Center for Embedded Computer Systems
University of California, Irvine

TLM Generation with ESE

Kyoungwon Kim

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{kyoungk1}@uci.edu

CECS Technical Report <10-07>
July 15, 2010

TLM Generation with ESE

Kyoungwon Kim
Center for Embedded Computer Systems
2010 AIRB
University of California, Irvine
Irvine, CA, 92697-2620
{kyoungk1}@uci.edu
http://www.cecs.uci.edu/

July 15, 2010

Abstract

Without a well-structured system design methodology, it is no more doable to cope with the dramatically increasing complexity in modern embedded system designs. Platform methodology, which often goes with hardware/software co-design with virtual prototyping, is the most popular solution [3]. Although it has been helpful for reduction of engineering costs and the length of design cycle due to the use of verified platform and virtual prototyping, it still has drawbacks that must be overcome.

The drawbacks mainly result from the followings; The starting point of a design is not behavioral specification of the system but a relatively inflexible platform instance. The refinement steps and models are not clearly defined and synthesis is not taken into consideration.

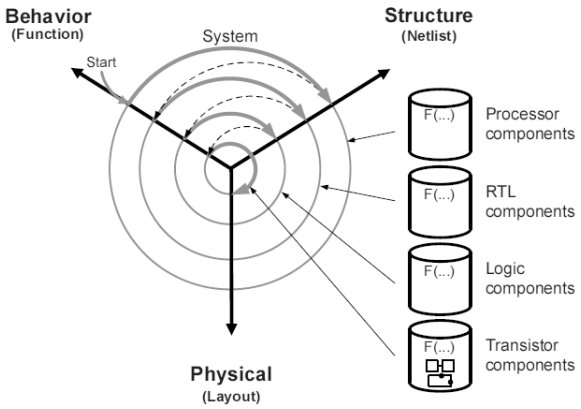
ESE(Embedded System Environment) overcomes these drawbacks while keeping the advantage of the current Platform methodology. A design starts with specification, which is flexible in nature and simplifying system level complexity by hiding unnecessary details. Refinement steps and models are firmly defined so that a design can be automatically synthesized. In the result, the productivity gain that designers can obtain with ESE is proved to be greater than 1000.

1 Introduction

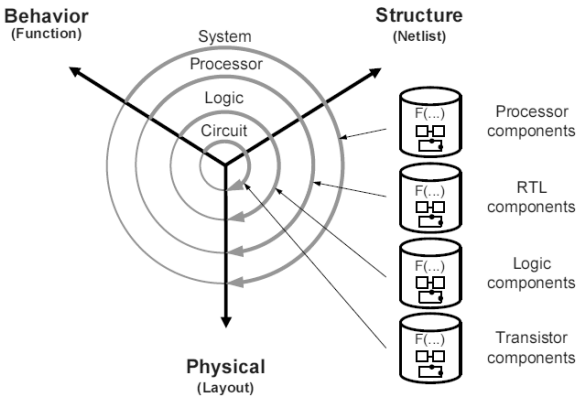
The complexity in embedded system designs has been dramatically growing over the past decades. Moreover, designers are more burdened with stricter time-to-market constraints, much higher non-recurring engineering costs, and so on. Since the design cycle has been becoming unbearably long and the cost of each iteration has been rising rapidly, productivity is no more a secondary issue.

Bridging the gap between productivity and design complexity turned out to be impossible with traditional design approaches such as the Top-down methodology or the Bottom-up methodology.

In the present and in reality, most of companies are likely to be armed with Platform methodology. Although it is not so necessary that this approach includes hardware/-software co-design with virtual prototyping, the co-design with virtual prototypes often goes with Platform methodology for large productivity gain. Although the current Platform methodology is so helpful, it still has critical drawbacks, which is the reason why we still need a novel and better design approach.



(a) The Top-down Methodology



(b) The Bottom-up Methodology

Figure 1: Two Traditional Methodologies : The Top-down and Bottom-up [3]

The novel system design methodology underlying ESE(Embedded System Environment) overcomes those drawback while keeping the best features of the previous ones. In this section, a historical review of system design methodologies will be followed by brief introduction to the concept of ESE. The best way to understand anything contemporary is to understand its history.

1.1 The Top-down and Bottom-up Methodology

The Top-down methodology is shown in the Figure 1(a). The design starts with an abstract model of the system in concern. Then, this model is further refined until its layout can be given. The main drawback stems from the fact that we cannot estimate metrics accurately on an upper level in Gajski Y-chart. It means that the impact of design decision on upper level is not predictable enough.

The Bottom-up methodology(Figure 1(b)) is another representative traditional design approach. The starting point is a set of presynthesized components. Each level generates library for its the next upper level. Critical defects of this approach mainly come out due to the fact that the needs or optimality on upper levels can hardly predicted on lower levels. Moreover, with this approach, system level complexity is often quite high.

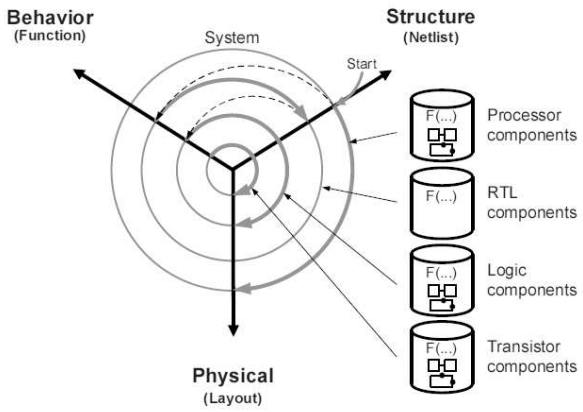


Figure 2: Platform Methodology [3]

1.2 Platform Methodology

Mainly due to the main drawbacks mentioned above, any of the two traditional design approaches is not so satisfying that designers can cope with the design complexity of contemporary and/or future embedded systems.

In reality, the most widely accepted design approach seems to be platform methodology. It is depicted in Figure 2 [3]. The reason why it is widely accepted is that design methodologies are often product-oriented and system platforms are already accessible for companies.

A platform is a partial design that is optimized not for a single application but for a set of applications in a specific domain such as multimedia. Legal compositions of elements and interconnects are limited and each of them is called a platform instance.

The design in Platform methodology starts with a platform instance on the system level. Standard components with well-defined layouts as well as custom components for application optimization are given.

In this approach, since the metrics of the components in the platform is often known. The impact of any design decision on the system level can be predictable. Moreover, when synthesizing custom components, the need on the system level can be propagated in this methodology.

However, in addition to that platform customization is still needed, since a platform is not so flexible in nature, designers cannot easily respond to future change in requirement.

1.3 Hardware/software Co-design with Virtual Prototyping

We can see how the trend of Platform methodology has changed in Figure 3. The future trend is actually what ESE is based on and will be explained later.

The main difference between the past trend and the present one is hardware/software co-design with virtual prototyping.

In the past, after once a system was partitioned into hardware and software, the software engineers had to wait until the production of the structural model of the hardware. Moreover, the co-simulation environment for software developers was not fast enough since too many details were included in the structural model of hardware and its counterpart, software. These two critically lengthened design cycle.

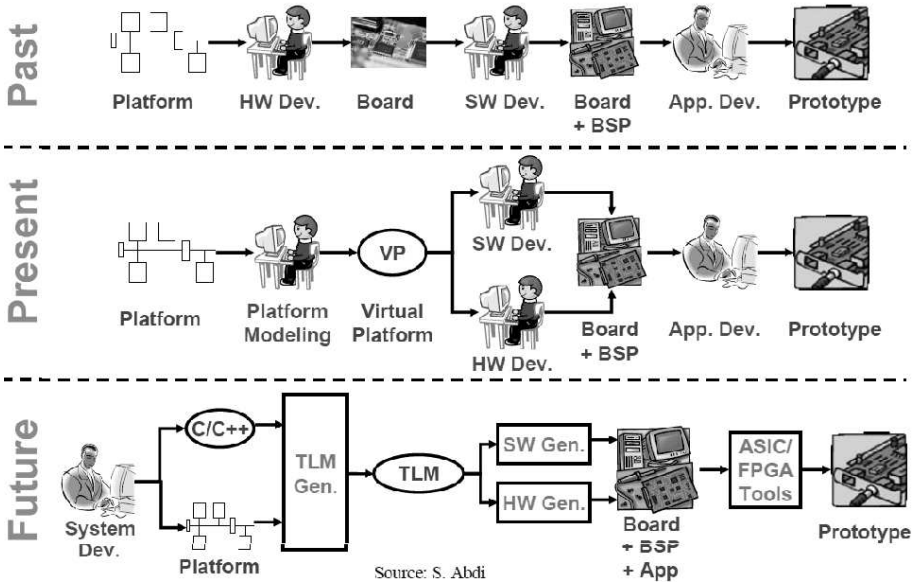


Figure 3: System Design Trend [2]

Virtual prototyping is almost a natural consequence to gain higher productivity by shortening design cycle. Virtual prototyping is to provide a co-simulation environment, into which hardware and software are integrated, before actual hardware is ready. Usually, implementation details are hidden to raise the level of abstraction. It reduce design cycle by enabling co-design and speeding up the simulation.

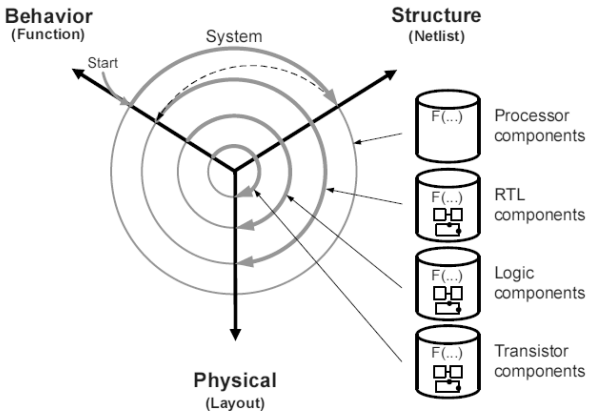
Transaction level model is one of the most well-known models that can serve as a virtual prototype but is not the only model used for virtual prototyping.

1.4 Embedded System Environment

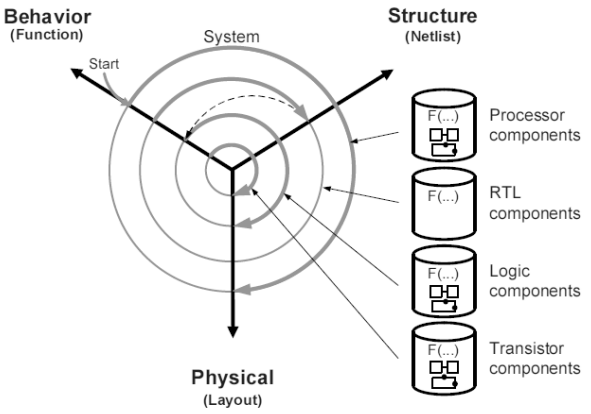
Platform methodology that goes with hardware/software co-design still has drawbacks that can never be ignored. As mentioned above, the starting point is not behavior of a system but its netlist. Due to this, it can be applied only narrowly, which means that it might be fine with contemporary embedded system designs but cannot satisfy future demands.

Moreover, models used for virtual prototyping is too simulation oriented and do not consider synthesis enough. The refinement steps and models are not defined solidly. With those lousy and unclear models, automatic synthesis is not easy.

The system design methodology underlying ESE overcomes these drawbacks. It leads the design to start with not a platform instance but behavioral specification of the system, which is flexible in nature. This is the same as we can see in the Figure 4, where the starting point is functional description on system level. In both of these approaches, the functional description is synthesized into the structure of processor components. However, unlike the meet-in-the-middle methodology in Figure 4(b), where the processor components are virtual and have no layout yet, in ESE, the layouts of those components are given only for the strength of Platform methodology to be preserved.



(a) Meet-in-the-Middle Methodology



(b) ESE Design Methodology

Figure 4: The Meet-in-the-middle and ESE Methodology [3]

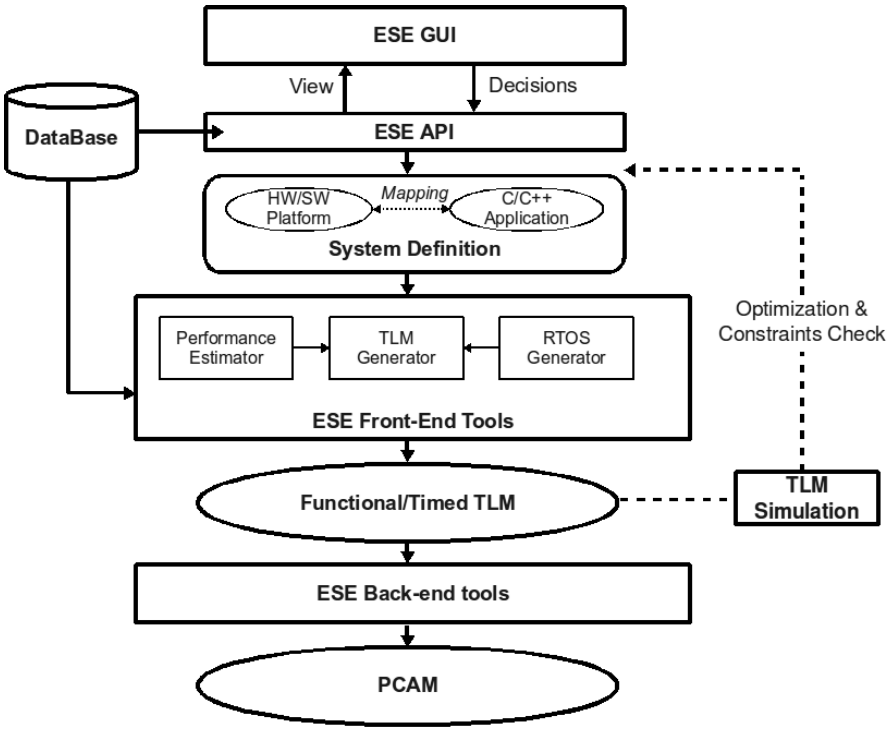


Figure 5: ESE Design Flow [3]

1.5 ESE Design Flow

ESE defines three models for system level synthesis; Specification model, TLM and PCAM. We claim that three is the minimal number of necessary models. We need specification model, which enables us to specify the system without considering implementation, to ease system design. Moreover, without any doubt, any practical system needs PCAM. However, specification model does not provide any information to estimate the implemented system in terms of timing, power consumption and so on. PCAM is not ready in early stages of a design process and the speed of simulation of PCAM is unfairly low. We need at least one more model, TLM.

The design flow with ESE depicted in Figure 5 goes on around these three models. A user of ESE, system designer, defines the system with ESE GUI on the top of ESE API to produce the specification model. The specification model is drawn inside the box, "System Definition". An application written in C/C++ is given without considering any implementation. The platform instance is retrieved from component library. The specification model is obtained by mapping an application and a platform instance. Then, TLM is generated by ESE front-end tools. The designer is allowed to estimate the system by simulation. ESE back-end generates PCAM from the TLM.

What is intended by this report is to provide far detailed explanation about ESE front-end by elucidating this design flow in great detail. Especially, TLM generation is mainly focused on. The remaining part of this report is organized as follows. One section, section 2 is assigned to discuss specification model. Three sections, section 3, 4 and 5 are written to explain TLM generation and followed by section 6, in which the improvement in TLM generator is discussed. The last section is conclusion.

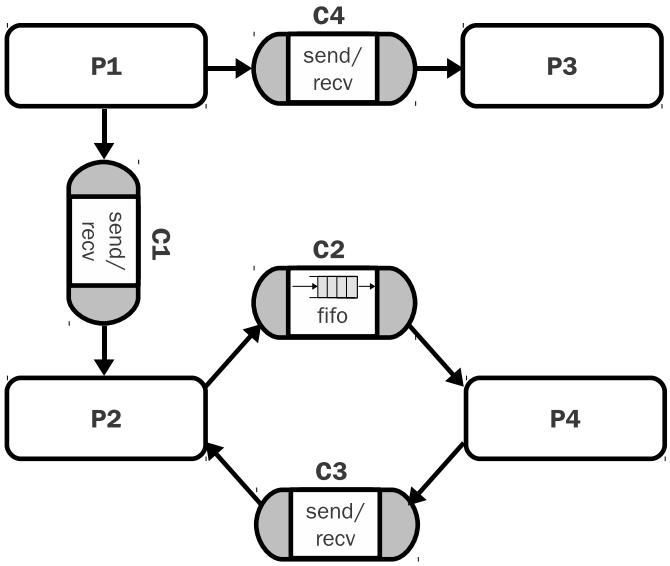


Figure 6: Communicating, Flattened Processes and Channels as Behavioral Specification Model in ESE

2 System Definition in ESE

The most important things that should be taken into consideration when we think about a system specification tool is as follows.

- Which model is used for the behavioral specification of the system?
- Which components should be included in the platform library?
- Mapping rule

A behavior of a system is currently represented as a set of flattened processes and abstract channels connected to processes. One simple example can be found in Figure 6. Four processes, P1, P2, P3 and P4 are communicating via process-to-process channels C1, C3 and C4 and a FIFO channel, C2.

Generally, a platform instance is decomposed of PEs, CEs and buses. A PE can be one of a CPU, a custom HW, a HW IP or memory. In the example in Figure 7, CPU0, HW0, IP0 and Memory0 are PE components, which are respectively a CPU, a custom HW, an IP and a memory. These components can be connected to buses such as Bus1 and Bus2. If two buses are not compatible in protocol, a transducer like Tx0 or a bridge should be introduced. Note that a transducer also have a FIFO in it, which is basically used when transferring data from a bus to another bus.

Users define their system by mapping these two, a platform instance and behavioral specification. Therefore, firstly, users specify the behavior of the system. Secondly, they select a platform instance from the platform library. The final step to define the system is mapping. Processes are mapped onto PEs. Channels are usually mapped onto routes. However, if two processes communicating via a channel are mapped onto the same PE, a channel is mapped onto a local memory of the PE. Moreover, if more than one processes are mapped onto the same PE, the PE must be a processor and has an RTOS on it. All of these are exemplified in Figure 8. Process P1 and P2 are mapped onto the same PE, CPU0, which is a processor. A RTOS should be specified

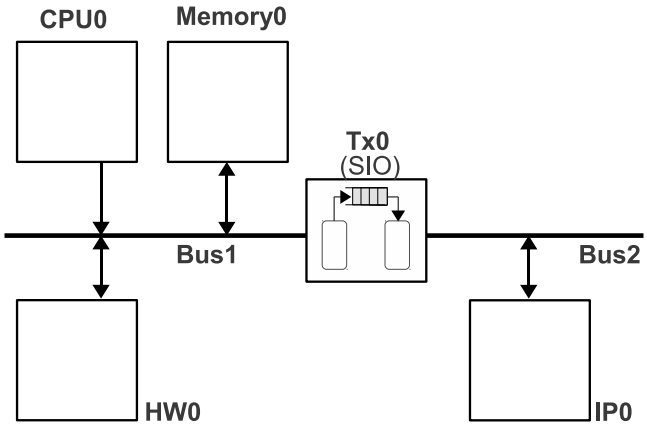


Figure 7: A Simple Example of a Platform Instance in ESE

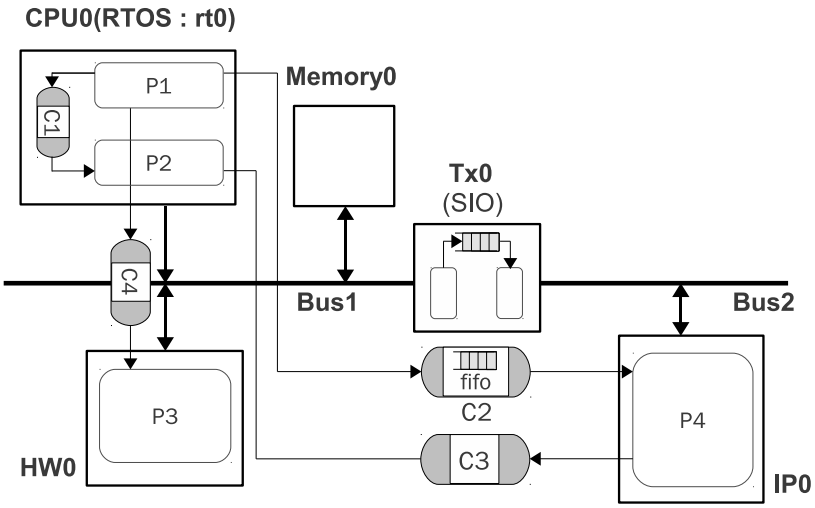


Figure 8: The Result of Mapping Figure 6 onto Figure 8

in this case. Process P3 and P4 are mapped onto HW0 and IP0 respectively, where HW0 is a custom HW and IP0 is a HW IP. Channel C1, C2 and C3 are mapped onto the route, "P1, Bus1, P3", "P1, Bus, Tx0, Bus2, IP0" and "P2, Bus1, Tx0, Bus2, IP0", respectively. However, C1 is mapped onto the local memory of CPU0 since both of P1 and P2 are mapped onto CPU0. For the same reason, CPU0 must be a processor and a RTOS should be introduced. That is the reason why a RTOS, rt0 should be specified for CPU0.

To represent these specification, platform and mapping, the well-defined data structure that is easy to manipulate for design space exploration is really essential. Currently, the data structure is unified and in form of an XML tree.

3 Refinement from Specification Model to Transaction Level Model

ESE front-end covers TLM generation. With TLM, designers can estimate their system before final implementation is ready, due to which design space exploration in

early design phases are allowed to them.

The goal of this section is as follows.

- Explaining how specification model is refined into TLM
- Explaining how ESE achieves this goal

Since computation is separated from communication in our TLM, we elucidate computation and communication separately. In addition to that, please note that understanding TLM generator is no hard if and only if we understand the *generated* TLM. Therefore, we have to focus on the generated TLM first and then the TLM generator can be easily understood.

The remaining part is organized as follows. In section 4, computation refinement will be explained. It is generation of PEs, processes and RTOS that are mainly focused on. The main topic of section 5 is generation of communication API for various types of channels.

4 Computation Refinement

In reality, TLM is even different from company to company. ESE defines two kinds of TLM; Functional and timed TLM. Timed TLM is different from functional one only in that RTOS may be introduced when multiple processes are mapped onto a single PE and it is annotated with timing. Therefore, we explain functional TLM first and timed TLM later.

4.1 Computation Refinement in Functional TLM

The components that serves as computational part in TLM are processes and PEs. In this section, the modeling style of PE and processes and their generation will be discussed.

PEs are modeled as a sc_module in SystemC, which resembles "entity" in VHDL. Processes mapped onto a PE are modeled as sub modules of the PE. Grammatically, each of these sub modules is also an independent sc_module. The PE sc_module instantiates processes in it and each of the process sc_modules calls users' function inside it, whose name is the same as that of the process. We can see it in Listing 1.

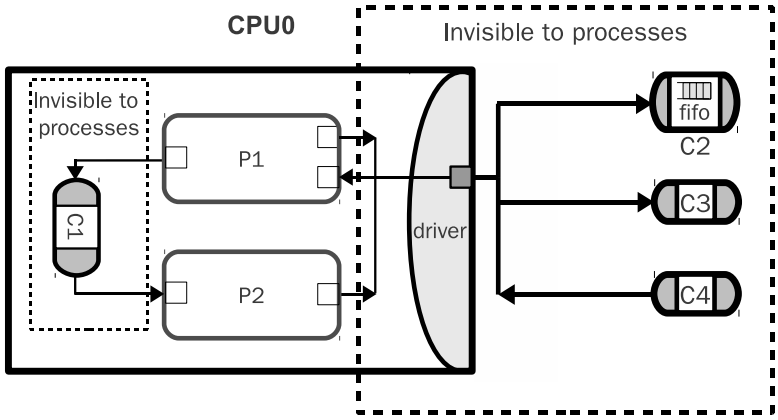


Figure 9: Processes and PEs in Functional TLM in View of Process/PE Generation

```
CPU0 : public sc_module {

  Process P1, P2;
  ...
  main()
  {
    P1 . main();
    P2 . main();
     ...
  }
};

P1 : public sc_module {

  ...
  main()
  {
    P1(); // call users' function
  }
};

P2 : public sc_module {

  ...
  main()
  {
    P2(); // call users' function
  }
};
```

Listing 1: Hierarchy Between Processes and PEs

The remaining part of modeling PE and processes are ports. To explain it, a typical example of PE/process modeling is introduced in Figure 9. This example is from Figure 8.

First of all, during generation of processes, channels are not taken into consideration. As we see in the figure, channels are not directly visible to processes. What is visible to processes are process ports and their interfaces called communication API. These ports are drawn as the small white squares in Figure 9. We need to insert these ports but do not have to care more about channels until generation of processes is done.

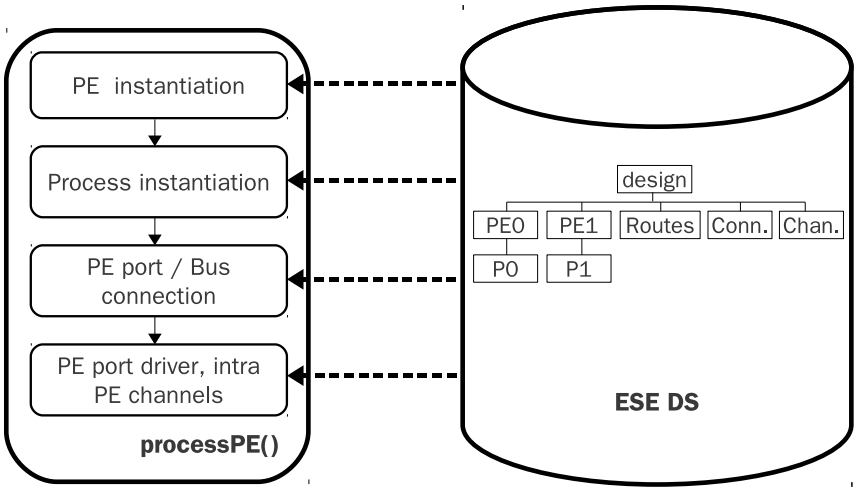


Figure 10: TLM Generation for PE and Processes

Secondly, during generation of PEs, the list of tasks are as follows.

- Implementing Intra PE Channels
- Inserting PE ports
- Implementing PE Ports Drivers to Implement Process Ports by Using PE Ports.

A channel is either an intra PE channel like C1 or external to the PE as C2, C3 and C4 are external to CPU0. In the first case, the channel should be implemented inside the PE to implement the interface of process ports. The second case needs to be explained in detail. A PE is connected to a bus via a PE port. In the figure, small gray square attached to CPU0 is the PE port and CPU0 has one PE port. We can see that this PE port is connected to Bus1 in Figure 8 and Bus1 implements the interface linked with this port. By the way, process ports to PE ports mapping is many-to-one. We can find an analogy in Linux. Linux provides processes with many logical "sockets". However, in fact, it usually has one or two Ethernet cards. Like Linux, in TLM, we may have multiple process ports that need to access the same PE port. To resolve this conflict, drivers have to be introduced inside PEs.

The generation procedures to produce processes and PEs are summarized in Figure 10.

- PEs are generated with PE ports. However, PE port drivers and intra PE channels are not yet implemented.
- Processes are generated in the way explained above.
- PE ports are connected to the proper buses.
- PE port drivers and intra PE channels are implemented.

In the example in Figure 9, the first step is to generate CPU0 with the PE port, the small gray square. The second step is to generate P1 and P2 with process ports, the small white squares. The third step is to connect the PE port to the bus, Bus1. The last step is to implement the PE port driver and the intra PE channel, C1. All of these steps are included in a single function, "processPE".

The information to do that is kept in the data structure, ESE DS, which is in the form of an XML tree as depicted in Figure 10. For example, to insert PE ports, we need to know the connectivity between PEs and buses and the list of these connectivity are kept in ESE DS. To do the work described above, we need the information about PEs, processes, channels, routes and connectives.

4.2 Computation Refinement in Timed TLM

Timed TLM is almost the same as functional one except the followings. First, timed TLM has RTOS model. If multiple processes are mapped onto a single PE, then, a RTOS should be introduced. Second, only timed TLM is annotated with timing information, which is estimated based on high level estimation techniques [5], [4].

Communication delay will be discussed in section 5. Computation delays are decomposed into two components; RTOS overhead and execution delays of applications. Dealing computation estimation in great detail is beyond the scope of this report. It will be briefly introduced and the relationship between estimation and TLM generation will be addressed in section 4.2.1.

Considering delays does not change the modeling style so much. On the contrary, introduction of RTOS models does affects the modeling style and the generation of TLM. These will be discussed in section 4.2.2.

4.2.1 Computation Estimation

Applications are annotated with timing by the estimation engine. The concept is roughly depicted in Figure 11. In the figure, two processes P1 and P2 mapped onto a PE, CPU0 is shown. As we explained, a process is modeled as sc_module. In the main thread of the sc_module, which is "main" in this example, users' function is called. The main thread of process P1, called main, is calling users' function, P1. However, the function that is actually called is different. In functional TLM, the function is users' original function shown at the top right of the figure. On the contrary, in timed TLM, it is the newly generated, time-annotated version of the original function. This new version of the function is found at the bottom right of the figure.

[4] shows its concept and algorithm to estimate execution delay of applications using PUM.

Users' function is given to the estimation engine as an input. During the estimation phase, P1 is converted into a control data flow graph. Every basic block in the graph is annotated with timing information by the estimation engine. The algorithm to estimate timing is well explained in [4]. As we can see Figure 11, two basic blocks, BB1 and BB2 are annotated with timing, BB1 delay and BB2 delay. These delays are estimation of the execution time of those two basic blocks on CPU0. The control flow graph cannot be executed directly under the TLM simulation environment. It needs to be generated as executable codes such as SystemC. A delay is in the form of wait functions.

Conceptually, the delay due to execution and that due to cache or branch prediction may be separated. The latter is quite complicated to estimate. Currently, the expectation value of the cost of load/store instructions considering cache is computed in static time based on the statistics. For example, with ARM9, we can obtain statistics related cache hit/miss rate. In addition to that, we can obtain the cost of a cache miss based on data sheet and statistics. The expectation value of the cost due to cache can be computed in static time. The delay can be inserted at the end of every

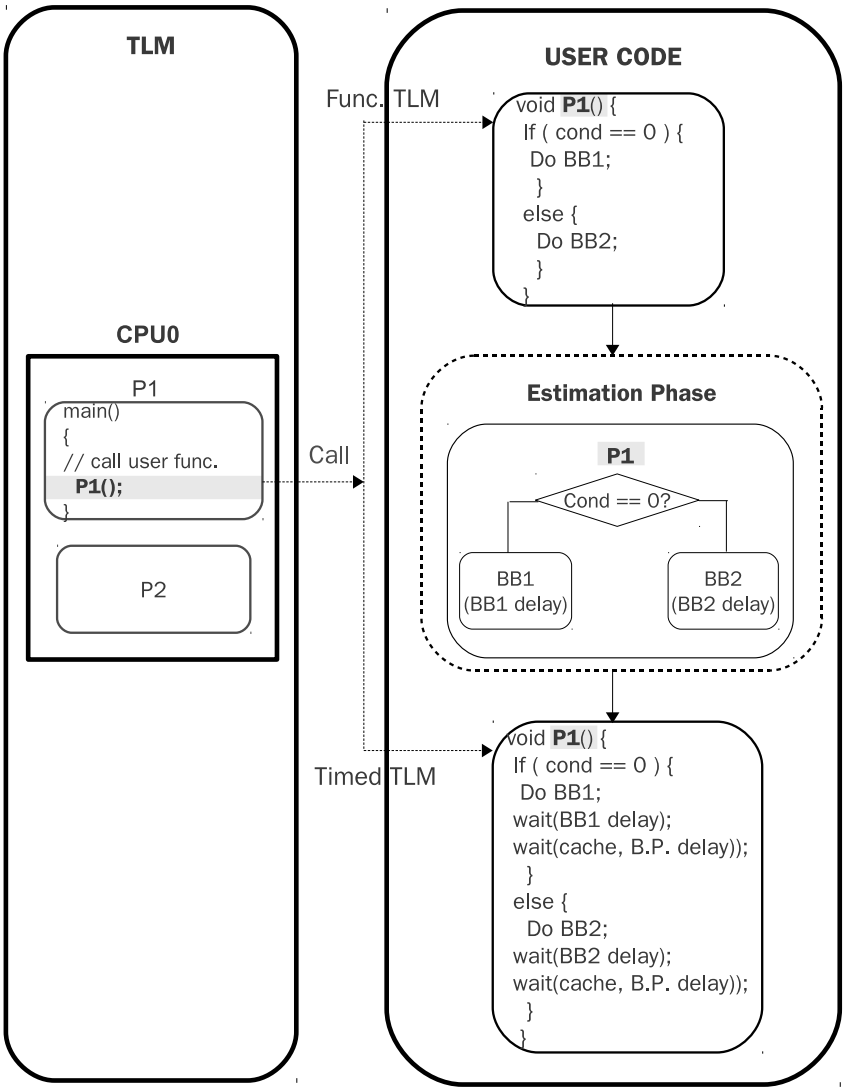


Figure 11: Timing Annotation To Users' Functions

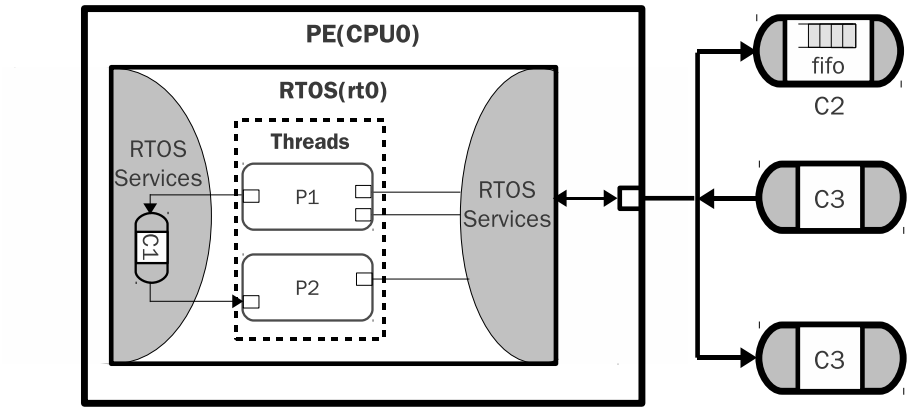


Figure 12: Processes/PEs Example with RTOS

basic block since the delay is computed based on statistics. The same rule is applied to the cost of branch instructions. Currently, while assuming that branch prediction policy is very simple one, we computes the expectation value and insert it at the end of every basic blocks. So, the timing annotated version of P1 includes those delays as additional wait functions.

However, note that the name of two different versions of the function is the same, P1. When generating CPU0, P1 and P2, we need to know the type and name of P1 but not its implementation. So, in the view of generation, timing annotation does not affect so much.

4.2.2 RTOS modeling

The modeling style changes at introduction of RTOS. It is depicted in Figure 12. The example in Figure 8 and 9 is used again. First of all, a RTOS, rt0 is introduced. This RTOS is modeled as a sc_module, which is, at the same time, a sub module of the PE, CPU0. Processes(P1 and P2) are no more independent sc_modules. Instead, they are threads managed by rt0. In functional TLM, since they were two sc_modules, the order of execution is dependent on the SystemC simulation kernel. Now, in timed TLM, it depends on the RTOS model. In Figure 9, drivers are inserted to bind process ports with PE ports. In timed TLM, communication API, the interfaces of process ports are implemented by using RTOS services. A intra PE channel such as C1 is located in the local memory of the PE, CPU0 and accessed by calling RTOS system calls.

The generation procedure is summarized as in Figure 13.

Above all, without RTOS, these steps are the same as those of function TLM generation. Or, they are as follows.

- PEs are generated with PE ports. RTOS models are included in the PEs
- RTOS models are retrieved from the SW library. Instead, sc_modules representing processes are not generated.
- PE ports are connected to the proper buses.

The first step is almost the same. It produces the CPU0 module in Figure 12. However, CPU0, the PE is generated so that it declares the RTOS, rt0 as its sub module instead of processes, P1 and P2. This is unlike the functional TLM case in Figure 9. Processes are invisible to CPU0.

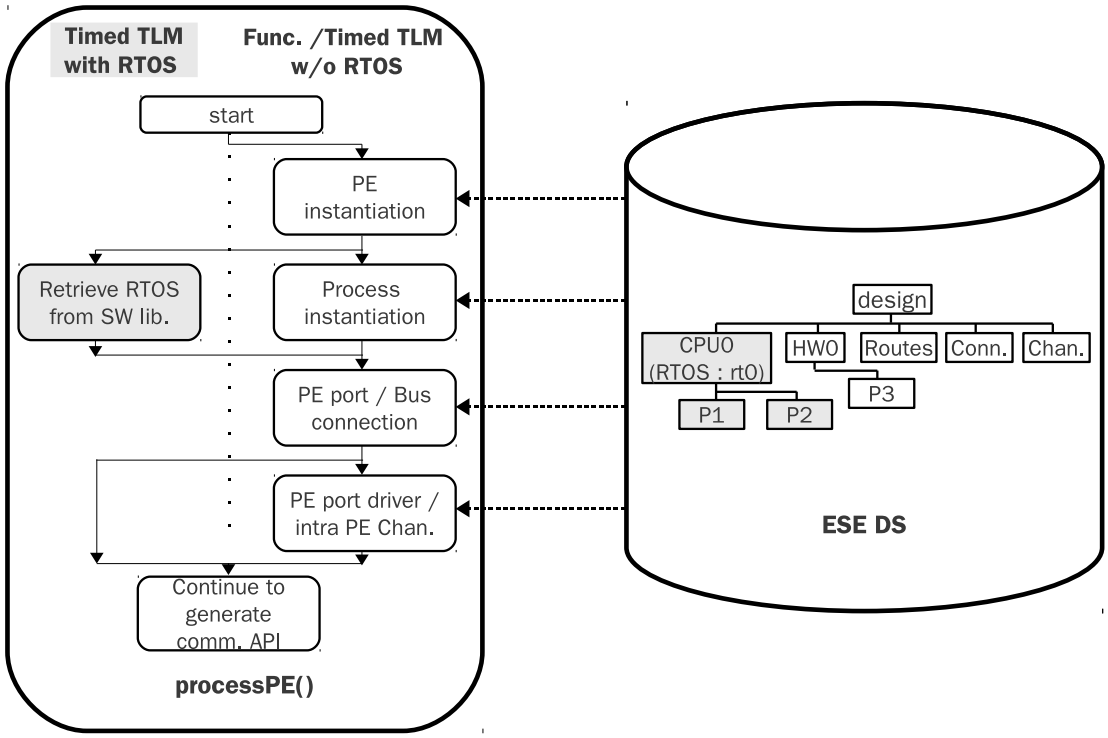


Figure 13: Modified Generation Steps when RTOS is present

In the second step in Figure 13, the RTOS model is retrieved from the SW library. The reason why RTOS models are not generated but retrieved will be explained later. As depicted at the right of the figure, the kind and configuration of the RTOS for CPU0 have to be specified and stored in ESE DS. As depicted in Figure 12, intra PE channels such as C1 are implemented as a part of RTOS services. In addition to that, RTOS rt0 takes the responsibility to map process ports onto PE ports so that communication API is implemented on the top of RTOS services.

The step in which PE ports are connected to the buses is the same. The last step in the functional TLM case is not necessary in the timed TLM case if a RTOS is present on the PE.

Now, we will explain why RTOS models are retrieved rather than generated. In fact, RTOS model does not need to be generated again and again at the every change of the system. RTOS services are common. Most RTOS services can be parameterized. For example, the scheduling policy may be different from system to system. However, we can parameterize it and do not need to implement the scheduler again and again.

It is depicted in Figure 14. Conceptually, all the TLM components including PEs, CEs or buses(the blue gray boxes) are generated after TLM component generation. However, ideally, RTOS models(the darker gray boxes) are not. It has been already made and stored in SW library. By retrieving the RTOS models and linking all the other TLM components with the RTOS model, we can obtain the final timed TLM.

5 Communication Refinement

In this section, the communication part will be discussed. Communication API generation is mainly focused on.

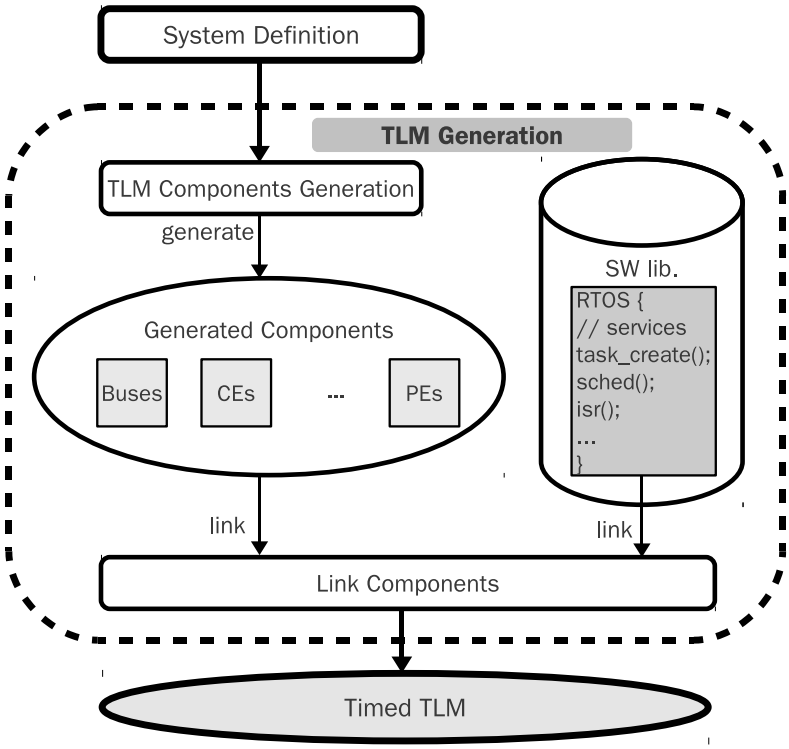


Figure 14: Linking The Generated Components with RTOS from the SW Library

Figure 15 enumerates all the kinds of channels that ESE currently supports. By explaining each of them, we can understanding TLM generation for communication.

In the figure, we can see that channels can be divided into three groups; Process-to-process, memory and FIFO channels. Each of these groups can be further decomposed into two or three groups depending on the relative location of the communicating processes or memory.

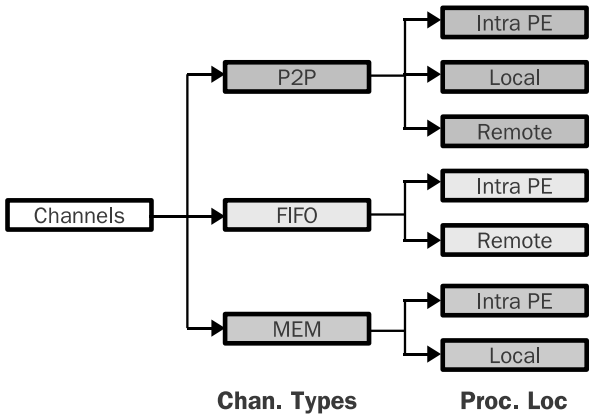
- Intra PE : Two processes(or one process and one memory) are mapped onto the same PE.
- Local Communication : Two partners are mapped onto different PEs. However, these two PEs are connected to the same PE and the data is directly sent via the bus.
- Remote Communication : The route that the channel is mapped onto includes one or more CEs in it.

In addition to that, each group of FIFO channels is still subdivided into two or three groups according to the location of the FIFO buffers. A FIFO buffer can be either in a transducer or in local memory of a reader/writer PE.

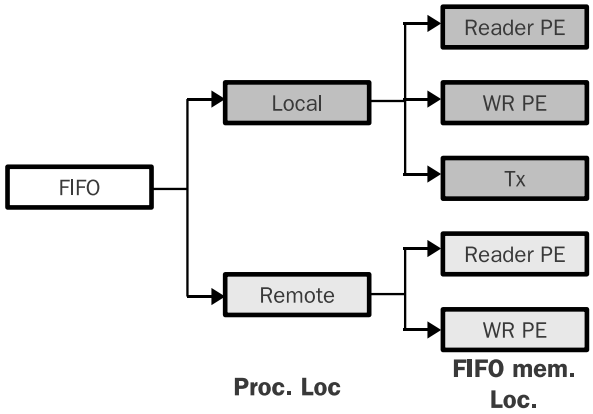
We will review all of them. However, all the channels are quite similar so that we can almost explain other channels by addressing the difference from P2P channels.

5.1 Process-to-process Channels

P2p channels are viewed to users as send/recv functions. As we have seen, there are three types of P2P channels; Intra PE, local communication and remote communication.



(a) Classification of Channels by the Semantics and Relative Locations of Processes/Memories



(b) Classification of FIFO Channels by the Location of FIFO Buffer

Figure 15: Classification of Channels in ESE

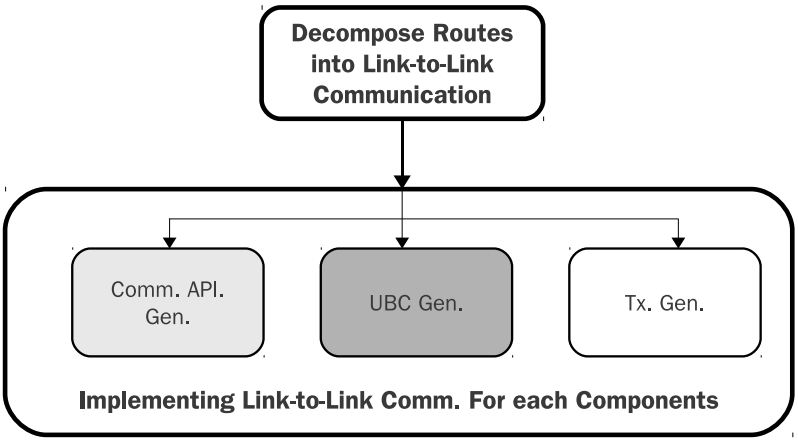


Figure 16: The Tasks for Generating TLM to Implement P2P Channels

Channel C1 the example in Figure 8, 9 and Figure 12 is an intra PE channel. Two processes, P1 and P2 are mapped onto the same PE. In this case, the channel is implemented with RTOS services. Since this case is trivial, we will not go deep into the generation of this kind of P2p channels.

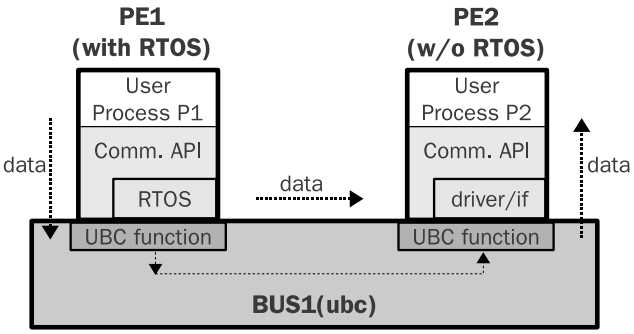
The rest two will be explained in this section.

Figure 16 shows the dependency graph of the tasks for TLM generation related with implementation of P2P channels. The first thing is to split every route into link-to-link communication. For example, in Figure 8, a send/recv channel C3 is mapped onto the route, "P2, Bus1, Tx0, Bus2, P4". In the view of P2P channel users, the communication via the channels are considered as in the network layer. However, because the communication via UBC is in the data link layer, the route should be split into one or more link-to-link communication. For example, the route "P2, Bus1, Tx0, Bus2, P4" is split into two link-to-link communications; P2 sends data to Tx0 via Bus1 and Tx0 propagates the data to P4 via Bus2. This step is not necessary in the local communication case, where two processes are mapped onto different PEs but the PEs are connected to the same bus.

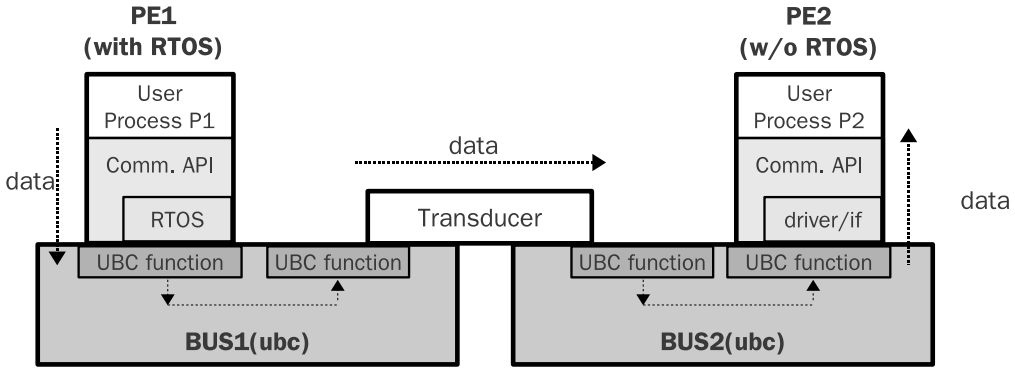
Once a route is decomposed into one or more link-to-link communications, the next step should be implementing every of them. As described in Figure 16, the three main things for it are communication API generation, UBC generation and transducer generation.

To understand it, we need to pay attention to Figure 17. Both of them show that P1 on PE1 sends data to P2 on PE2. PE1 has a RTOS while PE2 does not. In Figure 17(a), PE1 and PE2 is connected to the same bus, BUS1. On the contrary, in Figure 17(b), PE1 and PE2 are connected to different buses, Bus1 and Bus2 respectively. The route onto which the channel is mapped is "P1, Bus1, Tx, Bus2, P2" in Figure 17(b).

In the first example, a link-to-link communication needs to be implemented. P1 and P2 access the channel by calling communication API. If a PE, like PE1, has a RTOS, the communication API is implemented depending on the RTOS services. Otherwise, it is implemented using software drivers or hardware interfaces. In any case, software drivers, hardware interfaces and RTOS services are on the top of UBC functions. UBC functions are to provide bus functionality such as synchronized send/recv via the bus. In TLM, for example, two PEs connected to the same bus call the UBC send/recv functions, by which UBC provides synchronized data transfer. In this case, two tasks are given to implement a link-to-link communication; Generating UBC to implement



(a) P2P Channels in the "Local Communication" Case



(b) P2P Channels in the "Remote Communication" Case

Figure 17: Simplified Protocol Stack in P2P Channels

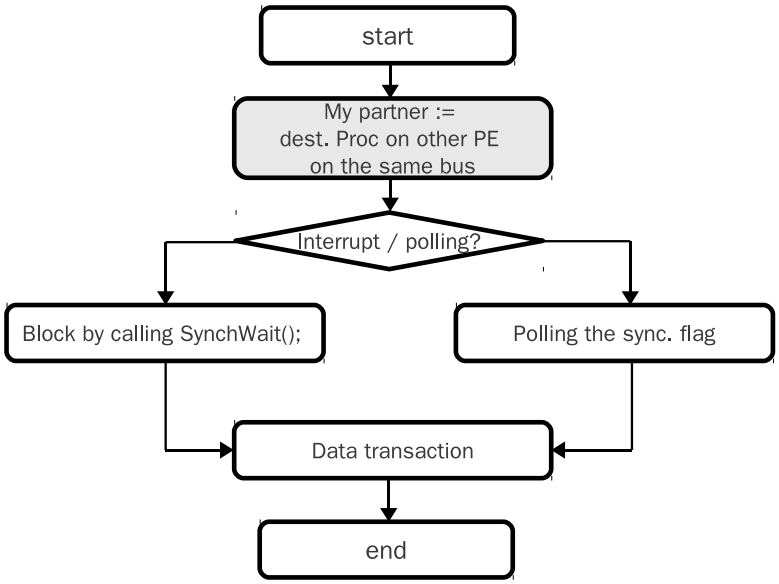
UBC functions and producing communication API.

In the second example, that in Figure 17(b), one of two partners participating in a link-to-link communication is a transducer. The rest is the same as the first example. So, implementing transducer is added to the tasks to implement a link-to-link communication.

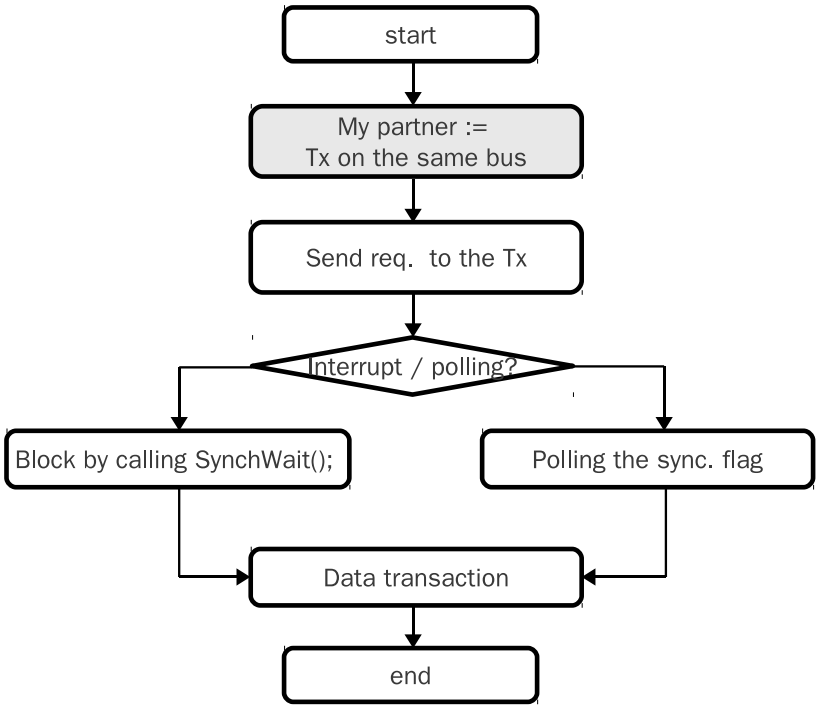
From now on, communication API generation will be explained in greater detail. UBC and transducer generation will be discussed at the end of this section, briefly.

Figure 18 shows the control flow of the generated send/recv functions. It tells us what we have to generate as communication API for P2P channels. The first job is setting the bus address. In the local communication case, the partner is the receiver. In the remote communication case, the partner is a transducer. That is the reason why setting bus address is different between two Figures, 18(a) and 18(b). If the communication partner is a transducer, sending a request to the transducer by writing a request in its request buffer is also necessary and that is the reason why a request is sent to the transducer just after setting the bus address in Figure 18(b). After setting the bus address, two main tasks follow; One is synchronization and the other is data transfer. Synchronization can be either depending on interrupt or done by polling the dedicated flag on the bus. The interrupt mechanism is provided by UBC as a UBC function, synchWait. Once the sender and receiver are synchronized with each other, data transfer is initiated.

Figure 19 shows the control flow of a send/recv function when RTOS model and timing annotation are taken into account. Most things are the same. However, as we can see the gray rounded rectangle named RTOS services, to be synchronized, the communication API should depend RTOS services instead of direct call of UBC



(a) Send/Recv Function in "Local Communication"



(b) Send/Recv Function in "Remote Communication"

Figure 18: Send/Recv Functions

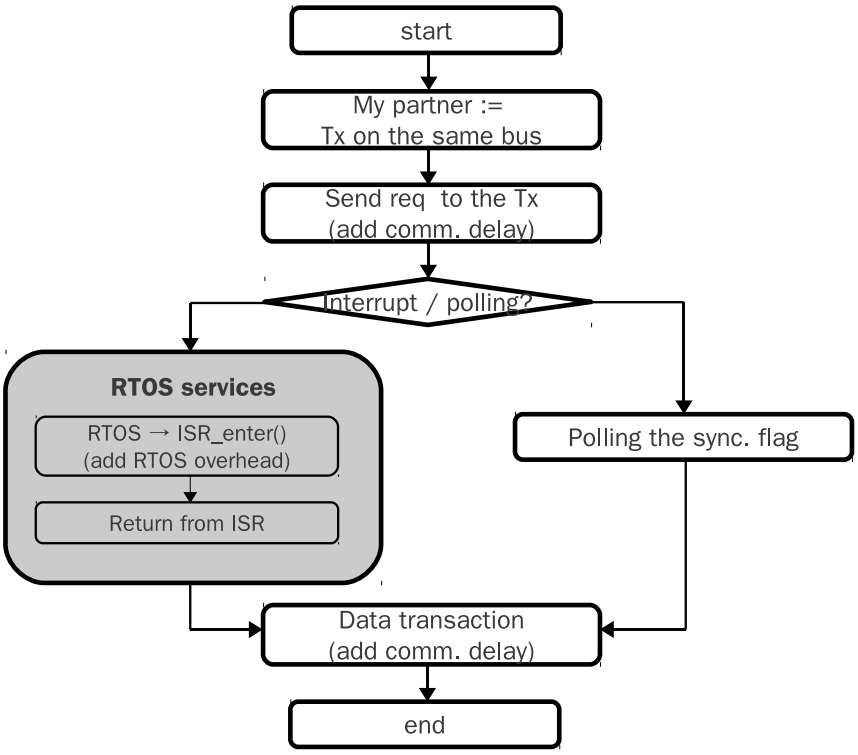


Figure 19: Send/Recv Functions with RTOS and Timing Annotation

functions. RTOS overhead is accompanied by RTOS services. For example, the overhead due to ISR_enter, interrupt service routine in the figure, contributes to the total delay in simulation. In addition to that, every communication related function is followed by accumulation of communication delay. How these communication delays are estimated will be explained in section 5.4.

In summary, communication API has to be generated as Figure 18 in functional TLM and as Figure 19 in timed TLM depending on the configuration. The control flow of send/recv functions are definitely simple, which leads their generation to be simple.

The generation of UBC and Tx are elucidated in [1] and [5] and will not be discussed here. Briefly, UBC has to provide five user functions, send/recv, read/write and memory service. In addition to that, UBC also endows its users with two types of synchronization functions, synchWait() and readFlag(). One is for interrupt and the other is for polling. Transducers have three main modules; I/O module, Tx request buffer and internal FIFO. Request buffers are active actors that check request buffer exposed externally and execute the transaction written in the buffer. I/O modules are directly connected to buses. According to the instruction of request buffer module, retrieve data from the bus and put it on the internal FIFO, or vice versa.

5.2 FIFO Channels

The classification of FIFO channels is shown in Figure 15(b). To be synthesized, FIFO channels cannot be intra PE channels. Two communicating processes must be mapped onto different PEs. A FIFO channel can be divided into two groups based on the location of processes. One is so called “local communication” case, while the other

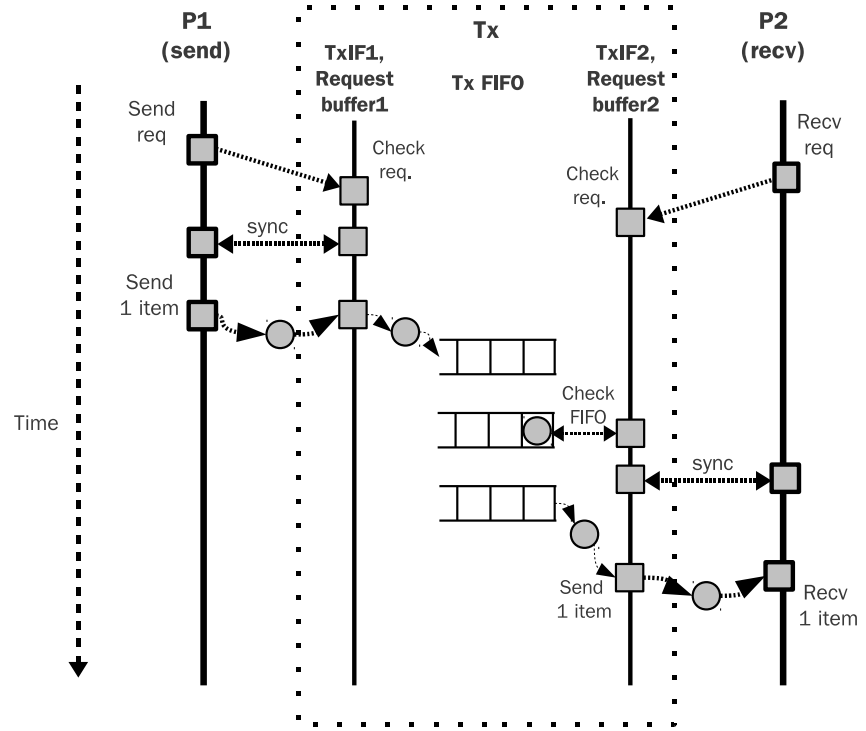
is “remote communication”. In the former, two PEs having the processes respectively are connected to the same bus. In the latter, two PEs are connected to the different buses and need one or more CEs to communicate with each other. In addition to that, each of two groups is subdivided into three groups depending on the location of the FIFO buffer. A FIFO buffer is needed to implement a FIFO channel. This buffer can reside either in a transducer or in the local memory of the reader/writer PE. The last two cases are almost the same in the view of implementation.

If the FIFO buffer is mapped onto a transducer, the FIFO channel is refined almost in the same way as a P2P channel. It is due to the fact that a transducer does have a FIFO buffer inside it. We can regard send/recv communication, where two PEs are connected to different buses, as FIFO write/read with the size of the FIFO is 1. We will show it with an example in Figure 20. In these examples, P1 sends data to P2 via Tx.

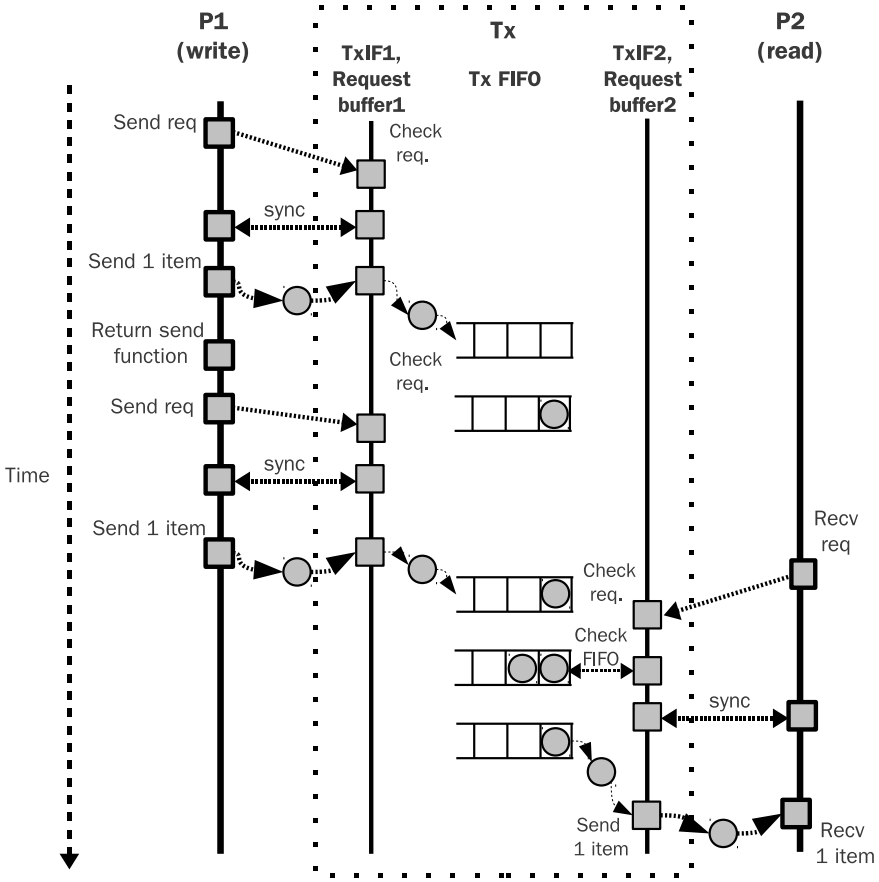
In Figure 20(a), the communication API, send, puts a request on the bus to write it on the request buffer of Tx, Request buffer1. Then, it tries to be synchronized with Tx. After synchronized, data transfer, after which the send function returns, is initiated. The communication API, recv, is called by P2. It puts a request on the bus to deliver it to the request buffer2 of Tx, which is on the P2’s side. Then, the recv function tries synchronization with Tx. Data transfer follows this synchronization. In view of P1 side of Tx, as soon as it checks the send request, begins trying to be synchronized with P1. After synchronization, it pulls the data from the bus and puts it onto the internal FIFO, Tx FIFO. The P2 side of Tx waits until the internal FIFO is not empty and initiates synchronization with P2. As we see, the communication is done through a FIFO. Rather, it is not necessary for the sender to check whether the FIFO is full or not. It is guaranteed not to be full in this case. In addition to that, the receiver does not check whether the FIFO is available or not and return when it is not available. Instead, the recv function assumes that the FIFO *is* available and waits until it is available.

Figure 20(b) shows communication through a FIFO channel. P1 writes data while P2 reads data. We omit checking the status of the FIFO. For a while, assume that the FIFO is not empty and available on reading it. The communication API, write is almost the same as send. It sends request, tries to be synchronized and sends data to Tx after synchronization. At the same time, the communication API, read, looks like recv. It sends a request, tries to be synchronized and gets the item from Tx. Transducer acts as it does in Figure 20(a). The P1 side checks requests from P1. If it finds a request, it tries to be synchronized with P1. Data transfer follows data synchronization. This data is moved into the internal FIFO, Tx FIFO, to feed P2. The P2 side, after getting the request from P2, waits until the FIFO is available. Once it is available, it is synchronized with P2 and sends the data in FIFO to P2. These are almost the same as send/recv in Figure 20(a).

The difference between those two are as follows. In Figure 20(b), after the first write, it can return and does return. Writing FIFO can be non-blocking and if it is the case, it can return just after writing data to the FIFO without considering the status of the reader. In addition to that, the communication API checks the number of rooms in the FIFO even though it is omitted in the Figure. If there are not enough rooms, it can return false before sending a write request to Tx. The same thing is applied to the read function. The communication API, read, has to check if the FIFO has enough items although the step is not depicted in the figure. This checking is done just before sending recv request. If the read is non-blocking, it can return false



(a) Send/Recv Function Scenario



(b) FIFO Write/Read Function Scenario, FIFO Buffer in Tx

Figure 20: Comparison between Send/Recv and FIFO Channel whose buffer is in Tx

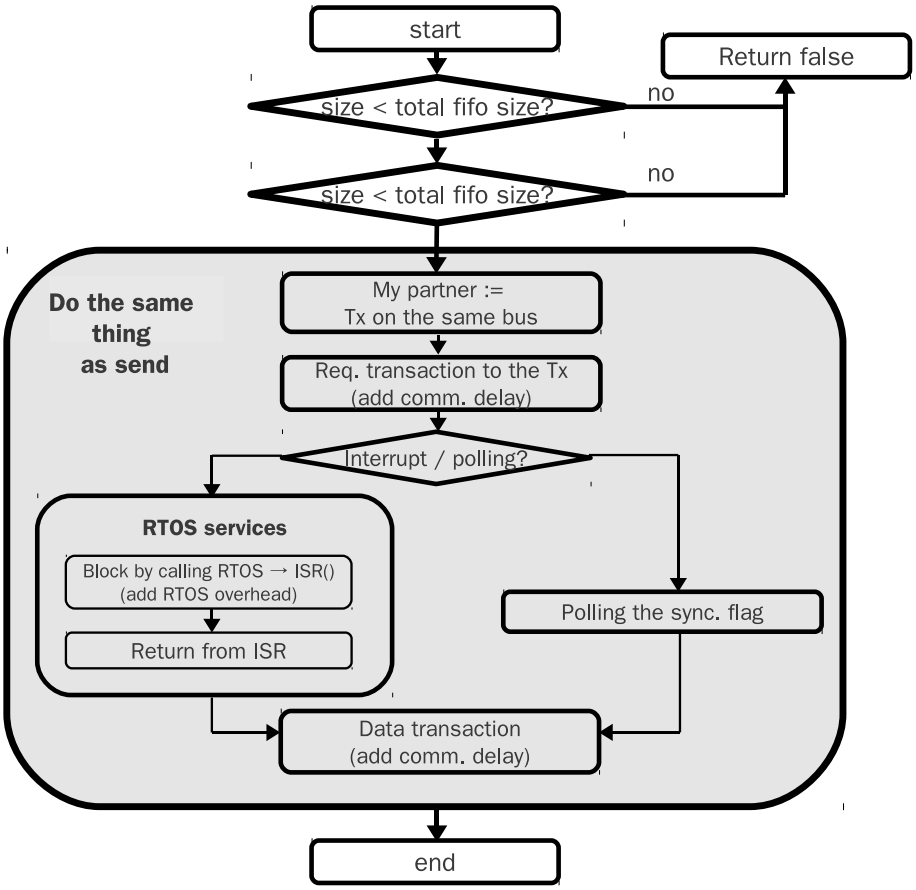


Figure 21: Control Flow of Write Function If PEs Are Not Connected to the Same Bus

immediately without waiting the FIFO by sending a read request.

The control flow of write function in this case is almost the same as send function. That of read function is also almost the same as recv function. As an example, the control flow of write function is described in Figure 21. The first step is to check if the FIFO has enough space or not. After this step, the rest is the same as send function.

The simplified protocol stack for this kind of FIFO channels are depicted in Figure 22. We can easily notice that it resembles the one in Figure 17(b). The only difference it that the former includes free/used checking to make it sure that the FIFO has enough spaces or items. Transducers are almost the same. However, it needs to expose the number of items or spaces available.

Now, we will explain the other cases of FIFO channels. In this case, the FIFO buffer is in the local memory of a PE instead of a transducer. We could not find these channels among the examples we have. Some codes for them are included in ese-2.0, however, it does not seem to work.

In this case, two PEs may or may not be on the same bus. If a process is not on the PE having the FIFO buffer, it is not easy for the process to know the status of the FIFO. The solution is introduction of a FIFO manager. Instead, at the expense of introduction of another process, this PE should be a processor with a RTOS.

The concept of FIFO managers is depicted in Figure 23. In this example, P1 on PE1 writes data while P2 on PE2 reads. Between these two, a logical FIFO channel is laid. However, in implementation level, between these two processes, a FIFO manager is introduced as depicted in Figure 23(a). It serves to P1 as the reader and to P2 as a

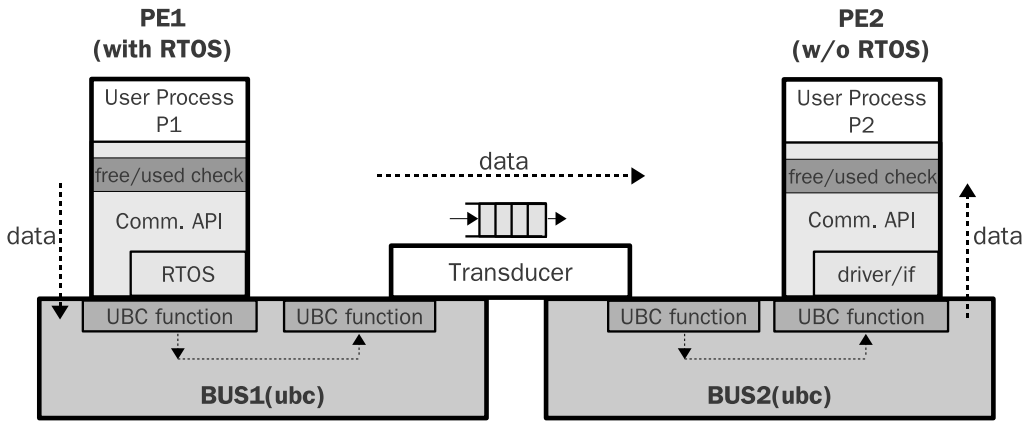


Figure 22: Protocol Stack Example for FIFO Channels

writer. For the time being, we will explain these two communications separately.

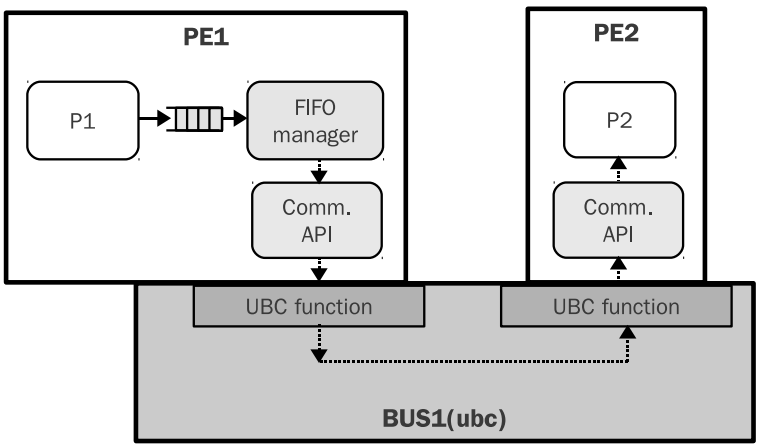
The communication P1 and the manager is simple. A FIFO is laid on the local memory of PE1 and managed by the RTOS. P1 writes data via the RTOS system call. The FIFO manager pops data from the FIFO with the help of RTOS. We can see it in Figure 23(b), where the simplified protocol stack is depicted. In the figure, between P1 and the manager, a FIFO is while being managed by RTOS.

In communication between P2 and the FIFO manager, actual data transfer is the same as send/recv. They will try to be synchronized with their communication partners respectively and data transfer will follow it. We can also see that the protocol stacks is not so different from each other in Figure 23(b). As P2P channels, the manager calls communication API, which is on the top of the RTOS and P2 calls communication API to transfer data from the FIFO manager.

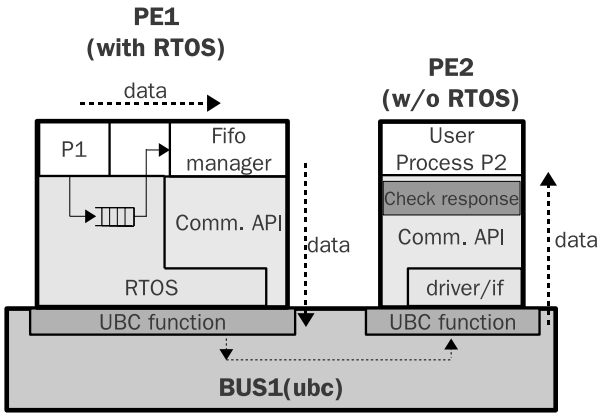
However, as the previous case of FIFO channels, where the FIFO buffer is in a transducer, P2 needs to check the status of the FIFO before these steps. To do that, P2 sends a request and waits for the response. This step is marked as a gray box, labeled “Check response” on P2’s side. At the same time, the FIFO manager should check the send request from P2 and report the status of its FIFO.

Figure 24 shows the control flow of FIFO managers. Figure 24(a) is explaining FIFO managers on the writer side, while Figure 24(a) that on the reader side. As mentioned above, the manager should check requests from the remote partner. In current implementation, the checking is a kind of blocking wait. If the manager has a request from the remote partner, for example, the manage on the writer side gets a request from the reader, it waits until its FIFO buffer has enough items to be served to the reader and sends data to the reader by the same mechanism as “send” function..

Figure 25 exemplifies how the FIFO manager works. The situation is the same as in Figure 23. For simplicity in explanation, we assume that the write is non-blocking and the read is blocking. The first thing that the FIFO manager is doing is to check the read request from P2, as we explained in Figure 24. For the first read request is checked, P1 commands to write 22 bytes(4 + 18) to the buffer, whose size is 15. So, P1 is blocked on calling the second write function at this point. After checking the read request, the FIFO manager is synchronized with P2 and initiates data transfer. After 6 bytes being sent, the buffer does not yet have enough room to complete the second write function call of P1. The manager does nothing for P1 just after the first read request is served. The second transaction between P2 and the manager, which

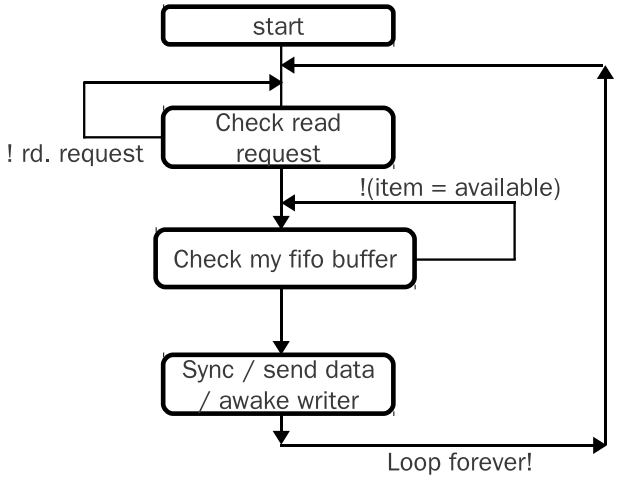


(a) Concept of FIFO Manager

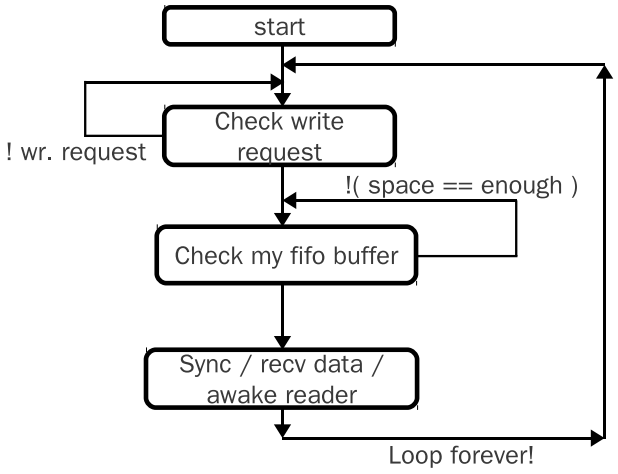


(b) Protocol Stack Including FIFO Manager

Figure 23:



(a) The Job of A FIFO Manager on the writer PE side



(b) The Job of A FIFO Manager on the reader PE side

Figure 24: Control Flow of a FIFO Manager

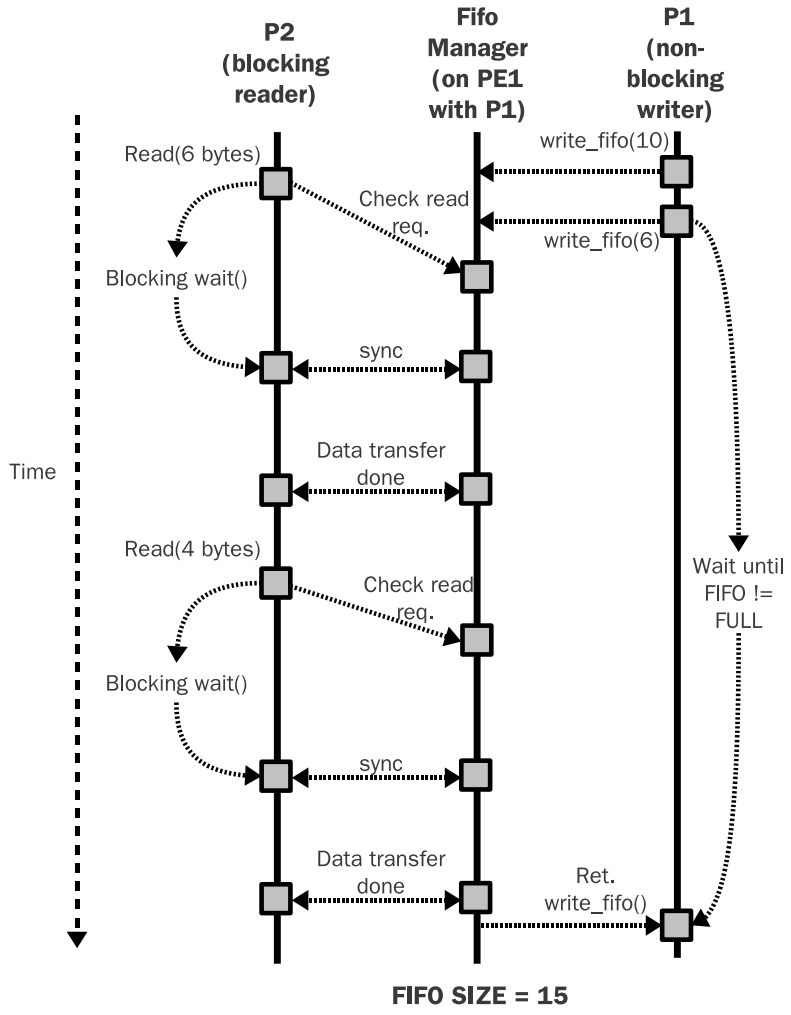


Figure 25: A FIFO with Its Size 15, Data Transaction Scenario [5]

is in the same way, makes enough room for P1. Due to the fact, at the end of second iteration of the FIFO manager, P1 is awakened.

The tasks to generate TLM related to this kind of FIFO channels are as shown in Figure 26. We can expect it is quite similar with the one to implement P2P channels. The main difference is that we have to generate a FIFO manager as well as communication API. Generation of communication API used by the FIFO manager and remote partner is the same as P2P channels or FIFO channels implemented with transducers. On the other hand, generating API for the local partner of the FIFO manager is almost the same as intra PE communication.

5.3 Memory Channels

Memory channels are grouped into two. First, the memory and the process accessing it are on the same PE. Second, these two are not on the same PE, but the PEs are connected to the same bus. Memory channels are not allowed if the process is remote from the memory. Communication API for memory channels is the simplest. If the memory and the process is on the same PE, API is generated with RTOS services followed by RTOS overheads. Or, if they are not on the same PE, API is implemented

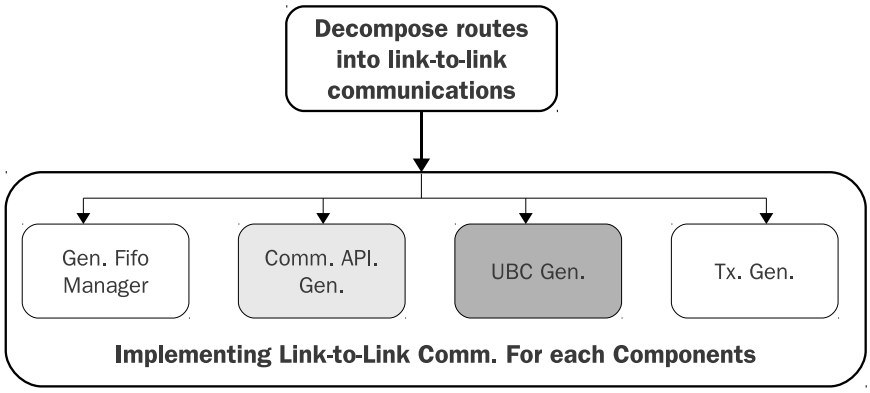


Figure 26: The Tasks for Generating TLM to Implement FIFO Channels with "READER/WRITER PE" implementation

on the top of UBC functions, followed by communication delays. RTOS model is not engaged in this case.

5.4 Communication Estimation

Delays due to communication delay are inserted in the form of wait function. For example, send/recv functions for P2P channels access bus to assert the bus address. The access is done by the UBC write function and followed by wait(communication delay).

In this section, estimation of communication delay (not annotation) will be explained.

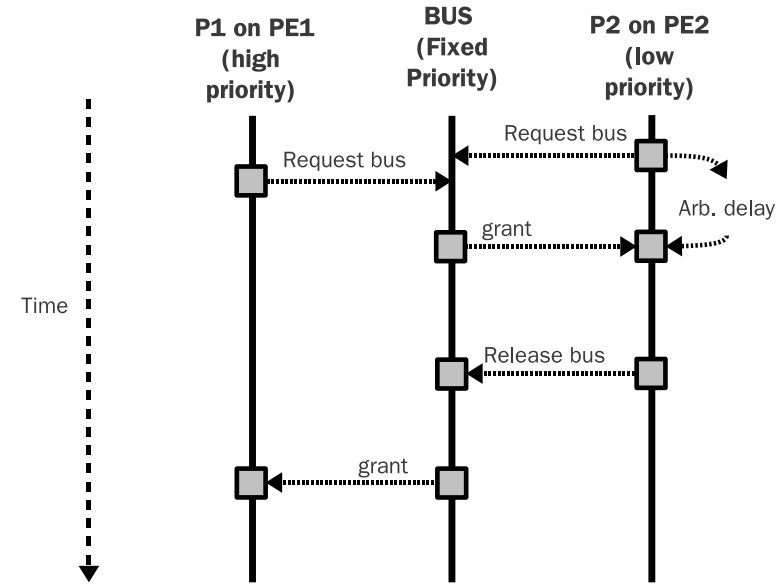
A transaction delay for any communication between two processes or between a process and a memory is obtained depending on Equation 1, where

- T_{arb} is arbitration delay
- T_{sync} is synchronization delay
- T_{dt} is data transfer delay

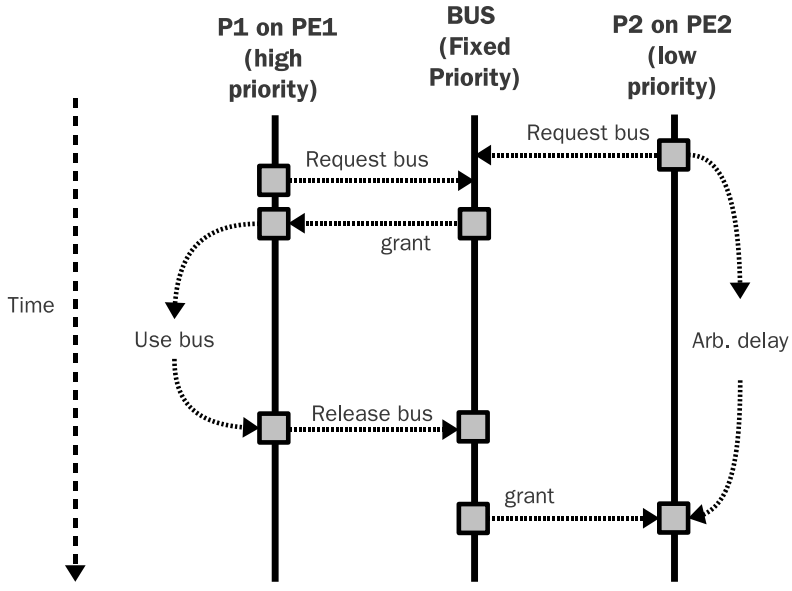
$$T_{total} = T_{arb} + T_{sync} + T_{dt} \quad (1)$$

A very simple example of T_{arb} is found in Figure 27. In Figure 27(a), the arbitration policy is FCFS, while it is "Fixed Priority" in Figure 27(b). PE1 and PE2 as well as some other invisible PEs are connected the bus. P2 on PE2 comes slightly earlier than P1 on PE1 but has lower priority. So, in the first figure, P2 is served first while in the second figure P1 is granted first. The delay from sending request to being granted is different. This kind of delay is T_{arb} and mainly depends on the order of getting the bus. UBC modules in Timed TLM is generated to implement the specified arbitration policy. In addition to that, arbitration itself has its own overhead, which is also taken into consideration.

An example of T_{sync} is shown in Figure 25. At the beginning, P2 requests read but needs to wait until the "sync" point. This delay is T_{sync}. It should be computed during simulation.



(a) First Come First Serve



(b) Fixed Priority

Figure 27: Arbitration Delay According to The Arbitration Policy

T_{dt} is data transfer delay. It is added just after returning from UBC send/recv/read-/write functions. It is estimated as follows.

$$T_{dt} = CA \cdot S + \lceil \frac{S \cdot 8}{B_{width}} \rceil \cdot D_{bus} \tag{2}$$

$$T_{dt} = CA + \lceil \frac{S \cdot 8}{B_{width}} \rceil \cdot D_{bus} \tag{3}$$

where,

- CA is Control/Address phase length in cycles
- S is size of the transaction in bytes
- B_{width} is bus width of the bus in bits
- D_{bus} is bus transaction delay in cycles

If the bus is specified to support burst mode, then, control and address do not need to be sent again and again. Therefore the equation 3 is applied. Otherwise, control and address have to be sent whenever the unit data is transferred.

This formula is hard-coded in the wait function that comes just after UBC send/recv/read-/write functions. For example, as we can see 21, at the end of read/write communication API, data is transferred by using UBC functions. After returning the UBC functions, a wait function representing the formulas is inserted.

Static information such as CA, B_{width} and D_{bus} is provided by ESE DS or the platform database. For example, if we use an OPB bus, whose CA, B_{width} and $D_{bus}\}$ are 2, 16 and 1, without burst mode, the read function looks like Listing 2.

```
1 #define OPB_CA 2
  #define OPB_B_width 16
3 #define OPB_D_bus 1
  ...
5
  extern ''C'' read( int S ) // read size
7 {
    ...
9   ubc -> read( S );
    wait( OBP_CA * OPB_S +
11        OPB_S * 8 / OPB_B_width * OPB_D_bus );
    ...
13 }
```

Listing 2: Read Function with Communication Delay

6 Improvement

We cannot claim the the design practices found in the TLM generator are good enough. TLM generation is decomposed into communication API generation/Process and PE generation/UBC generation and Tx generation. The way TLM components are grouped looks reasonable.

However, a single function that is 4K generates the entire processes, PEs and communication API. The second longest function, which is up to 2K, generates the entire UBC for itself and so on.

In addition to the size, lots of codes are duplicated. For example, as explained, write/read functions for FIFO channels are very similar with send/recv functions for P2P channels if the FIFO buffers are mapped onto transducers. However, the current TLM generator looks like Listing 3.

The list shows generation of communication API as an example. API for P2p channels are generated first in the first block. The block sends b, c, d and e to the file. These are mainly tons of c++ version of fprintf. After the generation is done, FIFO channels implemented with transducers are generated. In this case, a, b1, c, d and e have to be sent to the same file. Lots of generated codes are common. c, d and e are present in both cases. However, in the generator code, all of them are duplicated again and again. The entire generation process goes with the same story.

```
1  // generate PE, process and comm. API
   generator::processPE()
3  {
     ...
5    /* start comm API Gen */
     {
7      // gen. p2p channels
       // tons of C++ version of fprintf
9      fout << ''b'';
       fout << ''c'';
11     fout << ''d'';
       fout << ''e'';
13     ...
     }
15
     {
17     // gen. FIFO channels, the FIFO buffer is in a Tx
       // many duplication
19     fout << ''a'';    // add
       fout << ''b1'';  // slightly different
21     fout << ''c'';
       fout << ''d'';
23     fout << ''e'';

25     ...
     }
27
   }
```

Listing 3: Communication API Generation in processPE()

Finally, the current ESE DS is not so reasonable. Everything is messed up in it. We do not have any separated data structures for behavior specification such as in Figure 6, platform as exemplified in Figure 7 and their mapping. Instead, the data structure is unified as we see in Figure 28, which makes design space exploration harder. For example, when a user wants to delete CPU0(the gray box) and map process P1, P2 onto another PE, even though the user does not want to remove the processes, with the current unified data structure, he or she cannot avoid deleting them along with CPU0. So, at least, the data structure should be split into three parts; specification, platform and mapping.

In conclusion, It looks to us that modularization of the generator class and generate function is necessary. Common codes should not be duplicated but shared by function calls. The ESE DS needs to be improved by being split into at least three

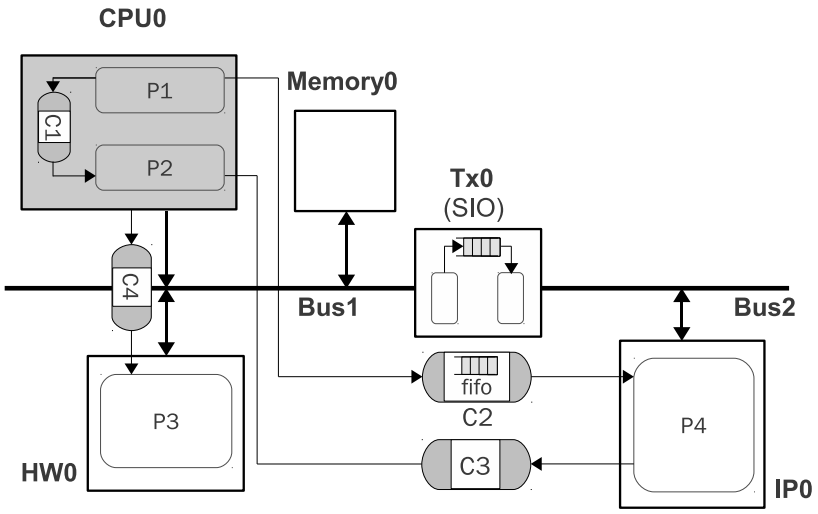


Figure 28: Deleting a PE, CPU0

entities, specification, platform and mapping. In addition to that, in the long run, the specification model, which does not fit the design philosophy of ESE, needs to be improved. It has been claimed that we can flattening any MoC to a set of communicating processes, but it is not a small deal.

7 Conclusions

ESE promises large productivity gain due to its solid concept. In Platform methodology, the design starts with a platform instance. Although it has drawbacks, it still reduces engineering cost and design cycle because components are already well verified. Moreover, if hardware/software co-design with virtual prototyping goes with Platform methodology, it clearly shortens design cycle and simplifies complexity further. However, the models are simulation oriented, automatic synthesis is not easy and a platform instance is usually less flexible.

Since the design methodology underlying ESE also utilizes platform and TLM, it shares the strong points of Platform methodology. Moreover, it overcomes the drawbacks of Platform methodology. Since the starting point of a design is specification, which is flexible in nature, ESE can be thought as an application oriented EDA. With well-define models, rules, transformations and refinements, the users of ESE can benefit from automatic synthesis, which shortens design cycle and makes design management easier.

However, still, ESE needs improvement. The generator is not manageable due to the lack of modularization and bad data structure. Data structures for specification, platform and mapping, which is now messed up in a single XML tree, should be clearly separated to be manipulated more easily. In the long run, The specification model does not seem to match the design philosophy of ESE and should be refined.

Moores law has been being proved to be true. Hundreds of cores can be integrated into a single chip. User demand is dramatically rising while time-to-market constrains is becoming stricter and engineering costs is not so low.

Acknowledgements

The work in this report would not have been possible without the help of professor Daniel. D. Gajski, who offered invaluable assistance, support and guidance. The author would also like to convey thanks to Y. Hwang, without whose knowledge and help, the work would not have been sucessful.

References

[1] S. Abdi and D. Gajski. A universal bus channel for transaction level modeling. 2006.

[2] D. D. Gajski. Icccas presentation. 2009.

[3] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis, Verification*. Springer, 2009.

[4] Y. Hwang, S. Abdi, and D. D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. *DATE*, 2008.

[5] L. Yu. Automatic generation and verification of transaction level modeling. 2009.