Grammar-based Hardware Synthesis of Data Communication Protocols

Johnny Öberg, Anshul Kumar, Ahmed Hemani
Electronic Systems Design Laboratory,
Royal Institute of Technology,
ESDLab/KTH-Electrum, Electrum 229, S-164 40 Kista, Sweden
Telephone: + 46 8 752 13 05
Email: johnny@ele.kth.se, kumar@ele.kth.se, ahmed@ele.kth.se

Abstract

For a synthesis methodology to support implementation independent design specification, a capability for design space exploration is essential. In this paper we present such a methodology for a specific domain: data communication protocols. A natural way to specify various elements of protocols is in terms of a grammar annotated with actions. Our language for protocol specification, called PRO-GRAM, is based on this idea. The hardware specification of the protocol is done by specifying the bit-patterns of the tokens the protocol is supposed to parse together with the actual grammar to parse the input stream. By specifying constraints on the input and output stream ports, the designer is allowed to explore alternative realisations with different widths of the I/O ports. The PRO-GRAM compiler outputs VHDL-code suitable for Logic Synthesis.

1. Introduction

Research in High Level Synthesis has made it possible to describe designs at behavioural level in languages like VHDL and to synthesize detailed circuits automatically. However, at the level of description supported by the present day HLS tools, a designer still cannot work entirely independent of the implementation. The need for languages and synthesis tools to support higher levels of design specification is evident from the recent research in system level synthesis [1][2]. For a synthesis methodology to support implementation independent design specification, a capability for design space exploration is essential. Thus, such a methodology is quite aptly termed specify-explore-refine methodology [3]. Design space exploration is a complex task in general, but can be more manageable in a specific application domain. In this paper we present a synthesis methodology which aims at supporting high level specifications and design space exploration for a specific domain: data communication protocols.

In the area of data communication protocols, several formal languages have been developed, for example Estelle, Lotos, SDL [4] and Promela [5]. These have been primarily used for verification and validation of the desired properties of protocols and for obtaining their software implementations rather than for synthesis of hardware implementations.

Our language for protocol specification, called PRO-GRAM (Protocol Grammar), is inspired by YACC [6] in terms of its notations but has been designed with the aim of hardware synthesis. In spirit, it has some similarity with LOTOS [4] in the sense that like LOTOS, specifications in PRO-GRAM deal with sequences of allowed events rather than states and state transitions as is the case with Estelle, SDL and Promela.

In section 2 we relate our work with some similar work done recently. Sections 3 gives a brief description of PRO-GRAM. In section 4 we describe the synthesis process and in section 5 we show the results of synthesising a small part of the F4 OAM ATM-protocol. Finally, in section 6 we summarize our contribution, draw some conclusions and discuss future research.

2. Related Research

Automatic generation of language recognizers from grammar specifications has been extensively used in the software area for a long time [7]. Synthesis of hardware from such specifications was recently reported by Seawright et. al. [8]. They describe a system, called Clairvoyant, which is used to synthesize some small to medium sized examples from Production-Based Specifications. The output of Clairvoyant is an FSM described in VHDL which can be handled by suitable low level synthesis tools. This system is targeted for detailed specification of communication interfaces and other control-dominated circuits including Communication Protocols. In Clairvoyant a design entity with a single process and a well defined boundary and interface is specified. All inputs and outputs

in the design entity are exactly described at clock cycle level.

Our approach is similar to this to the extent that the input is a production based specification and the output is in RT-level VHDL. However, PRO-GRAM is targeted for specification and synthesis of large systems and differs from the above approach in a significant way. The essential difference is that a specification in PRO-GRAM is at a higher level of abstraction. The synthesizer allows fast exploration of the lower level design alternatives and frees the designer of clock cycle level details and exact input-output description. Furthermore, multiple processes can be specified in PRO-GRAM and reading and writing of the multiple input and output streams can take place independently. This is very useful for structuring large designs.

In Clairvoyant all actions are specified in VHDL-code, similar to the software approaches used by, for instance, YACC [6][7]. PRO-GRAM, however, is using actions specified as lists of assignments. This helps in analysing the specifications for design space exploration as well as for design verification.

3. Describing HW Protocols as a Grammar

3.1. Protocol Specification Requirements

A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. The five elements of a protocol that are needed for a complete specification are [5]:

1) The service to be provided.
2) The assumptions about the environment.
3) The vocabulary of messages.
4) The encoding format of each message.
5) The procedure rules for consistency.

A natural way to specify the elements 1, 3, 4 and 5 is in terms of a grammar annotated with actions. The vocabulary of messages used to implement the protocol correspond to the grammar rules. The encoding format of each message in the vocabulary corresponds to the token terminals encoded into the grammar. The procedure rules guarding the consistency of message exchanges are also embedded in grammar rules. The service provided by the protocol corresponds to the actions in a grammar. We view the element 2, i.e. the assumptions about the environment, as a set of constraints posed on the synthesis process. The most natural place to specify these constraints is in the interface declaration section of our grammar.

3.2. The Grammar Specification

In PRO-GRAM, the grammar specification is divided into five distinct sections - Interface, Token definitions, Memory layout, Action values and Grammar rules.

Interface Declarations.

In the interface section the external interfaces and internal signals (interfaces between processes) are declared as:

```
%input input_cell_1 [bit]8 rate 155 Mbps
%internal vci [bit]16
%internal address [bit]8
%output output_cell_11 [bit]53*8
```

An input declaration consists of the name, the width and the bitrate of the input. Bitwidth specifications are given as constraints for the synthesizer. If the bitrate is given, it is for the PRO-GRAM compiler to guarantee that the produced circuit meets the throughput constraints. Declarations of outputs and internal signals lack the bitrate specification in the present implementation. The grammar Rules are completely independent of the constraints. Varying the bitwidths and bit rates allow design space exploration without changing the remaining part of the specification. The interface section ends with start rules as in YACC:

```
%start Input_Handler(input_cell_1)
```

One difference with YACC is that in addition to the start rule, the input stream that should be parsed with that rule is also given. The input stream is then inherited downwards in the hierarchy of rules until a redirection of the input stream is found, as explained later in this section. Another difference with YACC, is that in PRO-GRAM it is also possible to specify multiple concurrent processes by specifying multiple start rules.

Token definitions.

A Token is a pattern of bits read from an input stream or written to an output stream. Reading and writing tokens are the primary events. In a software parser, sequences of bytes are read from the input stream by a lexical analyser and the tokens produced are fed to the grammar parser. This division into two separate programs is not followed by us when parsing data communication protocols as sequences of bits. Examples of PRO-GRAM Tokens for an F4 OAM- protocol [9] are given in Figure 3.1.

The Token SIXA42 is an example of multiplicity where the pattern 6A (hexformat) is repeated 42 times.

Memory Layout.

Memories are specified in a slightly different way:

```
%memory connection_memory [[bit]8] connection_status
```

The size of the memory is given by the number in brackets as the number of address lines connected. Then the rule that specify the memory field layout is given. Currently, the memory port width is directly inferred from the memory field layout. However, it is possible to consider this also as a target for design space exploration.

Action Value Section.

Actions in the grammar specify assignment of values to

```
// GENERAL TOKENS
VCI_SEGMENT0000 0000 0000 0011
VCI_CONNECTION0000 0000 0000 0100
VCI_USER_1 0000 0000 0000 0001
VCI_USER_2 0000 0000 0000 0010
VCI_USER_5 0000 0000 0000 0101
VCI_HIGH_IS_ZERO0000 0000 0000
VCI_IDLE      0000 0000 0000 0000
SIXA42        [0110 1010]42
```

```
vci_user: VCI_USER_1
        | VCI_USER_2
        | VCI_USER_5
        | ^VCI_HIGH_IS_ZERO [bit]4;
vci_other: VCI_HIGH_IS_ZERO 0110
        | VCI_HIGH_IS_ZERO 0111
        | VCI_HIGH_IS_ZERO 1 [bit]3;
```

Figure 3.1. a) Examples of F4 OAM specific tokens. b) Usage of Tokens as Terminators

```
user_cell_action_1_highp = $connection_endnode
    $segment_endnode
    NOT_PRESENT_STATE
    NOT_PRESENT_STATE
    SN_LOOPBACK $sn_actreq $sn_deactreq
    $sn_gen $sn_term LOOPBACK_TIMEOUT
    PEER_RESPONSE_TIMEOUT
    $block_size_a_b $block_size_b_a
    $block_size_a_b_curr
    $block_size_b_a_curr $corr_tag $vci_sc $bip16_1
    $mcsn $tuc0 $tuc01 $directions $mcsn_bw
    $mcsn_prev $tuc01 $trcc0 ($trcc01+1)16
    $connection_rest27;
```

Figure 3.2. Example of an action value specification.

signals. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action value section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols.

Grammar Rules.

A grammar rule consists of a grammar symbol which serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols, terminals and actions. In Figure 3.3. two top level rules of the F4 OAM example are given. A redirection of the input stream is done by passing the new signal stream as a parameter to the subtree of productions. Actions are enclosed in curly brackets. A grammar symbol prefixed with $ refers to the bit pattern recognized by that symbol. The scope of such a reference is limited to the symbols already recognized in the current alternative.

In the signal assignment in Figure 3.3., the signal address is assigned the value of the rule production vpi. The symbol special_user_cell parses a stream of bits obtained from

```
Input_Handler:input_cell;
input_cell: gfc vpi VCI_SEGMENT { vci=VCI_SEGMENT; }
        pti clp hec
         { address = $vpi; } oam_segment_types
    | gfc vpi VCI_CONNECTION
        { vci=VCI_CONNECTION; } pti clp hec
         { address = $vpi; } oam_connection_types
    | gfc vpi vci_user {vci=$vci_user;} pti clp hec
        { address = $vpi; } user_cell_body
        special_user_cell(connection_memory[address])
        { output_cell_11 = special_user_cell_action;
         priority=PRIORITY_1; }
    | gfc vpi vci_other {vci=$vci_other;} pti clp hec
        user_cell_body
        { output_cell_11 = other_user_cell_action;
         priority=PRIORITY_0; }
    | gfc vpi VCI_IDLE {vci=VCI_IDLE;} pti clp hec
        user_cell_body;
```

Figure 3.3. Partial grammar specification for F4 OAM example.

```
    gfc: [bit]4;
    clp: bit;
    user_cell_body: [bit]424-40;

    oam_segment_types: ...
        | ...
        | [others]4 [bit]432-44;

    segment_fault_management:...
        | ...
        | error;
```

Figure 3.4. Example of terminals.

connection_memory.

Examples of Terminals can be found in Figure 3.4. A terminal can be any of the four following things: a token, a bit string, a special pattern or the keyword error. The keyword error denotes an error condition. There are two types of special patterns:

[bit]k

[others]k

where k is an integer. The first one specifies a string of k don't cares (either a 1 or a 0). The second one is the same as specifying an else clause i.e. all other combination of bits that has not previously been specified. This also matches any string of k bits. The others type of pattern or an error keyword can only appear as the last alternative in a grammar rule. In addition it is also possible to negate a pattern i.e. to specify that anything but this pattern matches the description. Any pattern or error condition not directly given corresponds to an error state in the synthesized hardware.

3.3. Restrictions to a full context free grammar.

Unrestricted recursion poses a problem when using a grammar to describe hardware since interpretation of arbitrary recursion requires an unbounded stack. Tail recursion

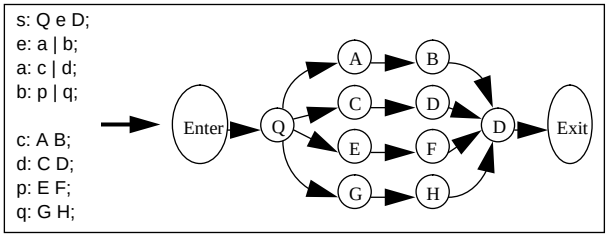

Figure 4.1. Grammar DAG example.

```
CheckConsistency();
for i=1 to Number_of_Start_Rules loop
    start_symbol=new symbol(ENTER,stream(i));
    exit_symbol=build_grammar_DAG(start_rule(i),
        start_symbol);
    ReduceExits(start_symbol);
    WordAlignTerminals(start_symbol,stream_size(i));
    WordAlignOutputs(start_symbol);
    ReduceGrammar(start_symbol);
    MarkAllStates(start_symbol);
    Output_FSM(start_symbol);
end loop;
```

Figure 4.2. The synthesis procedure.

or right-recursion causes no problem since it can be implemented without a stack. Therefore, we allow only right recursion. We disallow empty productions also.

An alternative to tail recursion is the usage of the Kleene Closure operator of regular expressions. This approach is followed in Clairvoyant [8] and also in regular expression based lexical analyzer generators e.g. LEX [10]. However, we decided to use a more homogeneous approach of specifying everything as production rules.

4. Synthesis from the Protocol Grammar

In order to synthesize a state machine, we transform the grammar specification into a Directed Acyclic Graph (DAG), called the Grammar DAG. The Grammar DAG is a series-parallel graph in which parallel subgraphs correspond to the alternatives for a grammar symbol and subgraphs in series correspond to the sequence of items within an alternative. All non-recursive references to non-terminal symbols are expanded to their subgraphs in the Grammar DAG. Thus, the nodes of the Grammar DAG correspond to terminals and recursive references to non-terminals. Figure 4.1. shows an example of grammar rules and the corresponding Grammar DAG.

The input is parsed, checked for consistency and conformance to the restrictions described in the previous section, and the Grammar DAG built and optimised using the procedure shown in Figure 4.2. The grammar DAG is built using a recursive algorithm shown in Figure 4.3.

After the DAG has been built all dummy Exit nodes are reduced out of the Graph. Only one Exit node remains after reduction: the top exit state that collects all different

```
build_grammar_DAG(rule,prev_symbol) return symbol
begin
    for i in all rule branches in rule loop
        next_symbol=prev_symbol;
        for j in all items in rule branch(i) loop
            if recursive symbol then
                if not the last symbol
                    Give Error Message and Terminate
                else
                    last_symbol=new symbol(RECURSION, prev_symbol);
                end if;
            else if symbol then
                last_symbol=build_grammar_DAG(sub_rule, next_symbol);
            else if redirect then
                next_symbol=new symbol(ENTER, stream);
                last_symbol->AddRedirection(next_symbol,
                        build_grammar_DAG(sub_rule, next_symbol));
            else if [constant, bit_string] then
                last_symbol=new symbol(TERMINAL, value);
                link(last_symbol, next_symbol);
            else if [others, any_bits, error] then
                last_symbol=new symbol([OTHERS, BITS, ERROR],
                        nr_of_bits);
                link(last_symbol, next_symbol);
            else if action then
                next_symbol=new symbol(ACTION, output_stream);
                last_symbol->AddAction(next_symbol,
                        build_Action(action_pointer, next_symbol));
            else
                Give Error Message and Terminate
            end if;
            next_symbol=last_symbol;
        end loop;
        branch_symbol[i]=last_symbol;
    end loop;
    exit_symbol=new symbol(EXIT, NULL);
    for all rule branches in rule loop
        link(exit_symbol,branch_symbol[i]);
    end loop;
    return exit_symbol;
end function;
```

Figure 4.3. The Build grammar DAG algorithm.

branches in the DAG.

When the grammar DAG has been constructed and all dummy Exits have been deleted, the resulting grammar DAG is word aligned onto its input stream. This is done by grouping together terminals which are of less size than the input stream width and splitting terminals that are wider than the input stream width. Sometimes the grouping of terminals is not a good decision if the input streams are too wide. Instead, a multiple clock scheme can be utilised, one for clocking the data in the input stream and a faster one for clocking the statemachine. The slower clock can be derived from the faster one. This is the solution traditionally applied when performing High-Level Synthesis. The same procedure is used to word align the outputs. However, in the output case no grouping needs to take place since the output assignment size must be an integral multiple of the output port size.

After the grammar DAG has been word aligned it must be reduced. For ease of description a designer is allowed to duplicate terminals and patterns in multiple branches as

```
ReduceGrammar(l)
begin
    Denote the set of all successors as Succ(l)
    if symbol_has_no_Successors then return;
    if symbol_has_one_successor then
        ReduceGrammar(only_successor);
    else
        for i,j in Succ(l), i != j loop
            if (Successors i and j are identical)
                MergeSuccs(i,j);
                DelSucc(j);
            end if;
        end loop;
        for i in all successors loop
             ReduceGrammar(successor[i]);
        end loop;
    end if;
end method;
```

Figure 4.4. The Grammar Reduction Algorithm

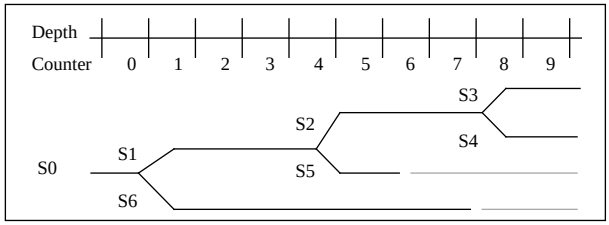

Figure 4.5. The Marking of Symbolic States in the Grammar DAG.

long as they can be reduced out later. This is also needed because when the Word Alignment of Terminals are performed, terminals are splitted and spread. If two specified constants have a long series of leading zeroes and they are word aligned to e.g. 8 bits, the produced terminals for these constants needs to be merged since the synthesized FSM must be a DFA. The grammar reduction algorithm is shown in Figure 4.4.Two terminal successors are identical if they have the same terminal patterns and the scheduled actions are identical. Two actions are identical if they refer to the same output and have the same value assigned to it. If any of the nodes to be merged is the entry point of the subgraph corresponding to a symbol with recursive rules, some nodes need to be duplicated.

After the grammar reduction states must be assigned to the grammar DAG. All branches are marked as distinct symbolic states. The State Marking is done recursively in linear time and will result in a Grammar DAG such as the one shown in Figure 4.5.

In order to reduce the next state logic, we split the states into two parts. The first part is called the “depth”, and records the passage of time, implemented either as a counter that is reset upon synchronisation or a one-hot-encoded token pipeline that is set upon synchronisation. The second part is called the “branch”, representing the branching decisions in the grammar DAG.

The PRO-GRAM compiler outputs Synopsys compatible

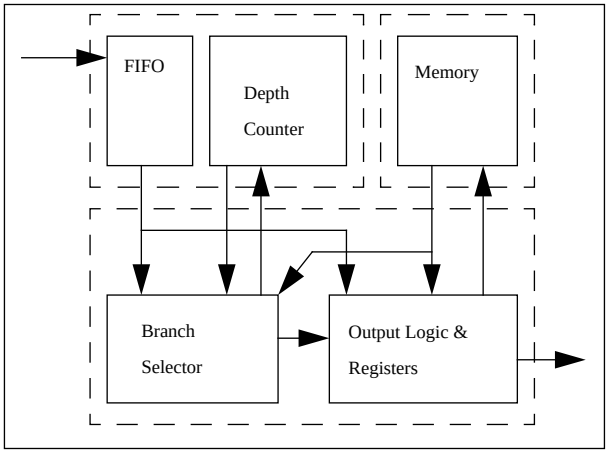

Figure 4.6. Target architecture with one input stream

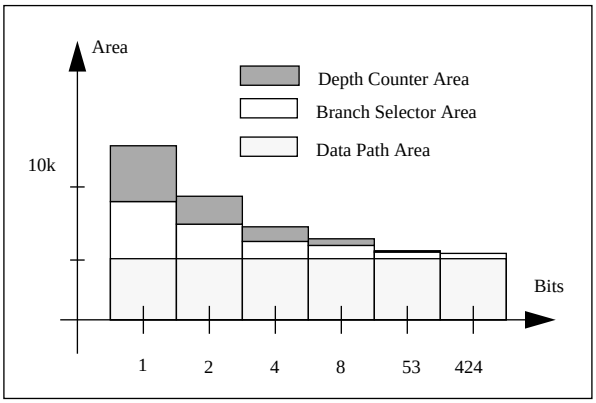

Figure 5.1. Design Space Exploration. Area vs. input port width

VHDL code. The general architecture of the synthesized design is shown in Figure 4.6.

It consists of a depth counter in the form of a token pipeline, one FIFO per Input Stream to allow the actions to refer to the values of previous parsed symbols, a branch selector in the form of a symbolic State Machine, memories and output logic and plus registers. Before Logic Synthesis, the token pipeline, the FIFO and the memories need to be extracted from the VHDL-code since these are poor candidates for optimisation by Logic Synthesis.

5. Results

A small part of the F4 Operation and Maintenance Protocol for ATM-switch systems [9] have been synthesised for different input port widths to test the strength of our approach. The design parses the incoming ATM-stream and extracts different types of OAM-cells. The synthesis results of the different versions are displayed in Table 1. and also plotted in Figure 5.1.

The datapath area includes the area of muxes and output

Input Port Width[a]	1	2	4	8	53	424
# of VHDL lines	3498	1817	1094	652	310	231
# of BS states	569	291	154	89	40	28
# of DC states	424	212	106	53	8	1
VHDL Generation Time (s)	1.89	1.70	1.69	1.65	1.62	1.61
FSM Synthesis Time (s)	12633[b]	2511[b]	50631	17605	3905	2056
Clock Period (ns)	6.5	12.5	25	50	100	200
Critical Path (ns)	19.35	17.58	21.75	47.63	49.56	43.80
Slack (ns)	-14.45	-6.68	1.65	0.77	48.84	154.60
BS + datapath area (gates)	8919[b]	7233[b]	5917	5597	5099	5028
DC area (gates)	4240	2120	1060	530	80	10

a.Only sub multiples of 424 (no. of bits in an ATM cell) were chosen as input width values.
b.The output Flip-flops were manually extracted from the VHDL-code to reduce the synthesis time.

Table 1. Synthesis of part of F4 OAM Functionality

registers. Memory and FIFO area is not included in the area figures.The lsi_10k was used as a target technology. Because the design performs a series to parallel conversion, the output width is fixed to 424 bits and therefore the datapath area is constant. The area of the datapath would increase linearly with the output width, were the output width to be varied as well.

The State Machine and Logic Synthesis was performed on a Convex Computer with 4 parallel processors and 512 MB of RAM. The VHDL generation was done on an HP 715 with 32 MB of RAM

6. Conclusions and Future Research

We have presented a methodology for specifying data communication protocols in an implementation independent manner and synthesizing hardware from such specifications. The approach appears promising. The synthesis tool developed can be used to explore alternative realisations with different widths of the I/O ports. A number of other design options can be included to explore a larger design space. These include alternative state partitioning and encoding choices, alternative synchronisation schemes and various pipelining options. The synchronisation schemes could be organized in the form of a macro library which could be accessed by the synthesizer.

The current State Marking Algorithm does not take into account the possibility of reusing states that drive the same outputs once the DAG has been splitted. This would reduce the number of states and speed up the Logic Synthesis Phase. However, it would also introduce a stack to remember the previous state. The size of the stack can be determined at compile time.

Scheduling of the output signal assignments also has many possibilities which have not been explored. Since the outputs are distributed over the control states, some kind of timing constraints that tell when the first word of the assignment should be outputted can be used to ensure a correct behaviour in a timing critical environment.

From our experiments it can be seen that VHDL generation time is very small but the time it takes to perform State Machine and Logic Synthesis is very long, particularly for small bit widths, because the state space of the design explodes. This could be partly fixed in the cases where the input is 1, 2, 4 and 8 bits. The number of extracted one-hot-states could be reduced away since most of the state control points already exist in the state machine.

The designs with 1 and 2-bit input do not meet the timing constraints because the delay in the decoding of the next-state logic is too long. Here a direct selection of one-hot-encoding for the whole statemachine would solve the state decoder delay problem. However, the datapaths contain some muxes, whose operations also needs to be spread out over time. This can be done by pipelining the datapath to meet the timing constraints.

References

[1] D. D. Gajski, F. Vahid, S. Narayan, J. Gong, “Specification and Design of Embedded Systems”, Prentice Hall, 1994.

[2] F. Vahid, S. Narayan, D. D. Gajski, “SpecCharts: A VHDL frontend for embedded systems”, IEEE Trans. on CAD, vol. 14, pp 694-706, 1995

[3] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, “System Design Methodologies: Aiming at the 100 h Design Cycle”, IEEE Trans. on VLSI Systems, vol. 4, pp 70-82, March 1996.

[4] Edited by K. J. Turner, “Using Formal Description Techniques”, John Wiley & Sons Ltd., 1993.

[5] G. J. Holzmann, “Specification and validation of Protocols”, Prentice-Hall International Inc., 1991.

[6] S. C. Johnson, “YACC, Yet another compiler compiler”, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.

[7] A.V. Aho, R. Sethi, J.D. Ullman, “Compilers, Principles, Techniques and Tools”, Addison-Wesley Publishing Company, 1986.

[8] A. Seawright, F. Brewer, “Clairvoyant: A Synthesis System for Production-Based Specification,” IEEE Trans. on VLSI Systems, vol. 2, pp 172-185, June 1994.

[9] “B-ISDN Operation and Maintenance interface principles and functions”, ITU-T Recommendation I.610.

[10] M. E. Lesk, “Lex-A lexical analyzer generator”, Computing Science Tech. Rep. 39, AT&T Bell Lab., Murray Hill, 1975.