Language-Based High Level Transaction Extraction on On-chip Buses[*]

Yi-Le Huang, Chun-Yao Wang, Richard Yeh[†], Shih-Chieh Chang, Yung-Chih Chen

Dept. of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan, Republic of China
kiwe@nthucad.cs.nthu.edu.tw
{wcyao,scchang,dr948308}@cs.nthu.edu.tw

[†]SpringSoft, Inc.
Hsinchu, Taiwan, Republic of China
richard_yeh@springsoft.com.tw

Abstract— With the increasing in silicon densities, SoC designs are the stream in modern electronics systems. Accordingly, the verification for SoC designs is crucial. One of the main problems in SoC verification is to verify whether the interface of a block works properly in its intended system. Transaction-based verification methodologies have been proposed to deal with this problem, and they allow users creating tests and writing test benches more easily. Furthermore, verifying interface designs in transaction level is very efficient. Previous work creates extractor manually for one on-chip bus (OCB), and the extra efforts are needed for another OCBs. In this paper, we present a language-based methodology to specify the bus behaviors in transaction level. Then the actual signals on the buses can be extracted to a higher level of abstraction. The bus behaviors displayed in transaction level significantly reduce the verification efforts for verification engineers. Furthermore, the corresponding transaction extractors are automatically generated. We demonstrate the success of our approach on AMBA AHB and Sonics' OCP buses.

I. Introduction

Block-based and platform-based design methodologies are the stream in the era of SoC. A complex design is partitioned into several smaller blocks with well-defined functionality and the corresponding interface by these design methodologies. These smaller blocks are called Intellectual Properties (IPs). The IPs can be obtained from internal departments or licensed from 3rd party IP vendors. It is very convenient and time-saving for SoC designers to reuse these pre-designed and pre-verified IP cores. In the past, designers negotiated the interface among IPs face to face for integrating an SoC design successfully. With the growth of design complexity, the interfaces become more complex and the information negotiation cannot achieve the goal, even be infested with bugs. Therefore, standard and efficient interface protocols play crucial roles in the SoC design methodology. Many OCBs, e.g., AMBA[4], CoreConnect[8], WISHBONE[10], and OCP[11], are promoted to fully address communication issues among blocks. With buses, functional blocks can exchange data easily. Designers only have to design interfaces between their IPs and the buses, rather than the interfaces to other blocks in its intended system. In addition, since IP suppliers create IP cores based on a specific bus, reusing those IP cores to integrate an SoC design would be much easier under the identical bus architecture.

Nevertheless, new problem occurs due to the usage of OCBs. OCB providers always introduce the bus behaviors by the bus protocol specifications. However, the bus specifications are written in natural languages with some auxiliary waveforms almost. Thus, the specifications are informal and possibly ambiguous for some properties due to the intrinsic characteristic of natural languages. There would exist many misunderstandings and different interpretations among designers when reading the same bus specification. Consequently, inconsistencies between the interface designs may occur. However, if OCB providers release the bus specification in the form of formal language, the mentioned problem will be solved. Furthermore, the bus protocol verification will be achieved easily and automatically as well.

One of the major verification problems for IP vendors and users is whether the interface of an IP interacts properly with its intended bus system. Current practice for the problem is to create the bus monitors manually. Monitors are programs or circuits that are used to observe the interface signals in the simulation runs. Then the verification is conducted by comparing the simulation results and the interpreted bus behaviors. There are two possible drawbacks for this practice. First, the interpreted bus behaviors may be erroneous. But this drawback will be eliminated when the description of bus specification is in formal forms. Secondly, it is time-consuming and labor-intensive for observing the detailed interface signals to determine if the interface is protocol compliant. It is especially true for more complicated bus protocols. Thus, displaying high level transactions in the bus monitor will improve the readability and shorten the verification time. Besides, the generation of the monitor is a tedious work. The repetitive efforts for creating different monitors for different OCBs are inevitable.

In this paper, we demonstrate an approach that generates corresponding monitors for different OCBs automatically. The monitor extracts the interface behaviors from the detailed signals to the high level transactions, and displays transactions in nWave environment[9]. That is, the input of the monitor is the detailed interface signals on the bus; the output is the corresponding transactions. The monitor is called transaction extractor in this work. The transaction extractor is generated with respect to the formally de-

[*]This work was supported in part by ROC National Science Council under Grant NSC93-2220-E-007-012.

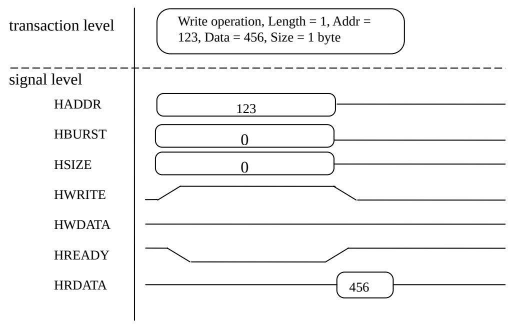


Fig. 1. Transaction level v.s signal level

scribed bus protocol. By examining extracted transactions, the level of verification is raised from signal level to transaction level. The formal bus protocol description in this work is obtained by a translation from official protocol specification document manually. Notice that this work does not emphasize the completeness and correctness of the translation of protocol specification from natural languages to formal languages, though we try our best for it. As long as the protocol is formally described, the proposed approach can be applied directly. We believe that the formal description of bus protocol without ambiguity is possible in the future.

The remainder of this paper is organized as follows. Section II presents the preliminary of this paper. Section III describes our approach for specifying the bus protocols formally. Section IV introduces that how to generate the transaction extractor based on corresponding formal protocol descriptions. Section V presents that how to enrich the extracted transactions with useful information for verification. We present experimental results in Section VI and conclusions in Section VII.

II. Preliminaries

To raise the verification level to a higher level of abstraction, the transaction-based methodology[5] has been proposed. The approach reduces the debugging and coverage analysis efforts by presenting verification information in transaction level. For example, the signal level information is shown in the lower half of Fig. 1. Its corresponding transaction level information is shown in the upper half of Fig. 1. It is obvious that transaction level information is much easier to read than signal level information. On the other hand, formal property languages such as OVL[2], CTL[7], or regular expressions, have been proposed to describe bus protocols formally. The properties extracted from bus protocols can be specified by them. Here we adopt regular expressions as the vehicle for describing bus protocols formally. This is because regular expressions are in excelling at describing behaviors over time. Furthermore, its readability is better than others. Fig. 2 and 3 show the examples of properties described by CTL and OVL respectively. They are not easy to be understood in general. Also, many public property specification languages are regular expression-based, e.g., Accellera's PSL[1]. Thus, these languages can be exploited directly with/without customized operations for achieving this goal. Besides, in nature the regular expressions are equivalent to FSMs (finite state machines). Since FSMs can be easily implemented by executable programming languages, this feature facilitates the automatic generation of transaction extractors.

Oliveira et al.[12] achieve automatic generation of IP interface monitors based on formal protocol descriptions. It provides the flexibility of creating monitors. But only signal level information is observed on the monitors. Ara et al.[3] propose a specification-based verification approach with CWL[6] that generates verification patterns, simulation checker, and coverage analyzer automatically. But the checker still only focuses on the signal level information, rather than transaction level information. Therefore, in this paper, we combine the advantages of presenting verification information in transaction level and automatic generation of monitors. The monitors which are used to observe the transaction level information are generated automatically.

$\pi \models \mathrm{f}U_I\mathrm{g}$ iff I-consistent, and $\exists$ (s_i , η_i) $\in$ π suchthat $s_i \models$ g, and for all (s_i , η_i) preceding (s_i , η_i) in π , $s_i \models$ f

Fig. 2. An example of a property described in CTL

```
assert_next #(1,2,0,0,0,"error: ERROR should be followed
by OKAY after 2 cycles") error_check3 (HCLK,HRESETn,
(HRESP==‘RSP_ERROR) && (HREADY==1'b0), (HRESP
==‘RSP_OKAY));
```

Fig. 3. An example of a property described in OVL

III. Protocol Description

In this section, we introduce in detail our regular expression-based descriptions for bus protocols. For convenience and user-friendliness, we quote some basic PSL syntax in our protocol descriptions. Examples taken from AMBA AHB will illustrate the concepts of our protocol descriptions.

There are two layers, symbol layer and sequence layer, in the protocol descriptions. In the symbol layer, symbols are used to describe behaviors or properties in one cycle. Some related interface signals are grouped into a specific symbol and therefore this symbol represents the combination of these signals. Besides, symbols can also be used to simplify the sequence layer.

In the symbol layer, users define some symbols to represent properties or characteristics in one cycle. Each symbol can be followed by Boolean expressions or arithmetic formulae consisting of defined symbols or reserved interface signals. It is similar to the variable declaration in C language and other programming languages. Users can declare Boolean-type symbols to represent internal one-cycle properties, and integer-type or string-type symbols to store information in a given cycle, as shown in Fig. 4. We explain some symbols that are defined in Fig. 4. In AMBA AHB, the control signals such as HSIZE, HPROT, and HBURST have to be stable during two cycles. Thus, *Control_stable* can be defined as shown in Fig. 4. All transfers within a burst must be aligned to the address boundary that is equal to the size of the transfer. We use the symbol *Address_align* to formally describe this property. In AMBA AHB, there are four kinds of address phases and we use symbols to describe the individual characteristics of them. They are shown as *NonseqAddress*, *SeqAddress*, *IdleAddress*, and *BusyAddress*

```
boolean Control_stable =((prev(HSIZE)==HSIZE) && (prev(HPROT)
==HPROT) && (prev(HBURST)==HBURST));
boolean Address_align =(((int)HADDR % (1<<HSIZE))==0)?1:0;
boolean NonseqAddress = (HRESETn && (HTRANS==2) &&
Address_align);
boolean SeqAddress = (HRESETn && (HTRANS==3) && Address_align
&& Control_stable);
boolean IdleAddress = (HRESETn && (HTRANS==0));
boolean BusyAddress = (HRESETn && (HTRANS==1) &&
Control_stable);
boolean ERROR_1 = (HRESETn && !HREADY && (HRESP==1));
boolean ERROR_2 = (HRESETn && HREADY && (HRESP==1));
int burst_counter = (HBURST == 0)?1:(1<<(HBURST/2+1));
```

Fig. 4. Examples of partial symbol layer of AMBA AHB

in Fig. 4. These symbols can contain some defined symbols as well. Furthermore, the defined symbols can be reused for describing multi-cycle behaviors concisely in the sequence layer and they are available to the sequences elsewhere.

The sequence layer is the kernel of our protocol description. Since a sequence in this layer usually contains some regular expressions, it is used to represent the behavior during several cycles, and it is adequate to describe transactions of bus protocols. A sequence consists of pre-declared sequences, symbols, and bus interface signals. The syntax of sequence declaration is similar to that in the temporal layer of PSL, as shown in Fig. 5. We assume that readers have been familiar with the regular expressions and Accellera's PSL, and skip the detail of them. Fig. 6 illustrates an example of the sequence declaration. The sequence *first_address* contains zero or many cycles that *NonseqAddress* is true and HREADY is deasserted and one cycle that *NonseqAddress* is true and HREADY is asserted.

```
Sequence_Declaration ::= sequence
Sequence_Name(Instantiated_Sequence_List) = Sequence
Instantiated_Sequence_List ::= sequence Name {,Name}
Sequence ::= {Regular Expressions}
```

Fig. 5. The syntax of sequence declaration

```
sequence first_address()=
{
(NonseqAddress && !HREADY )[*];
(NonseqAddress && HREADY )
}
```

Fig. 6. An example of sequence declaration

Fundamentally, a regular expression specifies a list of tokens. In this paper, tokens are Boolean expressions of bus interface signals, symbols, and pre-declared sequences. It excels at describing behaviors over time. Unfortunately, with the growth of interface protocol complexity, describing the full details of typical interface protocols with regular expressions reveals the limitations of it. That is, the complexity of using the regular expressions for describing interface protocol grows exponentially. As a result, we introduce two features, *pipeline operator* and *information storage*, in this paper to eliminate the exponential blow-up issue of describing protocols by public regular expression-based languages, such as PSL. We describe these features in the following two paragraphs.

Transactions are almost fully pipelined in high-performance bus protocols. Phases of transactions may be overlapped with each other in a given cycle. It would be very difficult to describe the pipelining behaviors with cycle-by-cycle statements, such as using pure regular expressions. This is because it requires users to manually specify all possible parallel behaviors, all possible combinations of phases. Hence we enhance the regular expressions by a new operator, pipeline operator ($\rightarrow$). It can easily describe the pipelining behaviors. If an event is described as (A $\rightarrow$ B), it means that there must be a sub-event B next to the sub-event A, and there is no "vacuum" between them. In addition, B would overlap with the subsequent event. For convenience, if an event contains a pipeline operator, it is called an overlapped event. The sub-event before the pipeline operator is called normal-part of it and that behind the pipeline operator is called overlapping-part. For example, in AMBA AHB, a transfer has an address phase and a data phase, and two transfers could be overlapping. The address phase of one transfer and the data phase of the previous transfer may be overlapped with each other in a given cycle. Therefore, we can use the pipeline operator to concatenate the phases for describing the pipelining transfers clearly and successfully. In Fig. 7, the timing diagram of the overlapped events with the pipeline operator is illustrated.

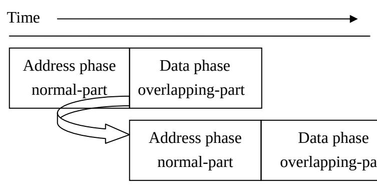


Fig. 7. The timing diagram of { (Address phase $\rightarrow$ Data phase)[*] }

Symbol declaration can be used to simplify the representations of regular expressions due to it is also allowed to store information. As a result, symbols become parts of regular expressions, and appear in Boolean expressions. Besides, symbols can be assigned to values or operated in an increment or decrement operation. Since operations on symbols are naturally mixed with regular expressions by the notation '&&', they are executed when the mixed tokens are matched. For example, in AMBA AHB, an incrementing burst can be of any length, but the upper limit is set by the fact that the address must not cross a 1KB boundary. It is convenient for users to describe this property by using symbols. In Fig. 8, the string-type symbol *last_address* is used to store the address of transfers in a burst. It is prepared for describing this property as "!((int)HADDR>>10 *XOR* (int)last_address>>10)". It states that the upper (32-10) bits of the address of the current transfer must be equal to that of previous transfer. On the other hand, in AMBA AHB, the signal of HMASTER must be stable during a transaction. By using pure regular expressions, transactions of each master are only slightly modified versions. Encoding every possible situations is necessary but painful. Symbols can be used to reduce the effort of describing this kind of behavior. In Fig. 8, the integer-type symbol *current_Master* is used to record which master is active on the bus. It is prepared for describing that the HMASTER has to be stable during a transaction in the symbol *stable_master*.

Besides, many behaviors are the same except some signals

```
// symbol layer
string last_address;
int current_Master;
boolean stable_master = (HMASTER == current_Master);

// sequence layer
sequence nonseq (sequence first_address, error_resp,
retry_resp, split_resp, ok_resp) =
{
((first_address && stable_master && !((int)HADDR>>10 XOR
(int)last_address>>10) && (last_address = HADDR))
$ (error_resp || retry_resp || split_resp || ok_resp))
}
```

Fig. 8. An example of a sequence with information storage

```
// symbol layer
int current_Master;

// sequence layer
sequence arbitration[2]() =
{
(<HBUSREQ%s> && !<HGRANT%s>)[*];
(<HBUSREQ%s> && <HGRANT%s> && !HREADY )[*];
(<HBUSREQ%s> && <HGRANT%s> && HREADY &&
(current_Master=<%d>))
}
```

Fig. 9. The arbitration phases of AMBA AHB described by a array-type sequence

in them, and those signals are usually named based on a specific rule. In general, users usually have to declare many sequences with similar behaviors. For example, in AMBA AHB, HBUSREQ0 and HGRANT0 are in correspondence with the master 0, while HBUSREQ1 and HGRANT1 are in correspondence with the master 1. The behaviors of the arbitration phase of masters are similar except the signals of their own. If there are two masters in the bus, users have to describe the arbitration phases of them with two individual but similar sequences. To reduce the effort of declaring many similar sequences, the array-type sequences are supported in our protocol descriptions. In the array-type sequences, a particular syntax, < *string*%s >, is allowed for presenting tokens of regular expressions. Besides, the index of array-type sequences may be useful information. It can be obtained by the syntax of < %d > if the type of it is integer and < %s > if string. With array-type sequences, users only need to specify the behaviors of arbitration phases one time, as shown in Fig. 9. Whatever the number of masters in the bus is, only one array-type sequence is required for specifying arbitration phases of them.

IV. Translations of Protocol Description to Extractor

The translations of the symbol layer and sequence layer are described in this section. The translations of the symbol layer are straightforward, similar to variable declarations and assignments in other programming languages. We just have to declare variables corresponding to symbols. Operations of symbols can be implemented by the primitive functions in the programming languages. Therefore, the translations of the symbol layer can be implemented easily.

The translations of the sequence layer can be implemented by creating the FSMs corresponding to sequences. The kernel of each sequence is the regular expressions, and the regular expressions are equivalent to FSMs in nature. Tokens are the transition functions of FSMs. If a token is a sequence, when it is processing, the FSM will hold its state. Then, the FSM transits to next state until it is matched. If it fails matching, the FSM will be reset and return to the initial state. Finally, if an FSM traverses from the initial state to the end state, it means that the behavior satisfies the sequence and a transaction will be extracted by the extractor. It, then, would be recorded in an FSDB-format file with the information of start time and end time.

Pipeline Operator:

For (A → B), if A is matched, the FSM transits to the state N+1 and a parallel engine will be attached to this state. This engine is used to check B. If it fails matching, the FSM will be reset. This can be seen in Fig. 10.

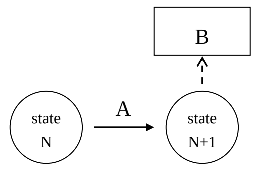


Fig. 10. Translations of (A → B)

Repeat Operator:

For (A[*];B), if A is matched, the FSM transits to the state N+1 and there is a self-loop of A in this state. Then, the state N+1 can transit to the state N+2 when matching B. Besides, the repeat operator * accepts empty string. If B is matched in the state N, the FSM transits to the state N+2 directly. This can be seen in Fig. 11.

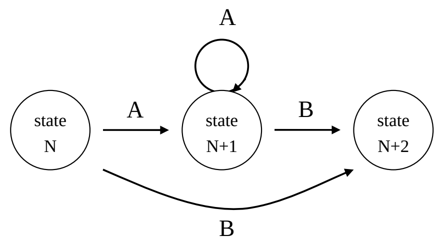


Fig. 11. Translations of (A[*];B)

For (A[*]), if the repeat operator * is attached to the last token, an additional state, N+2, is added to the FSM. If A is matched, the FSM will transit to the state N+1 and there is a self-loop of A in this state. Then, if A fails matching, the FSM will transit to the end state N+2. Because the repeat operator * accepts empty string, in state N, the FSM can transit directly to the end state N+2 if A fails matching. With the additional state, the FSM will match A as many as possible. Therefore, this feature makes the transaction extractor extracts transactions of the longest length. This can be seen in Fig. 12.

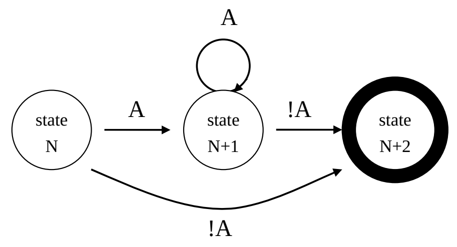


Fig. 12. Translations of (A[*])

For (A[+];B), it is similar to (A[*];B) but the repeat operator + does not accept empty string. Therefore, the transition arcs from the state N to the state N+2 are removed in Fig. 11 and 12.

Operations of Symbols:

Operations of symbols will be regarded as actions of states. If the FSM transits from the current state to the next state, the actions of the next state will be executed. Besides, if there is a self-loop in a state, the actions of this state will be executed once the transition function of the self-loop is matched.

In sum, we have successfully translated our protocol descriptions, including new features, into the executable transaction extractor. It raises the verification level to a higher level of abstraction. We have also demonstrated our methodology by generating the transaction extractors of ARM's AMBA AHB and Sonics' OCP, and the high level transactions are fully extracted successfully.

V. Attribute Description

To show the corresponding useful information upon the extracted transactions for verification, the details of transactions have to be kept. Thus, the attribute descriptions are used for obtaining information of transactions. The attribute description is specified in another file, but corresponds with the protocol description. Since different verification engineers could prefer seeing different information on the extracted transactions. The attribute description is customized by individuals who refer to the same protocol description. Thus, the well-specified protocol descriptions can be reused easily among verification engineers. In this section, we introduce that how to obtain useful information by specifying the attribute descriptions. The attribute description exploits sequences to store the information of transactions. Then, these sequences are used to map with the sequences in the protocol descriptions.

The features of attribute description are summarized as follows:

- **Using integer-type and string-type local variables to store the information.**
- **Supporting increment, decrement and assignment(+ or +=) operations on local variables.**
- **Providing hidden/visible attributes attached to local variables, only visible local variables could be treated as the transaction information.**
- **Being executed in overlapping-part if an operation is prefixed by '#', otherwise operations are executed in normal-part.**
- **Being regarded as a Boolean variables if an instantiated sequence appears in a Boolean expression. True if it is matched, otherwise false.**
- **Obtaining the information of instantiated sequences by the syntax of *sequence_name.variable_name*.**
- **Supporting the operation *enqueue* for adding relationships between extracted transactions.**

Finally, we illustrate the mapping between protocol descriptions and attribute descriptions with an example, as shown in Fig. 15. The operations, (address = HADDR) and (ReadorWrite = first_address.ReadorWrite), in Fig. 15(b) are mapping to *first_address* in Fig. 15(a). These operations are naturally concatenated by the notation '&&'. The information of address will be stored in the local variable *address*. The information of read/write will be stored in *ReadorWrite*. The operation, enqueue(split_resp,address), in Fig. 15(b) is mapping to (error_resp ∥ retry_resp ∥ split_resp ∥ ok_resp) in Fig. 15(a). If *split_resp* is true, *address* will be stored in an internal queue. Then, if *address* of another transaction is equivalent to that in the queue, the relationship between this transaction and that in the queue will be added. It is used for obtaining the information if transfers are related to SPLIT responses.

```
sequence nonseq (sequence first_address, error_resp,
retry_resp, split_resp, ok_resp) =
{
(first_address → ( error_resp ∥ retry_resp ∥ split_resp ∥
ok_resp ))
}
```

(a)A sequence in the protocol descriptions

```
sequence nonseq () =
{
visible int ReadorWrite;
visible string address="";

((address = HADDR) && (ReadorWrite =
first_address.ReadorWrite) && #enqueue(split_resp,address))
}
```

(b)The corresponding sequence in the attribute descriptions

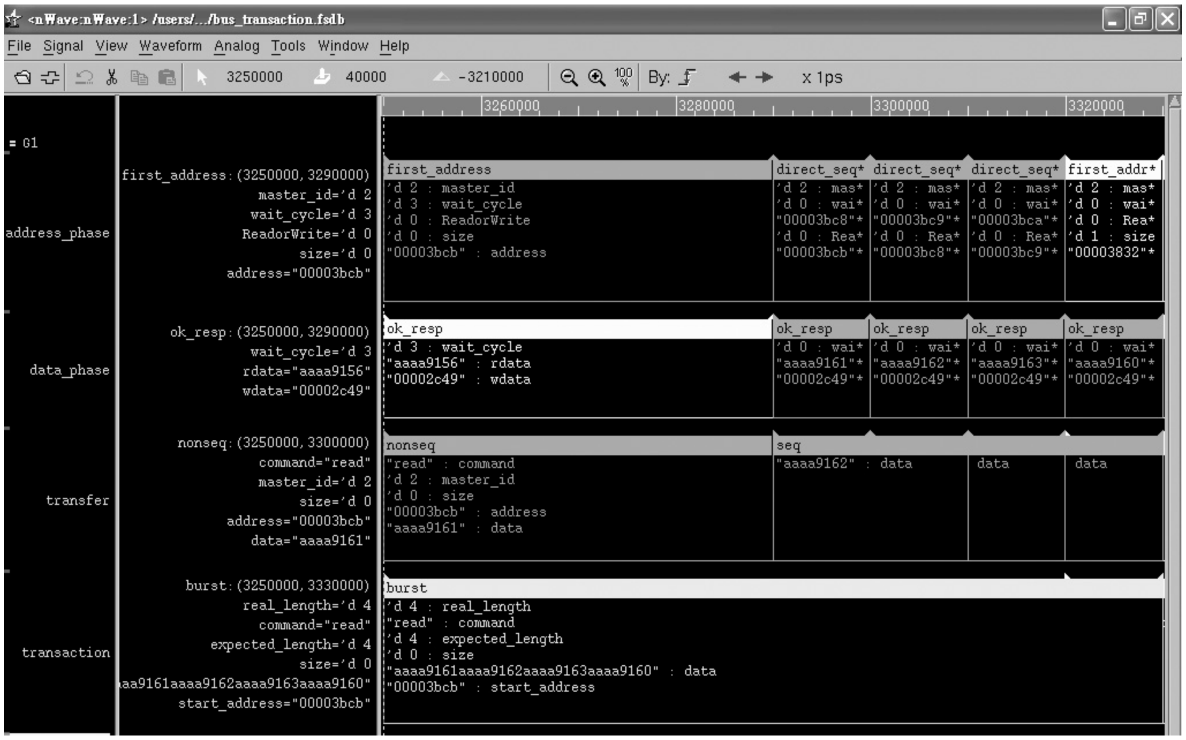
Fig. 15. Sequence mapping between protocol descriptions and attribute descriptions

With the attribute descriptions, extracted transactions are enriched by useful information. The relationships between specific transactions are available. Thus, the debugging process can be conducted in transaction level and the verification effort is reduced.

VI. Experimental Results

In this section, we show the experimental results of the automatically generated transaction extractor base on AMBA AHB and Sonic's OCP bus protocols. The results are shown in Fig. 13 and 14, respectively. Due to page limit, only sampled results are shown.

When different bus protocols have been formally described by our language-based methodology, a unique compiler will generate the corresponding transaction extractors without manual effort. The input of a transaction extractor is a simulation log file of bus interface signals dumped from an SoC design, and the output is the extracted transactions in FSDB-format. It can be displayed in nWave environment, as shown in Fig. 13 and 14. In Fig. 13, visible local variables of sequences are shown as the attributes of the transactions. In addition, the hierarchical relationships between transactions are also highlighted (it is displayed in gray color due to printing). In Fig. 13, an extracted transaction in AMBA AHB is displayed on the label *transaction* (Column 1) with attributes of address, data, and etc (Row 4). It is made up of four transfers, one *nonseq* and three *seq*, displayed on the label *transfer* (Row 3). The corresponding address phases and data phases are also shown on the label *address_phase* (Row 1) and *data_phase* (Row 2). In Fig. 14, a simple single write transfer in OCP is extracted as well. Please notice



<nWave:nWave:1> /users/.../bus_transaction.fsdb
File Signal View Waveform Analog Tools Window Help
3250000 40000 -3210000 By: x 1ps
3260000 3280000 3300000 3320000
G1
address_phase
data_phase
transfer
transaction

```
first_address:(3250000,3290000)
     master_id='d 2
    wait_cycle='d 3
   ReadorWrite='d 0
          size='d 0
   address="00003bcb"

ok_resp:(3250000,3290000)
    wait_cycle='d 3
   rdata="aaaa9156"
   wdata="00002c49"

nonseq:(3250000,3300000)
   command="read"
   master_id='d 2
        size='d 0
  address="00003bcb"
     data="aaaa9161"

burst:(3250000,3330000)
   real_length='d 4
   command="read"
   expected_length='d 4
          size='d 0
aa9161aaaa9162aaaa9163aaaa9160"
   start_address="00003bcb"
```

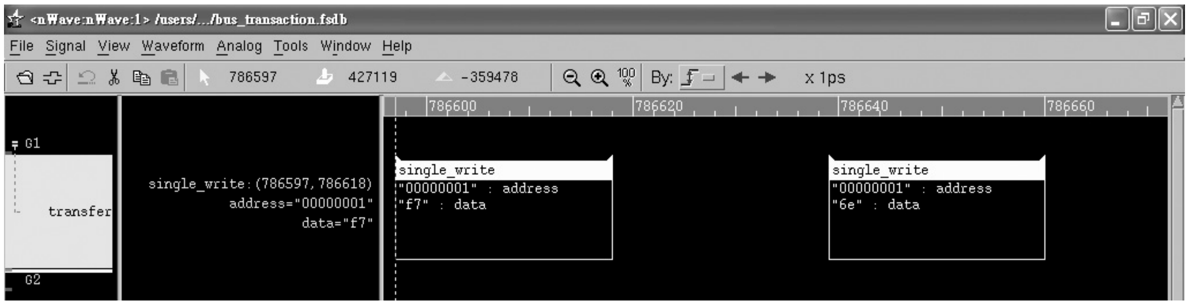
```
first_address           direct_seq*  direct_seq*  direct_seq*  first_addr*
'd 2 : master_id        'd 2 : mas*  'd 2 : mas*  'd 2 : mas*  'd 2 : mas*
'd 3 : wait_cycle       'd 0 : wai*  'd 0 : wai*  'd 0 : wai*  'd 0 : wai*
'd 0 : ReadorWrite      "00003bc8"*  "00003bc9"*  "00003bca"*  'd 0 : Rea*
'd 0 : size             'd 0 : Rea*  'd 0 : Rea*  'd 0 : Rea*  'd 1 : size
"00003bcb" : address    "00003bcb"*  "00003bc8"*  "00003bc9"*  "00003832"*

ok_resp                 ok_resp      ok_resp      ok_resp      ok_resp
'd 3 : wait_cycle       'd 0 : wai*  'd 0 : wai*  'd 0 : wai*  'd 0 : wai*
"aaaa9156" : rdata      "aaaa9161"*  "aaaa9162"*  "aaaa9163"*  "aaaa9160"*
"00002c49" : wdata      "00002c49"*  "00002c49"*  "00002c49"*  "00002c49"*

nonseq                  seq
"read" : command        "aaaa9162" : data         data         data
'd 2 : master_id
'd 0 : size
"00003bcb" : address
"aaaa9161" : data

burst
'd 4 : real_length
"read" : command
'd 4 : expected_length
'd 0 : size
"aaaa9161aaaa9162aaaa9163aaaa9160" : data
"00003bcb" : start_address
```

Fig. 13. Sampled experimental results of AMBA AHB



<nWave:nWave:1> /users/.../bus_transaction.fsdb
File Signal View Waveform Analog Tools Window Help
786597 427119 -359478 By: x 1ps
786600 786620 786640 786660
G1
transfer
G2

```
single_write:(786597,786618)
   address="00000001"
      data="f7"

single_write              single_write
"00000001" : address      "00000001" : address
"f7" : data               "6e" : data
```

Fig. 14. Sampled experimental results of Sonic's OCP

that if the transactions cannot be extracted successfully, it implies the behaviors of signals do not follow the protocol descriptions and they could be erroneous behaviors. Thus, the approach does not only achieve extraction but also achieve interface verification implicitly.

VII. Conclusions

In this paper, we propose a language-based methodology that demonstrates the automatic generation of transaction extractors is possible for various OCBs as long as the protocols are described formally. Those extracted transactions are also enriched by some useful attributes. As a result, the information presented for debugging is in terms of transactions, rather than signals and waveforms.

References

[1] Accellera. "Property Specification Language Reference Manual Version 1.01," 2003.

[2] Accellera. "Open Verification Library Assertion Monitor Reference Manual," June 2003.

[3] Koji Ara, Kei Suzuki. "A Proposal for Transaction-Level Verification with Component Wrapper Language" in *Proc. of Design, Automation and Test in Europe Conference and Exhibition*, pp. 82-87, 2003.

[4] ARM Limited. "AMBA Specification Rev (2.0)," 1999.

[5] Cadence Berkeley Labs. "The Transaction-Based Verification Methodology," August 2000.

[6] Component Wrapper Language Specifications and Users manual, available at http://koigakubo.hitachi.co.jp/.sl/cwl/html/ en/, 2002.

[7] Pallab Dasgupta, Arindam Chakrabarti, P.P. Chakrabarti. "Open Computation Tree Logic for Formal Verification of Modules," in *Proc. of International Conference on VLSI Design*, pp. 735-740, 2002.

[8] International Business Machines Corporation. "The CoreConnect^{TM} Bus Architecture," 1999.

[9] NOVAS Software, Inc. "FSDB API of Debussy 5.4," Mar 2003.

[10] OpenCores Organization. "WISHBONE SoC Architecture Specification Revision B.3," September 7, 2002.

[11] Sonics Incorporated. "Open Core Protocol Specification 1.0 Document Version 1.2," 2000.

[12] Marcio T. Oliveira, Alan J. Hu. "High Level Specification and Automatic Generation of IP Interface Monitors," in *Proc. of Design Automation Conference*, pp. 129-134, June 2002.