I. INTRODUCTION

Some 40 years ago Knuth [1] extended Context Free Grammars (CFG) by allowing the addition of context-sensitive properties and thus introducing the appealing formalism of Attribute Grammars (AG). Specifically, semantic rules and attributes were added to CFGs augmenting their expressional capabilities and therefore offering many advantages in the domain of language specification, analysis and translation. AGs allow high-level context-sensitive properties of individual constructs in a language to be described in a declarative way and to be automatically computed for any program in the language. The primary field of AG usage is in computer languages [4] but they are also convenient in fields such as Artificial Intelligence [5], [6], Pattern Recognition or even Biomedicine [2]. Regardless the field of application, in an AG, knowledge is represented using syntactic and semantic (attribute evaluation rules) notation. For the first (syntax) part, a parser is responsible for the recognition of the syntax and the construction of the parse tree as well. For the second part, that of the tree traversal and semantic evaluation, referred in the following as tree decoration, an evaluator is needed. The evaluator traverses the parse tree, which is covered with applicable rules in each node and executes the corresponding actions, associating attribute values in each node. Typically, the values of an attribute can be defined in terms of values of other attributes.

Attribute grammar evaluation is an operation that is usually divided into two subparts, the syntactic and the semantic. Concerning the syntactic analysis, one significant factor that can positively influence the performance of the CFG parser is undoubtedly the selected parsing algorithm. Two well-known parsing algorithms for general CFGs are the Earley’s algorithm [12] and the Cocke-Younger-Kassami (CYK) algorithm [20]. Both of them are basically a dynamic programming procedure and have a time complexity $O(n^3|G|)$, where $n$ is the length of the input string and $|G|$ is the size of the grammar. A closer look on the previously mentioned parsing algorithms shows that there are some strong connections [16] between the two. After the introduction of Earley’s and CYK algorithms, several modifications [16] and improvements via parallelization [13], [17], [18], [19] have been proposed for these algorithms. Chiang & Fu [13] and Cheng & Fu [17] have presented designs using VLSI arrays for the hardware implementation of the aforementioned parsing algorithms, although they do not propose an efficient implementation for the operator they use, while Ibbara [18] and Ra [19] presented software implementations running on parallel machines. All these approaches are not implemented in reconfigurable hardware and the scale of the hardware is input string length dependent. The hardware oriented approach was reinvigorated by presenting implementations in reconfigurable FPGA (Field Programmable Gate Arrays) boards of the CYK algorithm (Ciressan [21] and Bordim [22]) and Earley’s algorithm (Pavlatos [23]). The early architectures, proposed in [21] and [22], either fail to fully exploit the available parallelization of the parsing algorithms or demand excessive storage. Whereas the software approaches by Ibbara [18] and Ra [19], execute parts of the parsing algorithms sequentially and, thus, do not achieve the maximum possible speed-up. On the other hand, existing hardware methodologies must overcome the complexity imposed by the operations of the parsing algorithms, something that leads to increased storage needs. In order to relax the hardware complexity, most of the proposed architectures implement the CYK algorithm, whose basic operations are much simpler than those of Earley’s. The first FPGA implementation of Earley’s algorithm was given in [23]. The approach proposed in [7] uses a combinatorial circuit for the fundamental operation of Earley’s algorithm. In this manner, an decrease in time by a factor of one to two orders of magnitude is achieved, compared to previous hardware implementations [23], depending on the size of the grammar and the input string length. The speed-up,
II. THEORETICAL BACKGROUND

In order to provide a better perception of the implemented system and make the paper self-contained, some extra background knowledge must be given.

An Attribute Grammar (AG) is based upon a Context Free Grammar (CFG) and therefore the CFG definition follows. A CFG [4] is a quadruple $G = \{N,T,R,S\}$, where $N$ is the set of non-terminal symbols, $T$ is the set of terminal symbols, $R$ is the set of grammar rules and $S$ ($S \in N$) is the start symbol - the root of the grammar. $V = N \cup T$ is called vocabulary of the grammar. Grammar rules are written in the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$. Capital letters $A, B, C, \ldots$ denote non-terminal symbols, lowercase $a, b, c, \ldots$ terminal symbols, Greek lowercase $\alpha, \beta, \gamma, \ldots$ strings of terminals and non-terminals and $\lambda$ is the null string.

An AG is also a quadruple $AG = \{G, A, SR, d\}$, where $G$ is a CFG, $A = \cup A(X)$ where $A(X)$ is a finite set of attributes associated with each symbol $X \in V$. Each attribute represents a specific context-sensitive property of the corresponding symbol. The notation $X.a$ is used to indicate that attribute $a$ is an element of $A(X)$. $A(X)$ is partitioned into two disjoint sets; the set of synthesized attributes $AS(X)$ and the set of inherited attributes $AI(X)$. Synthesized attributes $X.s$ are those whose values are defined in terms of attributes at descendant nodes of node $X$ of the corresponding decorated parse tree. Inherited attributes

The proposed architecture has been tested for various AGs and the outcome was more than encouraging; opposed to software approaches the speed-up\(^2\) is two to three orders of magnitude, while the execution time of our implementation is about one order of magnitude lesser, when software runs on contemporary fast processors that run at extremely higher frequency. To the best of our knowledge this is the first attempt to implement hardware attribute grammar evaluators in its most general form of multi-pass evaluation using both inherited and synthesized attributes.

The proposed architecture has been tested for various AGs and the outcome was more than encouraging; opposed to software approaches the speed-up\(^2\) is two to three orders of magnitude, while the execution time of our implementation is about one order of magnitude lesser, when software runs on contemporary fast processors that run at extremely higher frequency. To the best of our knowledge this is the first proposed implementation of a multi-pass attribute evaluator in hardware.

Footnotes:
1. Look Ahead Left to right, Rightmost derivation parser with 1 lookahead symbol
2. The performance comparison is based on the clock cycles needed for the execution, consequently the speed-up applies to the ratio of the corresponding clock cycles consumed.
X,i are those whose values are defined in terms of attributes at the parent and (possibly) the left sibling nodes of node X of the corresponding decorated parse tree. The start symbol does not have inherited attributes. Each of the productions \( p \in R(p : X_0 \rightarrow X_1X_2\ldots X_n) \) of the CFG is augmented by a set of semantic rules \( SR(p) \) that define attributes in terms of other attributes of terminals and on terminals appearing in the same production. The way attributes are evaluated depends both on their dependencies to other attributes in the tree and also on the way the tree is traversed. Finally d is a function that gives for each attribute a its domain \( d(a) \).

Two important categories of AGs are S-attributed grammars and L-attributed grammars. S-attributed grammars are those having only synthesized attributes and can be evaluated in one bottom-up pass and thus is good match for LR parsing. Whereas L-attributed grammars are those that apart from synthesized attributes also support inherited attributes. Each inherited attribute depends only on attributes of terminals and on terminals appearing in the same production. The way attributes are evaluated depends both on their dependencies to other attributes in the tree and also on the way the tree is traversed. Finally d is a function that gives for each attribute a its domain \( d(a) \).

Let \( S \rightarrow \alpha, (\alpha \in T^*) \) be a derivation in an AG G. The corresponding derivation (parsing) tree is an ordered tree with root \( S \), leaves the terminal symbols in \( \alpha \), and nodes the rules that are used for the derivation process. The process of analyzing a string for syntactic correctness is known as parsing. An algorithm that decides whether or not a string \( a_1a_2a_3\ldots a_n \) (of length n) can be generated from a grammar \( G \) is called recognizer. If simultaneously the derivation (or parse) tree is constructed, the algorithm is called parser. In each node of the parse tree correspond certain attributes, both inherited and synthesized. After the construction of the parse tree, the latter is traversed and the attribute values are evaluated. The corresponding parse tree that also contains the attribute values, is called decorated parse tree and its production is the final goal of every attribute evaluation system. In case of multi-pass attribute grammars, the parse tree should be traversed more than one time in order to correctly evaluate the attributes.

### III. System Description

The proposed system is consisting of three major modules: the parallel parser, the tree constructor and the multi-pass attribute evaluator (see figure 1). The parser handles the recognition task and constructs the parse table, based on the given input string. When the parsing process is over, the parse tree is constructed and afterwards, while being traversed, the corresponding attributes are evaluated. It must be clarified that the first two modules may automatically be produced, based on the syntactic part of the given AG as well as the generic architecture for the multi-pass attribute handling, which is independent to the grammar. If complex functions are dictated by the semantics, special submodules are required. Therefore, these submodules cannot yet be produced automatically but have to be application specific functions provided by the user. In case semantic rules are given by the user described in Verilog, the whole system can be automatically generated, as shown in figure 1. Usually in the process of analyzing large input strings, the input string is divided into smaller strings of reasonable length. Therefore, a future aspect we are currently working on, is the application of pipeline techniques between the three major modules, when long input string have to be divided and executed in real time, which will lead into a further increase of performance. Obviously, the implementation using the pipelined techniques can also be utilized in applications where successive independent input strings are to be analyzed.

#### A. The Parallel Parser

In 1970 Earley [12] presented a top-down parser, whose basic innovation was the introduction of a symbol called dot “·” that does not belong to the grammar. The utility of the dot in a rule (now called dotted rule) is to separate the right part of the rule into two subparts. For the subpart on the left of the dot, it has been verified that it can generate the input string (or substring) examined so far. However, for the subpart on the right of the dot, it still remains to be checked whether or not it can generate the rest of the input string. When the dot is at the last position, the dotted rule is called completed. Prior to reading any input symbol, the dotted rules are in the form \( A \rightarrow \cdot \alpha \). As the reading of the input symbols commences, new dotted rules are created. If after reading the last input symbol a dotted rule of the form \( S \rightarrow \alpha \cdot \) exists, then the input string is
a sentence of the grammar. The way in which dotted rules are created, during the parsing of the input string, can be efficiently formulated by the operation $\otimes$ that was first introduced by Chiang & Fu [13]. This is a binary operation that takes as inputs either two sets of dotted rules or a terminal symbol and a set of dotted rules and produces a new set of dotted rules. In this way, Chiang & Fu’s [13] version of Earley’s [12] parallel parsing algorithm decides whether or not an input string $a_1a_2\ldots a_n$ of length $n$ is a sentence of a CFG $G$ by constructing a $(n+1) \times (n+1)$ parsing table $PT$ whose elements $pt(i,j)$ are sets of dotted rules.

In [7] a highly efficient architecture for the hardware implementation of CFG parsers was presented. Its efficiency stems from an innovative combinatorial circuit that implements the fundamental operation $\otimes$ in time complexity $O(\log_2 |G|)$, where $|G|$ is the size of the CFG. As shown in figure 2(a) PT cells are computed in parallel by applying the operation $\otimes$, which is now implemented by the proposed combinational circuit $C_{\otimes}$. $C_{\otimes}$ is constructed from the “characteristic” equations of the underlying CFG $G$, which are algorithmically derived. The computation of $pt(i,j)$ is based on the equation shown in figure 2(b). All $\otimes$ operations can be executed in parallel by multiple $C_{\otimes}$ circuits and the union of their outputs produces $pt(i,j)$ (see figure 2(c)). Hence, two levels of parallelism can be identified: a local or cell-level that corresponds to the parallel execution of the $\otimes$ operations inside each cell, and a global or architecture-level that corresponds to the tabular form of Earley’s algorithm. The proposed design achieves recognition of an input string of length $n$ in $O(n \log_2(n|G|))$ time. Taking also into consideration the hardware nature of the implementation, the architecture presented in [7] achieves a speed-up factor that varies from two orders of magnitude for toy-scale grammars to six orders of magnitude for large real life grammars compared to software approaches.

B. The Tree Constructor

In [3] certain modifications were made into the direction of converting the parallel parser presented in [7] to a suitable parser for attribute evaluation. The parse tree is a necessity for the attribute evaluation and hence the combinatorial parser must construct it. In case of ambiguous grammars, all possible parse trees should be constructed, in order that the implementation is applicable in numerous fields such as artificial intelligence, hardware compilers and pattern recognition. As it can easily be comprehended, the parser should keep its combinatorial nature, that grants the great speed-up. Therefore, all changes were made while bearing in mind this precondition.

The most essential extension is the storage for every created dotted rule of its origin. Due to the nature of the $\otimes$ operator, a dotted rule can be the result of an operation between two dotted rules or between a dotted rule and one terminal symbol of the input string. Thus, for every dotted rule a maximum number of two origins is stored, i.e. the cells which contain the dotted rules that created the new dotted rule. These data are required for proper placement of the rules in the parse tree at the tree construction phase and are encoded into a bit-vector, called source.

The parser in every execution step calculates the dotted rules for the cells that belong to the same diagonal, as shown in figure 2(a). At the end of each execution step, a whole column of the parse table is filled and therefore these data can be transmitted to the next module, the parse tree constructor, as shown in figure 1. It must be clarified that due to the parallel nature of the parser, at each execution step, all possible rules are created and stored into the parse table. Thus the parse table contains numerous dotted rules, many of which are not even needed for the derivation of the given input string. Furthermore, for the parse tree construction only completed dotted rules are used and therefore only those are transmitted from the parser module to the tree constructor. The separation of the completed dotted rule is carried extremely easy and fast by the usage of a bit-vector mask, that has set only the bits that correspond to the position of the completed dotted rules.

Once the whole input string is read by the parser, i.e. the final column of the parse table is filled and transmitted to the next
module, the parse tree construction may begin. The entire operation starts from the top right cell of the parse table, where the starting rule of the grammar is sought. Once found, it is placed into the root of the parse tree and the rules responsible for the creation of the start rule are fetched, via the usage of the source bit-vector. The source vector may have stored either two or one cell position, depending on whether the operands of ⊗ were just dotted rules or a dotted rule and a terminal symbol. For every rule found, the same procedure is followed recursively until a rule belonging to the parse table major diagonal is reached, which denotes the end of the particular branch and therefore the node is degenerated to a leaf. In this way, branch by branch the whole parse tree is constructed. If a completed dotted rule derives from more than one ⊗ operations, i.e. it has more than one origins, obviously the grammar is ambiguous and more than one parse trees should be constructed. For that reason, the process forks for each of the stored origins.

A detail that needs to be elucidated is the way useful completed rules are separated from others existing in the same cell and hence in the same bit-vector. In order to preserve the combinatorial nature of the parser, bit-vector masks are used for every rule. These masks are created based on the concept that a parent node in the parse tree has children nodes that have as left hand side symbol (lhss), of the corresponding rule, nonterminals that exist in the right hand side symbol (rhss) of the parent node. To clarify these concept, we consider the rule A → BC at the parent node. The rules at the children node can only have as lhss the nonterminal B or C and not any other. Therefore, the appropriate mask for this rule would allow only rules with the specific lhss. Because of the bit-vector encoding followed, these masks can easily be built and applied with the aid of AND gates. In case there are more than two nonterminals at the right hand side of the rule, still two masks are utilized, one for the rightmost nonterminal and one for all the others.

The construction of all the masks is an unsophisticated task that is carried out automatically together with the entire system construction. It must be noted that the usage of masks does not increase the parser complexity, since it only adds the need for some extra AND gates and consequently the parser retains its combinatorial nature and extreme performance. More details will be given later with examples.

C. Illustrative Example of Syntactic Analysis and Tree Construction

In order to further describe the methodology of syntactic analysis and tree construction, an illustrative example is given based on the AG \( G_{op} \) (see Table I(a)), which describes the basic arithmetic operations of addition and multiplication between two or more operands. The produced system can recognize input strings that describe arithmetic operations and furthermore evaluate the final result, resembling to a common (infix notation) calculator.

### Table I

<table>
<thead>
<tr>
<th>Rule Number</th>
<th>Syntax Rule</th>
<th>Semantic Rule in AG notation</th>
<th>Index</th>
<th>Rules</th>
<th>Descendants</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S \rightarrow E )</td>
<td>( S_E \rightarrow E_s )</td>
<td>0</td>
<td>0,2,3</td>
<td>1.2</td>
</tr>
<tr>
<td>1</td>
<td>( E_1 \rightarrow E + E_2 )</td>
<td>( E_{1s} = T_s + E_{2s} )</td>
<td>1</td>
<td>1.6,7</td>
<td>3.4</td>
</tr>
<tr>
<td>2</td>
<td>( E \rightarrow T )</td>
<td>( E_s = T_s )</td>
<td>2</td>
<td>1.3,8,11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( T \rightarrow F \times T )</td>
<td>( T_s = F_s \times T_s )</td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( T \rightarrow F )</td>
<td>( T_s = F_s )</td>
<td>4</td>
<td>3,14</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( F \rightarrow (E) )</td>
<td>( F_s = E_s )</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( F \rightarrow N )</td>
<td>( F_s = N_s )</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( N \rightarrow DN )</td>
<td>( N_s = 10 \times E_s + N_s )</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( N \rightarrow D )</td>
<td>( N_s = D_s )</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( D \rightarrow 0 )</td>
<td>( D_s = 0 )</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( D \rightarrow 9 )</td>
<td>( D_s = 9 )</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the expression 35 * 4 is given as input to the system, the first step is the construction of the parse table by the extended parser, which is shown in figure 3(a), where the main diagonal is not shown, since it contains initial rules that have been precomputed. The representation in the figure is not in bit-vector, for simplicity. Actually, the data are represented in bit-vectors of length 45, imposed by the possible number of dotted rules that can be derived from the 19 syntax rules of the grammar. This parse table contains both completed and not completed dotted rules, therefore the not completed rules must be screened out using the mask that has set only the bits that correspond to the position of the completed dotted rules. This mask is automatically produced as explained in section III-B.

After the mask application, the next step is to create the parse tree (see figure 3(b)), i.e. to keep only the useful completed rules. For this process, two sets of masks have been created for each syntax rule, that contains at least one nonterminal on the right hand side, i.e. the first 9 rules of the grammar. These masks are automatically constructed based on the methodology explained in section III-B. Having kept only these necessary rules, the parse tree is stored in the table form shown in table I(b). The first column is merely the index of the table, while the second column contains the rules and the third the descendant subtree (referring to the corresponding index in the same table). For example, the descendants of rule 3 in the tree of figure 3(b) are stored in index position 1 (rules 6 and 7) and 2 (rules 4, 6, 8 and 13), for the left and right branch respectively.
the one-pass attribute evaluation of that grammar, the tree should be traversed top-down and from left to right, as shown by the dashed line in figure 3(b).

D. Multi-pass Attribute Evaluation

In [3] a stack-based methodology was proposed that allows to achieve the attribute evaluation mainly by the usage of simple push and pop commands on specific stacks. Unfortunately this approach cannot be applied to multi-pass attribute grammars. In case of multi-pass attribute grammars, the parse tree should be traversed more than one time in order to correctly evaluate the attributes. Let us consider as an example the translation of conditional statement. The syntactic and semantic rules of that grammar is shown in Table II, where the terminal symbols are shown in quotation marks (i.e “ELSE”). We associate with each non-terminal symbol two synthesized attributes code and follow and one inherited start. The attribute otheralt is associated only with symbol CH. In table II attributes are presented as subscripts using letters c, f, s and o respectively (i.e $CS_c$ is the attribute code of symbol CS). The code begins from memory location start and the last location it occupies is follow -1. The value of otheralt is either the start address of the “ELSE” part or the address of the following statement. CON stands for concatenation, INCR for increase the value of the parameter by 1 and & means line transfer operation.

### Table II

**IF STATEMENT GRAMMAR $G_{ij}$**

<table>
<thead>
<tr>
<th>Rule Number</th>
<th>Syntax Rule</th>
<th>Semantic Rule in AG notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>GEN $\rightarrow$ CS</td>
<td>GEN$C_c$ = CON($CS_c$, &quot;$\text{END}$&quot;) $CS_c = 1$</td>
</tr>
<tr>
<td>1</td>
<td>CS $\rightarrow$ CH &quot;$\text{ELSE}$&quot; BS</td>
<td>$CS_f = \text{CON}(CS_c, &quot;$\text{BRANCH}$&quot;, CS_f, BS_c)$ $CS_f = BS_f, CH_c = CH_c, BS_c = \text{INCR}(BS_c), CH_c = BS_c$</td>
</tr>
<tr>
<td>2</td>
<td>CS $\rightarrow$ CH</td>
<td>$CS_c = CH_c, CS_t = CH_t, CH_c = CH_c, CH_o = CH_f$</td>
</tr>
<tr>
<td>3</td>
<td>CH $\rightarrow$ ICf US</td>
<td>$CH_c = \text{CON}(IC_c, &quot;$\text{BRANCHONFALSE}$&quot;, CH_o, &quot;$\text{US}$&quot;), $US_f = U S_f, IC_c = CH_c, US_o = \text{INCR}(IC_c)$</td>
</tr>
<tr>
<td>4</td>
<td>IC $\rightarrow$ IFf &quot;THEN&quot;</td>
<td>$IC_c = BS_c, IC_f = BS_f, BS_f = BS_c$</td>
</tr>
<tr>
<td>5</td>
<td>BS $\rightarrow$ CS</td>
<td>$BS_c = BS_c, BS_f = BS_f, BS_o = BS_o$</td>
</tr>
<tr>
<td>6</td>
<td>BS $\rightarrow$ &quot;B_1&quot;</td>
<td>$BS_f = &quot;$B_1&quot;, BS_{1f} = \text{INCR}(BS_{1f})$</td>
</tr>
<tr>
<td>7</td>
<td>BS $\rightarrow$ &quot;B_2&quot;</td>
<td>$BS_f = &quot;$B_2&quot;, BS_{2f} = \text{INCR}(BS_{2f})$</td>
</tr>
<tr>
<td>8</td>
<td>US $\rightarrow$ &quot;S_1&quot;</td>
<td>$US_f = &quot;$S_1&quot;, US_{1f} = \text{INCR}(US_{1f})$</td>
</tr>
<tr>
<td>9</td>
<td>US $\rightarrow$ &quot;S_2&quot;</td>
<td>$US_f = &quot;$S_2&quot;, US_{2f} = \text{INCR}(US_{2f})$</td>
</tr>
</tbody>
</table>

In figure 4(a) the parse tree for the input string “IF $B_1$ THEN $S_1$ ELSE IF $B_2$ THEN $S_2$" is shown. The evaluation of the attributes is also shown, according to the semantic rules. Each non-terminal symbol has its associated attributes near by. The value of the attribute code for the non-terminal GEN is the translation of the input string. Following figure 4(a), we can see that this value is: “$B_1 \text{ & BRANCH ON FALSE} 5 \ & S_1 \ & \text{BRANCH 8 } \ & B_2 \ & \text{BRANCH ON FALSE} 8 \ & S_2 \ & \text{END}$”. It
is obvious, from the arrows that show the progress of the attribute evaluation, that the parse tree should be traversed multiple times.

In order to handle the multi-pass attribute evaluation, the third subpart (see figure 1) of our system scan the parse tree as many times as it is required to evaluate the value of all attribute instances according to the flowchart of figure 4(b). Initially, the parse tree, that is stored in a table of the form of figure I(b), is augmented keeping the values of the attribute instances. Memory space is kept for each node of the parse tree and of each attribute. The attribute instances for which memory should be kept at our example are represented in figure 4(a) as square white boxes. Moreover, for each possible semantic action a hardware module is generated to carry out the attribute evaluation. The semantic actions as well as the input values for each semantic action are shown in the decorated parse tree of figure 4(a). The tree is traversed top-down and from left to right, as shown by the dashed line in figure 3(b) of the previous example and the values of the attributes of which the input values are available are evaluated. This is repeated until all attributes are evaluated. According to the flowchart of 4(b), if a value of an attribute cannot be evaluated then the value of variable $\text{flag}$ becomes equal to zero and the tree should be traversed again. For example the value of attribute $o_2$ cannot be evaluated during the first pass, in contrast to attribute $s_2$ that can be evaluated. In case semantic actions consist only of simple arithmetic expressions the hardware modules can be generated mechanically. If complex functions are dictated by the semantics, special submodules are required. Therefore, these submodules cannot yet be produced automatically but have to be application specific functions provided by the user. In case semantic rules are given by the user described in Verilog, the whole system can be automatically generated, as shown in figure 1. The multi-pass attribute evaluator contains hardware modules for all symbols and their semantic actions and as traverse the parse tree select the appropriate each time module and pass to it the correct inputs (values of previously evaluated attributes) in order to evaluate the attributes of the examined symbol.

**IV. CONCLUSION - FUTURE WORK**

This paper presents an innovative architectural design of an AG evaluator. We have designed an architecture, that exploits its characteristics for the hardware implementation of applications, that require multi-pass attribute evaluation to enhance their performance by integrating syntactic and semantic knowledge via AGs. Obviously, in order to construct an efficient attribute evaluation system, care should be given to both syntactic and semantic tasks. For the syntactic part, an efficient parsing algorithm has be chosen, capable of constructing the parse tree needed for the next operation of attribute evaluation. The attribute evaluation takes place on dedicated hardware modules. These modules are designed especially for the execution of the semantic rules necessary to evaluate attributes - both inherited and synthesized and thus are extremely effective. The proposed architecture has been tested for various AGs and the outcome was more than encouraging; opposed to previous approaches the speed-up is one to three orders of magnitude, depending on the implementation, the size of the grammar and the input string length.
This work is a part of a project\textsuperscript{1} for developing a platform (based on AGs) in order to automatically generate special purpose embedded systems. A future aspect, we are currently working on is to handle the multi-pass attribute evaluation using the dataflow computation paradigm [29].

\textbf{REFERENCES}


\textsuperscript{1}This work has been funded by the project PENED 2003. This project is part of the OPERATIONAL PROGRAMME “COMPETITIVENESS” and is co-funded by the European Social Fund (80%) and National Resources (20%).