arbitration is necessary to resolve concurrent bus accesses. In the transaction-level model, this arbitration is accomplished by arbitration channels.

```
1 // network model of transducer
2 behavior Tx_NET(
3     i_sender link_GBus,
4     i_receiver link_SIO)
5 {
6     void main(void) {
7         char buf[128];  // buffer
8         unsigned long int len;
9         while (true) {
10            link_SIO.receive(&len, sizeof(len));
11            link_SIO.receive(buf, len);
12            link_GBus.send(buf, len);
13        }
14    }
15 };
16
17 behavior Tx_TLM(
18     IToshibaGBusSlave cpuBus,
19     IRS232BusMaster sioBus,
20     i_send intrA)
21 {
22     // CPUBus slave interface (MAC and link layer)
23     ToshibaGBusSlaveLinkAccess cpuAccess(cpuBus);
24     ToshibaGBusSlaveLink cpuLink(cpuAccess, intrA, ADDR_SIO);
25
26     // RS232Bus master interface (MAC and link layer)
27     RS232BusLinkAccess sioAccess(sioBus);
28     RS232BusMasterLink sioLink(sioAccess);
29
30     // network model of transducer
31     Tx_NET Tx(cpuAccess, sioAccess);
32
33     void main(void) {
34         Tx.main();
35     }
36 };
```

Figure 20: An example of transducer in SpecC.

```
1  // synchronization channel for interrupt A
2  channel IntrA(
3      IToshiba_TX49H2IntVectors cpu) implements i_send
4  {
5      void send(void) {
6          cpu.int0handler();
7      }
8  };
9
10 // synchronization channel for interrupt B
11 channel IntrB(
12     IToshiba_TX49H2IntVectors cpu) implements i_send
13 {
14     void send(void) {
15         cpu.int1handler();
16     }
17 };
```

Figure 21: Examples of synchronization channels.

Rule 38 *Each master on the bus has to be connected to a arbitration channel.*

In the example shown in Figure 4, the arbitration is implemented by `c_mutex` channels. The instances of the arbitration channels are connected to the ports of the master components (`CPU` and `DMA` on the `CPUBus` and `Bridge` on the `SlaveBus`).

8 Protocol Channels

In the transaction-level model, protocol channels represent the abstract bus protocols that connect PEs, memories, and CEs. Bus protocols are represented by SpecC channels and should be instantiated in top-level behavior. Each channel implements the corresponding bus protocol and abstract away the pin-level communication functionality.

Rule 39 *Protocol channels must provide master and slave interfaces. They connect system components.*

As shown in Figure 22 and Figure 23, the interface provided by a protocol channel is invoked in the lowest layer of the protocol stack of a system component [1]. Master components use a master protocol interface, slave components use a slave protocol interface. They are connected by the protocol channel.

9 Example

In the example as shown in Figure 24, the design has three protocols: *SWProtocol*, *HWProtocol* and *MemProtocol*. Within each bus group, unique bus addresses and interrupts for synchronization are assigned to each logical link channel. On the *SWProtocol* side, the memory is assigned a range of addresses with a base address plus offsets for each stored variable.

As part of the MAC layer implementation in the transaction level model, arbitration protocol channel *Arbiter* is connected to protocol channels to regulate multiple bus accesses of masters on *SWProtocol*.

[1] Note that although the example is for the TLM, individual signals are handled in this example.

```
// master interface
interface IToshibaGBusMaster {
    bit[31:0] LoadCyle(bit[35:0] addr, bit[3:0] be);
    void StoreCycle(bit[35:0] addr, bit[31:0] data, bit[3:0] be);
    bit[31:0] LoadWord(bit[35:0] addr);
    void StoreWord(bit[35:0] addr, bit[31:0] val);
};

// slave interface
interface IToshibaGBusSlave {
    unsigned bit[5:0] Listen(bit[35:0] *addr, bit[35:0] mask);
    void WriteCycle(bit[31:0] val);
    bit[31:0] ReadCycle(void);
    void WriteWord(bit[31:0] val);
    bit[31:0] ReadWord(void);
};

// protocol channel
channel Toshiba_GBus( )
    implements IToshibaGBusMaster, IToshibaGBusSlave
{
    signal bit[35:0] GA;        // address
    signal bit[31:0] GDOUT;     // data output
    signal bit[31:0] GDIN;      // data input
    signal bit[3:0]  GBE = 0;   // byte enable
    signal bit[1]    GRD = 1;   // read
    signal bit[1]    GWR = 1;   // write
    signal bit[1]    GACK = 1;  // acknowledge

    bit[31:0] LoadCycle(bit[35:0] addr, bit[3:0] be) {
        bit[31:0] data;
        GA = addr;
        GBE = be;
        GRD = 0;
        waitfor (CPU_CLK/2);
        while(GACK != 0)
            wait(GACK falling);
        waitfor (CPU_CLK/2);
        data = GDIN;
        GRD = 1;
        return data;
    }
```

Figure 22: An example of a protocol channel (I).

```
43      void StoreCycle(bit[35:0] addr, bit[31:0] data, bit[3:0] be) {
44          GA = addr;
45          GBE = be;
46          GWR = 0;
47          GDOUT = data;
48          waitfor (CPU_CLK/2);
49          while(GACK != 0)
50              wait(GACK falling);
51          waitfor (CPU_CLK/2);
52          GWR = 1;
53      }
54      bit[31:0] LoadWord(bit[35:0] addr) {
55          return LoadCycle(addr, 1111b);
56      }
57      void StoreWord(bit[35:0] addr, bit[31:0] val) {
58          StoreCycle(addr, val, 1111b);
59      }
60
61      unsigned bit[5:0] Listen(bit[35:0] *addr, bit[35:0] mask) {
62          *addr = GA & mask;
63          return GBE @@ GRD @@ GWR;
64      }
65      void WriteCycle(bit[31:0] val) {
66          GDIN = val;
67          GACK = 0;
68          waitfor (5);
69          GACK = 1;
70      }
71      bit[31:0] ReadCycle(void) {
72          bit[31:0] val;
73
74          val = GDOUT;
75          GACK = 0;
76          waitfor (5);
77          GACK = 1;
78          return val;
79      }
80      bit[31:0] ReadDoubleWord(void) {
81          return ReadCycle();
82      }
83      void WriteWord(bit[31:0] val) {
84          WriteCycle(val);
85      }
86  };
```

Figure 23: An example of a protocol channel (II).

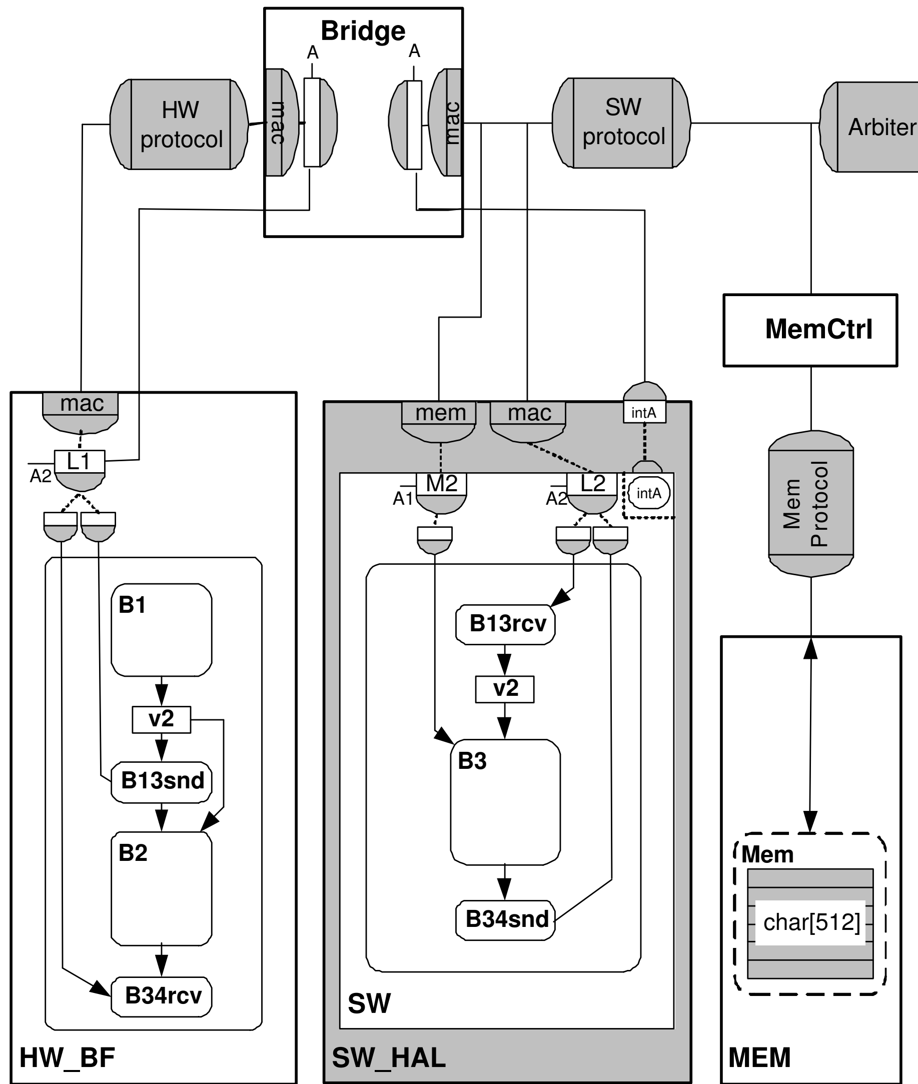

Figure 24: An example of transaction level model.

In the transaction level model (Figure 24), the application layer shell from the architecture model and the implementation of logical link and media access layers are instantiated inside a hardware abstraction layer shell of processor components or inside a bus-functional hardware shell for synthesizable components. Depending on the type of transfer (e.g. memory transfer or normal message passing transfer), different types of media access implementations can be implemented and inlined into a component.

The shaded parts (the shell *SW_HAL* for *SW* and MAC layer implementations) in the figure indicate that they are pulled from the (PE and bus) databases instead of being generated by tools. Logical link layer adapters are automatically generated and inserted into the application layer shell by the tool. In case of processors, a template for interrupt handlers is defined as part of the HAL shell in the PE database. The behavior of interrupt handlers is automatically generated by the communication synthesizer on top of this template.

References

[1] R. Dömer, A. Gerstlauer and D. D. Gajski. *SpecC Language Reference Manual, Version 2.0*, SpecC Technology Open Consortium (STOC), Japan, December 2002.

[2] SpecC Technology Open Consortium. http://www.specc.org.

[3] SpecC Compiler V2.2.0, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

[4] L. Cai, A. Gerstlauer, S. Abdi, J. Peng, D. Shin, H. Yu, R. Dömer and D. D. Gajski. *System-on-Chip Environment (SCE Version 2.2.0 Beta): Manual*, Technical Report CECS-TR-03-45, Center for Embedded Computer Systems, University of California, Irvine, December 2003.

[5] A. Gerstlauer, K. Ramineni, R. Dömer and D. D. Gajski. *System-on-Chip Specification Style Guide*, Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[6] J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Architecture Modeling Style Guide*, Technical Report CECS-TR-04-22, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[7] D. Shin, J. Peng, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Network Modeling Style Guide*, Technical Report CECS-TR-04-23, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[8] D. Shin, A. Gerstlauer, R. Dömer and D. D. Gajski. *System-on-Chip Communication Modeling Style Guide*, Technical Report CECS-TR-04-25, Center for Embedded Computer Systems, University of California, Irvine, July 2004.

[9] A. Gerstlauer, L. Cai, D. Shin, R. Dömer and D. D. Gajski. *System-on-Chip Component Models*, Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, August 2003.