Automating information retrieval from EDA software reports using effective parsing algorithms

Manish Bhati Siemens EDA, manish.bhati@siemens.com

2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

INTRODUCTION

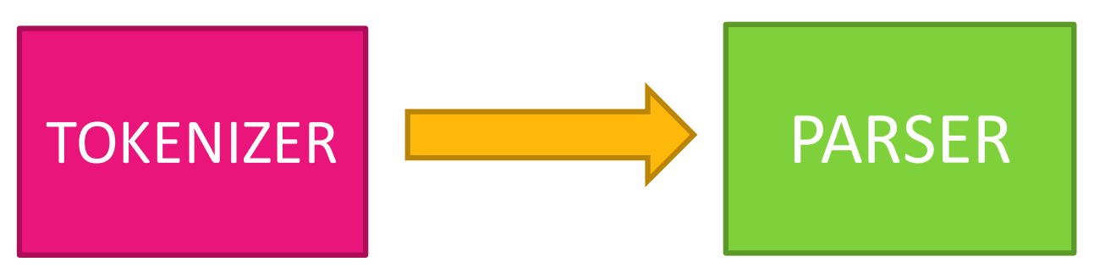
VLSI design has many stages of development right from requirement to final implementation in ASIC or FPGA. All these stages entail many EDA software where these software produce results in the form of reports. These reports have meaningful information to be extracted. Generally, these reports are structured, and engineers extract information through scripts. The most common way of extracting useful information is through the usage of regular expressions to find the desired patterns in the report/data files. Regular expressions help to a certain degree in extracting the patterns, but a robust parsing scheme is always required to extract the data.

Parsing algorithms are very specific to computer science especially in the field of compiler design. We can leverage parsing algorithms to parse the EDA tool reports. Also, sometimes the EDA tool require input data in some specific format like CSV. Consider an example where a verification team keeps its register initialization information in a proprietary format. To convert this data into CSV we may require parsing the proprietary format data. Engineers often face these challenges of data migration and hence a need for effectively parsing the data is inevitable.

Parsers can be coded in some language or generated through automation by some language agnostic tool like [1]. We will use [1] to automatically generate a parser saving lot of time. We will show an example of a Reset Tree expression generated by Questa RDC tool and will see that one can leverage this information to do meaningful tasks like AI/ML to extract more information.

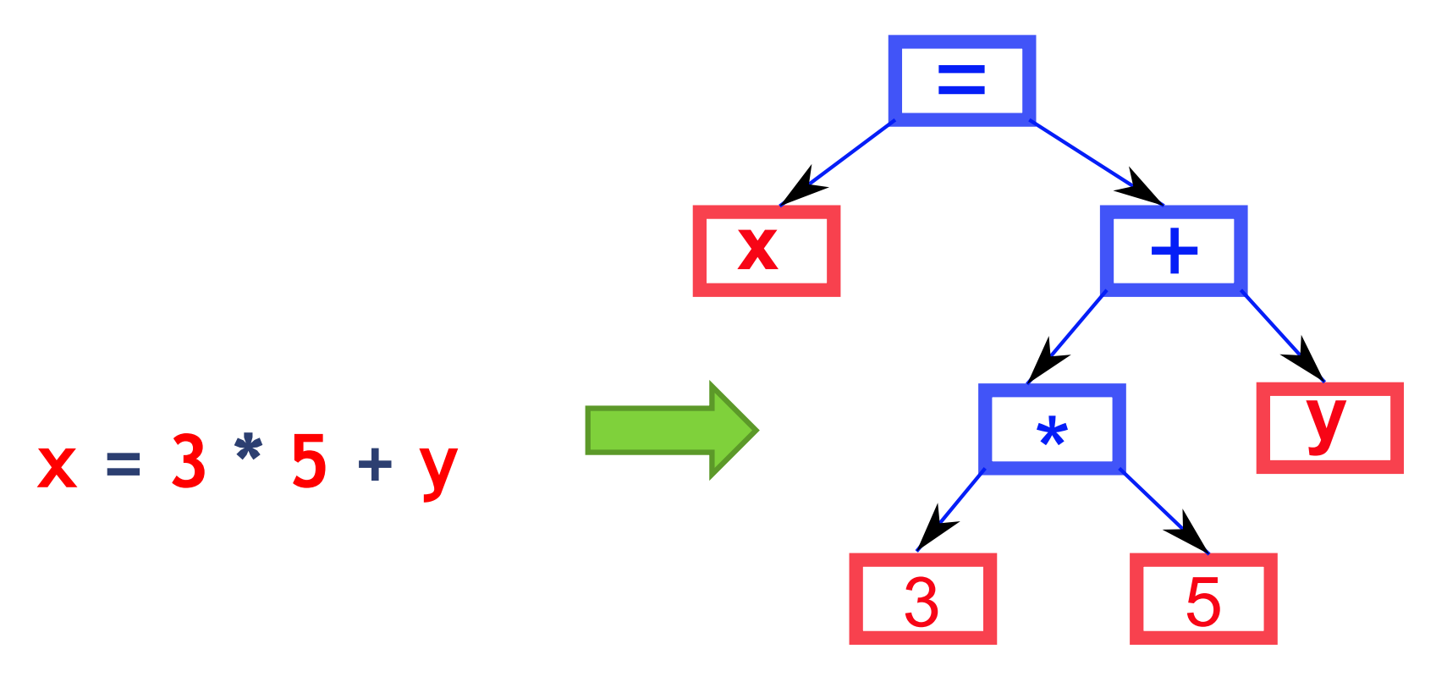
PARSING PRIMER

We will introduce some basic concepts which are required to create a grammar for a structured data. Parsing process is two stage process as shown below.



Tokenizer also called as lexer groups characters to produce a stream of representations also called as tokens.

Parser will do Syntactic Analysis of the grammar and produce a syntax tree.



Abstract Syntax Tree

Grammar is a set of restrictions on the alphabet (set of characters) for a specific language (set of strings). Formally grammar is a tuple of four elements.

G = (N,T,P,S) where

T:- Terminals e.g. {a,b}

S:- Starting Symbol from first production e.g. S

P:- Productions also called as rules e.g. S->aX

N:- non terminals or variables {S, X}

Grammar	Parsing of "ab"
S -> a X	"ab"
X -> b	S -> a X -> ab

A language-agnostic parser generator

There are many online parser generators. We have chosen 'Syntax' [1] as our main parser generator tool. Syntax is language agnostic when it comes to parser generation. The same grammar can be used for parser generation in different languages. Currently Syntax supports JavaScript, Python, PHP, Ruby, C#, Rust, and Java. The target language is determined by the output file extension. [1]

Syntax supports Yacc/Bison and JSON notations to define the grammars. Syntax also supports several LR parsing modes: LR(0), SLR(1), LALR(1), CLR(1) as well LL(1) mode. More details on this can be found at [2].

LR (Left Recursive) parsing, and its most practical version, the LALR(1), is widely used in automatically generated parsers. For out representation we will be using LALR(1).

Following command was used to generate the AST. "test.bnf" is our grammar file and "test.lt" is our actual data to be parsed.

```
[mbhati]$ syntax-cli --grammar grammar/test.bnf --mode LALR1 --file __tests__/test.lt
```

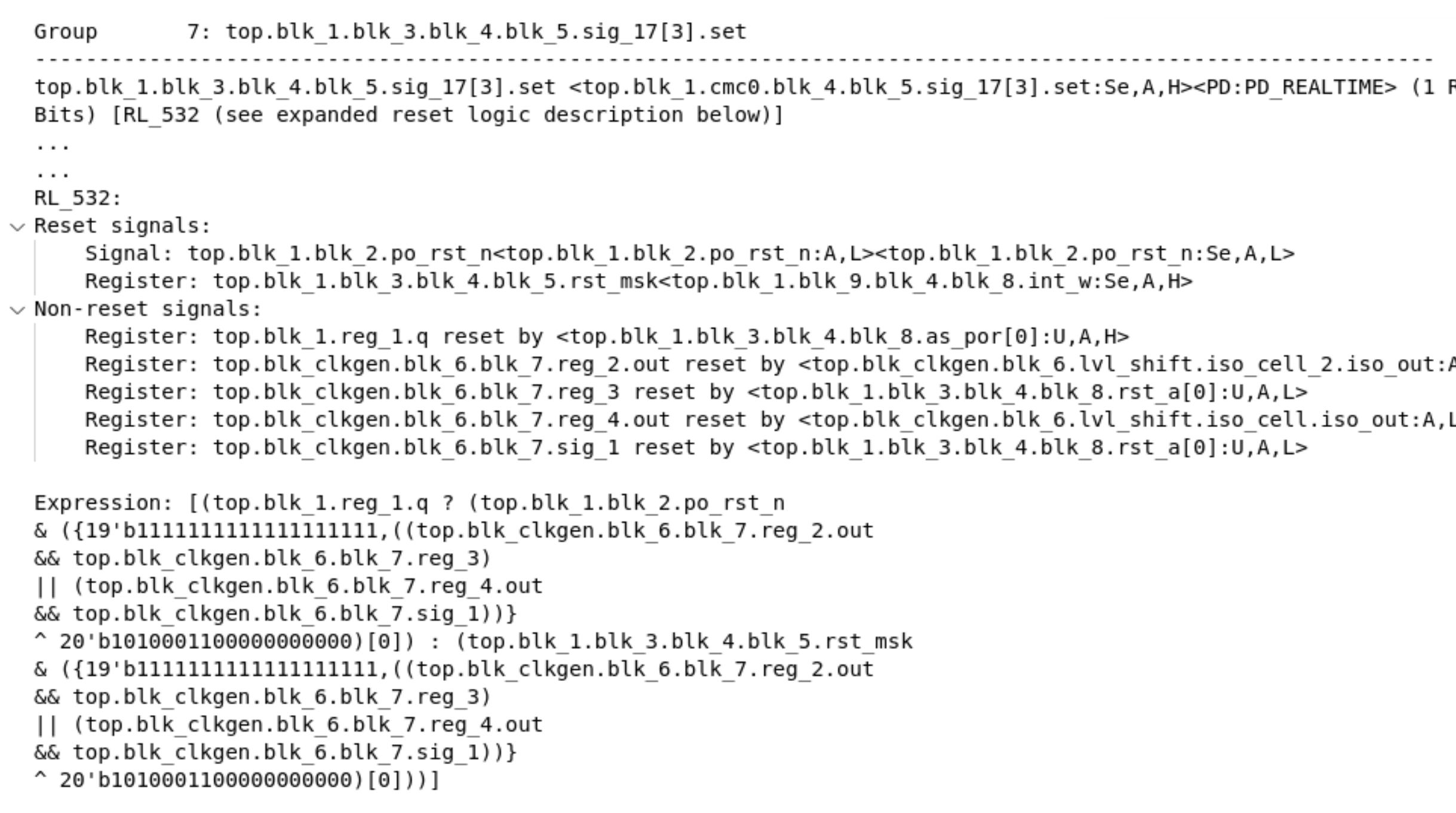
To generate the parser in Python language following command was used

```
[mbhati]$ syntax-cli --grammar grammar/test.bnf --mode LALR1 –o my_parser.py
```

The generated python file can be now used as a plugin in our main python file.

An Example Data from Tool Report

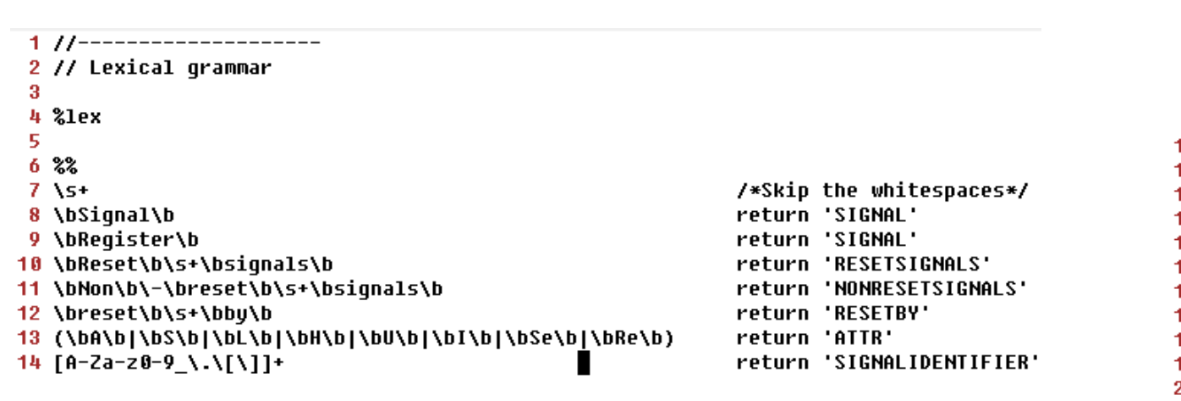
Consider the following data section from an RDC run report which depicts a lot about an inferred reset. There is a new inferred reset domain top.blk_1.blk_3.blk_4.blk_5.sig_17[3].set created, and the report also lists the expression responsible for this. The information contains three sections., viz Reset Signals, Non-Reset Signals and Expression. One can parse this information to create a data for further processing of this information. This information can also be used to do an automated root cause analysis on extra inferred reset domains. This is one such use case, but possibilities are endless.



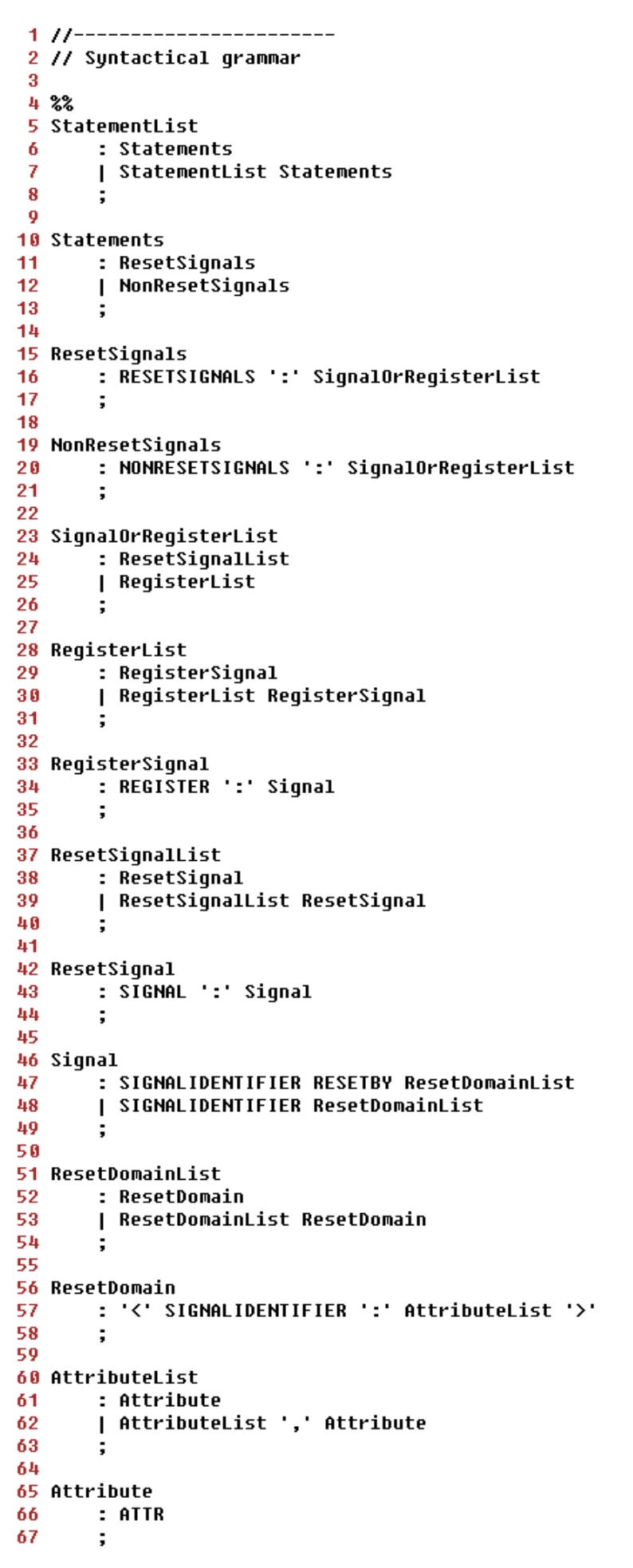
```
Group        7: top.blk_1.blk_3.blk_4.blk_5.sig_17[3].set
----------------------------------------------------------------------------------------------------------
top.blk_1.blk_3.blk_4.blk_5.sig_17[3].set <top.blk_1.cmc0.blk_4.blk_5.sig_17[3].set:Se,A,H><PD:PD_REALTIME> (1 Register Bits, 0 Latch
Bits) [RL_532 (see expanded reset logic description below)]
...
...
RL_532:
Reset signals:
    Signal: top.blk_1.blk_2.po_rst_n<top.blk_1.blk_2.po_rst_n:A,L><top.blk_1.blk_2.po_rst_n:Se,A,L>
    Register: top.blk_1.blk_3.blk_4.blk_5.rst_msk<top.blk_1.blk_9.blk_4.blk_8.int_w:Se,A,H>
Non-reset signals:
    Register: top.blk_1.reg_1.q reset by <top.blk_1.blk_3.blk_4.blk_8.as_por[0]:U,A,H>
    Register: top.blk_clkgen.blk_6.blk_7.reg_2.out reset by <top.blk_clkgen.blk_6.lvl_shift.iso_cell_2.iso_out:A,L>
    Register: top.blk_clkgen.blk_6.blk_7.reg_3 reset by <top.blk_1.blk_3.blk_4.blk_8.rst_a[0]:U,A,L>
    Register: top.blk_clkgen.blk_6.blk_7.reg_4.out reset by <top.blk_clkgen.blk_6.lvl_shift.iso_cell.iso_out:A,L>
    Register: top.blk_clkgen.blk_6.blk_7.sig_1 reset by <top.blk_1.blk_3.blk_4.blk_8.rst_a[0]:U,A,L>

Expression: [(top.blk_1.reg_1.q ? (top.blk_1.blk_2.po_rst_n
& ({19'b1111111111111111111,((top.blk_clkgen.blk_6.blk_7.reg_2.out
&& top.blk_clkgen.blk_6.blk_7.reg_3)
|| (top.blk_clkgen.blk_6.blk_7.reg_4.out
&& top.blk_clkgen.blk_6.blk_7.sig_1))}
^ 20'b10100011000000000000)[0]) : (top.blk_1.blk_3.blk_4.blk_5.rst_msk
& ({19'b1111111111111111111,((top.blk_clkgen.blk_6.blk_7.reg_2.out
&& top.blk_clkgen.blk_6.blk_7.reg_3)
|| (top.blk_clkgen.blk_6.blk_7.reg_4.out
&& top.blk_clkgen.blk_6.blk_7.sig_1))}
^ 20'b10100011000000000000)[0]))]
```

Grammar and Abstract Syntax Tree (JSON Data)

Tokenizer and Grammar for Parsing Reset Signal and Non-Reset signals data

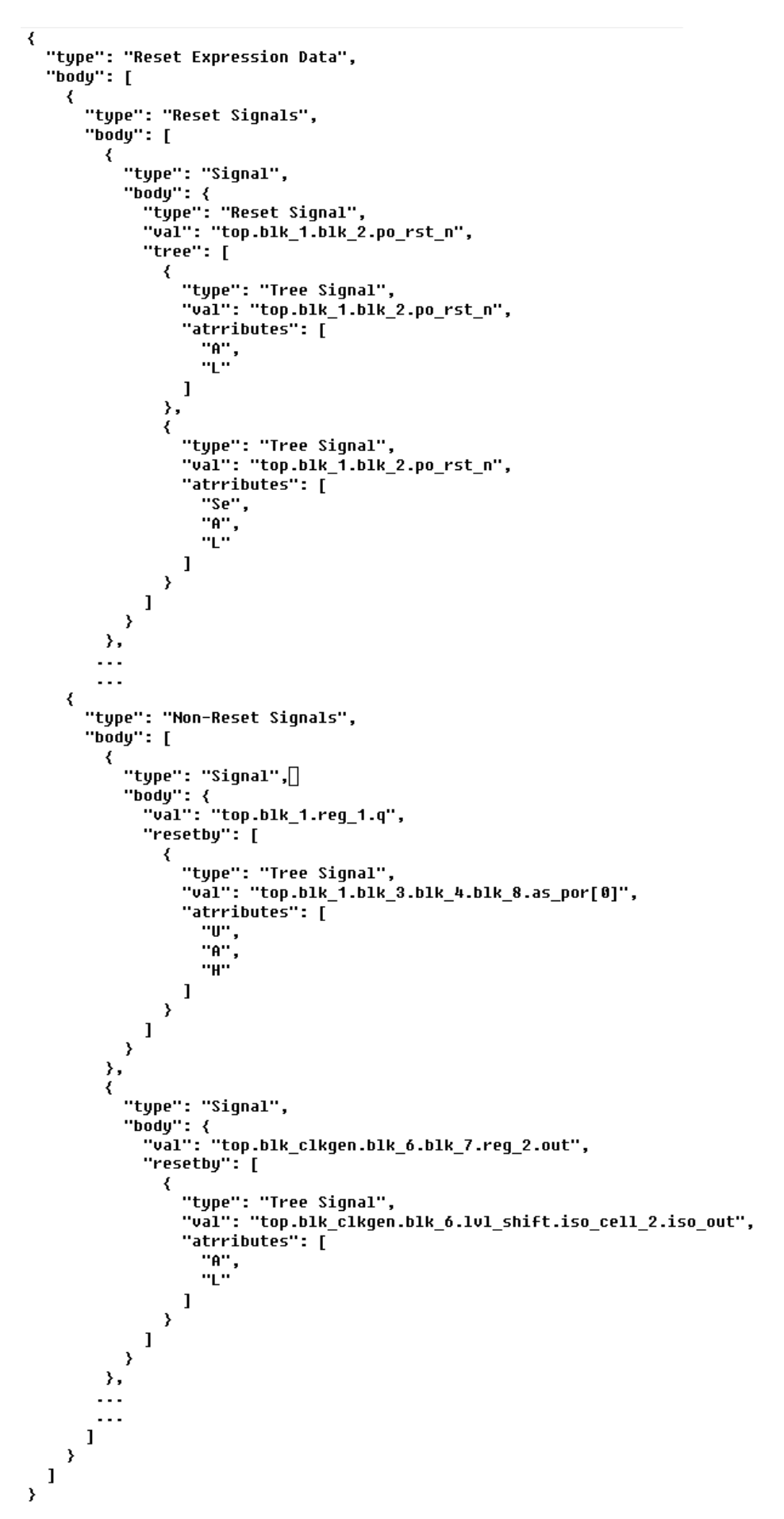


```
//--------------------
// Lexical grammar

%lex

%%
\s+                                                 /*Skip the whitespaces*/
\bSignal\b                                          return 'SIGNAL'
\bRegister\b                                        return 'SIGNAL'
\bReset\b\s+\bsignals\b                             return 'RESETSIGNALS'
\bNon\b\-\breset\b\s+\bsignals\b                    return 'NONRESETSIGNALS'
\breset\b\s+\bby\b                                  return 'RESETBY'
(\bA\b|\bS\b|\bL\b|\bH\b|\bU\b|\bI\b|\bSe\b|\bRe\b) return 'ATTR'
[A-Za-z0-9_\.\[\]]+                                 return 'SIGNALIDENTIFIER'
```



```
//----------------------
// Syntactical grammar

%%
StatementList
    : Statements
    | StatementList Statements
    ;

Statements
    : ResetSignals
    | NonResetSignals
    ;

ResetSignals
    : RESETSIGNALS ':' SignalOrRegisterList
    ;

NonResetSignals
    : NONRESETSIGNALS ':' SignalOrRegisterList
    ;

SignalOrRegisterList
    : ResetSignalList
    | RegisterList
    ;

RegisterList
    : RegisterSignal
    | RegisterList RegisterSignal
    ;

RegisterSignal
    : REGISTER ':' Signal
    ;

ResetSignalList
    : ResetSignal
    | ResetSignalList ResetSignal
    ;

ResetSignal
    : SIGNAL ':' Signal
    ;

Signal
    : SIGNALIDENTIFIER RESETBY ResetDomainList
    | SIGNALIDENTIFIER ResetDomainList
    ;

ResetDomainList
    : ResetDomain
    | ResetDomainList ResetDomain
    ;

ResetDomain
    : '<' SIGNALIDENTIFIER ':' AttributeList '>'
    ;

AttributeList
    : Attribute
    | AttributeList ',' Attribute
    ;

Attribute
    : ATTR
    ;
```

Abstract Syntax Tree Created (Shown for some signals only)



```
{
  "type": "Reset Expression Data",
  "body": [
    {
      "type": "Reset Signals",
      "body": [
        {
          "type": "Signal",
          "body": {
            "type": "Reset Signal",
            "val": "top.blk_1.blk_2.po_rst_n",
            "tree": [
              {
                "type": "Tree Signal",
                "val": "top.blk_1.blk_2.po_rst_n",
                "atrributes": [
                  "A",
                  "L"
                ]
              },
              {
                "type": "Tree Signal",
                "val": "top.blk_1.blk_2.po_rst_n",
                "atrributes": [
                  "Se",
                  "A",
                  "L"
                ]
              }
            ]
          }
        },
       ...
       ...
    {
      "type": "Non-Reset Signals",
      "body": [
        {
          "type": "Signal",
          "body": {
            "val": "top.blk_1.reg_1.q",
            "resetby": [
              {
                "type": "Tree Signal",
                "val": "top.blk_1.blk_3.blk_4.blk_8.as_por[0]",
                "atrributes": [
                  "U",
                  "A",
                  "H"
                ]
              }
            ]
          }
        },
        {
          "type": "Signal",
          "body": {
            "val": "top.blk_clkgen.blk_6.blk_7.reg_2.out",
            "resetby": [
              {
                "type": "Tree Signal",
                "val": "top.blk_clkgen.blk_6.lvl_shift.iso_cell_2.iso_out",
                "atrributes": [
                  "A",
                  "L"
                ]
              }
            ]
          }
        },
       ...
       ...
      ]
    }
  ]
}
```

Conclusion

We have tried to show how a complex structured data can be easily parsed using a properly designed grammar. We used 'Syntax' a language-agnostic parser generator which can generate parser code in the language of choice. With the grammar created we can create abstract syntax tree. Our AST simply contains the reset signal data in JSON format. JSON is a popular data format and can be easily parsed using language APIs.

In this experiment we were data mining deep reset inferences which were hard to identify in schematic and hard to manually refer the reports. The JSON representation of the data can help in doing machine learning and finding the root cause of too many reset inferences.

Parsers can also be hand coded but using an automated parser generator is a faster method to create a parser.

REFERENCES

[1] GIT Repository for Syntax, https://github.com/DmitrySoshnikov/syntax

[2] Syntax: language agnostic parser generator, https://dmitrysoshnikov.medium.com/syntax-language-agnostic-parser-generator-bd24468d7cfc#.1xmztsx3k