Communication Link Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski

Technical Report CECS-04-16
June 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, gajski}@cecs.uci.edu

Communication Link Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski

Technical Report CECS-04-16
June 10, 2004

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans, gerstl, gajski}@cecs.uci.edu

Abstract

Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs. In this report, we propose automatic generation of communication architecture from communication link model where system components communicate through logical links of network architecture. Automatic model refinement for communication architecture enables rapid design space exploration in order to achieve the required productivity gains. The experimental results show the benefits of our methodology and demonstrate the effectiveness of our automatic model generation for communication design.

Contents

List of Figures

Communication Link Synthesis for SoC

Dongwan Shin, Andreas Gerstlauer and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs. In this report, we propose automatic generation of communication architecture from communication link model where system components communicate through logical links of network architecture. Automatic model refinement for communication architecture enables rapid design space exploration in order to achieve the required productivity gains. The experimental results show the benefits of our methodology and demonstrate the effectiveness of our automatic model generation for communication design.

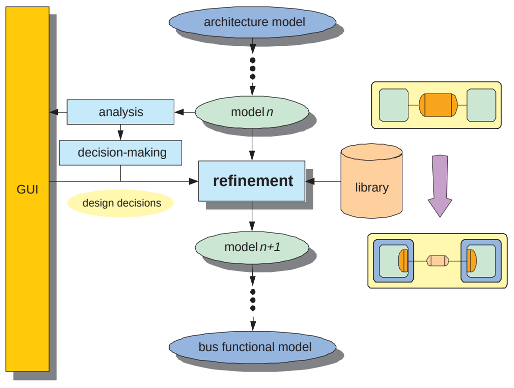


Figure 1. Refinement-based communication design methodology.

1. Introduction

With the ever increasing complexity of system level designs and the pressure of the time-to-market in the design of System-on-Chip (SoC), communication between components is becoming more and more important. Communication design for SoCs poses the unique challenges in order to cover a wide range of architectures while offering new opportunities for optimizations based on the application specific nature of system designs.

We propose refinement-based communication design methodology which is a set of models and transformations between models that subdivide the design flow into smaller, manageable steps as shown in Figure 1. With each step, a new model of the design is generated, where a model is a description of design at certain level of abstraction, usually captured in system level design languages. The abstraction level of each model is defined by the amount of implementation detail in terms of structure or order.

In each of tasks, users can make design decisions manually by using an interactive graphical user interface (GUI), for example, while transformations from one model into another can be accomplished automatically by refinement rules or model guidelines. After each refinement step in the synthesis flow, a corresponding model of system is generated, which means that design decisions made in each design task are reflected in the generated models.

Finally, metrics estimation, designers have to simulate generated model to verify the functionality and to estimate design metrics. In general, the design metrics are not satisfactory in the first trial. Therefore, many iterations of these tasks may be needed for each design step.

Figure 2 shows the communication synthesis flow [Ger03] which is divided into two tasks: *network synthesis* and *communication link synthesis*. During the network synthesis, the topology of communication architecture is defined and abstract message passing channels between system components are mapped into communication between adjacent communication stations (communicating system components, e.g. processing elements, communication elements) of the system architecture. The network topology of communication stations connected by logical link channels is defined, bridges and other communication elements are allocated as necessary, abstract message passing channels are routed over sets of logical link channels. The result of the network synthesis step is a refined communication link model of the system. The communication link model represents the topology of communication architecture where components and additional communication stations communicate with logical link channels.

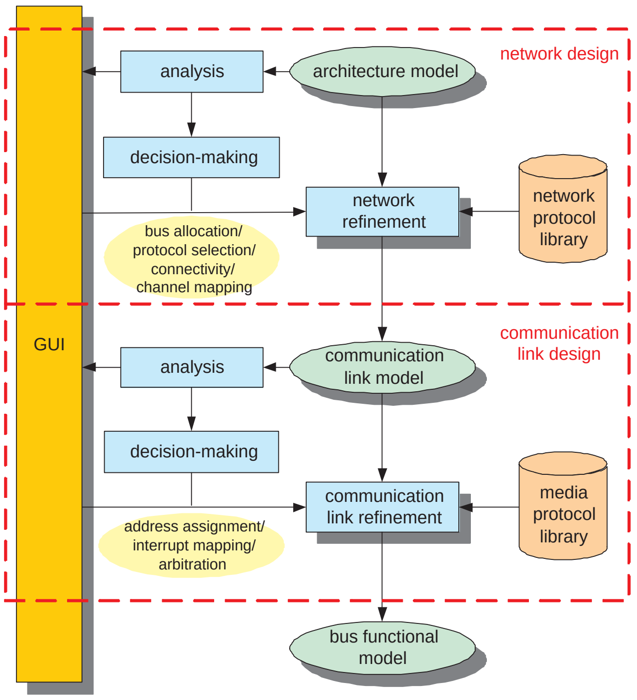


Figure 2. Communication synthesis flow.

There have been some reasons that we have chosen communication link model as an intermediate model in communication synthesis flow [Ger03]. First, designer would like to see the topology of communication architecture and estimate the performance of a communication architecture at early stage of communication synthesis. Secondly, the simulation speed of communication link model is almost same as that of architecture model according to experimental result in [Ger03]. Finally, communication delay of communication link model is not as accurate as that of bus functional model but the accuracy of communication delay in communication link model can be improved by efficient and accurate estimation tools.

Communication link synthesis is followed by network synthesis. Logical links channels between adjacent stations are then grouped and implemented over an actual communication medium (e.g. system busses). During communication link synthesis, each group of logical link channels can be grouped, and be implemented separately onto a communication medium with associated protocol. The parameters such as addresses and interrupts for synchronization are assigned to each logical link channel.

As a result of the communication synthesis process, a bus functional model of a system is generated. The bus functional model is a fully structural model where components are connected via pins and wires and communicate in a cycle-accurate manner based on media protocol timing specifications. In the backend process, behavioral descriptions of computation and communication in each component of the bus functional model are then synthesized into targeted hardware or software implementations.

In this report, we look at how we speed up the communication link synthesis process by enabling automatic model refinement. The rest of the report is organized as follows. Section 2 gives an overview of related works. Section 3 shows our communication link synthesis flow and Section 4 describes the media protocol library. Section 5 looks at the tasks of communication link refinement. Finally, we present experimental results in Section 6 and wind up with a summary and conclusion.

2. Related Works

Narayan and Gajski presented interface refinement step integrated in *SpecSyn* [GVNG97]. Their interface refinement step consists of two steps: bus generation and protocol generation. The bus generation algorithm [NG94a] determines the bit width of a bus implementing a set of abstract communication channels. Using the computed bus width, the protocol generation step [NG94b] defines the exact mechanism of transferring data over the bus. Different low level protocols, such as full handshake, half handshake, fixed delay and even hardwired ports are supported by *SpecSyn*.

Daveau et al. [DMBIJ97] proposed an algorithm which handles bus protocol selection and interface generation which is based on allocation of communication units. They support low level protocols such as bidirectional handshake, single and dual FIFO in a communication library.

Bolsen et al. [BML$^+$97] developed system environment called *CoWare*, which supports specification of heterogeneous communicating processes, which are specified in DFL, VHDL, or C. Their main focus is system integration and handling of communication in embedded systems. The heterogeneous specification is mapped onto different processing elements. The inter-process communication is abstracted with point-to-point communication channel with rendez-vous semantics. It is implemented by remote procedure call (RPC). Hardware/software communication channels are mapped onto a fixed communication architecture, which is based on several library models implementing different I/O scenarios. For software, the communication procedures are captured as parameterized C functions that are mapped onto a software model, i.e. they adapt to processor specific I/O handling, interrupt handling, etc. For hardware, a hardware interface cell is generated to connect with a handshake protocol to an I/O control unit.

Knudsen and Madsen [KM98] presented a communication estimation model and shows the importance of integrating communication protocol selection with hardware/-software partitioning by the use of this model.

In [GABP98], Gogniat et al. proposed the communication interface generation method from partitioned and scheduled system model for HW/SW interfaces for codesign of embedded systems.

Balarin et al. [BGJ+97] developed *POLIS* which focused on control dominated applications with system architectures composed of a single processor surrounded by custom or library hardware. *POLIS* uses Codesign Finite State Machines (CFSM) as the internal representation for a system description, separating communication, behavior and timing of the system. The communication model is globally asynchronous, locally synchronous, with nonblocking finite buffers between CFSMs. C code and HDL code are generated from the CFSMs mapped to software and hardware, respectively. Except for the I/O drivers and code generated from the CFSMs, software code consists of a generated application specific operating system for the selected processor. All communication within software or between software and hardware occurs through shared memory, I/O ports or memory mapped I/O. The synchronized hardware includes address decoders, multiplexers, latches and glue logic. Special purpose hardware must follow a simple, data/strobe based protocol in order to be interfaceable with other CFSMs.

Lyonnard et al. [LYBJ01] [CBG+02] presented interesting schemes for putting together heterogeneous components on a bus using wrappers for design of application specific multi-processor SoCs. In their approach, architecture is generated from an architecture template by setting the parameters for the four types of elements: processor local architectures, communication coprocessors, IP components and communication network.

However, most of research work has been on automatic decision-making on communication topology of system architecture. There has been little attention paid to automatic generation of network topology of communication architecture from the partitioned, scheduled architecture model. Samar et al. [ASG03] [SAG04] proposed automatic generation of communication model from the partitioned architecture model. But they addressed synchronization by interrupt handling and data formating issues but did not address transducer synthesis and integration of IPs into a design.

3. Refinement-based Communication Link Synthesis

Communication link synthesis is the process of moving from one model to the next, gradually transforming models and refining the logical link communication in the communication link model down to its bus functional implementation. Communication link synthesis groups logical links between adjacent communication stations, which are implemented over a system bus. As a result of the communication link synthesis, a bus functional model of a system is generated. The bus functional model is a fully structural model in which components are connected via pins and wires and communicate in time accurate manner based on protocol timing specifications of the system busses.

Communication link synthesis implements the functionality of link layer, media access layer and protocol layer and inlines them into corresponding components. The link layer defines the type of a communication station (e.g. master/slave on a bus) for each of its incoming or outgoing links. It is also responsible for implementing synchronization between communication stations, e.g. by interrupts or polling in case of interrupt sharing.

The media access layer is responsible for slicing blocks of bytes into bus word Furthermore, the media access layer resolves simultaneous bus accesses of components through arbitration. Depending on the arbitration scheme chosen, additional arbitration stations are introduced into the system as part of the media access layer.

Finally, the protocol layer 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 sides.

We begin with a communication link model, output model of the network synthesis process, which represents communication topology of the communication architecture. The components on top level of the design communicate with each other via logical link channels. Each channel comprises of the data itself and *send/receive* methods that enable the data transaction.

The user provides a set of design decisions such as *master/slave assignment for components*, *address assignment and interrupt mapping for components or messages*, and *arbitration scheme and bus access priorities*. System busses may be inserted in the design by instantiations from a media protocol library. With these inputs, the communication refinement tool produces an output model, bus functional model that reflects the bus architecture of the system. In the output model, the top level of the design consists of system components and wires of the system busses. The components themselves are refined to their bus functional models that communicate using the system busses.

3.1. Design Decisions

The refinement engine works on directions given to it by the communication link design decisions. The decision making process can either be automated or interactive as per the user's methodology. However, the decisions must input to the refinement engine using a specific format. Some typical features of the communication architecture include *master/slave selection*, *address assignment*, *interrupt mapping* and *priorities of components on a bus*. Based on these decisions, the refinement engine imports the required protocols from the media protocol library and generates interfaces and drivers for components so that they may talk over the system busses. For the purpose of our implementation, we annotated the input model with the set of design decisions. The refinement tool then detects and parses these annotations to perform the requisite model transformations.

3.1.1 Master/Slave Assignment

As a part of link layer implementation, system components in communication link model should be either master or slave on a bus. Then communication link refinement tool will take protocols for master or slave out of a media protocol library and insert into the corresponding components.

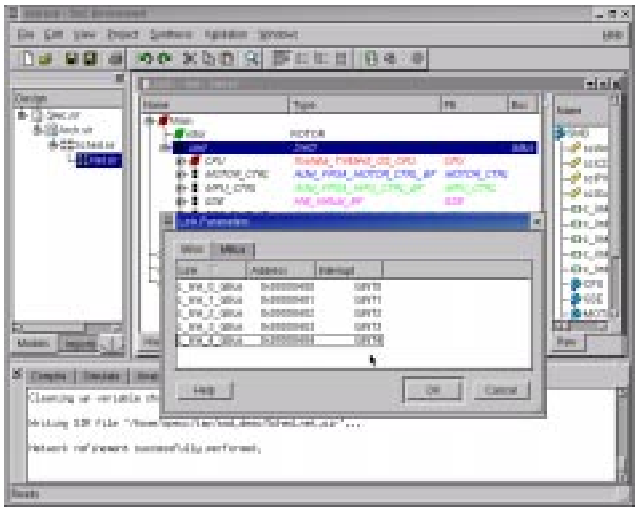
3.1.2 Interrupt Mapping

To implement the blocking semantics of the message passing communication, system should perform the proper synchronization of data transfers between components. Depending on the bus, synchronization can be inherent in the protocol. The synchronization can be performed by *interrupting* or *polling*. With polling, master have to poll all slaves to see if they want to do anything and thus polling might have serious effects on throughput of the system. But with interrupting, master can run along nicely until such time that a slave wants to signal an event, which means the system does not waste time going to the slave, the system lets slaves come to the master where they are ready. Generally, in bus based systems, programmable processors have interrupt controller to handle multiple interrupts from slaves which need to be arbitrated by interrupt controller.

The interrupt controller accepts requests from slaves, determines which of the incoming request is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the programmable processor. The interrupt controller places the service routine address (vector address) on the data bus. The programmable processor services the interrupting device from this address.

3.1.3 Address Assignment

As part of link layer implementation, addressing of channels need to be determined. In general, bus addresses are a combination of source component, destination component, and ID of the message to be transferred. The address of a message passing channel on a system bus should be assigned. The address will be used to distinguish messages polling as part of synchronization in case of interrupt sharing. Figure 3 show the screenshot of address assignment for channels.



```
Cleaning up variable ch[illegible]
Writing SIR file "[illegible]"...
Network refinement successfully performed.
```

Figure 3. A screenshot for address assignment and interrupt mapping for channels.

3.1.4 Arbitration

If the bus supports multiple masters connected to the bus, it has to supply an arbitration scheme that is used to regulate accesses to the shared bus wires. Arbitration can be distributed or centralized. In a centralized arbitration scheme, the master side of the arbitration protocol instantiated in each master communicates with the slave side of the arbitration protocol instantiated in an additional arbiter component attached to the bus. In a distributed arbitration scheme, there is no slave side of the arbitration protocol and the master sides of the protocol in each master regulate accesses among themselves. For the arbitration, we have to select arbitration scheme and priorities of masters on the system bus, which are annotated into each component.

3.2. Transaction Level Modeling of Input Model

The input model of the communication link synthesis is the communication link model which is generated from network synthesis as shown in Figure 4. The communication link model reflects communication topology of the communication architecture where components communicating via logical link channels which implement message passing semantics.

For each data item (variable) communicated between components, the upper layers of protocol stack such as presentation layer, session layer, transport layer and network layer are inlined into the corresponding components. In

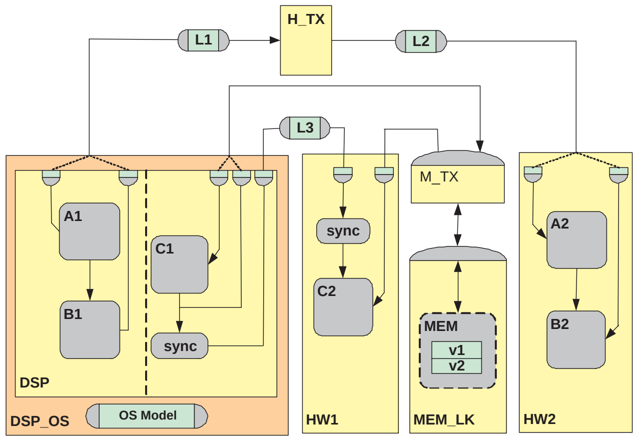


Figure 4. Communication link model.

the communication link model, end-to-end channels have been replaced with point-to-point logic link channels between components that will later be physically directly connected through bus wires. In the communication link model, communication elements are inserted from network protocol library and synthesized in order to connect two different busses.

3.3. Bus Functional Model of Communication Architecture

Bus functional model is the final result of the system synthesis process and defines structure of the system architecture in terms of components and connections. Computation in specification has been mapped onto components and communication onto busses. At the top-level of behavior hierarchy, a design consists of concurrent, non-terminating system components that communicate through the busses.

Inside the component, behavior models of bus drivers and bus interfaces (protocol stack) describe the communication functionality of the component, i.e. the implementations of all communication layers in protocol stack are inlined into the components in form of channel adapters and implements the abstract transaction in architecture model over busses. Those bus adapters (a stack of communication layers) specify how the component implements the semantics of the abstract transactions by driving and sampling the wires of the system bus. Behavioral blocks inside the component, in turn, connect to the equivalent message passing channel interface provided by bus adapters.

Figure 5 shows an example of bus functional model which is refined from the communication link model in Figure 4. The logical link channels in the communication link model are inlined and connected to next higher layer (presentation layer). MAC and protocol layer channel adapters are taken out of media protocol library and inserted into the bus functional model of the corresponding components and connected to the corresponding inlined logical link adapters.

Additional communication elements such as interrupt controller (*PIC*) and arbiter (*Arbiter*) are inserted into the bus functional models in order to resolve multiple accesses of masters on a bus.

Inside a programmable component (*DSP*), interrupt service routine (*ISR*) and interrupt handling methods are generated and inserted for synchronization between other components (*HW1* and *HW2*).

4. Media Protocol Library

The media protocol library [GCS+03] is a set of channels that model the protocols of system busses. These channels provide the standard read/write methods for the bus protocol. Additional methods may be required for more complex designs that support arbitration, multiple interrupt signals etc. Each bus transaction also requires definition of a master and slave. Therefore, the protocol library must provide for unique channels for both master and slave sides. The ports of the bus protocol channel represent the actual bus wires which are later exposed in the bus functional model.

4.1. Bus Database

The bus database contains models of busses including associated protocols where the term “bus” refers to communication structures in general, e.g. networks and their protocols. Models taken out of the bus database are inserted by communication link refinement to implement communication inside the components connected to the busses of the system.

Bus models in the bus database consist of a stack of two layers: physical layer and media access layer. At the bottom of the stack, the physical layer is connected to the actual physical bus wires and it implements the bus primitives defined by the bus protocol for data transfers, synchronization and arbitration. On top of the physical layer, the media access layer provides an abstraction of external communication into data links and memory accesses by using and combining bus primitives to regulate media accesses and slice abstract data into bus words.

Each physical layer can have two separate sides with different implementations for bus masters and bus slaves. The different models of the two physical layers for bus models are stored as channels. Each channel provides a protocol implementation for one single component connected to the bus, i.e. each connected component implements a bus protocol by creating internal instances of the required protocol models. Physical layer models connect to the bus

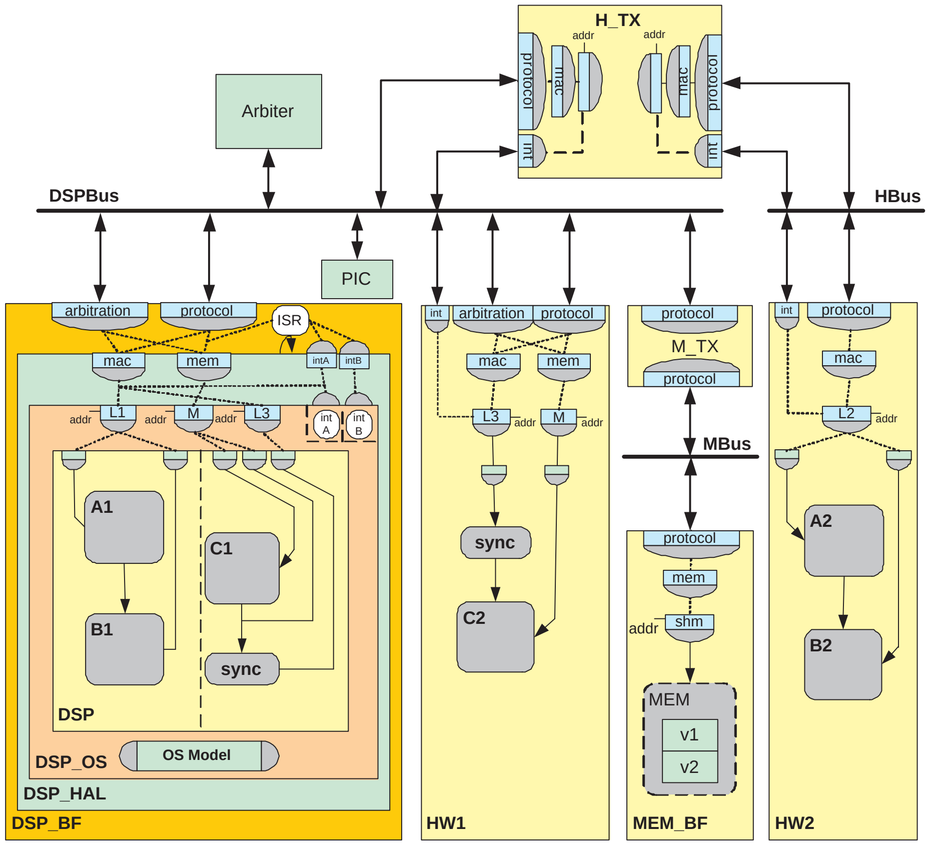


Figure 5. Bus functional model.

wires through ports of the model and pins of the component. Higher level models are stacked on top of each other via interfaces implemented by each model where a model calls the methods of the model beneath it via ports of corresponding interface type. These channels provide for the standard *read/write* methods for the bus protocol. Additional methods may be required for more complex designs that support arbitration, multiple interrupt signals etc.

At the top level of the bus database, all channels and behaviors that are part of the same bus model are then grouped together under a single, top level bus channel that acts as a container representing the overall bus protocol in the bus database.

4.1.1 Physical Layer

Physical layer models provide primitives for atomic bus transactions at their interfaces and they describe the basic timing of events (value changes) on the wires of the bus for each primitive. Timing diagrams of bus transactions are represented through state machines and associated timing constraints defining the possible sequences of driving and sampling bus wires. Physical layer protocol models are used by communication link refinement to provide models of the protocol behavior on the wires of the bus both for system simulation and for synthesis of protocol implementations in actual hardware.

In general, a bus can have separate physical protocols for basic data transfers, synchronization, and arbitration. A protocol model provides a description for implementation of the protocol in a component connected to the bus wires. Each physical protocol can have two separate models with different implementations for bus master and bus slave type component.

Data Transfer Protocol The mandatory data transfer protocol is the core of the bus and it describes primitives for transferring native bus words distinguished by bus addresses. It has to provides methods for all atomic bus cycles available over the core bus, including special types like burst mode, etc.

In case of a master/slave arrangement (separate models for master and slave side), the master side is actively initiating bus cycles and the slave slide can only passively listen on the bus for the start of a cycle to participate in. Consequently, methods on the slave side are considered blocking and only return when the transfer has been completed successfully with the corresponding master. In order to enable polling, methods on the master side, on the other hand, must not block even if no corresponding slave is available to successfully complete the transfer (note that in order to satisfy this requirement, active slaves might be necessary to answer and decline a transfer if no data is available through higher layers).

In case there is no distinction between masters and slaves (master model and slave model are the same), the single data transfer protocol model acts both as master and slave where for each transfer the sending side is assumed to be the master and the receiving side the slave. Apart from that, the same restrictions for the sending (master) and receiving (slave) methods apply.

Synchronization Protocol As explained in the previous section, the data transfer protocol generally only supplies inherent one way synchronization from master to slave. However, in order to implement reliable communication with guaranteed data delivery, two way synchronization between communication partners is required. Therefore, a bus can supply an optional, distinct synchronization protocol to efficiently send events from slave to master. Usually, this means an interrupt protocol and interrupt wires through which a slave can send interrupts to a master. In the same manner as data transfer protocols, interrupt protocols can be arranged as separate models for master side and slave side, or as one common model acting as both master (when sending) and slave (when receiving).

If the synchronization protocol annotations in the bus channel point to the data transfer protocol model(s), two way synchronization with blocking transfers on both sides has to be implemented as part of the data transfer protocol, and no separate synchronization protocol is available or necessary. Similarly, if no synchronization protocol is supplied, no event transfer mechanism is available as part of the bus.

Arbitration Protocol If the bus supports multiple masters connected to the bus, it has to supply an arbitration protocol that is used to regulate accesses to the shared bus wires. In a centralized arbitration scheme, the master side of the arbitration protocol instantiated in each master communicates with the slave side of the arbitration protocol instantiated in an additional arbiter component attached to the bus. In a distributed arbitration scheme, there is no slave side of the arbitration protocol and the master sides of the protocol in each master regulate accesses among themselves.

The master side of the arbitration protocol can either be provided as a separate physical layer protocol model or it can be a built-in part of the data transfer protocol in which case the arbitration master annotation points back to the data transfer protocol model. In case of a separate arbitration protocol, the master side arbitration protocol model has to implement a interface for acquiring and releasing access to the bus. In either case, arbitration has to be made available for each component that will act as a data transfer protocol master if multiple masters sharing the bus are supported.

4.1.2 Media Access Layer

Media access layer models abstract accesses to the actual physical medium through the protocol into canonical interfaces for regulated, non-conflicting exchange or communication of data of arbitrary size and type. Hence, the media access layer regulates conflicting bus accesses in case the bus supports multiple masters through the bus arbitration protocol, and it slices data chunks into bus words or frames that are transmitted using the primitives (and possibly choosing among modes) of the bus data transfer protocol. Note that the media access layer does not implement any additional synchronization (e.g. through the synchronization protocol) but rather inherits the synchronization semantics from the underlying data transfer protocol.

The media access layer consists of two parts, models for implementation of bi-directional data links and models for accesses to shared memories connected to the bus. Mandatory data link models provide primitives to create point-to-point logical links for exchanging data between two communication partners attached to the bus. Optional memory access models are required if the bus supports shared memories and addressing of and access to storage of the component. In both cases, media access layer models can consist of separate implementations for use in bus master and bus slave type components.

Since they will be accessed and used by automatically generated code, media access layer interfaces have to adhere to a certain canonical format. A media access layer interface defines a set of methods for transferring a block of data over the bus using a bus address to distinguish between different logical connections made over the same physical bus. Consequently, a media access method has no return value and takes three parameters: a bus address of integral type corresponding to the range of addresses available on the bus, a pointer to a data block, and the size of the data block in bytes. A media access layer interface has to define exactly two methods for reading and writing a block of data from/to the bus where the names of the methods have to match the corresponding annotations at the bus channel.

Data Link Layer The data link part of the media access layer is used to transfer streams of data packets between logical endpoints inside components attached to the bus where two logical endpoints define a bi-directional, point-to-point logical link. Since streams only support sequential access (no random access), bus addresses are used to distinguish among different logical links on the bus only, i.e. data link models use the same bus address supplied as parameter for all bytes in a packet. Addresses supplied to data link model methods are used as addresses on the bus where addresses can be assumed to be aligned on bus word boundaries as defined in the bus channel annotations.

The data link part of the media access layer can provide different master and slave sides of the model if the underlying data transfer protocol in the physical layer differentiates between bus masters and slaves and if separate functionality is needed. For example, acquiring and releasing access to the bus through calls to the arbitration protocol is only needed on the master side, if supported by the bus at all.

Memory Access The memory access part of the media access layer provides methods for accessing bytes of data stored in a shared memory component attached to the bus. Since memories need to support random access, data bytes in all memories attached to the bus have to be individually distinguishable. Therefore, bus addresses are used to select among different characters stored in memory where each character holds a certain amount of bytes as defined by the bus and where consecutive bytes in memory are accessed as consecutive characters on the bus. Consequently, for each memory access the address supplied is the address of the first character in the block of data to be accessed and the length of the block divided by the bus character size determines the range of bus addresses accessed. A media access layer memory model consists of master and slave sides for initiating and serving shared memory accesses over the bus. The master side provides methods with names matching the corresponding bus channel annotations for reading and writing blocks of data bytes from/to memory used in components that access shared storage over the bus. The slave side, on the other hand, provides methods for serving incoming random memory accesses used in components that provide shared storage (e.g. dedicated shared memory components).

4.2. Bus Functional Component Database

For components with fixed, pre-defined interfaces and communication functionality, the component database has to contain a bus functional model of the component. A bus functional component model accurately describes the component interface at the pin level and it provides a simulation model of communication aspects of the component on top of any computation functionality as defined by the behavior model of the component, if any.

Bus functional models can be thought of as additional communication layers that wrap around the component behavioral model. A bus functional model can consist of several layers of behaviors that create a hierarchy or tree of behavior instantiations. At minimum, a top level bus functional layer has to exist that provides a pin accurate model of the component. Through this layer and its optional sublayer instance hierarchy, the bus functional component model describes the communication behavior of the component at its pins and it has to provide the same computational functionality as the behavioral model of the component.

For IPs with fixed computation and communication functionality the bus functional IP model provides a timing accurate and data accurate descriptions in terms of signals that can be observed at the pins of the component. Bus functional IP models only have to provide a single bus functional layer but they can consist of several hierarchical layers internally.

For programmable component with flexible computation behavior (i.e. no functionality provided in the component behavior, but fixed, pre-defined interfaces and communication functionality, the bus functional component model has to provide a hierarchy with at least two layers: a top level bus functional layer describing the component pin interface on the outside and an internal empty hardware abstraction layer (HAL) shell at the leaf of the bus functional model hierarchy describing the interface for accessing the communication implementation of component from the programmable computation on the inside.

A component is considered programmable in terms of its computation if the bus functional component model provides a hardware abstraction layer (HAL) shell. Communication link synthesis will use the HAL shell of the bus functional component model as a template and modify it to implement computation on the component on top of the services provided by the HAL shell. In terms of services, the HAL shell defines the insertion point for implementing the computation of the component and it provides communication services for stream and memory I/O and interrupt handling. The HAL shell together with the outer bus functional layers model the corresponding capabilities of the pre-defined component implementation (e.g. number and type of external interfaces, amount and level of interrupts, etc.).

The HAL shell marks the boundary between hardware and software in programmable components. The code for the HAL shell and the sub-behavior hierarchy inserted into the HAL shell by communication link refinement will later be implemented in software. On top of an implementation of the HAL shell on the target processor taken out of the OS databases, target specific code will be generated for the HAL model. Layers of the component model above up to and including the bus functional layer shell represent the hardware implementation of the component and will later be replaced with a description of the real component hardware taken out of the databases for manufacturing.

If no hardware abstraction layer shell is provided, on the other hand, the bus functional component model is considered to be a self contained simulation model of the complete component including computation and communication that will be plugged into the system simulation as is. Figure 6 shows an example of PE bus functional component database for a programmable processor. In this example, HAL, HW and BF shell are implemented in the library, but the functionality inside OS shell, for example, interrupt handlers and semaphores need to be generated by communication link refinement tool.

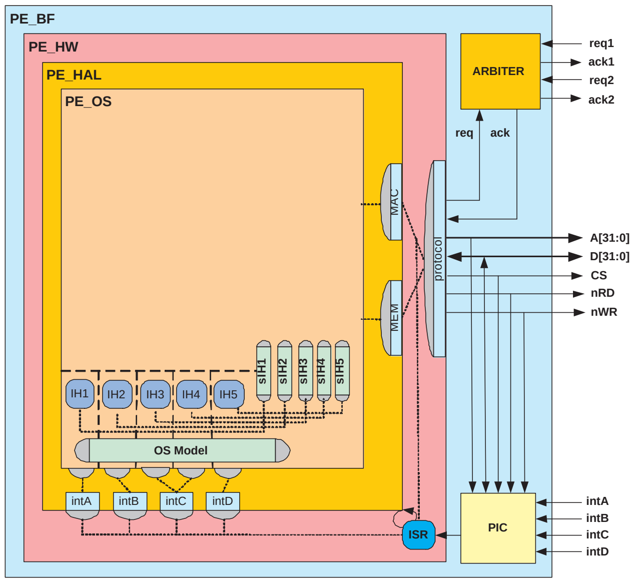


Figure 6. PE bus functional model for programmable component.

Interrupt Handling Bus functional models for programmable components as shown in Figure 6 have to include a definition of the interrupt capabilities of the component and its interrupt handling. As described previously, the top level bus functional layer defines the interrupt pins available at the physical component interface and the HAL model provides corresponding empty interrupt handler templates. The different layers of the bus functional component model then describe the component's interrupt behavior of detecting interrupts, suspending regular computation, and executing the HAL interrupt handlers during system simulation.

In order to support more complex interrupt capabilities with more than one source of interrupts, different priorities and masking, the component database needs to include interrupt controllers as part of the bus functional component models. Interrupt controllers sit in front of the basic component core model and are modeled by adding another layer to the bus functional component model between the processor core and the outer bus functional layer. Typically, the interrupt controller provides a set of interrupt lines at the pins of the top level bus functional layer while internally communicating with the core via the component bus and the core's interrupt condition input. The core then interrupts normal computation and executes the appropriate handler depending on the inputs received from the interrupt controller.

5. Tasks for Communication Link Refinement

Communication link refinement tool refines the input communication link model into bus functional model of a system. The refinement process can be divided into five steps, namely, *channel grouping*, *PE bus functional model instantiation*, *protocol stack generation and insertion*, *communication element synthesis and insertion*, and *port mapping and bus wiring*, each can be further divided into substeps.

5.1. Channel Grouping

First task of communication link refinement is channel grouping. With the help of channel mapping decision of users, link layer channels between components will be grouped into corresponding system busses. For example, the channels *L1* and *L3* in Figure 4 are grouped to *DSPBus* and the channel *L2* is grouped to *HBus*.

5.2. PE Bus Functional Model Instantiation

The bus functional models for components with fixed, predefined protocol interfaces, for example, programmable processor, IP, system memory and FPGA, need to be taken out of and be instantiated from PE bus functional library. The bus functional model instantiation process is shown in Algorithm 1

Algorithm 1. InstantiateBF (IR_{Design})

```
1:  for all $Bus \in IR_{design}$ do
2:      $BusWire_{Bus}$ = CreateBusWire ($IR_{design}$, $Bus$)
3:  end for
4:  for all $B_{PE} \in IR_{design}$ do
5:      if HasBFModel ($B_{PE}$)) then
6:          CreateInstance ($IR_{design}$, $B_{PE}$, $BusWire_{Bus_{PE}}$)
7:      else
8:          CreatePort ($B_{PE}$, $BusWire_{Bus_{PE}}$)
9:      end if
10: end for
```

For example, PE bus functional model of the DSP processor (*DSP_BF*) and the memory (*MEM_BF*) are taken out of a PE bus functional library and instantiated into the bus functional model as shown in Figure 5.

5.3. Protocol Stack Generation and Insertion

During communication link design, link layer, MAC layer and protocol layer are inserted into components. The MAC layer and protocol layer are a part of the media protocol library. Therefore they need to be taken out of media protocol library and instantiated into corresponding components. But link layer has to be generated during communication link refinement.

5.3.1 Link Layer Adapter Channel

The link layer is responsible for implementing synchronization as shown in Figure 7. In the figure, synchronization is achieved by interrupts (*intr* in line 5, line 9, line 17, line 21). *cCPUBusMasterLink* and *cCPUBusSlaveLink* channel is an adapter channel in master side and slave side respectively. The interface methods in the master side wait for interrupt from slaves (line 5, line 9). If the master gets interrupt from a slave, it will invoke MAC layer interface method to access the corresponding bus protocol (line 6, line 9). In the slave side, the slave sends interrupt to a master to notify data transfer request (line 17, line 21). The address port in the channels represents the addresses of the message or the the registers in the slaves (line 18, line 22).

The link layer adapter channel to access memory component does not have interrupt line because memory component is always ready for memory access. It will read/write data to assigned range of addresses of the memory component (line 7, line 10). The address port (*addr*) in the channels represents the base address of the memory component which then is assigned a range of addresses with a base address plus offsets (*offset* in line 7 and line 10) for messages to enable memory mapped I/O.

5.3.2 Protocol Stack Insertion

The protocol stack insertion process is shown in Algorithm 2. The input to Algorithm 2 is the internal representation for the whole design IR_{Design}. Each component behavior B_{PE} inside the design is checked if it is software component or hardware component (line 2 and line 11). If the component is software, the link layers (C_{link}) as shown in Figure 7 for general bus accesses and Figure 8 for memory accesses, are created with the corresponding operating system (OS) interface port (P_{os}) inside its application shell ($B_{PE_{app}}$) (line line 4 – line 5). Also OS adapter channel (C_{os}) will be instantiated with the link channel interface port (P_{link}) inside the OS shell ($B_{PE_{OS}}$) (line 3, line 6). In case of

Figure 7. Link layer adapter channel for general bus access.

```
 1 channel cCPUBusMasterLink( ICPUBusLinkAccess mac,
       i_receive intr,
 2     in unsigned long long int addr ) implements
           i_tranceiver
 3 {
 4     void receive(void *data, unsigned long int
           len) {
 5         intr.receive();
 6         mac.read(addr, data, len);
 7     }
 8     void send(const void *data, unsigned long int
            len) {
 9         intr.receive();
10         mac.write(addr, data, len);
11     }
12 };
13 channel cCPUBusSlaveLink( ICPUBusLinkAccess mac,
      i_send intr,
14     in unsigned long long int addr ) implements
           i_tranceiver
15 {
16     void receive(void *data, unsigned long int
           len) {
17         intr.send();
18         mac.read(addr, data, len);
19     }
20     void send(const void *data, unsigned long int
            len) {
21         intr.send();
22         mac.write(addr, data, len);
23     }
24 };
```

Figure 8. Link layer adapter channel for memory access.

```
 1 channel ShmLinkMem(ICPUBusMasterMemAccess shm, in
       unsigned long long int addr)
 2         implements IMemLink
 3 {
 4     void read(unsigned long int offset, void *
           data, unsigned long int len) {
 5         shm.read(addr + offset, data, len);
 6     }
 7     void write(unsigned long int offset, void *
           data, unsigned long int len) {
 8         shm.write(addr + offset, data, len);
 9     }
10 };
```

PE connected to the shared memory, the memory interface port (P_{mem}) will be created in the OS shell of the software component (line 7 – line 9).

If the component is hardware, all protocol stacks including link layer, MAC layer and protocol layer are taken out of media protocol library and inserted into the model of hardware components (line 12 – line 14). The protocol layer adapter will be inserted with corresponding bus wires (*BusWires*) on top of bus functional model. Also if the component accesses the shared memory, the MAC and protocol layer for memory access will be inserted (line 15 – line 18).

Algorithm 2. InsertStack (IR_{Design})

```
1:  for all $B_{PE} \in IR_{design}$ do
2:    if $B_{PE} == SW$ then
3:      CreatePort ($B_{PE_{OS}}$, $P_{MAC}$)
4:      CreatePort ($B_{PE_{app}}$, $P_{semap}$)
5:      CreateInstance ($B_{PE_{app}}$, $S_{MAC}$, $P_{semap}$)
6:      CreateInstance ($B_{PE_{OS}}$, $S_{semap}$, $P_{MAC}$)
7:      if $B_{PE}$ is connected to memory $mem$ then
8:        CreatePort ($B_{PE_{OS}}$, $P_{mem}$)
9:      end if
10:     CreateInstance ($B_{PE_{HAL}}$, $B_{PE_{OS}}$, $\{P_{MAC}, P_{mem}\}$)
11:   else if $B_{PE} == HW$ then
12:     CreateInstance ($B_{PE}$, $S_{proto}$, $BusWire$)
13:     CreateInstance ($B_{PE}$, $S_{MAC}$, $S_{proto}$)
14:     CreateInstance ($B_{PE}$, $S_{link}$, $S_{MAC}$)
15:     if $B_{PE}$ is connected to memory $mem$ then
16:       CreateInstance ($B_{PE}$, $S_{mem\_proto}$, $BusWire$)
17:       CreateInstance ($B_{PE}$, $S_{mem\_MAC}$, $S_{proto}$)
18:     end if
19:   end if
20: end for
```

For example, the functionality of link layer, MAC layer and protocol layer is inlined into the custom hardware component (*HW1*) and the channel adapters are connected to each other to access bus wires in the bus functional model as shown in Figure 5.

5.4. Communication Element Synthesis and Insertion

As part of bus functional model, communication elements such as transducer, arbiter and interrupt controller might have to be inserted into the system architecture. The transducer that translate between incompatible bus protocols will act as bridges connecting two busses or as bus interfaces for components with fixed, predefined protocols. Arbiter and interrupt controller will resolve the multiple accesses of masters on busses and multiple accesses of slaves to processors, respectively. Like the other components, the behaviors of the communication elements are instantiated and added to the set of concurrent, non-terminating components at the top level of the design.

5.4.1 Memory Interface Controller

Since memory components have their own fixed interface protocol, they cannot be directly connected to the system bus and memory interface controller, therefore, was inserted into communication link model during network synthesis. The behavior of the memory interface controller should be refined into pin-accurate protocol model.

For shared memory transfers, protocol channels provide split transactions that allow the shared memory to list to and serve incoming accesses. Therefore, memory interface controller might have to support split transaction of the memory. Figure 9 shows an example of the memory interface controller which connects system bus (*CPU bus*) and memory component (*SRAM bus*).

The memory interface controller connects to the system bus on the one hand (line 3 – line 12) and to memory bus on the other hand (line 14 – line 18) through corresponding sets of ports. Since the memory interface controller has to act as both master (for communication on the memory bus) and slave (for communication on the system bus), it connects with *read* and *write* interfaces of the control lines.

For implementation of the memory bus protocol, the memory interface controller instantiates a memory bus adapter (line 21). For memory accesses, protocol addresses have to be used to distinguish among individual addressable words in the memory (*GA* line 29) and addresses have to be incremented properly according to alignment for all successive transfers in a consecutive blocks of data (line 34 – line 40 and line 49 – line 55).

For example, the implementation of a memory interface controller (*M_TX*) is generated and inserted into the bus functional model as shown in Figure 5.

5.4.2 Interrupt Handling

As we described in Section 4.2, bus functional models for programmable components have to include a definition of the interrupt capabilities of the component and its interrupt handling. The top level bus functional shell defines the interrupt pins available at the physical component interface and the HAL model provides corresponding empty interrupt handler templates. Therefore interrupt lines from slaves need to be connected the interrupt pins on the programmable components. Also the empty interrupt handler need to be filled and instantiated in HAL shell and call interrupt tasks in operation system shell of the programmable components.

Figure 9. An example of memory interface controller.

```
#define GBUSCLK 5
behavior MemCtrlBF(
    // CPU bus
    in signal bit[31:0] GA,
    in signal bit[31:0] GDOUT,
    out signal bit[31:0] GDIN,
    in signal bit[7:0] GBE,
    in signal bit[0:0] GRD,
    in signal bit[0:0] GWR,
    out signal bit[0:0] GACK,
    in signal bit[0:0] GID,
    in signal bit[0:0] GBSTART,
    // SRAM bus
    out signal bit[18:0] A,
    inout signal bit[7:0] IO,
    out signal bit[0:0] CS,
    out signal bit[0:0] OE,
    out signal bit[0:0] WE)
{
  // protocol adapter for SRAM
  MasterMem mem(A, IO, CS, OE, WE);
  void main() {
    bit [31:0] addr, data;
    bit[7:0] p;
    unsigned int i;
    while(true) {
      do {
        wait(GBSTART falling);
      } while((GID != 0) || (GA < 0x0ffff || GA >
           0x4ffff));
      addr = GA;
      if(!GRD) { // write
        data = 0;
        for (i=0; i<4; i++) {
          p = 0;
          data = data << 8;
          if (GBE[3 - i]) {
            mem.read(addr++, &p);
            data += p;
          }
        }
        GDIN = data; // drive data
        GACK = 0;    // acknowledge
        waitfor(GBUSCLK);
        GACK = 1;
      }
      else if(!GWR) { // read
        data = GDOUT; // take data
        for (i=0; i<4; i++) {
          if (GBE[3 - i]) {
            p = data[31:24];
            mem.write(addr++, p);
          }
          data <<= 8;
        }
        GACK = 0; // acknowledge
        waitfor(GBUSCLK);
        GACK = 1;
      }
    }
  }
};
```

Figure 10 shows interrupt handling task in operation system shell in the programmable component. All link adapter channels connected to the programmable component have corresponding interrupt handling tasks which will be invoked by the interrupt handler in the operation system shell of the component model (line 17 – line 20 in Figure 11).

Figure 10. Interrupt handling task.

```
behavior CPU_IntTask(OSAPI os, i_send ev)
    implements CPU_IntTask, OS_TASK_INIT
{
    void start(void) {
        os.task_resume(os_task_id);
    }
    void main(void) {
        os.task_activate(os_task_id);
        while(true) {
            os.task_sleep();
            ev.send();
        }
    }
    void os_task_create(void) {
        os_task_id = os.task_create("intHander",
            2, 0, 0);
    }
};
```

Figure 11 shows the refined operation system shell for the component. Inside the model, we can see an OS API channel which implements scheduling policy for operation system as shown in line 4. The FCFS is the OS API which has first-come first-service scheduling policy. The OS API will be connected to semaphore (line 6 – line 8), interrupt tasks (line 12 – line 14) shown in Figure 10 and application shell of the component (line 16).

Figure 12 shows interrupt handling method in HAL shell of the programmable component. The HAL shell has the interrupt handler for each interrupt line (line 7 – line 12). For example, the processor has four lines of interrupts, there are four interrupt handlers in HAL shell. In the figure, *intHandler* method is invoked whenever interrupts come into the processor. The *intHandler* method will take with the interrupt controller (PIC) outside of hardware shell of the component and get the interrupt status and invoke interrupt handling routines (*int0Handler*, *int1Hander*). Again the interrupt handling routines in the HAL shell will call interrupt handling methods in the operation system shell (line 8, line 11).

Algorithm 3 shows the process which inserts interrupt handling methods in OS shell and HAL shell of the component. First, all channels (C_{PE}) in the software components have the assigned interrupt line and the corresponding semaphore and interrupt handling tasks and methods are inserted into OS shell of the component (line 4 – line 6). The interrupt handling methods are invoked in HAL shell of the

Figure 11. Interrupt handling method in OS shell.

```
behavior CPU_OS( ICPUBusLinkAccess mac_Bus, ICPUBusMasterMemAccess mlink_Bus)
    implements IntHandlers_CPU
{
    FCFS os;    // OS channel
    Shmlink_M1 IShmlink_M1(mlink_Bus, (2048ull));    // memory
    // os semaphore
    c_os_handshake Flag_m_link_0_Bus(os);
    c_os_handshake Flag_m_link_1_Bus(os);
    // link channels
    CPUBusMasterDLink m_link_0_BusDriver(mac_Bus, Flag_m_link_0_Bus, (1024ull));
    CPUBusMasterDLink m_link_1_BusDriver(mac_Bus, Flag_m_link_1_Bus, (1025ull));
    // interrupt tasks
    CPU_IntTask m_link_0_Bus_intTask(os, Flag_m_link_0_Bus);
    CPU_IntTask m_link_1_Bus_intTask(os, Flag_m_link_1_Bus);
    // application
    CPU_SW CPU(m_link_0_BusDriver, m_link_1_BusDriver, IShmlink_M1, os);
    void m_link_0_Bus_intHandler(void) {    // interrupt handler
        os.ienter();
        m_link_0_Bus_intTask.start();    // call interrupt task
        os.ireturn();
    }
    void m_link_1_Bus_intHandler(void) {    /* ... */ }
    void main(void) {
        os.init();
        CPU.os_task_create();
        os.start();
        m_link_0_Bus_intTask.os_task_create();
        m_link_1_Bus_intTask.os_task_create();
        par {
            CPU.main();
            m_link_0_Bus_intTask.main();
            m_link_1_Bus_intTask.main();
        }
        os.task_terminate();
    }
};
```

Figure 12. Interrupt handling method in HAL shell.

```
behavior CPU_HAL( ICPUBusMaster mac, inout bit[31:0] SR, inout bit[31:0] CR)
    implements ICPUBusIntVectors
{
    CPUBusMasterLinkAccess link(mac);    // MAC layer
    CPUBusMasterMemAccess mem(mac);      // MAC layer for memory
    CPU_OS CPU(link, mem);               // OS shell
    void int0handler(unsigned int num) {
        CPU.m_link_0_Bus_intHandler();
    }
    void int1handler(unsigned int num) {
        CPU.m_link_1_Bus_intHandler();
    }
    /* ... */
    void intHandler(void) {  /* ... */ }
    void main(void) {
        CPU.main();
    }
};
```

component (line 7). In case of the hardware component, The interrupt line will be inserted as output port of the component (line 10).

Algorithm 3. InsertISR (IR_{Design})

```
1:  for all $B_{PE} \in IR_{design}$ do
2:      if $B_{PE} == SW$ then
3:          for all $C_{PE} \in IR_{design}$ do
4:              CreateOSSemaphore ($B_{PE_{OS}}, C_{PE}$)
5:              CreateIntrHandlerTask ($B_{PE_{OS}}, C_{PE}$)
6:              CreateIntrHandlerMethod ($B_{PE_{OS}}, C_{PE}$)
7:              CreateIntrHandlerMethodCall ($B_{PE_{HAL}}, C_{PE}$)
8:          end for
9:      else if $B_{PE} == HW$ then
10:         CreateInstance ($B_{PE}, M_{intr}$);
11:     end if
12: end for
```

For example, DSP processor are connected to two logical link channels which are assigned to two different interrupts (*intA* and *intB*) by user decision on interrupt mapping. The interrupt handler tasks and methods for the two interrupts are generated in OS shell and HAL shell of the bus functional of DSP processor (Figure 5). Also hardware components *HW1* and *HW2* have the implementation of interrupt generation.

5.4.3 Arbitration

For busses that support arbitration, the user may designate more than one master Essentially, the master side protocol should have a special method which is annotated to be identified as the bus arbitration method. If such a channel method is not found, we have to generate a centralized bus arbiter as per the requirements. The arbitration mechanism will be instantiated from the protocol library, because it is the part of bus protocol. The protocol layer in a master has the method to request bus access. Based on design decisions, we generate a priority-based or round-robin arbitration unit. The arbiter behavior is exactly like that of an interrupt controller, except that it resolves conflicts between masters.

5.4.4 Communication Element Insertion

Algorithm 4 shows the communication element insertion process. First, for each slave on a bus, we have to create an interrupt port and connect the created interrupt port with corresponding interrupt wire on the bus (line 2 – line 6) because the synchronization between master and slave is implemented.

If the number of master components are connected with a bus is more than one, then the bus should have arbiter to resolve multiple accesses of the masters. All masters need to have arbitration ports to be connected to the arbiter on the bus (line 9 – line 11). The arbiter will be instantiated with the created arbitration wires on the top level of a design (line 18).

If a master is connected with more than one slave, it will have interrupt controller to resolved multiple access from the slaves. The interrupt controller will be inserted with the created interrupt wires (line 13).

For example, arbiter (*arbiter*) on *DSPBus* and programmable interrupt controller (*PIC*) for DSP are taken out of media protocol library and instantiated into the bus functional model as shown in Figure 5.

5.5. Port Mapping and Bus Wiring

Now all protocol stacks and communication elements are inserted into corresponding components, the components on top of the design need to be connected to each other through wires of the busses. The ports are already changed during protocol stack and communication elements insertion. The connections between port and busses are defined through the port mapping as shown in Algorithm 5. The interrupt lines and arbitration lines are connected based on the priorities of the components connected.

Algorithm 4. InsertCE (IR_{Design})

```
1:  for all $Bus \in IR_{design}$ do
2:      for all $B_s \in$ slaves connected the $Bus$ do
3:          $IntrWire_s$ = CreateIntrWire ($IR_{design}$, $B_s$)
4:          $P_i$ = CreatePort ($B_s$, $IntrWire_s$)
5:          Connect ($P_i$, $IntrWire_s$)
6:      end for
7:      if number of masters on the $Bus > 1$ then
8:          for all $B_m \in$ masters on the $Bus$ do
9:              $ArbitWire_m$ = CreateArbitWire ($IR_{design}$, $B_m$)
10:             $P_a$ = CreatePort ($B_m$, $ArbitWire_m$)
11:             Connect ($P_a$, $ArbitWire_m$)
12:             if number of slaves connected to master $B_m > 1$
                then
13:                 CreateInstance ($IR_{design}$, $B_i$, $IntrWire_s$);
14:             else
15:                 Connect ($B_{sw}$, $P_i$, $IntrWire_s$)
16:             end if
17:         end for
18:         CreateInstance ($IR_{design}$, $B_a$, $ArbitWire_m$);
19:     end if
20: end for
```

Algorithm 5. Wiring (IR_{Design})

```
1: for all PE $B_{PE} \in IR_{design}$ do
2:     for all port $P_{PE} \in B_{PE}$ do
3:         $w_{P_{PE}}$ = FindBusWire ($IR_{design}$, $P_{PE}$)
4:         Connect ($B_{PE}$, $w_{P_{PE}}$, $P_{PE}$)
5:     end for
6:     CreateInstance ($IR_{design}$, $B_{PE}$, $B_{PE_{BF}}$)
7: end for
```

6. Experimental Results

Based on the described methodology and algorithms, we developed a communication link refinement tool, *SpecC Communication Link Refinement (sccr)*, which takes communication link model in SpecC. For experiments, we used the communication link models ([SGG04]) which were generated by network refinement tool from the architecture models as shown in [SGG04].

Table 1 shows the characteristics of the partitioned architecture models of these examples. Different architectures using Motorola DSP56600 processor (DSP), MIPS based CPU (CPU) and custom hardware (HW) were generated and various bus architectures were tested. In Table 1, the number of channels represents the number of message passing channels in top level of architecture model and the total traffic refers to the amount of data exchanged between components.

Table 2 shows design decisions which are made during communication link synthesis. In this table, addresses and interrupts for channels are decided by users. Priorities for arbitration are assigned into master components on busses.

We have to measure the quality of the generated models in order to be able to assess the model quality. For the communication link refinement, we can use two quality of metrics: the code size of the models and the readability of the refined model.

Table 3 shows the results of communication link refinement. We used the Lines of Code (LOC) metric. Modified lines of code by automatic refinement is calculated by $modified = inserted - library + deleted$ in fifth column. We assume that a person can modify 10 LOC per hour, thus, manual takes several hundred hours for reasonably complex designs. Automatic refinement on the other hand completes in the order of a second. In order to compute the productivity gain, we assume that design decisions (address assignment, interrupt mapping, arbitration) for communication link refinement takes 5 minutes to do. The productivity gain is around 1000 times as a result of automatic refinement. For example, gain for arch1 of JPEG is calculated by $427 = (35.6 \text{ hr.})/(0.10 \text{ sec.} + \ 5 \text{ min.})$.

7. Conclusions

In this report, we presented a methodology and algorithms to automatically generate bus functional models from communication link model. During communication link synthesis, logical links between adjacent components are grouped and implemented over a system bus. As a result of communication link refinement, bus functional model is developed in order to reflect communication architecture of a system. The functionality of link layer, MAC layer and protocol layer need to be inlined into components during the communication link refinement.

Communication link refinement tool was developed and integrated into SoC design environment. Using an industrial strength example, the feasibility and benefits of the approach have been demonstrated and huge productivity gain is obtained with communication link refinement tool. Our main contribution in the report is the automation of a time consuming and error prone process to achieve better designer productivity. It also enables designers to evaluate several design points during exploration.

References

[ASG03] Samar Abdi, Dongwan Shin, and Daniel D. Gajski. Automatic communication refinement in system-level design. In *Proceedings of the Design Automation Conference*, pages 300–305, June 2003.

[BGJ+97] Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciano Lavagno, Alberto L. Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[BML+97] Ivo Bolsens, Hugo De Man, Bill Lin, Karl V. Rompay, Steven Vercauteren, and Diederik Verkest. Hardware/Software co-design of the digital telecommunication systems. *Proceedings of the IEEE*, March 1997.

[CBG+02] W. O. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-baed design approach for multicore SoCs. In *Proceedings of the Design Automation Conference*, pages 789–794, June 2002.

[DMBIJ97] Jean-Marc Daveau, Gilberto F. Marchioro, Tarek Ben-Ismail, and Ahmed A. Jerraya. Protocol selection and interface generation for HW-SW codesign. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, March 1997.

[GABP98] Guy Gogniat, Michel Auguin, Luc Bianco, and Alain Pegatoquet. Communication synthesis and HW/SW integration for embedded system design. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 49–53, March 1998.

Table 1. Characteristics of the partitioned architecture models.

Examples		Components	Buses	# Channels	Traffic
JPEG	arch1	1 DSP, 1 HW	1 DSP Bus	7	7560
	arch2	1 DSP, 1 DCT IP	1 DSP Bus	2	2160
Vocoder	arch1	1 DSP, 1 HW	1 DSP Bus	12	46944
	arch2	1 DSP, 2 HW	1 DSP Bus	22	56724
	arch3	1 DSP, 3 HW	1 DSP Bus	42	76284
	arch4	2 DSP, 2 HW	1 DSP Bus	29	52160

Table 2. Design decisions for communication link synthesis.

Examples		Channels	Address	Interrupt	Medium (master/slave)
JPEG	arch1	linkHW	0x0000300x	intA	DSP Bus (DSP/HW)
	arch2	linkIP	0x0000300x	intA	DSP Bus (DSP/IP)
Vocoder	arch1	linkHW	0x0000300x	intA	DSP Bus (DSP/HW)
	arch2	linkHW1	0x0000300x	intA	DSP Bus (DSP/(HW1, HW2))
		linkHW2	0x0000400x	intB	
	arch3	linkHW1	0x0000300x	intA	DSP Bus (DSP/(HW1, HW2, HW3))
		linkHW2	0x0000400x	intB	
		linkHW3	0x0000500x	intC	
	arch4	linkDSP1HW1	0x0000300x	intA	DSP1 Bus (DSP1, DSP2)/(HW1, HW2)
		linkDSP2HW2	0x0000300x	intA	

[GCS+03] Andreas Gerstlauer, Lucai Cai, Dongwan Shin, Rainer Dömer, and Daniel D. Gajski. System-on-Chip component models. Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, August 2003.

[Ger03] Andreas Gerstlauer. Communication abstractions for system-level design and synthesis. Technical Report CECS-TR-03-30, Center for Embedded Computer Systems, University of California, Irvine, October 2003.

[GVNG97] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, March 1997.

[KM98] Peter Knudsen and Jan Madsen. Integrating communication protocol selection with partitioning Hardware/Software codesign. In *Proceedings of the International Symposium on System Synthesis*, pages 111–116, December 1998.

[LYBJ01] Damien Lyonnard, Sunjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor System-on-Chip. In *Proceedings of the Design Automation Conference*, pages 518–523, June 2001.

[NG94a] Sanjiv Narayan and Daniel D. Gajski. Protocol generation for communication channels. In *Proceedings of the Design Automation Conference*, pages 547–551, June 1994.

[NG94b] Sanjiv Narayan and Daniel D. Gajski. Synthesis of system-level bus interfaces. In *Proceedings of the European Design Automation Conference*, pages 395–399, March 1994.

[SAG04] Dongwan Shin, Samar Abdi, and Daniel D. Gajski. Automatic generation of bus functional models from transaction level models. pages 756–758, January 2004.

[SGG04] Dongwan Shin, Andreas Gerstlauer, and Daniel D. Gajski. Network synthesis for SoC. Technical Report CECS-TR-04-15, Center for Embedded Computer Systems, University of California, Irvine, June 2004.

Table 3. Experiment results of communication link refinement.

Examples		Lines of Code			Automatic refinement	Manual refinement	Productivity gain
		Link	BF	Modified (inserted (lib)/deleted)			
JPEG	arch1	2969	3668	356 (766 (487)/77)	0.10 s	35.6 hr	427
	arch2	2763	3462	321 (761 (512)/72)	0.10 s	32.1 hr	385
Vocoder	arch1	10980	11740	341 (794 (487)/34)	0.31 s	34.1 hr	409
	arch2	11415	12205	405 (841 (487)/51)	0.33 s	40.5 hr	486
	arch3	12276	13096	489 (897 (487)/77)	0.39 s	48.9 hr	587
	arch4	14033	15220	757 (1309 (674)/122)	0.84 s	75.7 hr	908