International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:4, 2007

An Embedded System for Artificial Intelligence Applications

Ioannis P. Panagopoulos, Christos C. Pavlatos, and George K. Papakonstantinou

***Abstract*—**Conventional approaches in the implementation of logic programming applications on embedded systems are solely of software nature. As a consequence, a compiler is needed that transforms the initial declarative logic program to its equivalent procedural one, to be programmed to the microprocessor. This approach increases the complexity of the final implementation and reduces the overall system's performance. On the contrary, presenting hardware implementations which are only capable of supporting logic programs prevents their use in applications where logic programs need to be intertwined with traditional procedural ones, for a specific application. We exploit HW/SW codesign methods to present a microprocessor, capable of supporting hybrid applications using both programming approaches. We take advantage of the close relationship between attribute grammar (AG) evaluation and knowledge engineering methods to present a programmable hardware parser that performs logic derivations and combine it with an extension of a conventional RISC microprocessor that performs the unification process to report the success or failure of those derivations. The extended RISC microprocessor is still capable of executing conventional procedural programs, thus hybrid applications can be implemented. The presented implementation is programmable, supports the execution of hybrid applications, increases the performance of logic derivations (experimental analysis yields an approximate 1000% increase in performance) and reduces the complexity of the final implemented code. The proposed hardware design is supported by a proposed extended C-language called C-AG.

***Keywords*—** Attribute Grammars, Logic Programming, RISC microprocessor.

I. Introduction

Knowledge engineering and logic programming approaches have extensively been used in many application domains such as medicine , scheduling and planning , control , artificial intelligence [1] etc. Therefore, the possibility of exploiting such approaches in embedded systems is of crucial importance. Since many of those applications need to conform to very strict real-time margins, one of the key requirements for the efficiency of such systems is that of performance.

In order to meet this requirement, we exercise an innovative HW/SW codesign method by presenting a special purpose hardware extension to RISC microprocessors, based on attribute grammar evaluation, well suited for the efficient implementation of logic programming applications in embedded systems.

Knowledge engineering tools are based on the declarative programming model. On the other hand, the nature of computation supported today by existing microprocessors is solely procedural. As a consequence, the implementation of logic programs in existing microprocessors is bound to the use of a software compiler performing the declarative to procedural translation for their execution. This translation mechanism affects both the complexity and speed of the final implementation, since it usually imposes the implementation of a software stack for logic derivations, increasing exponentially memory I/O references. As a consequence, software implementations of logic programs, in existing embedded platforms, negatively affect design efficiency. The existence of processors capable of supporting the declarative programming model would greatly improve execution performance and simplicity of the generated code.

Extensive efforts in the implementation of machines for logic programming have been mainly encountered in the 5th generation computing era which envisioned a number of interconnected parallel machines for AI applications [3][4]. Powerful processors have been introduced working on UMA and NUMA computers [2][3] in the effort of increasing the efficiency and parallelization of declarative programs implemented for PROLOG inference engines. Although the overall speed-up achieved, following such approaches, has been satisfactory, the cost for the implementation of such systems, along with their size, prevented their use in small scale applications in embedded system environments. Additionally, the implemented machines were solely optimized for the logic programming model, which is not always suited for all application domains. The introduction of embedded systems [5] seems to present new challenges and requirements in the implementation of processors with optimized logic inference capabilities. Embedded systems do not target generality since they are oriented for small-scale applications running on dedicated hardware. Additionally, their restricted computational power (required for constraint satisfaction), turns approaches for increasing performance

Manuscript received December 23, 2003.

I.P. Panagopoulos is with the National Technical University of Athens, Zografou Campus, Athens, Greece (e-mail: ioannis@cslab.ece.ntua.gr., Tel: ++30210-7722495)

C.C. Pavlatos is with the National Technical University of Athens, Zografou Campus, Athens, Greece (e-mail: cpavlatos@cslab.ece.ntua.gr, Tel: ++30210-7721529).

G. K. Papakonstantinou is with the National Technical University of Athens, Zografou Campus, Athens, Greece (e-mail: papakon@cslab.ece.ntua.gr, Tel: ++30210-7722495).

extremely useful for design efficiency. As a result, the effort of designing hardware capable of supporting the declarative programming model for logic derivations can now lead to intelligent embedded designs which are considerably more efficient compared to the traditional procedural ones.

In this paper we propose an extension of the RISC-architecture microprocessor for knowledge representation, based on attribute grammars evaluation, in the effort of achieving design efficiency for intelligent embedded systems. We have chosen to follow the attribute grammar (AG) approach for the implementation of inference engines, since AGs have been proven to support both the declarative and procedural programming model encountered in existing knowledge representation systems [6][7][8][9]. Our contribution is summarized in the following:

- We introduce a programmable hardware implementation of an extended parser which is capable of handling all required derivations in logic programming applications.
- We propose a modified version of the RISC microprocessor which allows programs following the declarative execution model to be executed. This modification extends and not substitutes the conventional procedural execution and thus hybrid applications may be programmed.
- We combine the extended hardware parser with the modified RISC microprocessor to present an AG evaluation system, capable of supporting inference processes in a knowledge base.
- We allow design flexibility, since both the microprocessor and the extended hardware parser are programmable allowing the implementation of any desired knowledge base.

The rest of the paper is organized as follows. In Section II, we present the close relation between logic programming and AGs and current software and hardware implementations of AG evaluators. In Section III, a brief overview of our approach is provided. In Section IV, we present the proposed C-extensions in the introduced C-AG language. In Section V, we provide a detailed description of the compilation process used for programming the extended microprocessor along with the preprocessors introduced to assist in the compilation process. In Section VI, we present implementation details of our approach. In Section VII, an example application is presented and used for the evaluation of the efficiency of our approach. In Section VIII, we demonstrate the approach we followed for the implementation of the proposed design. Finally, conclusions and future work are presented at Section IX.

II. Attribute Grammars and Logic Programming

A. Introduction to Attribute Grammars

An *attribute grammar* (AG) is based upon a *context free grammar* (CFG). A CFG is a 4-tuple *G = (N, T, P, Z)*, where *N* is the set of non-terminal symbols, *T* is the set of terminal symbols, *P* is the set of grammar rules (a subset of N x (N $\cup$ T)* written in the form A→α, where A $\in$ N and α $\in$ (N $\cup$ T)*) and Z ($Z \in N$) is the start symbol (the root of the grammar). An AG is a 4–tuple AG = {G, A, SR, d} where G is a context-free grammar, A = $\cup$ A(X) where A(X) is a finite set of attributes associated with each symbol X $\in$ V (V=(N $\cup$ T)). 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 the values defined in terms of attributes at descendant nodes of node *X* of the corresponding semantic tree (decorated tree). Inherited attributes *X.i* are values defined in terms of attributes at the parent and (possibly) sibling nodes of node *X* of the corresponding semantic tree. From the definition, the start symbol does not have inherited attributes while the terminal symbols do not have synthesized attributes. Each of the productions $p \in P$ ($p : X_0 \rightarrow X_1 ... X_n$) of the CFG is augmented by a set of *semantic rules SR(p)* that define attributes evaluation rules and conditions in terms of other attributes of terminals and non 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, each attribute *a* is associated with a specific domain *d*(*α*).

The syntax rules of the AG define all possible derivations from a specific non terminal symbol. If only terminal symbols are used to determine the success of those derivations (based on comparisons with the tokens of an input string), then *parsing* is performed. If terminal symbols and attribute instance values determine the success of those derivations then, *semantically driven parsing* is performed. It is possible to omit all terminal symbols in the AG (replace them with *nil* tokens) and store their information in attributes at the leaf nodes of the tree (definite clause AG approach [10]), in order to perform semantically driven parsing without the use of terminal symbols. Finally, if no terminal symbols exist and no input string in used, then parsing is *degenerate* and tree derivations are only controlled by semantic conditions on attribute instance values. In our approach, we do not use any terminal symbols since the implementation's main purpose is for logic programming applications. Apart from that, using the proposed implementation, semantically driven parsing can be performed through the definite clause AG approach, while degenerate parsing can be realized through the evaluation and checking of attribute instance values. As a consequence, all expressive power of AGs is preserved. This, as it will be shown in later sections, ensures the maximum possible flexibility in the design.

B. Logic Programming Using Attribute Grammars

Attribute grammars have extensively been used for logic programming applications [6][11][12]. In [8] [9] an effective method based on an extension of the Floyd's parser [13] is

presented that transforms a logic programming program to its AG equivalent representation. This method introduces a number of equivalent syntax rules for the inference rules and a number of attributes and semantic conditions for the unification process in the inference procedure and can be used as a complete inference engine. The basic concepts underlying this approach are the following:

Every inference rule in the initial logic program can be transformed to an equivalent syntax rule consisting solely of non-terminal symbols. For example:

$$R_O(t_{01},t_{02},...t_{ok_0}) \leftarrow R_1(t_{11},t_{12},...t_{1k_1})R_2(t_{21},t_{22},...t_{2k_2})..R_m(t_{m1},t_{m2},...t_{mk_m})$$

is transformed to the syntax rule: $R_O = R_1 R_2 ... R_m \,|\,.$

("|." represents the end of the rule). In case there are two or more alternatives for a single inference rule, those are transformed to an equal number of alternatives in the corresponding syntax rule. For example:

$$R_O(t_{01},t_{02},...,t_{ok_0}) \leftarrow R_1(t_{11},t_{12},...,t_{1k_1})R_2(t_{21},t_{22},...,t_{2k_2})...R_p(t_{p1},t_{p2},...,t_{pk_p})$$

$$R_O(t_{01},t_{02},...,t_{ok_0}) \leftarrow D_1(t_{11},t_{12},...,t_{1l_1})D_2(t_{21},t_{22},...,t_{2l_2})...D_q(t_{q1},t_{q2},...,t_{ql_q})$$

are transformed to the syntax rule: $R_O = R_1 R_2 ... R_p \,|\, D_1 D_2 ... D_q \,|\,.$

("|" represents the alternative meta-symbol). Finally, facts of the inference rules are transformed to terminal leaf nodes of the syntax tree referring to the empty string. For example the facts:

$R_g(a,b),\ R_g(c,d),\ R_g(e,f)$

are transformed to: $R_g = |||\,.$

(which are three "nil" symbols separated by the alternative meta-symbol). Obviously, parsing is degenerate since there are no terminal symbols. For every variable existing in the initial predicates, two attributes are attached to the corresponding node of the syntax tree one synthesized and one inherited. Those attributes assist in the unification process of the inference engine. The attribute evaluation rules are constructed based on the initial logic program. A detailed method for specifying those transformation rules can be found in [8][9][12] and can be easily performed automatically by a suitable tool. Attributes at the leaf nodes of the tree are assigned values from the constants in the facts of the logic program. The inference process is carried out during tree derivations and an *EVAL* function is called at the insertion/visit of each node that computes the attribute rules performing the unification procedure. Semantic conditions, on the result of the unification procedure, determine the success or failure of those derivations in the inference procedure (a meta-variable *flag* is used to hold this information). In general, the addition of attribute evaluation rules which determine the creation and form of the constructed syntax tree forms a full degenerate semantically driven parser which can be effectively used in logic programming derivations. Additional semantic rules can be further added to increase the inference power beyond the one affected by the PROLOG rules, leading to the implementation of semantically driven parsers [14], theorem provers [15] and inference engines with fuzziness and uncertainty [16].

TABLE I
AG EQUIVALENT REPRESENTATION OF THE KNOWLEDGE BASE OF THE "SUCCESSOR" PROBLEM

Informal Definition of the Knowledge Base	Equivalent AG evaluation Syntax and Semantic rules

Goal (X,Y) if Successor (X,Y)
Successor (X,Y) if Parent (Z,X) and Successor (Z,Y)
Successor(X,Y) if Parent (Y,X)
Parent (j,b)
Parent (j,l)
Parent (b,a)
Parent (b,p)
	Goal = Successor |.
	Successor.ia1=Goal.ia1;
	Goal.sa2=Successor.sa2;
	Successor = Parent Successor |.
	Parent.ia2=Successor[1].ia1;
	Successor[2].ia1=Parent.sa1;
	Successor = Parent |.
	Parent.ia2=Successor.ia1;
	Successor.sa2=Parent.sa1;
	Parent = |.
	if ((Parent.ia1!=nil) && (Parent.ia1!="j"))
	flag=0; else Parent.sa1="j";
	if ((Parent.ia2!=nil) && (Parent.ia2!="b"))
	flag=0; else Parent.sa2="b";
	Parent= |.
	if ((Parent.ia1!=nil) && (Parent.ia1!="j"))
	flag=0; else Parent.sa1="j";
	if ((Parent.ia2!=nil) && (Parent.ia2!="l"))
	flag=0; else Parent.sa2="l";
	…

In order to clarify the aforementioned transformation, we demonstrate a toy-scale example of a logic program which is transformed to its AG equivalent one. Consider that we have the knowledge base illustrated in Table I (First Column) and we want to ask the question *"p is successor of whom?"* i.e. *Successor (p,?)*. The syntax rules which form the equivalent AG evaluator are illustrated in Table I (Second Column) along with the attribute evaluation rules and semantic conditions to be used for the inference and unification process.

The question asked has two solutions, which are *"j"* and *"b"*. The corresponding parse trees, decorated with the unification attributes are illustrated in Fig. 1.

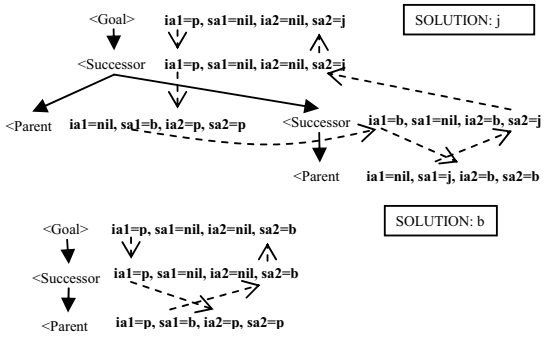


Fig. 1 Parse trees for the "successor" example leading to solutions (Note that tree traversal is top-bottom, left to right)

C. Attribute Grammar Software/Hardware Implementations

Several software [8][17][18][19] and hardware [14][21][22] implementations of attribute grammar evaluators exist in literature. Nevertheless, existing hardware implementations for AGs although highly efficient in terms of performance, restrict their use to a specific AG (hardwired in the design) and as a consequence lack the programmability required in embedded systems. The proposed implementation is based on a modified version of the AG evaluator presented in [8] [9] and its hardware implementation presented in [20]. The choice of this AG evaluator, due to its simplicity, allows a simple and fast hardware implementation and can become easily reprogrammable. It is also semantically driven (supports dynamic parsing by exploiting tree-pruning techniques to increase efficiency and prevent the memory explosion problem for storing all possible parse trees) and gives all possible solutions (non-deterministic), i.e. it can provide all possible parse trees for a specific input string. Due to the close relation between attribute grammars and logic programs, it can be easily extended to be used in intelligent embedded systems for constraint logic programming applications [23] and can be extended to support fuzziness and uncertainty [15] [16]. Finally, there are already software implementations of the parser and its use in various application domains providing a sufficient development environment for the evaluation of the design [8] [9] [20] [12].

III. Overview of Our Approach

Conventional approaches, for the incorporation of a logic program in embedded systems, follow the method illustrated in Fig. 2(a). A logic program is initially captured using a Logic Programming Language (such as PROLOG). Then, a compiler is used that performs the transition from the declarative algorithm to the behaviorally equivalent procedural one, performing the inference process. The final algorithm is programmed to the microprocessor for execution. Our approach (Fig. 2(b)) starts from a conventional PROLOG like language for the definition of the knowledge base. The initial specification is transformed, through the use of a preprocessor, to its AG equivalent representation, written in the proposed extended C-language, called C-AG.

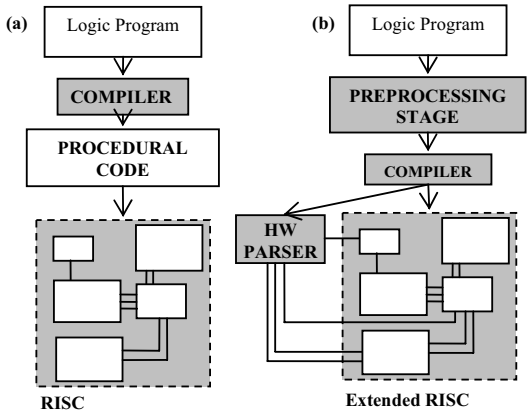


Fig. 2 (a) Conventional approach (b) Proposed approach

We further introduce an extension to the RISC microprocessor which consists of a programmable hardware parser, for the definition of the syntax rules, that performs the required tree derivations (logic derivations) and a conventional RISC part that handles the attribute evaluation rules (unification process) and possibly executes conventional sequential code that co-exists in the hybrid procedural-declarative application. The C-AG language is finally transformed to a conventional program expressed in C through the use of a preprocessor that adds inline assembly code (related to the extended RISC's features and supporting the declarative execution) to the final program. The use of the preprocessor is imperative to prevent any modifications that would be required to the conventional C-compiler, adding additional effort in translating existing C-compilers to support the innovative extended RISC's features. The mapping of the C-AG program to the extended RISC microprocessor is illustrated in Fig. 3.

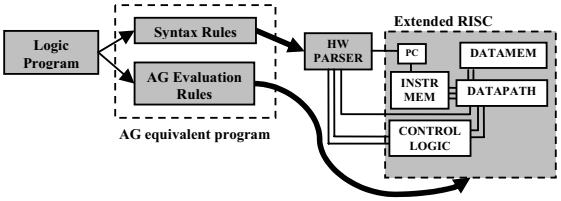


Fig. 3 Overview of the mapping process of the final AG equivalent program written in C-AG

The basic idea in the proposed architecture is to equip the microprocessor with two modes of operation: one declarative and one procedural. In the procedural mode, the processor functions in the conventional way (the hardware parser extension is disabled) and program execution is sequential through consecutive increases or jumps of the Program Counter (PC). On the declarative mode, the hardware parser is enabled and performs the required tree derivations (logic derivations) based on its internally stored syntax rules (capturing the knowledge base). Nodes of the constructed tree are stored in a specially designed stack within the hardware parser. There is a one-to-one correspondence between a stack

line and the node in the parse tree it represents. Consequently, the position of the node in the stack can be used as the identification number of the specific node (NID-Node IDentification number). A stack line also holds an encoding of the non-terminal symbol (predicate) associated with the node along with information on the dependency of the node to other nodes (predicates) of the tree (represented as locations in the stack).

Upon each creation/visit to a node, attribute evaluation rules need to be executed in order to perform unification and possibly determine the success or failure of the unification process. Attribute evaluation rules are organized as blocks of code in the RISC microprocessor. Each block is associated with its corresponding non-terminal symbol. The hardware parser uses the associated non-terminal symbol's encoding (stored in the stack line), for each node, to determine (by controlling the value of the PC) the block of code that needs to be executed (the block of code that performs unification). A space is also reserved in the embedded system's memory for storing attribute instances. The latter are dynamically organized in blocks (associated with a specific stack line-node) and can be referenced based on the NID of the node.

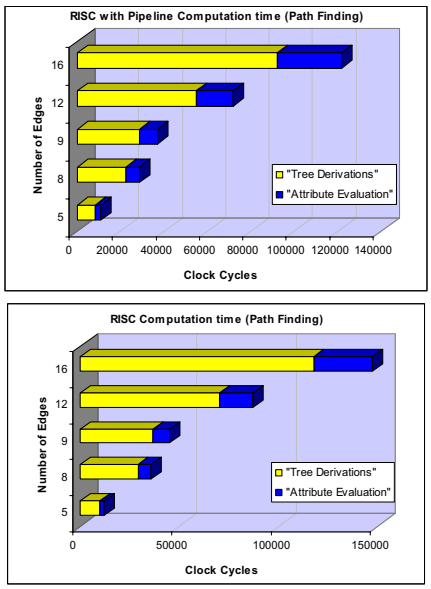


Fig. 4 Computational Time required for performing tree derivations and attribute evaluation in a RISC and a RISC with pipeline microprocessor

By using the proposed approach and the microprocessor's extended features, apart from the fact of the declarative-procedural coexistence, we greatly reduce the complexity of the implemented program (no translation to procedural code is needed). We also achieve a great improvement to the system's performance (av. 1000%). This is mainly due to the fact that the programmable hardware implementation of the parser relieves the microprocessor from the additional computation time that would be required for performing the additional tree derivations (logic derivations).

In Fig. 4, it is clear that the computational time required for performing such tree derivations by a RISC and also a RISC with pipelining capabilities microprocessor constitutes approximately the 80% of the total time required to perform the inference process (attribute evaluation computations related to the unification process consist of simple assignment operations and therefore are not computationally intensive). Mapping the tree derivation process to hardware would therefore surely improve the overall embedded system's performance.

In the following sections of this paper, we present in more detail, the extended C-AG language, the required preprocessors' features for the compilation process and details of the extended RISC's hardware implementation.

IV. The Extended C-AG Language

In the effort of supporting our proposed implementation, specific language constructs were needed to enable hybrid declarative-procedural computations. For that reason, we had to incorporate a specific syntax for the specification of the syntax and attribute grammar evaluation rules, along with a way to enable the control of tree derivations and perform the required switching mechanism from declarative to procedural code and vice-versa. Nowadays, the most well known programming language syntax for the specification of grammars is YACC [24]. Although YACC is widely accepted as a valuable tool for the specification of syntax rules and parsing, it has been shown that its use is restrictive for AGs [24] (eg. it does not support inherited attributes). Several other attribute grammar evaluators also do exist (following the conventional approach) defining an AG-like syntax such as ELI [24] and FNC2 [25]. Those systems also allow the specification of blocks of sequential code within the attribute evaluation rules. Such mechanism increases the expressiveness of AGs, required for real-life applications. We have chosen to follow an ELI-like syntax for the specification of the AG in the introduced extended C language (C-AG) and performed the required modifications to be able to integrate it into conventional procedural code. An overview of the introduced programming template is illustrated in Fig.5. AGs within the conventional C-code are defined as functions. Those functions are distinguished from conventional ones by being prefixed with the string *"AG_"*. Within the function, syntax rules are defined along with their associated attribute evaluation rules. We allow at every syntax rule a sequential block of code that is executed upon a successful creation of the sub-tree starting from the non terminal defined at the Left-Hand Side of the specific rule. Reserved keywords "*setflag/clrflag*" within the attribute evaluation rules are used to determine the success/failure of a derivation.

Finally, in *main(),* the switch to declarative code is performed through the reserved work *"switchdecl"* whose parameter specifies the function defining the AG to be executed. It should be noted that the scope of conventional variables within the program remains intact and attribute evaluation rules are able to access the same variables that

could be accessed in a conventional C-program. This allows data sharing between procedural and declarative code.

```
…Conventional C inlcudes, variable, function declarations
<return type> AG_<grammar_name> (…){
   ….
   RULE: <NT> ::= <NT> <NT> … <NT> END
   {
     attribute evaluation rules;
     {..} procedural code
   }
   RULE: <NT> ::= <NT> <NT> … <NT> END
   {
     attribute evaluation rules;
     if (condition (<NT>.<attribute>)) clrflag/setflag;
   }
   …. return ….; }
int main (int argc,char argv[])
{
          <return type> variable;
          ….
          variable= switchdecl
                    (AG_<grammar_name>(…));
           exit(0);
}
```

Fig. 5 A general template of the C-AG language

V. The Preprocessors

As mentioned before, two preprocessors are used for the mapping process of the desired application to the extended RISC microprocessor. The first (Logic Preprocessor) receives the initial logic program and presents the resulting AG equivalent one written in C-AG. The designer can additionally include any other procedural-declarative desired code to the resulting program written in C-AG. The hybrid program is then received by the second preprocessor (C-AG to C preprocessor) which translates the C-AG program to its equivalent C one, by adding inline assembly code, where needed, to take advantage of the extended RISC microprocessor's additional features. The overview of this translation mechanism is presented in Fig. 6.

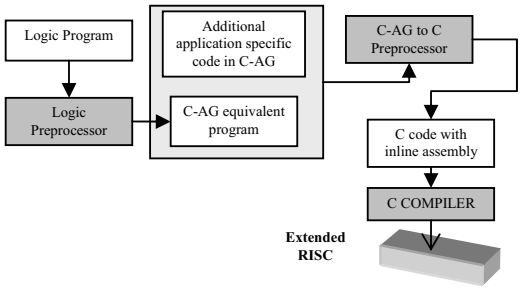


Fig. 6. The stages of the preprocessing phase

The Logic Preprocessor performs automatically the transition from the Logic Program to the C-AG equivalent representation, based on rules for the logic programming to AG transformation. Since those rules have been already well documented in [8][9][12] their description and explanation is beyond the scope of this paper. Therefore, we will consider having ready the C-AG equivalent program that will be further preprocessed to be mapped to the extended microprocessor. Consider, for example, the C-AG representation of the "successor" example presented in Section II (Fig. 6).

```
#include <stdio.h>
void AG_Successor (char successors[10], char p){
 int i;
  i=-1;
 RULE: G::=S END {
   G.ia1=p;
   S.ia1=G,ia1;
   {successors[i++]=G.sa2}
 }
 RULE: S::= P S END {
   P.ia1=S[1].ia1;
   S[2].ia1]=P.sa1;
 }
 RULE: S::= P END {
  P.ia2=S.ia1;
  S.sa2=P.sa1;
 }
 RULE: P::=  END {
   if ((P.ia1!='j')&&(P.ia1!=nil)) clrflag;
   else  P.sa1='j';
   if ((P.ia2!='b')&&(P.ia2!=nil)) clrflag;
   else  P.sa1='b';
 }
 … (Rules for other facts follow here)  }
int main (int argc,char argv[]) {
 int j;
 char successors[10];
 for (j=0;j<10;i++)  successors[i]=' ';
 switchdecl(AG_successor (successors,'p'));
 j=0;
 while (successors[j]!=' ')
  printf ("%c, ",successors[j++]);
exit(0);
}
```

Fig. 6 Output of the "Logic Preprocessor" for the "successor" example (Code in italics represents additional code added by the designer after the preprocessing stage)

The resulting C-AG program is further preprocessed by the "C-AG to C" preprocessor. The program written in C-AG is separated in the fragments of code (attribute evaluation rules, procedural code) to be mapped to software and the syntax rules that program the hardware parser (HW/SW Codesign). The syntax rules of the AG are extracted from the program, encoded and loaded to the hardware parser's "Rules Memory". The preprocessor determines automatically the time AG evaluation rules need to be executed, depending on the position in the syntax rule (tree derivations have reached so far) and transforms the rules accordingly. Conventional C-code is finally produced augmented with inline assembly instructions using the extended instruction set of the proposed implementation. Then, a conventional C-compiler is used to produce the final binary executable. The schematic diagram of the "C-AG to C" preprocessor along with the description of the resulting code is illustrated in Fig. 7.

Program begins execution like in a conventional RISC microprocessor. When the "*swichdecl*" instruction is encountered, the encoded syntax rules of the corresponding AG are loaded into the hardware parser's "Rules Memory" and the hardware parser takes control of the microprocessor's

program counter. The hardware parser performs tree derivations and, according to the non terminal symbol processed, defines the position of the attribute evaluation rules to be processed by the microprocessor to report success/failure of the derivations (in this case the unification process).

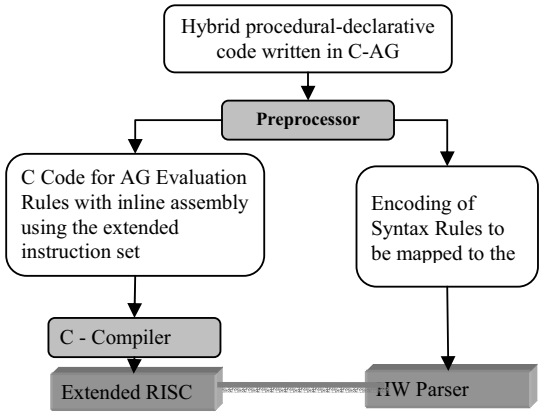


Fig. 7 Substages of the "C-AG to C" preprocessor

When there are no valid tree derivations left, the hardware parser completes and the microprocessor regains control of the program counter. In the following subsections, a more detailed description is provided on the encoding of the syntax rules used to be stored in the parser along with the description of the inline code and final representation of the C-code to be compiled.

A. Syntax Rules Encodings

The extracted syntax rules from the initial program are initially transformed to the following representation: *RULE: A::= B C END* and *RULE: A::=D E END* are transformed to *A=BC| DE|*. where the symbol *"|"* represents alternative productions and the combination of "|." denotes the end of the rule. The proposed encoding used for the non-terminal symbols of the grammar uses $P = \lceil \log_2(N-2) \rceil$ bits, where *(N-2)* is total number of non-terminal symbols. The additional two symbols have the encoding "000.." which is used to represent the symbol "|" and encoding "11111…" which is used to represent the symbol ".". Such an encoding on one hand is the smallest possible, achieving the maximum possible reduction in the number of bits needed to represent the grammar and on the other hand, as it will be shown, provides an easy way of storing the grammar rules into the *"Rules Memory"* in the hardware parser. Every sub-goal (the left hand side non-terminal symbol in each syntax rule) is assigned a value according to the previously mentioned encoding. We use the encoding of each sub-goal as the reference address to the *"Rules Memory"* location where the Right-Hand-Side of the grammatical rule is stored.

To better illustrate this, we demonstrate the resulting encoding of the "successor" example. Fig. 8(b) represents the encoding for the non-terminal symbols of the grammar and Fig. 8(c) represents the way the rules are stored in the *"Rules Memory"* in the hardware parser.

(a)

```
G = S |.
    S = P S | P |.
        P = | | | | |.
```

(b)

Goal	Encoding (Binary)	enc(Goal)
G	0001	1
S	0010	2
P	0011	3
|	0000	F
.	1111	0

(c

Address	Data					
1	2	0	F	F	F	F
2	3	2	0	3	0	F
3	0	0	0	0	0	F

Fig. 8 (a) Syntax rules of a the "successor" example grammar. (b) Example grammar's encoding *(Each non terminal symbol is assigned a specific 4bit encoding)* (c) Contents of "Rules Memory".

B. AG Evaluation Rules Translation

As previously stated, the "C-AG to C" preprocessor automatically determines when attribute evaluation computations need to be performed; depending on the position in a syntax rule i.e. tree derivations have reached so far. For example, an initial attribute evaluation rule of the form: *B.inh=f(A.inh)* associated with the syntax rule *A=B|*. is automatically transformed to: *When a node associated with B is visited: inh[current]=inh[father]*. Note that the notation *<attribute name>[<related_node>]* illustrates the dependency of the attribute *<attribute name>* in the currently visited node with attributes in its neighboring nodes in the derived tree. This implies the transformation of the initial attribute evaluation rules associated with non terminal symbols in the syntax rules, to semantically equivalent attribute evaluation rules related to node dependencies in the derived tree. In our previous example such a transformation can be expressed as: *"when a node related to the non-terminal symbol B in the rule A=B|. is visited, the attribute inh of the node can be calculated from the attribute inh of its father (which is a node associated with the non-terminal A)*". Those transformations are automatically performed by the "C-AG to C" preprocessor which determines the attribute dependencies expressed in the syntax rules and transforms them to attribute dependencies related to dependencies on the nodes of the tree. Table II demonstrates such transformations for various types of dependencies among non-terminal symbols in a syntax rule (the notation *(enc(NT),i)* is explained right after Table II and is used to access the memory address i.e. label corresponding to a specific non-terminal node, where the execution of the specific computation will have to take place during tree derivations).

In order for such transformation to take place, we need to be able to distinguish non-terminal symbols at the right hand side (RHS) of the syntax rule. For that reason non terminal symbols at the RHS are defined by a tuple *(x,i)* where *x* is the sub-goal's encoding for the specific syntax rule (LHS non terminal symbol) and *i* is the position of the needed non-

terminal symbol at the RHS of the syntax rule starting from *0*. For example in the rule *A=BC|., (enc(A), 0)* defines *B*, *(enc(A),1)* defines *C*, *(enc(A),2)* defines the *"|"* and so on *(enc(NT) gives the encoding of the non terminal symbol NT).*

TABLE II
TRANSFORMATION OF ATTRIBUTE EVALUATION RULES FROM NT-RELATED TO NODE-RELATED (INDICES I, J, K IMPLY "FOR ALL" ATTRIBUTES HAVING SUCH COMPUTATION DEPENDENCE, "$TEMP_i$" ARE ADDITIONAL ATTRIBUTES INTRODUCED FOR THE REQUIRED COMPUTATION)

Attribute Evaluation Rules	Attribute Evaluation Equivalent (traversal dependent)
A → B C |. $A.syn_i$ = f ($B.syn_j$,$C.syn_k$)	**(enc(A),2):** syn_i[current]= f(syn_j[predsub],syn_k[sub])
A → B C |. $B.inh_i$=f($A.inh_j$); C.inh=h($A.inh_k$);	**(enc(A),0):** inh_i[sub]=f(inh_j[current]); **(enc(A),1):** inh_i[sub]=h(inh_k[current]);
A → B1 B2 B3 … Bn |. $A.syn_i$= f(B1.a_1,B2.a_2,…, Bn.a_n);	**(enc(A),1):** $temp_1$[current]= a_1[sub]; **(enc(A),2):** $temp_2$[current]= a_2[sub]; … **(enc(A),n-1):** $temp_n$[current]= a_{n-1}[sub]; **(enc(A),n):** syn_i[current]=f($temp_1$[current], $temp_2$[current], …,$temp_n$[current] , a_n[sub])
A → B C |. $C.inh_i$=f($B.inh_j$);	**(enc(A),1):** inh_i[sub]=f(inh_j[sub]);
A → B C D E|. $D.inh_i$=f($B.inh_j$);	**(enc(A),1):** temp[l]=inh_j[sub]; **(enc(A),2):** inh_i[sub]=f(temp[current]);

The instruction memory is divided into blocks where each block holds the attribute evaluation rules of a specific syntax rule (referenced by the value of *x* in the tuple *(x,i)*). This block is further subdivided into sub-blocks holding the attribute evaluation rules for every position within the rule (indicated by the index *i* in the tuple *(x,i)*). Those blocks and sub-blocks can either be organized with fixed sizes or a mapping table can be used that points to the memory location for every tuple of the form *(x,i)*. The resulting memory organization for the AG evaluation rules for the "successor" example is illustrated in Table III. (Fixed block sizes are used in this example reserving 256bytes per block (for every syntax rule) and 32bytes per sub-block (for every position within the syntax rule).

TABLE III
MEMORY LAYOUT OF THE ATTRIBUTE EVALUATION RULES.

Address	Content
Base to Base+999	**Sequential Code**
Base+1000 to Base+1031	Attribute Evaluation Rules **(enc(G),0)**
Base+1031 to Base+1063	Attribute Evaluation Rules **(enc(G),1)**
Base+1064 to Base+1095	Attribute Evaluation Rules **(enc(G),2)**
Base+1096 to Base+1255	**UNUSED**
Base+1256 to Base 1287	Attribute Evaluation Rules **(enc(S),0)**
Base+1288 to Base 1319	Attribute Evaluation Rules **(enc(S),1)**
Base+1320 to Base 1351	Attribute Evaluation Rules **(enc(S),2)**
Base+1352 to Base+1383	Attribute Evaluation Rules **(enc(S),3)**
Base+1384 to Base+1415	Attribute Evaluation Rules **(enc(S),4)**
Base+1416 to Base+1447	Attribute Evaluation Rules **(enc(S),5)**
Base+1448 to Base+1511	**UNUSED**
Base+1512 to Base 1543	Attribute Evaluation Rules **(enc(P),0)**
…	…
Base+1768 to Base+xxxx	**Sequential Code**

C. Extended RISC Instruction Set

The memory for attribute instances values is also divided into blocks, where each block corresponds to a node located in the stack of the hardware parser and can be referenced by its NID. In order for the RISC to be able to access attribute instances values of current and neighboring nodes whenever a tuple *(x,i)* is dispatched from the parser for attribute evaluation, additional information is needed from the hardware parser indicating the position of the parent, son, sibling and son's sibling node of the currently visited node. Those can be used in the microprocessor's attribute evaluation rules for accessing attributes of neighboring nodes. For that reason we have extended the RISC microprocessor's instruction set to incorporate additional instructions related to attribute referencing. Moreover, additional instructions are needed for implementing the required switching mechanism between procedural and declarative code and for uploading the rules to the parser's "Rules Memory". Finally, two additional instructions are used for controlling the meta-variable flag indicating the success/failure of semantic conditions on attribute instance values. The complete table of

the added instructions is illustrated in Table IV. *(Implementation issues for the incorporation of this extended instruction set to the RISC microprocessor will be given in Section 6 where hardware implementation issues are explained in detail)*

TABLE IV
ADDED INSTRUCTIONS TO THE RISC MICROPROCESSOR'S INSTRUCTION SET FOR ATTRIBUTE REFERENCING

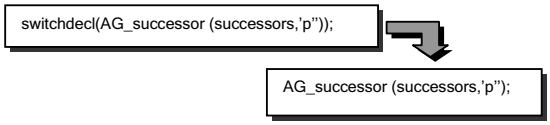
Instruction	Description
ldsub addr,dest **ldsup addr, dest** **ldsib addr, dest** **ldssib addr, dest** **ldcur addr, dest**	Load the value at memory location addr+(index depending on position in stack) and store it to register dest. (index positions refer to current(cur), son(sub), sibling(sib), parent(sup) and son's sibling node(ssib)).
stsub addr,value **stsup addr,value** **stsib addr,value** **stssib addr,value** **stcur addr,value**	Store to memory location addr+(index depending on position in stack) the value *"value"*. (index positions refer to current(cur), son(sub), sibling(sib), parent(sup) and son's sibling node(ssib)).
prgdecl addr	Performs the switch between declarative and procedural code. Specifically, the assembly instruction loads to the PC the base memory address where AG evaluation rules are stored and pushes into the stack the return value of the PC (after all logic derivations have taken place and all solutions have been found)
setflag/clrflag	Allows the manipulation of the previously described meta-variable flag which is dispatched to the Hardware parser in order to indicate a successful/unsuccessful derivation
ldrules index,addr	Load the 32bit value located at memory address *"addr"* to the" Rules Memory" of the hardware parser at position *"index"*.
initstk	Assembly instruction used to initialize the parser's stack contents

D. C-Code with Inline Assembly

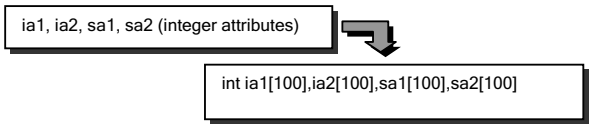
As previously mentioned, the end-product of the preprocessing stage is software code written in C with specific inline assembly blocks related to the extended microprocessor's instruction set. During the preprocessing phase the C-AG code is transformed as follows:

Initially the preprocessor extracts the grammar from the C-AG code and inserts a global array of 32bit encodings of the rules of the grammar (logic formulas) following the encoding presented in Subsection V(B):

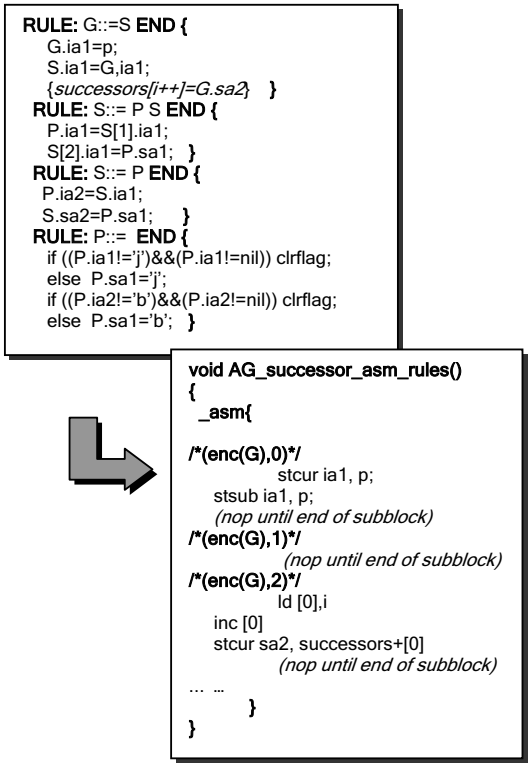
The call to the attribute evaluation function through the use of *"switchdecl"* is transformed to a conventional call to a function:



For every attribute used within the attribute evaluation rules, a space is reserved in the global section of the final C program:



The initial attribute evaluation rules, based on syntax rules, are transformed to attribute evaluation rules based on nodes of the constructed tree following the rules specified in Table II: *(The notation [i] denotes the index number i of the register in the register file)*



Inline assembly is added at the location of the attribute evaluation rules which initializes the *"Rules Memory"* and the parser's stack and initiates the declarative mode. The assembly instruction *"prgdecl"* pushes into the stack the location of the PC and then jumps to the attribute evaluation

function.

```
void AG_Successor (char successors[10], char p)
{
  int i;
  i=-1;
  _asm {
        ld [0],0
        ldrules [0],AG_successor_rules+[0];
        ...
        initstk;
        prgdecl AG_succesor_asm_rules;
          }
  ...
```

The whole purpose of the preprocessing stage is to provide a final C-program which, when compiled and executed by the microprocessor handles both the declarative and procedural code. In specific, the microprocessor can be functioning at two modes of operation. The procedural one follows the conventional steps of program execution. At the declarative one, (call to an AG evaluation function in the program) the processor switches to the declarative mode. The flowchart of the two mode switching mechanism is illustrated in Fig. 9.

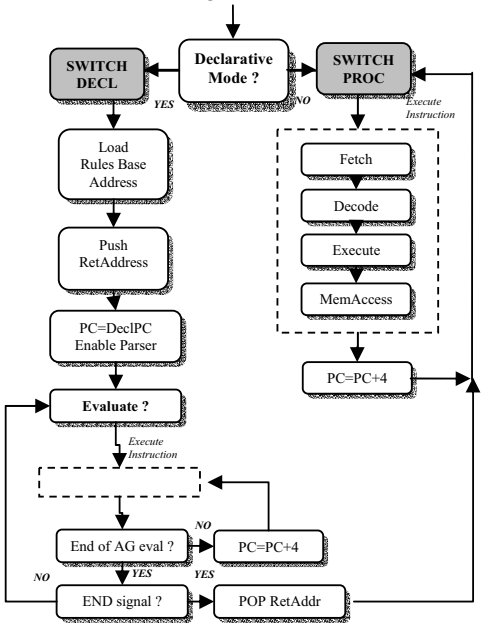


Fig. 9 Flowchart of the two modes of operation of the microprocessor

During the declarative mode, the processor initially loads the grammar rules into the hardware parser's *"Rules Memory"*, specifies the base address of the AG evaluation rules (the address of the AG evaluation function after the completion of the *"C-AG to C"* preprocessor), pushes into the stack the return address of the PC (when declarative mode completes) and waits for a signal from the hardware parser indicating that AG evaluation is required. The processor at each such call, jumps to the AG evaluation block related to the non terminal associated with the currently parsed node, performs the AG computation and waits for further requests. At the end of the derivation process, the processor pops from the stack the return address and resumes normal, procedural execution. In the next section all hardware implementation issues required for the hardware parser and the extended RISC microprocessor, in order to support such operation, are clearly defined.

VI. HARDWARE IMPLEMENTATION ISSUES

A. The Hardware Parser

An overview of the parser's flowchart is illustrated in Fig. 10.

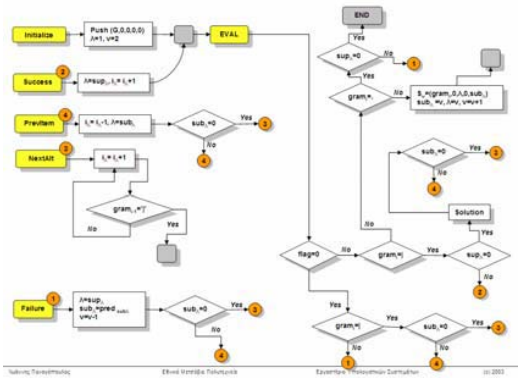


Fig. 10 The implemented parser's flowchart

The parser uses a stack to hold the nodes of the constructed parse tree. Those nodes are defined as five-tuples of the form $(goal_\lambda, i_\lambda, sup_\lambda, sub_\lambda, pred_\lambda)$ in a stack S. $goal_\lambda$, is the non terminal which is currently tried, λ is the element of the stack (current node) which is currently active, v is the new empty position in the stack, i_λ is the place in the definition of the non terminal goal at which tree derivations have reached so far, sup_λ is the location in the stack S of the goal's superior and sub_λ, $pred_\lambda$ are the locations in the stack S of the goal's most recent subordinate and sibling respectively. The parsing that takes place is degenerate. The parser has been extended to incorporate calls to an attribute evaluator procedure (*EVAL*) for attribute evaluation whose results control tree derivations through the use of the meta-variable *flag*. A more detailed description of a similar software implementation of the extended parser can be found in [20][8].

An overview of the proposed programmable hardware implementation of the parser is illustrated in Fig. 11. The grammatical rules are encoded and stored within the parser in the *"Rules Memory"*. Another memory element within the parser is used as the stack *S* that stores nodes visited of the constructed tree. The stack is a two-input, two-output memory allowing two memory read/write operations to be executed simultaneously. We have chosen to integrate this stack within the hardware parser in the effort of increasing performance in the construction of the tree.

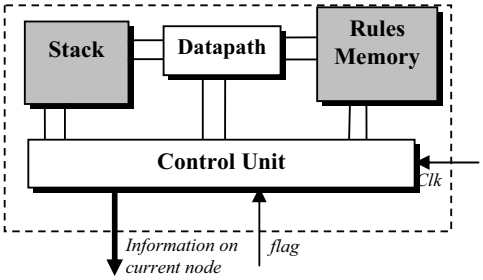
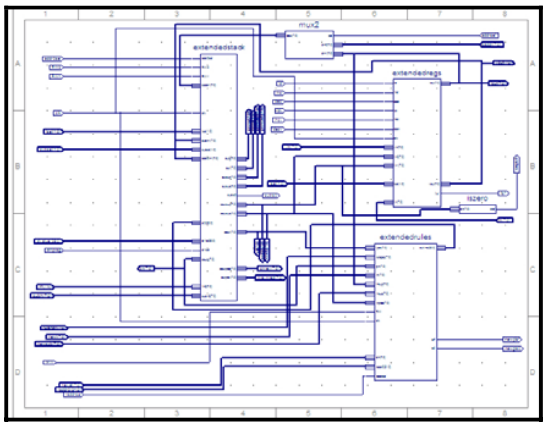
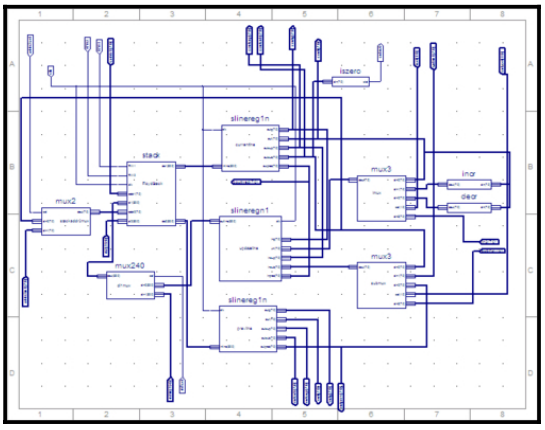


Fig. 11 A general overview of the extended RISC microprocessor's hardware parser

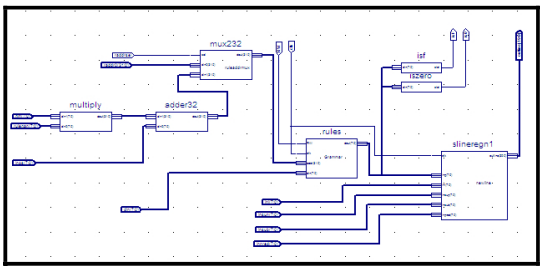
Two additional registers are used to store the current and new position in the stack: the λ register and the v register respectively. The detailed hardware implementation of the parser is provided in Fig. 12 and has been obtained by the implementation of the hardware in the XILINX ISE 6.0 environment [27].



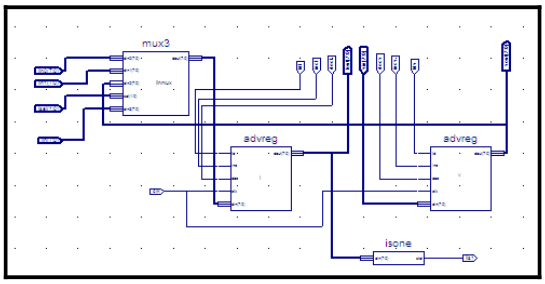
(a) Top Most Level



(b) Extended Stack



(c) Extended Rules



(d) Extended Registers

Fig. 12 Detailed implementation of the datapath of the hardware parser. (a) Top level design (b) The Stack's implementation (c) The "Rules Memory implementation (d) The register's implementation

A Finite State Machine controller drives the control signals and implements the parsing algorithm. The design is driven by the microprocessor's clock. Since the operations that occur in a single cycle in the parser are primitive RTL operations, the hardware parser's computations within a single cycle do not impose any upper limit to the microprocessor's clock, provided that they are implemented using the same technology used for the implementation of the microprocessor. At any moment, the sub-goal's encoding and the index, within the current syntax rule in the current stack line, is used to determine the next symbol to be evaluated (Value at the output port of the *"Rules Memory"*). The new symbol is either a new non terminal causing a new node to be added to the stack line or defines the end of the syntax rule. After the addition of a new node, the hardware parser triggers the microprocessor for attribute evaluation and monitors the value of the flag. If the derivation is unsuccessful or a new symbol represents the end of the rule, new alternatives are evaluated. The *"Rules Memory"* is of k*32bit size where k is its maximum size. The non terminal symbols of each syntax rule are stored in consecutive memory positions at fixed intervals. Each stack line holds information on a specific node of the constructed parse tree as described by the software parser. This means that with a *P bit* encoding of non terminal symbols and k at maximum symbols allowed at the RHS of a production, each stack line is of $4*log_2N + log_2k$ bits. For an efficient and realistic implementation, a typical size for the stack is 1,5KB and for the "Rules Memory" 2KB. This means that for an 8bit encoding of non terminal symbols, the hardware can support up to 300 nodes in each tree, 254 non terminals and up to 254 syntax rules with a maximum of 8 non

terminals at the RHS of each rule.

In order for this hardware implementation to support attribute instance referencing, the parser has been extended to dispatch attribute referencing base addresses of neighboring nodes (their positions in the stack).

B. RISC Modifications

For our analysis and experimental evaluation we have used the description of the RISC microprocessor presented in [26]. Based on this description we have implemented a soft-core in VHDL of a traditional microprocessor, which we have extended for AG evaluations. The hardware parser takes control of the PC and dispatches the tuple *(x,i)* to define the attribute evaluation computations to be executed for each node. The modification in the PCs datapath for such action is illustrated in Fig. 13.

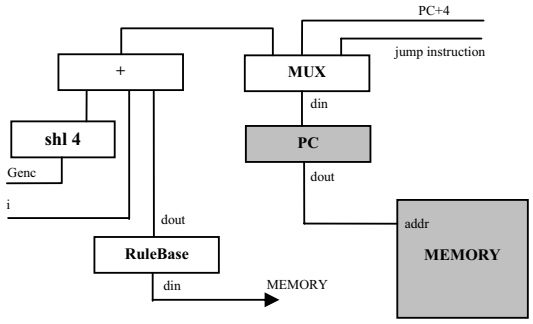


Fig. 13 The Program Counter's new Datapath

If more than one computation instructions are defined for a visited node, those are executed sequentially. Attributes associated with every node in the parse tree are stored in a designated area in the microprocessor's data memory. We have chosen to use this memory for attribute storage in order to enable other parts of the program's code (sequential or declarative) to be able to access them. Attributes in this memory area are also organized into blocks. The RISC microprocessor's instruction set is extended to support such attribute referencing instructions. Those are assigned their own encodings and introduce additional hardware to the microprocessor's soft core for their implementation (since those instructions are similar to any other indexed memory access instructions, they do not impose any additional delay in the designated microprocessor's clock cycle time). In Fig.14 additional registers are introduced storing the NIDs of neighboring nodes which are accessed upon execution of the additional introduced attribute referencing instructions.

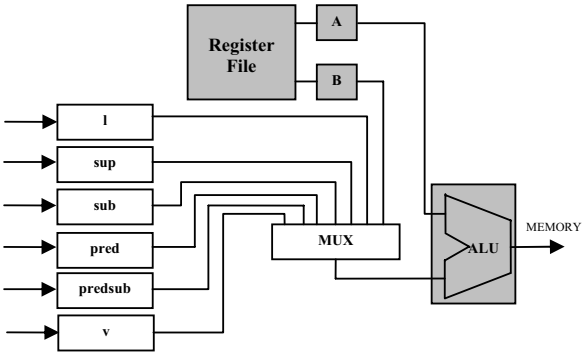


Fig. 14 Added Registers for attribute referencing instruction support

VII. Case Study and Benchmarking

As an illustrative example , we want to implement a logic program for finding paths from an arbitrary node *x* to another node *z* in a directed acyclic graph For a graph of *k* nodes with each node represented by a number *i*, where *0<i<k* we define the predicate *connected (i, j)* which is true whenever there is a directed edge leading from *i* to *j*. Such program is illustrated in Fig. 15.

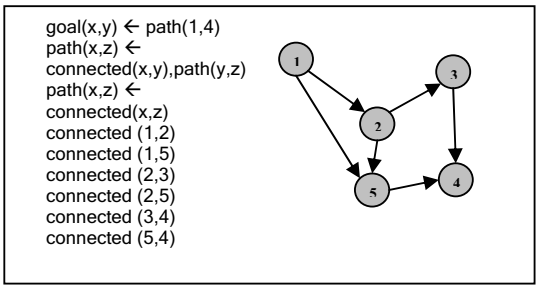


Fig. 15 Example directed acyclic graph and Logic Program for finding a path in a directed acyclic graph

We want the program to be able to answer questions of the form *"path(1,4)?"* that is, "*are there any paths from 1 to 4 and if so, which are these?"*. Based on the previously mentioned transformations, we can transform this logic program to its AG equivalent written in the C-AG language (Fig. 16) where the non-terminal *G* corresponds to *Goal*, *P* to *Path*, *C* to *Connected* and the formed syntax rules correspond to the logical formulas of the initial logic program. The initial attribute evaluation rules are determined by the application of the "Logic Preprocessor" to the initial logic program. An additional attribute *index* is then added and used in order to determine the number of steps taken for visiting a specific node. A two-dimensional array called *"paths[i][j]"* is used, where at each *i* we hold the nodes that should be visited for a solution and at each *j* we store the actual nodes. The *index* attribute is used to determine the location of a newly visited node that has led to a successful derivation. Whenever a node is added to the array, the next position in the array is filled with the value *"-1"*. This is chosen in order to denote the end

of the created path so far, since backtracking could have left "garbage" in the next locations in the array. There is also a variable *np* which is initially *0* and is incremented whenever a succesful tree deriation i.e a succesful path from *1* to *4* has been found. It should be noted that, although the implementation of the graph finding example may seem complicated, we deliberately followed such solution to the problem in order to clearly emphasize the intertwining achieved between procedural-declarative code and their data sharing possibilities. A simpler solution would also be possible, that allows the attribute grammar evaluation process itself to present the solutions in a form of a linked list of visited nodes.

```
int paths [10][10]; int np;
void AG_findpath (int paths[ ] [10],int start,int end) {
  RULE: G::=P END {
    P.inh1=start;P.inh2=end;P.index=0;
    {np++;}
  }
  RULE: P::= C P END {
    C.inh1=P[1].inh1;P[2].inh2=P[1].inh2;
P[1].inh1=C.syn2;
    P[2].index=P[1].index+1; C.index=P[1].index;
  }
  RULE: P::= C END {
    C.inh1=P.inh1;C.inh2=P.inh2; C.index=P.index;
  }
  RULE: C::=  END {
    if ((C.inh1!=1)&&(C.inh1!=nil)) clrflag;
    else
   {C.syn2=2;paths[np][C.index]=2;
     paths[np][C.index+1]=-1;}
  } … (Other facts) }
int main (int argc,char argv[]) {
    int i,j;
    for (i=0;i<100;i++)
            for (j=0;j<100;j++) paths[i][j]=-1;
            rcv_sensor_input();
            switchdecl(AG_findpath (paths),1,4);
            selectpath(); control_motors();
            for (i=0;i<np;i++) {
              j=0;
             while (j!=-1) printf ("%d ",paths[i][j++]);
            exit(0);}
```

Fig. 16 Program in C-AG

The program specification remains simple and expressive enough while procedural and declarative constructs are combined in a single program Such a path finding mechanism can be used in robotic applications in embedded systems (Fig. 17) where the graph represents visiting paths of a space separated in rooms. A robot uses the extended microprocessor to locate possible paths for reaching a specific room while the procedural part receives input from sensors (function *rcv_sensor_input()*), decides which path to follow from the ones found (function *selectpath()*) and controls the motors (function *control_motors()*).

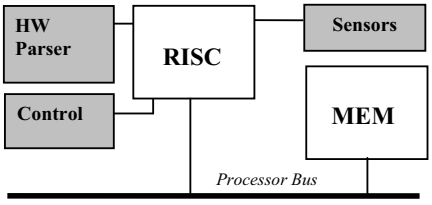


Fig. 17 Embedded system with proposed implementation

In order to evaluate the performance of the proposed implementation, we have used this implementation and simulated it for various sizes of graphs. The time required for the evaluation of attribute has not been considered. This does not affect the correctness of the results, since in both approaches (proposed and conventional) attribute evaluation occurs in the microprocessor. Therefore, attribute evaluation time does not get affected by our proposed implementation (and therefore no improvement in this time is achieved). For a various number of tree derivations (size of the parse tree constructed) instruction-level simulation with exact execution times has given the results illustrated in Fig.18, Fig.19 compared to both a conventional RISC implementation and a RISC with a pipeline with 5 stages. We have not considered any additional delay for memory I/O references.

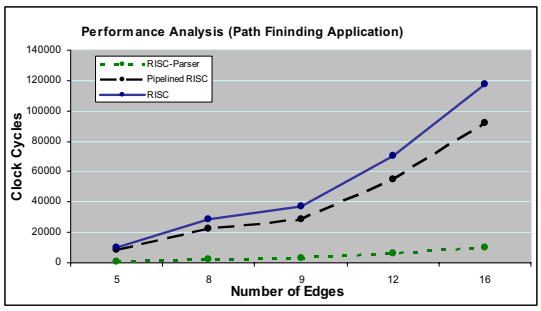


Fig. 18 Performance analysis in clock cycles for possible implementations

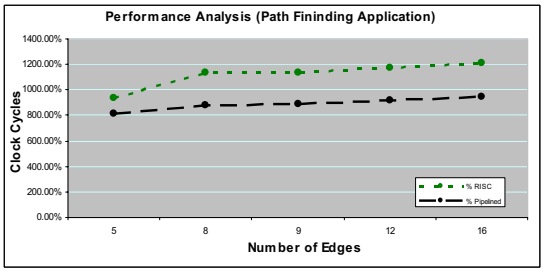


Fig. 19 Increase in performance achieved by our method compared to conventional approaches

Performance using the proposed programmable attribute grammar evaluator has increased by an average of 1000% compared to the purely software approach in a conventional microprocessor. Apparently the increase in performance is larger if we take into consideration memory I/O delays.

VIII. Implementation Details

The proposed approach has initially been verified for its correctness in a testing and evaluation environment that we have implemented in software, using the VISUAL C++ 6.0 environment [28] (Fig. 20). The implemented program accepts as input the syntax and attributes evaluation rules for a given grammar and provides graphically, the way tree derivation process is carried out. It additionally estimates the time required for the completion of the tree derivation for the simple *RISC* approach, the *RISC with pipeline* approach and the proposed *Hardware Parser-RISC* hybrid approach

Having verified the correctness of our approach and getting encouraging results in estimated performance we achieve, we focused on the actual hardware implementation of the intial idea. The proposed implementation has been implemented in synthesizable Verilog in the XILINX ISE 6.0 environment [27] and has been simulated for various FPGA technologies at the net-list level. RISC times have been accurately simulated by a similar implementation of a RISC architecture in synthesizable Verilog in the same environment. The actual optimization results have completely verified our initial expectations.

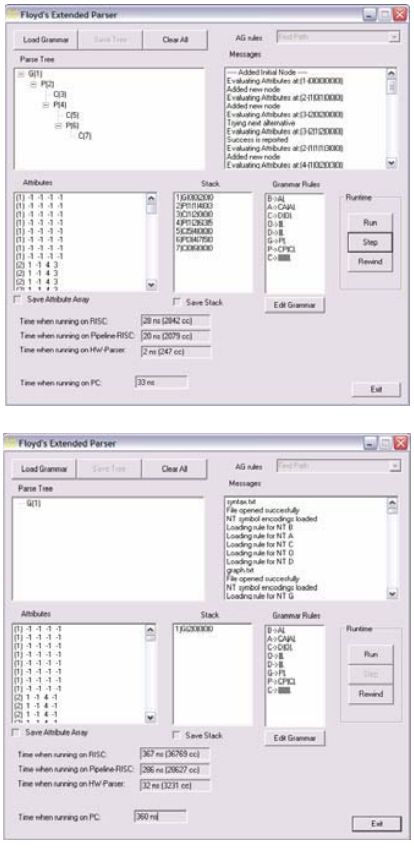


Fig. 20 Two Snapshots of the execution of the software implementation

IX. Conclusion and Future Work

In this paper we have proposed an extended RISC microprocessor for logic programming applications. Towards this goal we have presented a RISC extension that supports the execution of hybrid combinations of declarative-procedural code and a C-extended language that allows such programs to be written. We have also proposed, a hardware programmable implementation of a parser that is attached to the microprocessor, in order to define the execution sequence of attribute evaluation rules creating a programmable semantically driven parser to be used for knowledge representation. As a result, we proposed a complete extended RISC microprocessor, which while supporting all conventional facilities for the execution of procedural programs, is capable of increasing the performance of logic programming computations and allows design flexibility required in embedded system applications. In the case where there is the need of reducing the final proposed system's size and cost, one possible modification to the presented extension is illustrated in Fig. 21. It is possible for the *"Stack Memory"* of the hardware parser to reside in the system's memory and not within the parser. Such an approach reduces the total space required for the implementation but also results in a negative impact to the maximum possible increase in performance.

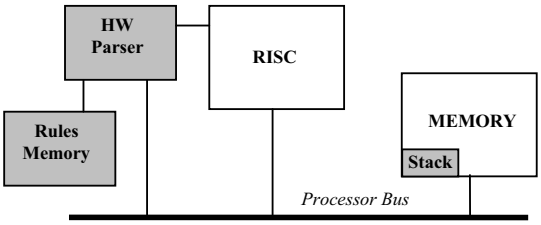


Fig. 21 Abstract architecture when the hardware parser's stack resides in Memory

Another possible modification to the proposed implementation concerns improving the extended microprocessor with pipelining capabilities. In other words, the hardware parser after each call to an EVAL function for the evaluation of attribute instances does not block until attribute instances are evaluated to determine the value of the *flag*, but continues execution. In that case the hardware parser assumes that the derivation has been successful and continues creating the corresponding derivation tree. In the case where attribute evaluation has been successful the hardware parser will have already computed a part of the rest of the derivation tree. In the different case the computed derivations are flushed and a new alternative is evaluated. Such pipelining technique is expected to improve even more the system's performance and is one of our main focus points in future releases of the implementation. Our current efforts are also focused in increasing the efficiency of the implemented hardware parser by using different more efficient parsing algorithms such as programmable modifications of the ones presented in [21]

space were drawn from the Cauchy distribution, mainly because its fatter tails are more likely than the Gaussian distribution to produce the rapid and/or large shift in attention allocation that has been reported by some empirical studies. However, with a proper experimenter-defined parameter setting (e.g., initial temperature & temperature decreasing rate), such shifts in attention might have been achieved with the Gaussian distribution. Moreover, it may be possible that shifts could be drawn from other types of distributions, including rectangular, skewed, or multi-modal distributions. Further simulation and empirical studies seem useful for investigating this issue.

ACKNOWLEDGMENT

Authors thank Stephen Jose Hanson, Catherine Hanson, Yasuaki Sakamoto, Areti Chouchourelou, and researchers at RUMBA for their helpful comments and suggestions.

REFERENCES

[1] Kruschke, J. K. (1992). ALCOVE: An exemplar-based connectionist model of category learning, *Psychological Review, 99.* 22-44.
[2] Kruschke, J. K., & Johansen, M. K. (1999). A model of probabilistic category learning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *25*, 1083-1119.
[3] Love, B. C. & Medin, D. L. (1998). SUSTAIN: A model of human category learning. *Proceeding of the Fifteenth National Conference on AI (AAAI-98),* 671-676.
[4] Matsuka, T. (2002). Attention processes in computational models of category learning. Unpublished doctoral dissertation. Columbia University, New York, NY.
[5] Matsuka, T. & Corter, J. E. (2003). Empirical studies on attention processes in category learning. Poster presented at 44th Annual Meeting of the Psychonomic Society. Vancouver, BC, Canada.
[6] Matsuka, T., Corter, J. E. & Markman, A. B. (2003). Allocation of attention in neural network models of categorization. Under review
[7] Bower, G. H. & Trabasso, T. R. (1963). Reversals prior to solution in concept identification. *Journal of Experimental Psychology, 66*, 409-418.
[8] Rehder, B. & Hoffman, A. B. (2003). Eyetracking and selective attention in category learning [CD-ROM]. Proceedings of the 25th Annual Meeting of the Cognitive Science Society, Boston, 2003.
[9] Macho, S. (1997). Effect of relevance shifts in category acquisition: A test of neural networks. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 23,* 30-53.
[10] Ingber, L. (1998). Very fast simulated annealing. *Journal of Mathematical Modelling, 12:* 967-973.
[11] Nosofsky, R. M. (1986). Attention, similarity and the identification –categorization relationship. *Journal of Experimental Psychology: General, 115*, 39-57
[12] Nosofsky, R. M., Palmeri, T. J., McKinley, S. C. (1994). Rule-plus-exception model of classification learning. *Psychological Review, 101*, 53-79
[13] Shepard, R. N., Hovland, C. L., & Jenkins, H. M. (1961). Learning and memorization of classification. *Psychological Monograph, 75* (13).
[14] Bettman, J. R., Johnson, E. J., Luce, M. F., Payne, J. W. (1993). Correlation, conflict, and Choice. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 19,* 931-951.
[15] Nosofsky, R. M., Gluck, M. A., Palmeri, T. J., McKinley, S. C., & Glauthier, P. (1994). Comparing models of rule-based classification learning: A replication and extension of Shepard, Hovland, and Jenkins (1961). *Memory and Cognition, 22,* 352-369.
[16] Matsuka, T. (In press). Generalized exploratory model of human category learning. *International Journal of Computational Intelligence*.