Protocol Transducer Synthesis using Divide and Conquer approach

Shota Watanabe
Dept. of Electronic Engineering
University of Tokyo

Kenshu Seto
VLSI Design and Education Center
University of Tokyo

Yuji Ishikawa
Dept. of Electronic Engineering
University of Tokyo

Satoshi Komatsu
VLSI Design and Education Center
University of Tokyo

Masahiro Fujita
VLSI Design and Education Center
University of Tokyo

e-mail : {shota, seto, yuji, komatsu}@cad.t.u-tokyo.ac.jp, fujita@ee.t.u-tokyo.ac.jp

Abstract— One of the efficient design methodologies for large scale System on a Chip (SoC) is IP-based design. In this methodology, a system is considered as a set of components and interconnects among them. The designers try to reuse existing IPs as much as possible. Communications among components have to be conducted using a common protocol, however, IPs available today use various communication protocols. Thus the protocol conversion is one of the most important topics in IP-based design[5]. In this paper, we propose a method for automatic protocol transducer synthesis which is applicable to complex protocols. The main idea of our proposed method is division of the exploration space into smaller ones for avoiding explosion of the exploration space, namely with a divide and conquer approach. We demonstrate our method by synthesizing transducers which translate between the real and complicated protocols with advanced features such as non-blocking transactions and out-of-order transactions.

I. Introduction

With rising complexity of the circuits on a single chip, IP-based design methodology is attracting attention to shorten the design periods. By reusing existing designs for a new design, the design period is expected to be much shortened. In the IP-based design, the most significant issue is the connectabilities among IPs. Since IPs usually have their own interfaces, IPs cannot communicate with one another if they use different protocols. In other words, an IP's interface limits IPs which can communicate with it.

In actual designs, designers usually insert protocol transducers (also called wrappers or bridges) between IPs with incompatible protocols, which consume additional time to design. As a result, the advantage of IP-based design is reduced. To resolve the problem, automatic synthesis of protocol transducers is an attractive solution [5]. However, the state-of-the-art protocols have various complicated functionalities such as non-blocking (pipelined) and out-of-order transactions[1, 2]. Unfortunately, the methods proposed so far are hard to deal with such advanced features in complicated protocols[4, 6], because the input protocol specifications of the previous methods are not good at describing these features.

In this work, we propose a protocol transducer synthesis method which is applicable to those complicated protocols. The basic idea of our method is divide-and-conquer approach. The main reason why the existing methods are difficult to deal with the complicated protocols is the explosion of the exploration space due to the complicated protocol specifications. Since an automaton is often adopted as a formal description of a protocol, modeling a protocol supporting advanced features results in quite large size of the specification automaton. Trying to synthesize the transducer from these complicated protocol specifications with the previous methods fails because of the large exploration space. Therefore, we divide the protocol specifications into smaller ones called $Sequence$s. A $Sequence$ corresponds to one operation such as Single Read, Burst Write, etc., and it consists of a set of automata. We apply $Sequence$ level transducer synthesis to one pair of $Sequence$s, and we call the synthesis result as a partial transducer. Then, we construct the whole transducer from a set of partial transducers. The synthesized transducer consists of several FSMs so that it can handle parallel transactions. With this approach, the protocol transducer for complicated protocols such as those supporting non-blocking and out-of-order can be synthesized, and we demonstrate it through experiments.

The rest of this paper is organized as follows: After discussing related work in Section II, we briefly explain the outline of our method in Section III. The detail of each process of the method are explained in Section IV,V,VI, and VII. Then we demonstrate our method by experiments in Section VIII. The last section gives concluding remarks.

II. Related Work

Roughly speaking, the approaches for protocol transducer synthesis are classified into two types: one is library-based method[9, 10, 11], and the other is protocol specification based method[4, 6, 7]. The library based method uses a library which has transducer designs for various protocols. The target transducer is obtained by combining some of the designs in the library with small modification. However, the usefulness of the synthesizer which uses the library based approach depends on the richness of the library. On the other hand, the specification based method takes two formal protocol specifications as inputs and automatically synthesize a transducer as output. The protocol specifications are written in formal descriptions such as automata descriptions[6], regular expressions which are equivalent to automata[4] or sequence charts[7].

In [4], the two input protocol specifications described in the form of regular expressions are translated into two equivalent automata. The synthesizer judges whether each state of the product automaton of them is legal or not in terms of data dependency. The output transducer is an FSM which is the subset of the product automaton that consists of legal states. We employ this method as a part of the proposed method with several extensions. The details of the extensions are explained in SectionVI-B.

In [6], D'silva $et\ al.$ proposed a formal approach that is a product automaton based synthesis method. This method can handle multiple clock speeds by inserting redundant states to the automaton which runs at higher clock speed. Their approach can handle non-blocking transactions only in the limited case. It cannot accept new requests when response data corresponding to the previous request is not ready because of slave's long latency, although it can accept a new request simultaneously with the previous response.

In the above two approaches, there are following limitations because a protocol is described in a single automaton, while our approach represents a protocol in a set of automata. Since input automata grow into large size in case the target protocols are complicated, the exploration space often becomes too large to handle. Additionally, this representation makes the description of parallelism (interleaving) complex, therefore it becomes difficult to handle the advanced features in the state-of-the-art protocols.

In [7], Roychoudhury $et\ al.$ used High-level Message Sequence Charts (HMSC) as the formal description of a protocol. They proposed a transducer synthesis method that takes HMSCs as input. An HMSC is a scenario based specification similar to the abstracted automaton which has a "sequence chart" as a node of the automaton. This method represents a protocol in a single HSMC so that it has the same limitations as we pointed out above.

To resolve the limitations of the previous works, we propose a specification based method that takes protocol specifications in the hierarchical automata form, and outputs the RTL description of the transducer. The proposed method can deal with complicated protocols with non-blocking and out-of-order transactions.

III. Outline of Proposed Approach

A. Protocol Classification

In this section, we classify protocols into three types and explain the classification rules. We consider that a protocol consists of requests and responses. A module which issues requests is called master, and one which issues responses is called slave. We classify protocols into the following types by the issue timing and the order of requests/responses.

1. **Blocking Protocol**
   The blocking protocol is a protocol in which the master cannot issue the next request until the slave returns the response for the previous request. AMBA AHB protocol[3] is a blocking protocol. Hereafter, "BK" stands for "Blocking".
2. **Non-Blocking Protocol**
   The non-blocking protocol is a protocol in which the master can issue the next request before the slave returns the response. The slave of a non-blocking protocol has to return responses in the same order as the corresponding requests' order. Open Core Protocol(OCP)[1] is a non-blocking when it is configured to have only basic signals. Hereafter, "NB" stands for "Non-Blocking".
3. **Out-of-Order Protocol**
   The out-of-order protocol is a protocol in which the master can issue the next request before the slave returns the response, and the slave can respond in different order from the requests'. The master detects correspondences between requests and responses by comparing tags buried in requests and responses. AMBA AXI[2] and OCP with tag extension are out-of-order protocols. In the following sections, we call OCP with tag extension as "Tagged OCP". Hereafter, "OoO" stands for "Out-of-Order".

B. Basic Idea

The novel idea of our method is employing divide-and-conquer approach to the protocol transducer synthesis for complicated protocols. The main reason why the existing methods are difficult to deal with complicated protocols is the explosion of the exploration space caused by the complicated protocol specifications. As a single automaton is often adopted as a formal description of a protocol, modeling a protocol supporting advanced feature can result in a quite large size of the specification automaton. So, we divide the protocol specifications into smaller ones called $Sequence$s (divide process). A $Sequence$ corresponds to one operation in the protocol such as Single Read, Burst Write, etc, and it consists of several automata. We apply $Sequence$ level transducer synthesis to one pair of $Sequence$s, and we call the synthesis result as a partial transducer (conquer process). Then, we construct the whole transducer from a set of partial transducers (combine process).

The $Sequence$ level synthesis employs automaton level synthesis. As the automaton level synthesizer, we employed an existing work which is done by Passerone $et\ al.$[4]. However, Passerone's synthesizer needs several extensions to be used in our work because it cannot deal with loops in the specification automata. The details of the extensions are explained in Section VI-B. We call the result of an automaton level synthesis as an element transducer, which is an FSM.

Figure 1 shows the outline of our approach. With this approach, the protocol transducer for complicated protocols can be synthesized.

For the ease of explanation, we define two terms: "starting condition" and "return to initial edge". "starting condition" is a state transition condition which is associated to the edge from the initial state of an FSM to other states. "return to initial edge" is a state transition whose destination is the initial state of an FSM. Also, we use descriptions such as ($protocol\ type1$, $protocol\ type2$) when the master uses a protocol whose type is $protocol\ type1$, and the slave uses $protocol\ type2$.

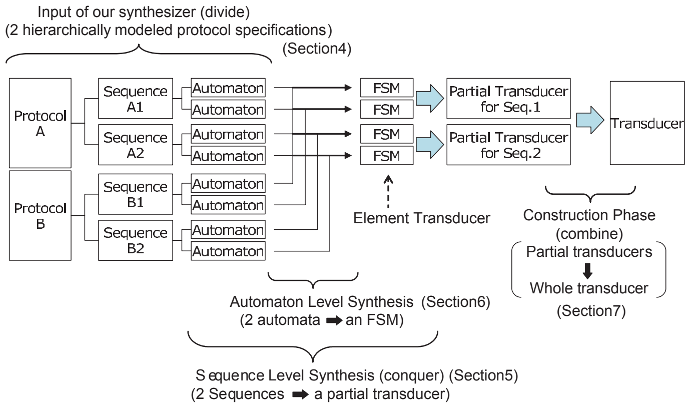

Input of our synthesizer (divide)
(2 hierarchically modeled protocol specifications)
(Section4)
Protocol A
Protocol B
Sequence A1
Sequence A2
Sequence B1
Sequence B2
Automaton
FSM
Partial Transducer for Seq.1
Partial Transducer for Seq.2
Transducer
Element Transducer
Construction Phase (combine)
Partial transducers
Whole transducer
(Section7)
Automaton Level Synthesis (Section6)
(2 automata ➡ an FSM)
Sequence Level Synthesis (conquer) (Section5)
(2 Sequences ➡ a partial transducer)

Fig. 1. Outline of proposed protocol transducer synthesis

IV. Proposed Protocol Modeling

We propose a protocol specification model shown in Figure 2 to overcome the explosion of the solution space. The novelties of the proposed protocol model are the following:

- Consider a protocol as a set of $Sequence$s.
- A $Sequence$ consists of either a single automaton or two automata.

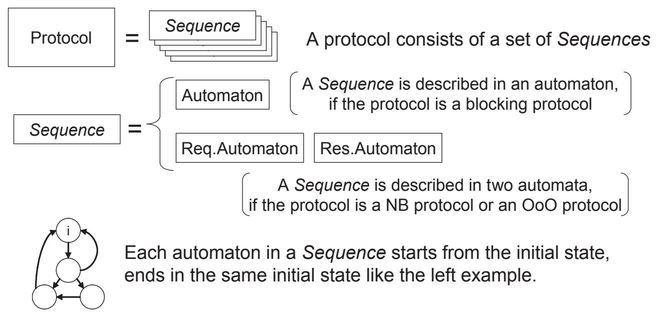

Protocol
=
Sequence
A protocol consists of a set of Sequences
Sequence
=
Automaton
A Sequence is described in an automaton, if the protocol is a blocking protocol
Req.Automaton
Res.Automaton
A Sequence is described in two automata, if the protocol is a NB protocol or an OoO protocol
i
Each automaton in a Sequence starts from the initial state, ends in the same initial state like the left example.

Fig. 2. Proposed Protocol Modeling Method

- For blocking protocols, a *Sequence* has an automaton which accepts both request procedure and response procedure.
- For NB or OoO protocols, a *Sequence* has two automata: a request automaton, and a response automaton. Each handles either requests or responses.
- Each automaton in a *Sequence* starts from and ends in the same initial state.

Figure 4 shows an example of this modeling method which describes essential parts of OCP[1]. In this protocol model, each *Sequence* corresponds to a "Timing Chart" which appears in protocol specifications written in natural languages. So, we believe this modeling scheme is a natural translation of specification into a formal description.

Using this model has the following advantages. First, we can apply transducer synthesis for each *Sequence* independently. The independent application of transducer synthesis to each *Sequence* reduces the total amount of computation. Also by modeling requests and responses independently, non-blocking and out-of-order transactions can be easily handled. Second, modeling of a whole protocol is facilitated. By describing in a unit of a *Sequence*, number of states in each automaton becomes relatively small. Hence unintentional errors in the protocol modeling process can be suppressed. Moreover, in [12], Y.Kakiuchi *et al.* use divided protocol model that is similar to ours for formal verification of protocol itself. So, the protocol specification can be verified by applying their verification method before the transducer synthesis.

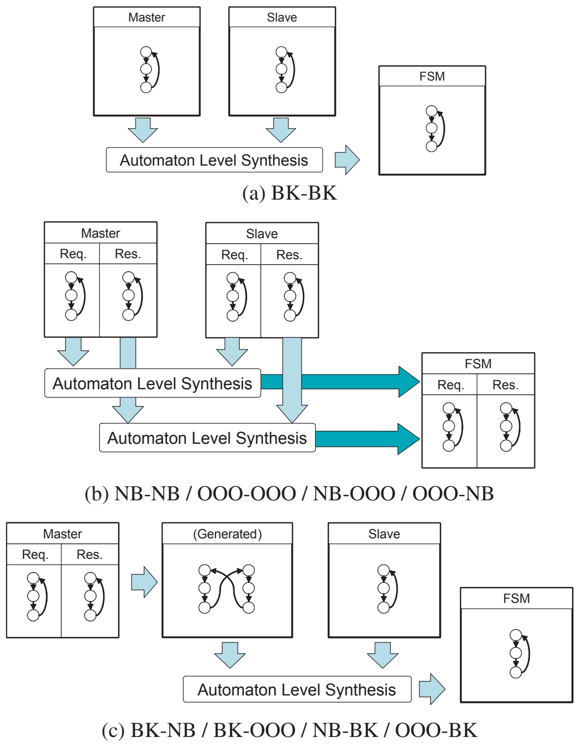
V. Sequence Level Synthesis

In this section, we explain the method to synthesize a partial transducer from a pair of *Sequence*s (one from each protocol). First, we have to select a target *Sequence* from each protocol. The selected *Sequence*s have the same semantics, such as single read (get the value stored in the specified address). Although this selection is not automated and it has to be done by users, it should not be so difficult in general.

Each of the selected *Sequence* has one or two automata depending on the protocol's type. We apply automaton level synthesis to the automata in the target *Sequence* pair considering the types of the protocols. As the automaton level synthesizer, we employ the algorithm proposed by Passerone *et al.*[4]. We modified their algorithm to allow loops in the input automata. The outline of their algorithm and the details of the extensions are explained in Section VI.

The output of a *Sequence* level synthesis is a partial transducer. A partial transducer consists of one or two FSMs, and it is used to construct the whole transducer. The construction method is explained in Section VII.

As explained in Section IV, *Sequence*s from NB or OoO protocols have two automata, and *Sequence*s from blocking protocols have an automaton. So, we have to apply *Sequence* level synthesis in the following way for each case :
•**Both are BK protocols(see Figure 3-(a)):** Simply apply automaton level synthesis to the two automata. Then we have an FSM as the partial transducer.
•**Both are NB or OoO protocols(see Figure 3-(b)):** Apply automaton level synthesis to the two request automata, and apply again to the two response automata. Then we have two FSMs as the partial transducer.
•**One is blocking, the other is NB or OoO(see Figure 3-(c)):** Make an automaton by connecting the request automaton and the response automaton in the *Sequence* which belongs to the NB or an OoO protocol . Then, apply automaton level synthesis to the generated automaton with the automaton in the blocking protocol's *Sequence*. The connection of the request automaton and the response automaton is simply done by changing the destinations of the return to initial edges to the other's initial states.


Master
Slave
FSM
Automaton Level Synthesis
(a) BK-BK
Master
Req.
Res.
Slave
Req.
Res.
Automaton Level Synthesis
Automaton Level Synthesis
FSM
Req.
Res.
(b) NB-NB / OOO-OOO / NB-OOO / OOO-NB
Master
Req.
Res.
(Generated)
Slave
FSM
Automaton Level Synthesis
(c) BK-NB / BK-OOO / NB-BK / OOO-BK

note: [BK]:blocking protocol, [NB]:Non-blocking protocol, [OOO]: Out-of-Order protocol

Fig. 3. *Sequence* Level Synthesis

VI. Automaton Level Synthesis

A. Passerone's Method

Passerone *et al.* proposed a protocol transducer synthesis method which synthesizes a transducer FSM from two protocol specifications in [4]. Figure 6 shows the outline of Passerone's method.

According to their algorithm, the synthesizer makes a product graph of the two protocol specification. Each state in the graph corresponds to the pair of states from each automaton and the synthesizer judges if each state is consistent or not. We say "a state in the product graph is inconsistent" when the state causes to output un-received data. The output FSM is generated by selecting proper states from the consistent states. The

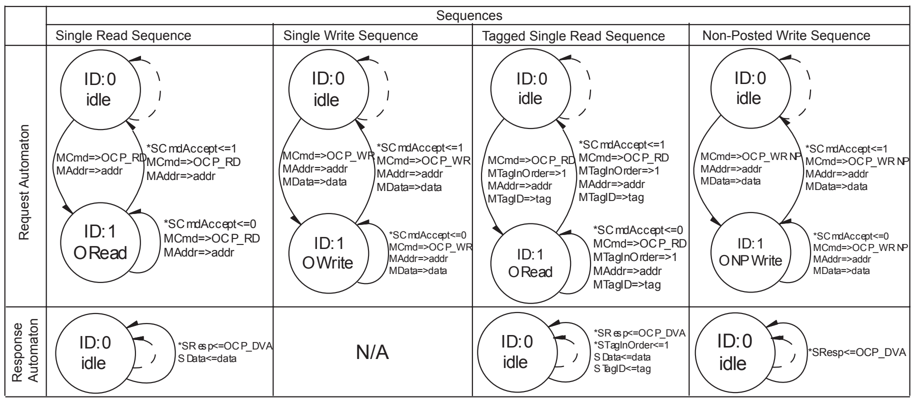


Fig. 4. Example protocol model: OCP

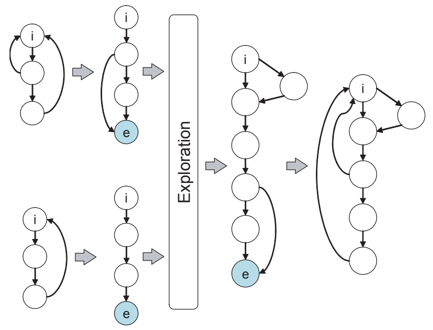


Fig. 5. Handling of loops

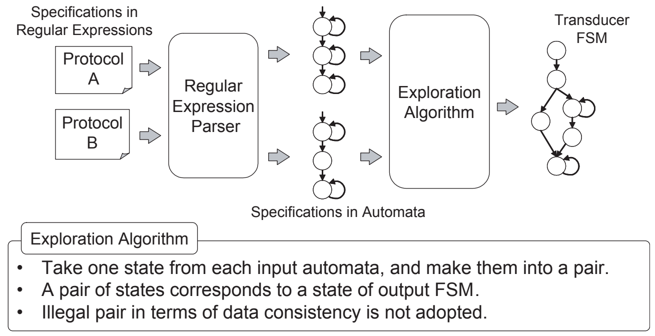

Specifications in Regular Expressions
Protocol A
Protocol B
Regular Expression Parser
Exploration Algorithm
Transducer FSM
Specifications in Automata
Exploration Algorithm

Fig. 6. Outline of Passerone's method

novel aspect of their method is using "Equivalence Classes" to minimize the latency of the transducer. The equivalence class is a subset of available transitions which have the same input. By choosing the best transitions from each equivalence class, an output FSM can have the least latency. Since we employ their method as an automaton level synthesizer, our method can also synthesize transducers with the least latency.

B. Extension to Passerone's Method

B.1 Handling of Loops

To use Passerone's method as an automaton level synthesizer, we modified Passerone's method so that it can deal with the cases in which the specification automata include loops. In this section, we explain the details of the extension. The point of the extension is adding an "end state" which acts as a substitute for the initial state. The outline of this scheme is shown in Figure 5. Before exploration, we add an "end state" to each input automaton, and change the destinations of the return to initial edges to the end state. As a result, the input automata become to have no loop edges except for immediate loops(loops to the same state). Thereby, the input automata are acceptable by Passerone's method. After application of Passerone's method, the output FSM has a state which corresponds to the pair of end states (we call this state end state of the output FSM). Finally, we change the destinations of all the incoming edges into the end state of the FSM to its initial state.

B.2 Multiple Data Sequences

In a complex protocol, some $Sequence$s have arbitrary number of data transactions (e.g. Imprecise Burst of OCP[1], Undefined-Length Burst of AMBA AHB[3]). Automata in this kind of $Sequence$ have loops which are not immediate loops nor return to initial loops. We call this kind of $Sequence$s Multiple Data $Sequence$s.

In order to deal with multiple data $Sequence$, we introduce "Super State". A super state is a state which behaves like a normal state, but contains a state-graph in itself. Introduction of super states helps us to apply transducer synthesis to Multiple Data $Sequence$s.

It is applied in the following way:

1. Extract a transaction unit from the multiple data $Sequence$. (Kernel Part Graph)
2. Replace the extracted states with a super state. (Shell Graph)
3. Synthesize two FSMs, one from the Kernel Part Graphs and the other from the Shell Graphs.
4. Combine them into one, to make the solution transducer.

The outline of this scheme is shown in Figure 7. Figure 7-(a) shows extraction of Kernel Part Graph from an automaton of a multiple data sequence. Figure 7-(b) shows transducer synthesis flow. This extraction of a transaction from a multiple data $Sequence$ is also applicable to nested case. Additionally, by regarding a single data sequence as a superstate, we can divide a multiple data transaction into a series of single data transactions, namely we can translate from a burst $Sequence$ to a set of single $Sequence$s.

VII. Construction of the Whole Transducer

After a set of $Sequence$ level syntheses, we have several partial transducers. In each partial transducer, there can be one FSM or two FSMs depending on the combination of the protocol types. We construct the whole transducer from these partial transducers. The construction process consists of two phases. The first phase is uniting FSMs, and the second one is the insertion of a buffer. The details of these processes are explained in the following sections. Figure 8 shows the outline of the construction process.

A. Uniting FSMs

In case that at least one of the master and the slave uses blocking protocol, each partial transducer consists of an FSM.

Exploration Algorithm
- Take one state from each input automata, and make them into a pair.
- A pair of states corresponds to a state of output FSM.
- Illegal pair in terms of data consistency is not adopted.

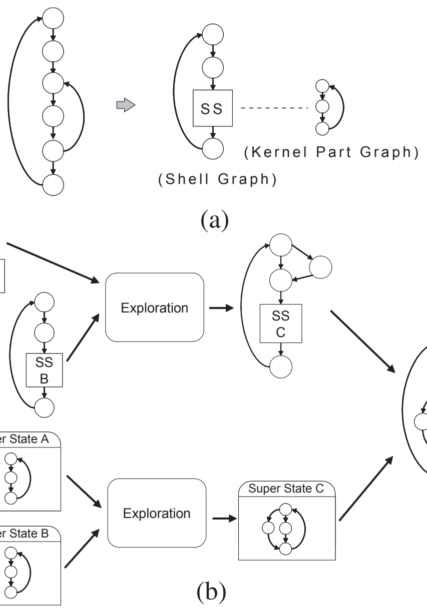


Fig. 7. Synthesis for Multiple Data Sequence

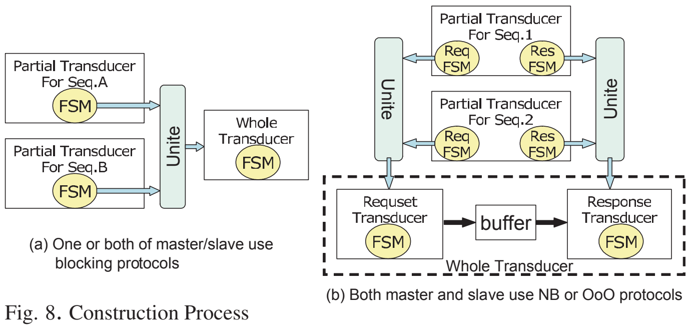

Partial Transducer For Seq.A
FSM
Partial Transducer For Seq.B
FSM
Unite
Whole Transducer
FSM
(a) One or both of master/slave use blocking protocols
Partial Transducer For Seq.1
Req FSM
Res FSM
Partial Transducer For Seq.2
Req FSM
Res FSM
Unite
Unite
Requset Transducer
FSM
buffer
Response Transducer
FSM
Whole Transducer
(b) Both master and slave use NB or OoO protocols

Fig. 8. Construction Process

In this case, we construct the whole transducer by uniting every FSM in the partial transducers, and skip the buffer insertion as shown in Figure 8-(a). Otherwise, every partial transducer consists of two FSMs: a request FSM and a response FSM. In this case, we generate two FSMs, one by uniting every request FSM and the other by uniting every response FSM. We call the generated FSMs as "request transducer" and "response transducer" as shown in Figure 8-(b).

We unite FSMs simply by co-owning the initial states. To unite FSM_A and FSM_B, we remove the initial state of FSM_B and modify the transitions from/to the initial state of FSM_B into transitions from/to the initial state of FSM_A.

B. Insertion of a buffer

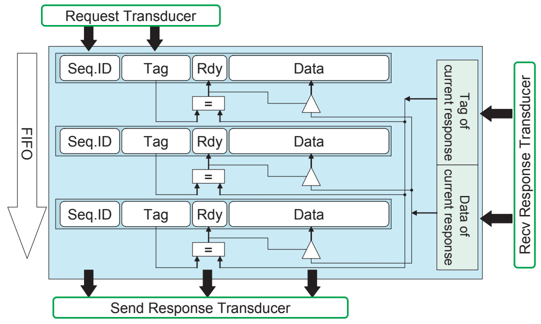
If both the master protocol and the slave protocol are not blocking protocols, we have to insert a buffer between the request transducer and the response transducer. In case (NB,NB), the responses from the slave keep the order of the requests. So, the inserted buffer should be a FIFO. Each entry of the FIFO buffer stores "$Sequence$ ID"'s. A $Sequence$ ID is a unique number among all partial transducers. We modify the request/response transducers to allow accesses to the inserted buffer. The modifications are done in the following way:

1. Add "FIFO-push" to the all the return to initial edges in the request transducer.
2. Change all the starting conditions of the response transducer into "original condition $\wedge$ the $Sequence$ ID in the front entry(the oldest entry) of FIFO equals to the $Sequence$ ID of the $Sequence$"
3. Add "FIFO-pop" to the all the return to initial edges in the response transducer.

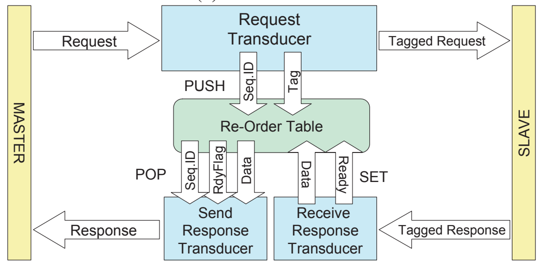
This process is also usable in case (OoO,NB), since the master can accept request-ordered responses. In this case each entry of the FIFO becomes [$Sequence$ ID, tag].

(NB,OoO) is the most difficult case. The order of the responses is not equal to the request order, although the master expects responses in the request order. In this case, we have to insert a "re-order table" and modify the response transducer. Figure 9 shows the architecture of the re-order table. The re-order table is basically a FIFO buffer, however each entry of the buffer has a "ready flag(Rdy)". The response transducer can pop this FIFO buffer only when the ready flag of the front entry is set. To allow transducer to access this re-order table, we generated a "receive response transducer" and a "send response transducer" from the response transducer (Figure 9-(b)). The receive response transducer is generated by removing all I/Os with the master from the response transducer and adding "ready flag set" to the return to initial edges. The send response transducer is generate by removing I/Os with the slave from the response transducer and adding "ready flag of the front entry is set $\wedge$ the $Sequence$ ID in the front entry of FIFO equals to the $Sequence$ ID of the $Sequence$" to the start conditions.

In case (OoO,OoO), we need no buffer as long as we are using the same tag.



(a) re-order buffer



(b) (NB,OoO) transducer

Fig. 9. Architecture of (NB,OoO) transducer

VIII. Experimental Results

We implemented a transducer synthesis tool based on our method and carried out several experiments. The inputs of our tool are XML descriptions that describe protocol specifications in the proposed hierarchical model, and the output is RTL descriptions of the whole transducer. We synthesized the transducers under several conditions, and checked if they work correctly by RTL simulations. We used ModelSim XE III as the RTL simulator. Table I shows the experimental results. The column "$Seqs$" shows the number of total $Sequence$s which we modeled for the each input protocol, and the column "Time" shows the consumed time for each transducer synthesis. The number in the column “Gates” are gate count of the transducer after logic synthesis by Xilinx ISE v.8.1. The experiments were carried out on a Windows XP computer with 2GHz Athlon 64 processor, and 1GB of memory. In the following sections, we show the simulation waveforms for the notable two results.

TABLE I
EXPERIMENTAL RESULTS

Master	Slave	Type	Seqs	Time	Gates
OCP	AHB	(NB,BK)	4	1.1[s]	2,352
AHB	OCP	(BK,NB)	4	1.3[s]	1,843
OCP	OCP	(NB,NB)	2	1.9[s]	1,568
OCP	TaggedOCP	(NB,OoO)	2	2.2[s]	3,514
TaggedOCP	AXI	(OoO,OoO)	2	4.8[s]	1,377
AXI	OCP	(OoO,NB)	2	4.9[s]	1,731

A. NonBlocking⇔NonBlocking

We synthesized a transducer which translates a non-blocking protocol into another non-blocking protocol. As the target protocols, we used two differently configured OCP[1]. The master uses single read and non-posted write $Sequence$s, and the slave uses single read, single write $Sequence$s. Non-posted write is a single write operation which requires the slave to return response. Because the master requires a response for write, although the slave does not return any response, the transducer has to return a response to a non-posted write request on behalf of the slave.

The waveforms of the simulation are shown in Figure 10. The requests and the responses are overlapped and translated correctly. Also we can see the transducer inserts the responses to the non-posted write requests in the correct order.

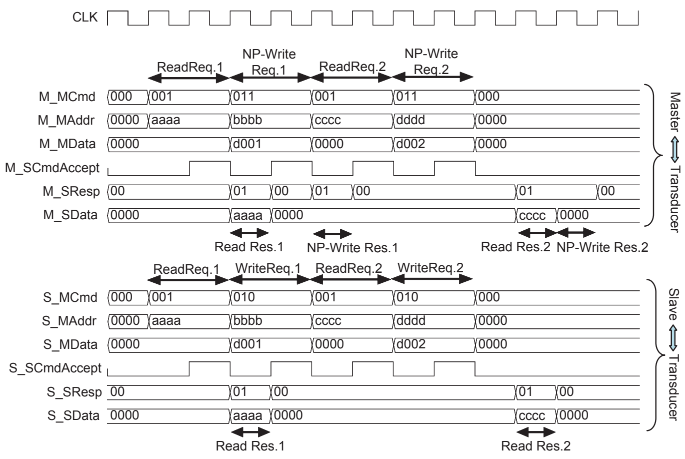


Fig. 10. Wave forms of OCP(RD, NPWR) - OCP(RD, WR) Transducer

B. NonBlocking⇔Out-of-Order

As another example, we show a transducer which translates a non-blocking protocol into an out-of-order protocol. The master uses OCP with only basic signals, and the slave uses Tagged OCP. Figure 11 shows the wave forms for the synthesized transducer. Two read requests are issued by the master and the transducer translates the requests into two tagged-read requests. The slave returns tagged-responses in the opposite order as the order of requests. The responses for the master are re-ordered by the transducer, and the master receives the responses in the same order as it requested.

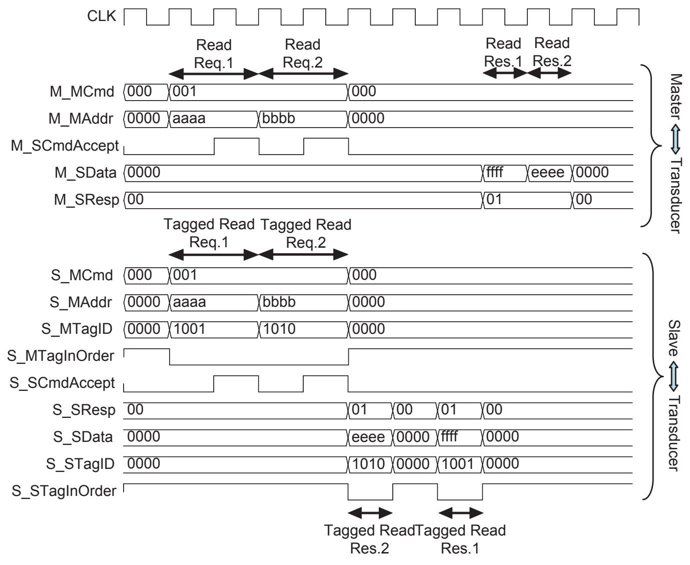


Fig. 11. Wave forms of OCP-TaggedOCP Transducer

IX. CONCLUSION

In this paper, we proposed a protocol transducer synthesis method which used divide-and-conquer approach. We used several automata in the hierarchical structure as a specification of a protocol to describe complicated protocol concisely. We applied $Sequence$ level synthesis which employs automata level syntheses to synthesize a partial transducer. Then we constructed the whole transducer from the partial transducers. In this approach we could synthesize protocol transducers which can handle non-blocking and out-of-order transactions, and we demonstrated it in the experiments.

REFERENCES

[1] Open Core Protocol Specification version 2.1

[2] AMBA 3 AXI Specification v.1.0

[3] AMBA 2 AHB Specification Rev.2.0

[4] R.Passerone, J.A.Rowson, A.Sangoivannni-Vincentelli, “Automatic Synthesis of Interfaces between Incompatible Protocols”, Proc. of the 35th. Design Automation Conference, pp.8-13, 1998

[5] Daniel D. Gajski, Allen C.-H. Wu, Viraphol Chaiyakul, Shojiro Mori, Tom Nukiyama, Pierre Bricaud, “Essential Issues for IP Reuse”, Proc. of ASP-DAC, pp.43-48, 2000

[6] Vijay D’silva, S. Ramesh, Arcot Sowmya, ”Bridge Over Troubled Wrappers : Automated Interface Synthesis”, 17th International Conference on VLSI Design, p.189, 2004

[7] Abhik Roychoudhury, P.S.Thiagarajan, Tuan-Anh Tran, Vera A. Zvereva, ”Automatic Generation of Protocol Converters from Scenario-Based Specifications”, RTSS’04, pp.447-458

[8] J. Smith and G. De Micheli, ”Automated Composition of Hardware Components”, Proc. of Design Automation Conference, 1998.

[9] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya, ”Automatic generation of embedded memory wrapper for multiprocessor SoC.”, Proc. of the 39th Design Automation Conference, June 2002.

[10] Drew Wingard. ”MicroNetwork-based integration of SOCs.”, Proc. of the 38th Design Automation Conference, June 2001.

[11] A.Grasset, F.Rousseau, A.A.Jerraya, “Automatic Generation of Component Wrappers by Composition of Hardware Library Elements Starting from Communication Service Specification”, The 16th IEEE International Workshop of Rapid System Prototyping, pp.47-53, 2005

[12] Y.Kakiuchi, A.Kitajima, K.Hamaguchi, T.Kashiwabara “Behavioral Model Construction for Formal Verification of Advanced On-Chip Bus Protocol”, The Workshop on Synthesis And System Integration of Mixed Information technologies, pp.282-289, 2004