A generic platform for the SoC implementation of grammar-based applications

Alexandros C. Dimopoulos, Christos Pavlatos, Panagiota Karanasou and George Papakonstantinou
National Technical University of Athens
Department of Electrical and Computer Engineering
Zografou 15773, Athens, Greece
Email: {alexdem, pavlatos, pkaran, papakon}@cslab.ece.ntua.gr

***Abstract*—In this paper a generic platform for the hardware implementation of grammar-based applications is presented. The proposed platform, given the specification of the application in the formalism of Attribute Grammars, automatically produces the necessary hardware modules for the syntactic and semantic analysis of input strings belonging to that grammar. The produced implementation tackles with the recognition task of the input string, using Earley's parallel parsing algorithm. The attribute evaluation makes usage of a stack-based methodology. The hardware modules are described in Verilog Hardware Description Language (Verilog HDL) and synthesized in a Xilinx Virtex-5 ML506 FPGA. Finally, two examples from the area of arithmetic expression evaluation and from the question answering area are given, for the illustration of the proposed system.**

I. Introduction

Attribute grammars (AGs) were introduced by Knuth [1] in 1968. The addition of attributes and semantic rules to Context free grammars (CFG) augmented their expressional capabilities, making them, by this way, a really useful tool for a considerable number of applications. AGs [2] have been extensively utilized in Artificial Intelligence applications [3], [4], [5], structural pattern recognition [6], compiler construction [7], text editing [8] e.t.c.. However, the additional complexity imposed by the added characteristics, dictates the need for hardware solutions of the whole procedure (parsing and attribute evaluation) as an attractive alternative to classical software solutions in order to speed-up the execution time.

In this paper, a generic platform for hardware implementations of grammar based applications is presented. The end-user provides the target application described in AG - both syntax and semantics - and the platform produces the appropriate hardware modules for the recognition of any input string belonging to the specific grammar, as well as the module for the evaluation of its attributes. The recognition task of the input string takes place on a hardware implementation [9] of Earley's parallel parsing algorithm [10], which is now extended so as to allow the evaluation of attributes. Previous attempts were using a general purpose processor, softcore [11] or external [12], for the semantics evaluation. In this paper, the semantics evaluation takes place in a separate special purpose (for the specific application) module in the same chip where the parsing lays (SoC). The output of the produced system, after the evaluation of the semantics, may be executed into another dedicated hardware module in the same chip, as presented in the first example of Section IV. Alternatively, the output code may be executed on an external processor or even on a remote computer as presented in the second example of Section IV. The high computational cost of the parsing task, in conjunction with a possible large amount of input sentences, to be processed simultaneously, justify the hardware implementation of the grammar (syntax and semantics).

The proposed architecture has been implemented in synthesizable Verilog in the Xilinx ISE 9.1 [13] environment, while the generated source has been simulated for validation, synthesized and tested on a Xilinx Virtex-5 ML506 FPGA. The system can be used in applications where a large amount of input data must be processed simultaneously and extract information. The extracted information can be statistics about the preferences of clients, the profiling of the users, intelligent extraction of keywords for web browsers etc.

The rest of the paper is organized as follows. In Section II, the necessary theoretical background for understanding the parsing algorithm as well as the Attribute Grammars is presented. All the necessary extensions to the parser module, which make attribute evaluation feasible, are introduced in Section III. In Section IV, two examples from the area of arithmetic expression evaluation and from the area of question answering, are given for the illustration of the proposed system. Finally, in Section V, we summarize the proposed implementation and present future work.

II. Theoretical Background

A Context Free Grammar [7] (CFG) 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 (a subset of $N \times (N \cup T)*$ written in the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)*$) and S ($S \in N$) is the start symbol (the root of the grammar). We use capital letters $A, B, C, \ldots$ to denote non terminal symbols, lowercases $a, b, c, \ldots$ to denote terminal symbols and Greek lowercases $\alpha, \beta, \gamma, \ldots$ for $(N \cup T)*$ strings, λ is the null string and $V = N \cup T$ is called vocabulary. $A \rightarrow \alpha$ means that α can derive from A after the application of one or more rules. Let $S \rightarrow \alpha$, $(\alpha \in T*)$ be a derivation in G. The corresponding derivation (parsing) tree is

an ordered tree with root S, leaves the terminal symbols in α, and nodes the rules that are used for the derivation process. The process of analyzing a string for syntactic correctness is known as parsing. A parser is an algorithm that decides whether or not a string $a_1a_2a_3 \dots a_n$ (of length n) can be generated from a grammar G and simultaneously constructs the derivation (or parse) tree.

An Attribute Grammar [2] is based upon a CFG. An AG is 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 semantic tree. Inherited attributes $X.i$ are those whose values are defined in terms of attributes at the parent and (possibly) the sibling nodes of node X of the corresponding semantic tree. The start symbol does not have inherited attributes. Each of the productions $p \in R$ ($p : X_0 \rightarrow X_1X_2 \dots X_n$) 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 will be 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). If we limit the computing power to the one of S-attribute grammars [2] then we have "action" grammars. Action grammar, in this paper, is defined as a CFG, in which actions may be incorporated at the end of the rules. These actions (semantics) decorate the parse tree at the corresponding nodes. When the tree is traversed top-down, the actions are executed from left to right, after the syntactic analysis.

III. Overview of our Approach

The proposed implementation follows the architecture shown in Fig. 1, i.e. it is divided into three major components, responsible for the parsing, the parse tree construction and the production and execution of the corresponding actions. The parser handles the recognition task and constructs the parse table. When the parsing process is over, the parse tree is constructed and afterwards, while being traversed, the corresponding actions are send to the execution submodule. This actions, executed by the submodule may lead into the production of an intermediate code executed on an abstract machine (e.g. SQL server, Java VM) or may be executed on a dedicated hardware component, coexisting on the same FPGA board with the rest of the modules. In example 1, presented in Section IV, the dedicated hardware component evaluates arithmetic operations, while in example 2, actions are transformed into SQL queries to be executed on an SQL server.

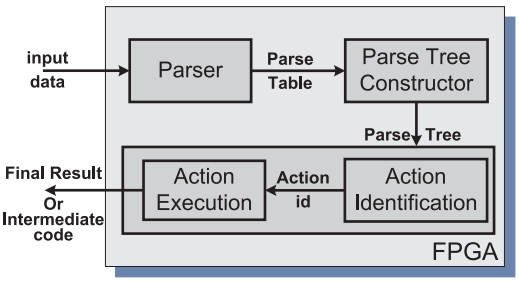

Fig. 1. Overview of our Approach

A. The Parsing Task

The parsing task may be reduced to the procedure of filling a two dimension table (parsing table: pt ()). Chiang & Fu [10] proved that the construction of the parsing table can be parallelized with respect to the length of the input string n, by computing at step k the cells pt(i, j) for which $j - i = k \geq 1$. Only the elements on or above the diagonal are used. In [9], [12] a parallel architecture (see Fig. 2) has been presented that uses $n + 1$ elements $(P_1, P_2, \dots, P_{n+1})$ to compute the parse table in O(n) time where n is the input string length. Every processing element is computing one cell pt(i, j) in each execution time and the next execution time is used again to compute the cell that belongs to the same column and is one row higher pt(i-1, j). In addition one processing element is required to control the whole process and one more to handle the attribute evaluation process. The n elements that are used for the parallel parsing are following the design presented in [9] (see Fig. 3). After the end of each execution step k (t_{ek}), the computation of one parsing processing element terminates. At the next execution step this processing element should transmit the cells that it has computed, to the next processing (t_{ck}). Each processing element repeatedly calculates a cell, checks if it should transmit some cells and then if it should receive any.

More specifically, the parse table produced by the parser

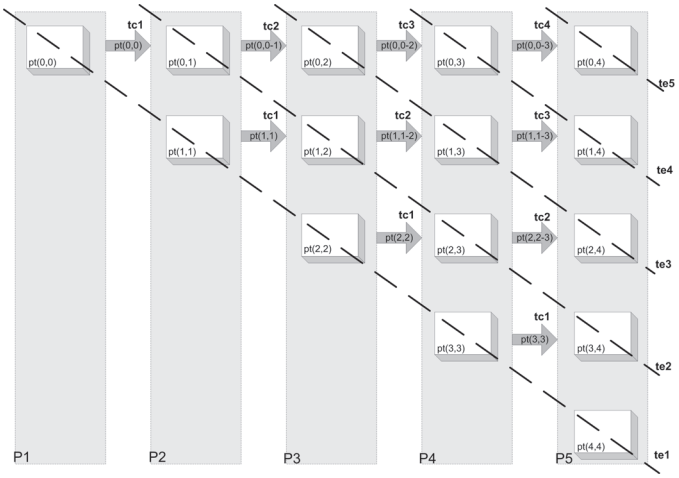

Fig. 2. The parallel architecture for the construction of Parsing Table

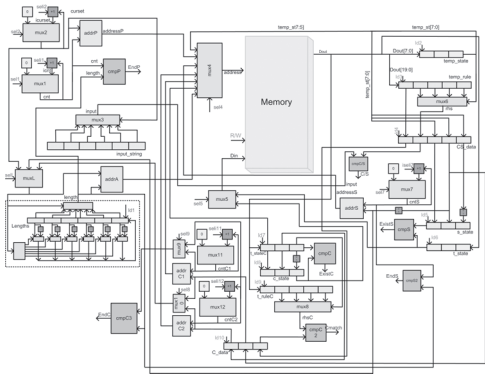

Fig. 3. The architecture of Processing Elements

is filled starting from its most left column and continuing to the right successively. For each entry of the parse table, extra data (source state - SS) must be kept, in order to keep track of the origin of each produced rule, to be used by the next task of parse tree construction. In this way, the hardware implementation [9] of Earley's parallel parsing algorithm, gives a speedup of 2 orders of magnitude compared to conventional software approaches.

B. Constructing the Parse Tree

Based on the abovementioned parse table, the parse tree is exported, using the SS data. Firstly, the root of the parse tree is located and then successively the following node of the tree is located, until the tree is fully constructed. The search of the next element, is based on fields of the examined element that correspond to the origin of each rule - where it was first created or to the state from which it came from - so the parse table is not parsed again, but only backtracking occurs. It is noted that the parse tree is constructed top to bottom and from right to left, as shown in Fig. 4.

C. Action Identification and Execution

The final scope - that of semantics evaluation - is carried out after the parse tree is fully constructed. There is a rule-to-rule correspondence between syntax and semantic rules. Conventional approaches in the evaluation of AGs are based on the construction of the parse tree that is after traversed one or more times, depending on the form of the grammar, in order to evaluate the attributes. In our approach, the parse tree is traversed from bottom-up and from left to right and for every node, the corresponding semantic rule is evaluated. The main idea of the algorithm, is to first evaluate all the subnodes of a node before evaluating the node itself. Furthermore, for every semantic rule, no real operation is made at the evaluation phase but a control signal - an action - is send to be executed later. Therefore, every semantic rule is corresponded to an action. In this way, after the construction of the parse tree, a one-pass attribute evaluation that generates the actions, is carried out by the action identification submodule. These actions are then executed by the action execution submodule, which uses a stack-based approach to cope with the necessary operations. The abovementioned transformation from the AG notation to Actions can easily be automated by the usage of a preprocessor. The needed actions result following the methodology described below:

- For each synthesized attribute, a stack is defined.
- As the tree is traversed, when the last symbol of a rule is reached, the synthesized attributes of the left hand side symbol (lhss) of the rule may be evaluated by unstacking (pop up) the values of the corresponding attributes of the right hand side symbols (rhss). The result, computed based on the semantic rule, is then stored (pushed) into the stack of the synthesized attribute referring to lhss.
- If a semantic rule has a simple transfer form, no action is needed.

The computing power of the proposed system is the one of S-attribute grammar [2]. Yacc [14] has the same computing power, so we could say that the system proposed is a hardware implementation of a Yacc metacompiler. Nevertheless, the parser utilized in our system is much more powerful than the one of Yacc. It is a non-deterministic CFG parser, finding all ambiguous solutions.

IV. Illustrative Examples

In the case of the first illustrative example given below, arithmetic expressions can be described through an AG with the usage of synthesized attributes. The semantic rules have been transformed into simple push and pop actions that are executed on a separate module embedded into the same FPGA board. In the second example, a Natural Language interface is developped, to translate English sentences to SQL queries. This application falls in the area of question-answering. The semantic rules have been transformed into actions, that finally construct SQL queries, send to be executed to an SQL server.

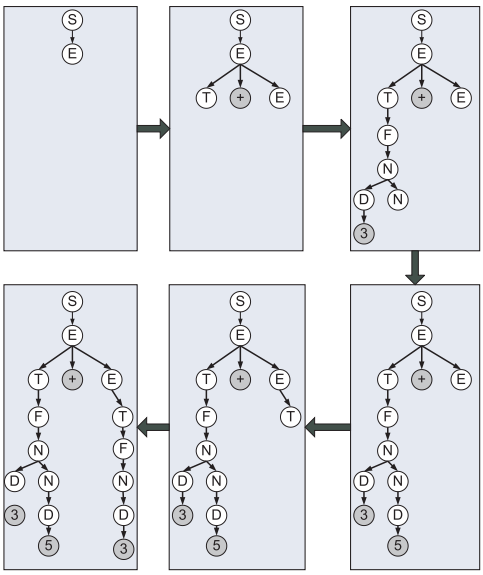

Fig. 4. Parse Tree Construction

A. Example 1

In a first approach to present the platform, an illustrative example is given based on the AG G_{op} (see Table I), which describes the basic arithmetic operations of addition and multiplication between two or more operands. The produced system can recognized input strings that describe arithmetic operations and furthermore evaluate the final result, resembling to a common (infix notation) calculator. According to our methodology presented in the previous section, only one stack is needed since only one synthesized attribute exist in the grammar G_{op}. Furthermore, in the case of rules, where a simple assignment is imposed by the semantics, no action is needed. In contrary, all other semantic rules have been transformed into simple push and pop actions.

Conventional approaches would recognize whether the input string $35+3*(4+20)$ belongs to G_{op}, using the syntax rules. On success, the parse tree would be produced (Fig. 5). At the next stage, the parse tree would be traversed and each time a node was traversed, semantic rules would be evaluated.

In our approach, the semantic rules have been transformed into the actions shown in the 3^{rd} column of Table I. For each node of the tree, the corresponding action id is outputted. The last submodule receives these simple actions and executes them. In this way, the final result (the number 107) appears at the output of our Soc. The actions produced by our system as well as the stack for the input string $35+3*(4+20)$ is summarized in Table II. The 1^{st} column of Table II refers to the numbered nodes of the parse tree shown in Fig. 5.

B. Example 2

Natural Language (NL) processing is a very attractive method of human-computer interaction and may be applied to a considerable number of fields such as intelligent embedded systems, intelligent interfaces, learning systems, etc [15], [16]. It is clear that automatically extracting linguistic information from a text can be an extremely powerful method for NL processing systems.

In order to show how we can build a natural language interface, using the system proposed, we have chosen a question-answering example [17] from the area of airline flights. In this example, the system can receive sentences belonging to a subset of NL. When the syntactic recognition of the input sentence is completed, using the created parse tree, the semantics are evaluated and the FPGA sends SQL queries generated to a data-management machine that has access to a data-

TABLE I
ARITHMETIC OPERATION GRAMMAR G_{op}

Syntax Rule	Semantic Rule in AG notation	Corresponding Actions in Action Grammar notation
$S \rightarrow E$	$S_s = E_s$	[pop result]
$E_1 \rightarrow T + E_2$	$E_{1s} = T_s + E_{2s}$	[pop x] [pop y] [evaluate x+y][push result]
$E \rightarrow T$	$E_s = T_s$	-
$T \rightarrow F * T$	$T_s = F_s * T_s$	[pop x] [pop y] [evaluate x*y][push result]
$T \rightarrow F$	$T_s = F_s$	-
$F \rightarrow (E)$	$F_s = E_s$	-
$F \rightarrow N$	$F_s = N_s$	-
$N \rightarrow DN$	$N_s = 10 * D_s + N_s$	[pop x] [pop y] [evaluate 10y+x][push result]
$N \rightarrow D$	$N_s = D_s$	-
$D \rightarrow 0$	$D_s = 0$	[push 0]
...	...	...
$D \rightarrow 9$	$D_s = 9$	[push 9]

TABLE II
ACTION EXECUTION G_{op}

Node	Action	Stack
1	push 3	3
2	push 5	3, 5
3	pop x; pop y; push 10y+x	35
4	push 3	35, 3
5	push 4	35, 3, 4
6	push 2	35, 3, 4, 2
7	push 0	35, 3, 4, 2, 0
8	pop x; pop y; push 10y+x	35, 3, 4, 20
9	pop x; pop y; push x+y	35, 3, 24
10	pop x; pop y; push x*y	35, 72
11	pop x; pop y; push x+y	107
12	pop result	-

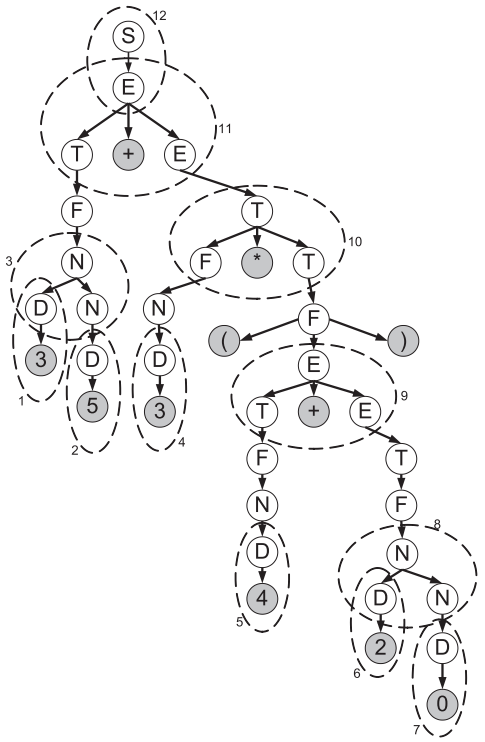

Fig. 5. Parse Tree for 35+3*(4+20)

TABLE III
NATURAL LANGUAGE GRAMMAR G_{NL}

Syntax Rules	
$PR \rightarrow NP\ VP$	$SB \rightarrow OB$
$NP \rightarrow QS\ W\ VP$	$SB \rightarrow OB\ P\ OB$
$NP \rightarrow QS$	$QF \rightarrow A\ SET$
$NP \rightarrow OB$	$QF \rightarrow E\ SET$
$QS \rightarrow A\ SET$	$OB \rightarrow Object\ Names$
$QS \rightarrow E\ SET$	$SET \rightarrow Class\ Names$
$VP \rightarrow VP_1\ VP_2\ SP$	$REL \rightarrow Relation\ Phrases$
$VP \rightarrow SP$	$AT \rightarrow Property\ Names$
$VP_1 \rightarrow IP\ VP_1$	$NRL \rightarrow Numerical\ Relations$
$VP_2 \rightarrow SP\ CNJ\ VP_2$	$N \rightarrow Numbers$
$IP \rightarrow REL\ QF\ W$	$W \rightarrow Relative\ Pronouns$
$SP \rightarrow REL\ SB$	$A \rightarrow Determiner$ "A"
$SP \rightarrow AT\ NRL\ N$	$Each \rightarrow Determiner$ "$Each$"
$SB \rightarrow QF$	$CNJ \rightarrow Conjuction\ Words$

base in order to produce the final result (answer). The system can be used in applications where a lot of input sentences must be processed simultaneously and extract information. The extracted information can be statistics about the preferences of clients, the profiling of the users, intelligent extraction of keywords for web browsers etc.

In Table III the syntax rules of the underlying AG G_{NL} are shown, which recognize questions concerning airline flights. The full AG grammar can be found in [17], where a subset of English accepted by the system, uses words belonging to classes like: class names, object names, property names e.t.c. The sentences of the subset of English are questions concerning airline flights and the answer after the processing of the intermediate code by the abstract data-management machine is YES or NO. In that paper the intermediate code was executed by an abstract toy-scale machine accessing a simplified data base. On the contrary, in this paper the intermediate code is SQL queries which can be processed by any real-life server. An illustrative simple question is:

A FLIGHT DEPARTS FROM ATHENS?

This question can be syntactically analyzed into a noun phrase consisting of a determiner and a common noun and a verb phrase consisting of a verb, a preposition and a proper noun. The determiner corresponds to a quantifier, the common noun to a class name, the verb and the preposition to a relation and the proper noun to an object. The nouns and the verb will be used as parameters by the commands that will be generated by the determiner and syntactic structures.

TABLE IV
CORRESPONDING ACTIONS FOR G_{NL}

Syntax Rule	Actions
$PR \rightarrow NP\ VP$	pop x; pop y; push conc(y,x,;)
$QS \rightarrow A\ SET$	pop x; pop y; push conc(y,x)
$VP \rightarrow SP$	
$SP \rightarrow REL\ SB$	pop x; pop y; push conc(y,x)
$SB \rightarrow OB$	
$OB \rightarrow Athens$	push ""ATH""
$SET \rightarrow Flights$	push "from Flights"
$REL \rightarrow Departs\ From$	push "where DepartsFrom="
$A \rightarrow A$	push "select FlightNo"

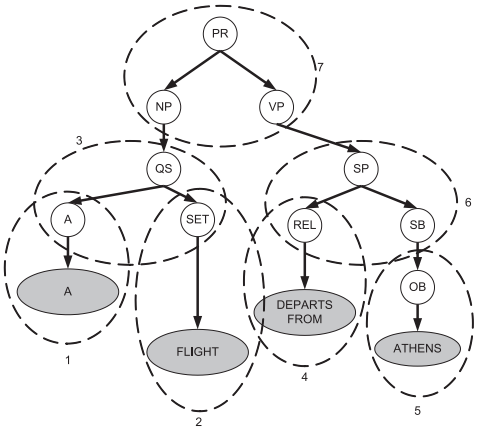

Fig. 6. Parse Tree for NL example

Following the same course as in example 1, the semantic rules are corresponded to simple actions and a stack is used. For the question, "A FLIGHT DEPARTS FROM ATHENS?" the parse tree is presented in Fig. 6. For the same input string, in Table IV the transformed semantic rules are presented, only for the necessary nodes, where conc $(par_1, \ldots, par_n)$, which stands for the concatenation of the contains of $par_1, \ldots, par_n$. Given a database that consists of a Flights Relation: Flights(FlightNo id, DepartsFrom char[3], ArrivesAt char[3], Airline char[5], ...), the generated system produces the SQL query: Select FlightNo From Flights Where DepartsFrom="ATH";, as shown in Table V, that can be executed on an SQL server.

More complex questions can also be handled by the proposed

TABLE V
ACTION OUTPUT G_{NL}

Node	Action	Stack
1	push "select FlightNo"	"select FlightNo"
2	push "from Flights"	"select FlightNo", "from Flights"
3	pop x; pop y; push conc(y,x)	"select FlightNo from Flights"
4	push "where DepartsFrom="	"select FlightNo from Flights", "where DepartsFrom="
5	push ""ATH""	"select FlightNo from Flights", "where DepartsFrom=", ""ATH""
6	pop x; pop y; push conc(y,x)	"select FlightNo from Flights", "where DepartsFrom="ATH""
7	pop x; pop y; push conc(y,x,;)	"select FlightNo from Flights where DepartsFrom="ATH";"

architecture, producing the corresponding SQL code, susch as:

- AIRLINE-X FLIES FROM ATHENS TO NEW-YORK?
- EACH FLIGHT WHICH IS CONNECTED TO A FLIGHT WHICH BELONGS TO AIRLINE-X DEPARTS FROM A CITY WHICH IS LINKED TO EACH CITY WHICH BELONGS TO GREECE?

V. Conclusion and Future Work

This work is a part of a project[1] for developing a platform (based on AGs) in order to automatically generate special purpose embedded systems. In this paper a generic platform for the hardware implementation of grammar-based applications is presented. The proposed platform, given the specification of the application in the formalism of Attribute Grammar, automatically produces the necessary hardware modules for the syntactic and semantic analysis of input strings belonging to that grammar. The produced implementation tackles with the recognition task of the input string using a hardware implementation [9] of Earley's parallel parsing algorithm, giving a speedup of 2 orders of magnitudes compared to conventional software approaches. The attribute evaluation makes usage of a stack-based methodology.
Our future work remains focused in implementing the proposed architecture using a faster parser, that is based exclusively on combinatorial circuits, i.e. the one proposed in [18]. A drastic further increase of the speed-up is expected using this parser. Furthermore, the prospective of extending the platform so as to tackle with inherited attributes and multi-pass AGs [2] and the automatic matching of semantic rules to action is a high priority.

References

[1] D. E. Knuth, "Semantics of context free languages," *Math. Syst.Theory*, vol. 2, pp. 127–145, 1971.

[2] J. Paakki, "Attribute grammar paradigms a high-level methodology in language implementation," *ACM Comput. Surv.*, vol. 27, no. 2, pp. 196–255, 1995.

[3] G. Papakonstantinou and J. Kontos, "Knowledge representation with attribute grammars," *The Computer Journal*, vol. 29, 1986.

[4] G. Papakonstantinou, C. Moraitis, and T. Panayiotopoulos, "An attribute grammar interpreter as a knowledge engineering tool," *Applied Informatics*, vol. 9, pp. 382–388, 1986.

[5] I. Panagopoulos, C. Pavlatos, and G. Papakonstantinou, "An embedded system for artificial intelligence applications," *International Journal of Computational Intelligence*, 2004.

[6] K. Fu, *Syntactic Pattern recognition and Applications*. Prentice-Hall, 1982.

[7] A. Aho, R. Sethi, and J. Ullman, *Compilers - Principles, Techniques and Tools*. Reading, MA: MADDISON-WESLEY, 1986.

[8] A. Demers, T. Reps, and T. Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors," in *Conf. Rec. 8th Annu. ACM symp. Principles Programming Languages*, 1981, pp. 415 – 418.

[9] C. Pavlatos, I. Panagopoulos, and G. Papakonstantinou, "A programmable pipelined coprocessor for parsing applications," *Workshop on Application Specific Processors (WASP) CODES.*, September 2004.

[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 (75%) and National Resources (25%).

[10] Y. T. Chiang and K.-S. Fu, "Parallel parsing algorithms and VLSI implementations for syntactic pattern recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, pp. 302–314, 1984.

[11] C. Pavlatos, A. Dimopoulos, and G. Papakonstantinou, "An embedded system for the electrocardiogram recognition," *EMBEC'05*, November 2005.

[12] C. Pavlatos, A.Dimopoulos, and G. Papakonstantinou, "An intelligent embedded system for control applications," in *Workshop on Modeling and Control of Complex Systems*, Cyprus, 2005.

[13] Xilinx Official WebSite http://www.xilinx.com.

[14] S. C. Johnson, *Yaccyet another compiler compiler*, ser. Computing Science Technical Report 32. Murray Hill, N.J.: AT&T Bell Laboratories, 1975.

[15] Y. Li, H. Yang, and H. V. Jagadish, "NaLIX: an interactive natural language interface for querying xml," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, Baltimore, Maryland, 2005, pp. 900–902.

[16] A. Yates, O. Etzioni, and D. Weld, "A reliable natural language interface to household appliance," in *Proceedings of the 8th international conference on Intelligent user interfaces*, 2003, pp. 189 – 196.

[17] C. Pavlatos, A.Dimopoulos, and G. Papakonstantinou, "Hardware natural language interface," in *4th IFIP Conference on Artificial Intelligence Applications & Innovations (AIAI)*, Athens, Greece, September 2007.

[18] C. Pavlatos, A. Dimopoulos, A. Koulouris, T. Andronikos, I. Panagopoulos, and G. Papakonstantinou, "Efficient reconfigurable embedded parsers," *Computer Languages, Systems & Structures*, 2007.