AN INTERACTIVE HIGH-LEVEL SYNTHESIS TOOL BASED ON ATTRIBUTE GRAMMARS

IOANNIS POULAKIS, GEORGE ECONOMAKOS, GEORGE PAPAKONSTANTINOU and PANAYOTIS TSANAKAS

Dept. of Electrical and Computer Engineering
National Technical University of Athens
Zographou Campus, GR-15773 Athens,GREECE

Abstract Computer-aided synthesis of digital circuits from behavioral level specifications offers an effective way to deal with the increasing complexity of digital hardware design. A high level synthesis tool transforms an abstract algorithmic description into a detailed register transfer level implementation. Even though considerable research has taken place, regarding high-level synthesis, practical implementations are just emerging. This happens due to the fact that designers demand interaction at both the specification and implementation level. This paper explores an original idea, as well as the corresponding implementation, for the design of a grammar based interactive design environment, which allows designers supplement high-level synthesis optimizations with their implementation preferences. The suggested methodology raises the feasibility for high level design space exploration by enabling synthesis results to be directly modifiable by the user. IMACS/IEEE CSCC'99 Proceedings, Pages:6381-6387

Key-Words: Attribute grammars, High-level synthesis, context free grammars, attributed behavior.

1. Introduction

Attribute grammars (AGs) were devised by Knuth [8], as an extension of context-free grammars. The original motivation has been to facilitate the compiler specification and development procedure. Generally an AG can be seen as a mapping from a language described by a context free grammar (CFG) into a user defined domain. Since the beginning, attribute grammars have been a subject for continuous research, both from a conceptual and a practical point of view. Numerous of automated AG based systems, that generate different kinds of language processors from their high level specifications, have been produced. The ability to develop such systems is one of the main advantage of AGs over other formal specification methods.

High-level synthesis (HLS) [5], [9], [10], [16], systems have been proved to be very effective in supporting the design of Very Large Scale Integration (VLSI) digital circuits. HLS accepts a behavioral specification of a digital system, along with a set of constraints on the resulting hardware, and finds a structure that implements the given behavior while satisfying the given constraints. The behavior is usually described as an algorithm, similar to a programming language description. The output structure is a register transfer level implementation which includes a data-path and a control unit. Control activates components of the data-path to realize the required behavior. The objective of synthesis is to find a structure that meets the constraints while optimizing some cost function like the required hardware resources or the power consumed. The main advantages of HLS is that the produced design is more portable, moreover it is very easy to incorporate changes in the specification level. The specification is independent of the target technology and consequently the design life is likely to increase.

Although HLS has become a hot research topic, the designers still prefer semi-automated (with automatic optimizations starting at lower abstraction levels) or even manual methodologies. This happens because a fully automated design offers little interaction, so it is hard to verify that all constraints are satisfied. Generally, graphical user interfaces are being used, in order to make HLS environments more productive. A robust language based HLS interactive methodology, called the attributed-behavior specification, has been presented in [1]. Following this approach, the input to HLS is not just a behavioral description but a set of properties

(attributes) that must hold in any implementation. These attributes refer to the textual algorithmic description. The aim of that work was to introduce three relationships (path, cost, delay) between the attributes. The user has to provide both a textual algorithmic and relation description. Synthesis is the task of defining the values of all attributes that have not been supplied.

The same idea was recently adopted by Seawright *et al* [13] to develop Clairvoyant, a system to perform hardware compilation using production-based specifications

As an extension to Clairvoyant, Oberg *et al* [12] have presented PRO-GRAM, a YACC-like grammar-based synthesis environment for data communication protocols.

Similar to PRO-GRAM was the extension of Clairvoyant presented by Seawright *et al* [14], the system called Dali. Dali uses a graphical hierarchical representation to describe systems that process structured data streams or handle structured control protocols.

AGs were introduced to the field of design automation by Jones *et al* [6], who presented an AG based solution to the incremental evaluation of properties and conditions in VLSI circuits for the development of interactive design editors. This work involved circular AGs with increased evaluation time, and did not study the feasibility of applying them for practical design systems.

A tool for silicon compilation, was the syntax-directed system developed by Keutzer *et al* [7], which was based on the same ideas with our current work.

This paper adds user interaction to the AGENDA [2] AG driven HLS environment and extends the work presented in [3]. It follows the idea of the attributed-behavioral system specification (where a system is described as a pair of a behavior and a set of attributes that must hold for any implementation) and the fact that in the AGENDA environment, all HLS transformations are performed through attribute evaluation rules. Allowing the direct manipulation of some attribute values by the user (with proper initializations), his/her intentions or design preferences can be directly passed into any implementation. Thus, interaction is introduced. By iterating over this process, all serial/parallel tradeoffs in behavioral modeling are handled in a unified, formal environment.

This paper proposes an attributed-behavioral synthesis methodology, using AGs as a formal underlying framework over which HLS is performed, following and extending the work presented in [4] and [5]. The advantage of using the attributed-behavior approach at the algorithmic level, is the flexibility that stems from the use of a single interactive synthesis tool and modeling language. As far as AGs are concerned, they can support a formal and flexible implementation framework for attributed-behavioral design optimizations. In fact, the attributed-behavioral specification can be synthesized using the same techniques found in [2] and [3] with the simple addition of proper attribute initializations. Overall, our approach responds to the constraint relationships provided by the engineer and produces a design that best meets these requirements.

2. Proposed Methodology

Despite the fact that HLS has been a hot research topic, it has gained little acceptance from industry. This is partially due to the fact that designers require much more interaction with the synthesis process than what design automation is offering. Since design complexity prohibits interaction at the lower levels, HLS is forced to support it. A methodology that can increase HLS interactivity for language based design entry is the attributed-behavior circuit specification.

An attributed-behavior specification consists of a *textual algorithmic description* accompanied by a set of properties (attributes) that must hold in any implementation of the description. The textual algorithmic description is a procedural specification with syntax similar to conventional programming languages. For the scheduling problem, we have defined four attributes, four operator relationships in the temporal domain, called *step, delay* [3], *c_step* and *group*. All the relationships, step, c_step, delay and group, refer to an operator and they are denoted themselves like referential operators in the textual algorithmic description. The hardware interpretations of the relationships are:

Step: [3] Assigns the control step in which the referential operation will be executed,(syntax $o_i[n]$).

Delay:[3]Assigns the number of control steps that the operation will be delayed. (syntax $o_i[\hat{}n]$).

c_step: Assigns the control step in which the referential operation will be executed. There is no further check for the desired control step. The desired control step is selected regardless the status of the operands. For instance if we want operation o_i to be executed in the n^{th} control step we add the token [&n] in front of o_i in the behavioral description ([&n]o_i).

Group: In the textual algorithmic description we can discriminate some of the operators in groups.

We can have more than one groups but each operator may belong in one group only. Grouping the operators means that the operations will be executed at the same control step The question is how shall we decide the control step the operations will be executed. In this case we suggest two alternatives: i)The user can group the operators and define the control step on which they will be executed. ii)The user will only group the operators and let the system decide for the control step they will be executed.

In the first case the user can use step or c_step attributes to define the control step while in the second case the selected control step is the latest one among the control steps of the operations in the group, assuming that no grouping was performed.

All the relationships are used to encode the designer's preferences regarding the implementation of the behavior and thus, have a direct impact on the operation of the current scheduling algorithm (CSA).

The designer may use the attributes in two ways:

a) as operators in the textual algorithmic description and b) as operations performed by the user after the schedule is calculated

The two cases are totally autonomous. The user can apply the four relationships in the textual algorithmic description so as to set the constraints related to the problem or/and after the schedule is done to make additional necessary changes to the schedule. In this part we can see a more powerful interaction between the designer and the system. The designer knows the arisen schedule and he may make changes to the value of the control step for each operation, to set new relationships among the operators.

The implementation of an attributed-behavior HLS environment can be made using any suitable method. However, there exists a strong correlation between this interactive specification method and the formal specification and implementation framework of the AGENDA AG driven HLS environment [2].

The attributed-behavior interactive synthesis technique has been implemented on top of the AGENDA formal AG based environment. Except from the work described in [2] and [3], attributes are also attached to the step, c_step, delay and group relationship grammar symbols, that hold their corresponding numerical values (Notice the difference between the attributes of the behavioral description and the attributes of the non-terminal grammar symbols. The syntax o_i[n], o_i[&n] or o_i[^n] denote that n is a step, c_step or delay attribute of o_i and the syntax o_i[g n] denotes that o_i belongs to group n. Using AGs, both o_i and n are non-terminal symbols that can have different attributes attached, according to the AG driven application that is to be constructed). Scheduling attribute evaluation rules can use these numerical attribute values to implement the modified scheduling heuristic. In fact, the way scheduling attributes propagate through the parse tree is not changed at all. Only proper initialization rules are attached to the leaves. For example, consider the following operation parsing rule with a step relationship.

```
operation →
operand1 operator [number]operand2 (1)
```

Considering the above modified CSA scheduling heuristic's requirements, it is implied that inputs must be scheduled before they are needed and that each output must be scheduled in the next control step after all its inputs have been scheduled. This can be accomplished by using a *synthesized* attribute `s_cs` (whose value depends on values of successor nodes in the parse tree) to pass scheduling information from inputs to outputs, with the following evaluation rule, in any syntactic rule similar to (1). In the following CSA is the control step estimated by the currently used scheduling algorithm .

```
operation.step=number
operation.s_cs=valid(operation.step) ?
operation.step :
operation.CSA
```

The step relationship is involved using the shorthand [? :] conditional operator (used exactly like in the C programming language). As it can be seen, it has a straightforward correspondence with an attribute of the AG formalism. In fact, constraint relationships can be regarded as the initial values (under conditions) of the scheduling attribute `s_cs`.

The same applies in the case of the delay relationship. The following is an operation parsing rule with a delay relationship.

```
operation →
 operand1 operator[^number]operand2 (2)
```

Scheduling is performed with the following evaluation rule.

```
operation.delay=number
operation.s_cs=operation.delay+
            +operation.CSA;
```

In the case of the c_step relationship, the parsing rule (3) is similar to step and delay parsing rules.

```
operation →
operand1 operator [&number] operand2(3)
```

It is obvious that inputs are scheduled independently from outputs thus the designer has to be sure that inputs will have the desired values in the control step he defines. No synthesized attributes are needed to pass information from input to output. The attribute *s_cs* is used to define the control step of the related operation and to pass scheduling information to higher nodes in the parse tree. Scheduling information arises from the following evaluation rule.

```
operation.step=number
operation.s_cs= operation.step
```

In the case of group relationship the parsing rule has two similar forms depending on the way it is used. If group relationship is used combined with *step* or *c_step* relationships then the parsing rule is as presented in (4).

```
operation →
operand1 operator [&number1][g number2]
operand2                                (4)
operation.step=number1
operation.s_group_cs=operation.step
operation.group=number2
insert_group(operation.s_group_cs,
      operation.group,operation.flag);
operation.s_cs=get_group_cs(operation.
          group,operation.i_group_cs);
```

If group relationship is used naturally (not related with other relations) then the parsing rule is as presented in (5).

```
operation →
operand1 operator [g number]operand2(5)
operation.step=operation.CSA
operation.s_group_cs=operation.step
operation.group=number
insert_group(operation.s_group_cs,
      operation.group,operation.flag);
operation.s_cs=get_group_cs(operation.
          group,operation.i_group_cs);
```

The basic difference between (4) and (5), is the way the decision of the control step is made. In (4) *group* is combined with *c_step* relationship, so the control step is decided by the user, while in (5) the control step is estimated by the current scheduling algorithm. In both (4) and (5) we can see two functions **`insert_group()`** and **`get_group_cs()`**.

insert_group() is used to relate the estimated control step of the current operator with the other members of the same group. The **`operation.flag`** attribute is used to specify weather parsing rule (4) or (5) was used. If operation.flag is set then **`number`** is the value of the group's control step, otherwise the control step of the group is the latest control step of all the members of the group.
get_group_cs() is used to retrieve the final value for the control step of the group.

Considering the above heuristic's (4), (5) requirements, it is obvious that the output is related not only with inputs **`operand1`** and **`operand2.`** In (4) the control step of all the group members is set to **`number1.`** In (5) as we have mentioned earlier the control step for each of the group members arises from the maximum value of the control steps of the operations, using the current scheduling algorithm. To respect this relationship before we decide the control step of the group we have to know the latest control step of all operations. To accomplish this we have introduced two more attributes, one synthesized ***s_group*** and one inherited ***i_group***. As we can see in the parse tree of fig.1 attribute *s_group* moves upwards in the parse tree from the grouped nodes towards the root, carrying information about the control step the operation would be executed using CSA. When *s_group* reaches the root, *i_group* inherits it's value and carries it downward towards the rest nodes of the same group. When *i_group* reaches these nodes it means that all *s_group* attributes have been calculated and then the control step can be calculated.

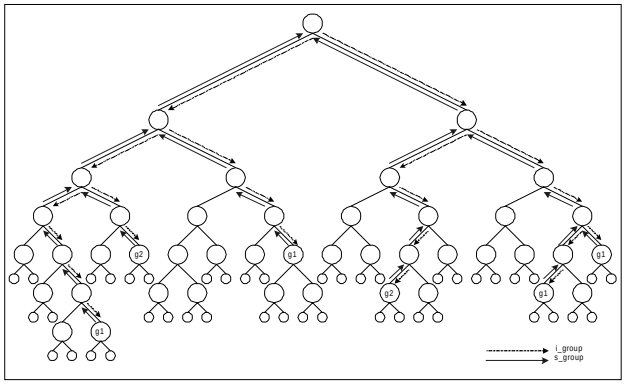

Fig1. s_group and i_group transfer information within a parse tree

All the attributes can be used more than one time independently or combined (ex [&2] [g 1]) in the parse tree of the behavioral description. A symbol

used as input in one application is used as output in the next and so scheduling attributes are passed in a bottom up fashion throughout the whole tree. In this way, the whole behavioral description is scheduled.
The step, c_step, delay and group attributes are not depending on the scheduling algorithm. They can be used in any scheduling algorithm to meet constraints set by the hardware. To use the attributes we suggest in any scheduling algorithm we have to reform the algorithm as presented in fig.2. In the following with CSA(n) we mean the control step that is calculated by the current scheduling algorithm.

Using an attributed-behavior HLS environment, designers can increase their interaction with the synthesis process. They can iterate through different implementations of the desired functionality changing the supplied step, c_step, delay and group attributes until a desired implementation is generated.

The advantage of this technique is the declarative and modular notation (as presented above) for the specification and implementation of each scheduling heuristic. With the adoption of the attributed-behavioral technique, user interaction is considerably increased. Using step, c_step, delay and group attributes in a scheduling algorithm increases its flexibility. For example, in the case of a resource constrained problem, certain scheduling heuristics may be trapped to local minimal solutions and fail to give the desired solution. In such cases, the designer may be able to aid the synthesis tool using step, c_step, delay and group to bind resources with specific operations. An example of resource usage minimization using an attributed-behavioral specification is presented in the following section

```
for each operation o_i
 if o_i is scheduled in control step n using c_step
      cstep(o_i) = n
 else if o_i is scheduled in control step n using step
    if n is a valid control step
      cstep(o_i) = n
    else
      cstep(o_i) = CSA(n)
 else if o_i is scheduled using delay
    cstep(o_i) = n+CSA(n)
 else if o_i is scheduled using group
    cstep(o_i) = maximum cstep of all the o_i of the same group
 else                              /* Normal scheduling */
    cstep(o_i) = CSA(n)
```

Fig.2 Reformed scheduling algorithm to include step, c_step, delay and group attributes

3. Experimental Results

The technique presented so far has been implemented on top of the SDP [15] attribute evaluator generator tool. The implementation consists of an AG, following the syntax of SDP. Considering that the scheduling AG has already been constructed from our previous work, the implementation of the attributed-behavioral specification was straightforward. This is verified by the fact that for 862 lines of AG code required for the implementation of AGENDA (basic scheduling and housekeeping), 1378 were used to implement the ideas of this paper.

```
Program inverse_A;
Begin
D:=a1*b2*c3+b1*c2*a3+c1*a2*b3-a3*b2*c1-
b3*c2*a1-c3*a2*b1;

if (D>0)
Begin
A1:=b2*c3-b3*c2; A2:= b3*c1-b1*c3; A3:=b1*c2-b2*c1;
B1:= a3*c1-a2*c3; B2:=a1*c3-a3*c1; B3:= a3*c1-a1*c3;
C1:=a2*b3-a3*b2; C2:= a3*b1-a1*b3; C3:=a1*b2-a2*b1;
a11:=A1/D; a21:=B1/D; a31:=C1/D;
b11:=A2/D; b21:=B2/D; b31:=C3/D;
c11:=A3/D; c21:=B3/D; c31:=C3/D;
End
End.
```

Fig.3 Estimation of inverse matrix A^{-1} using ASAP scheduling.

As a design example of the resulting environment we present part of our work for fast transitory effects in electric power systems, the estimation of the inverse matrix A^{-1} of A. For simplicity we present the case of a 3x3 matrix.

In Fig.3 we can see the behavioral specification of the algorithm without the use of the attributes we propose.
In fig.4 we can see the scheduling tree using ASAP local scheduling algorithm. Alternatively we could use ASAP global scheduling. The difference between the two schedules is that the first would require 18 multipliers and 9 control steps and the second would first would require 24 multipliers and 8 control steps. It is obvious that with both scheduling schemes the resulting implementation, is quite expensive (18-24 multipliers, 9 dividers and 9 subtracters require lot of resources).

We can rewrite the behavioral description as an attributed-behavior description using c_step, group and delay relationships. Since we know the specifications of the algorithm we can reform it

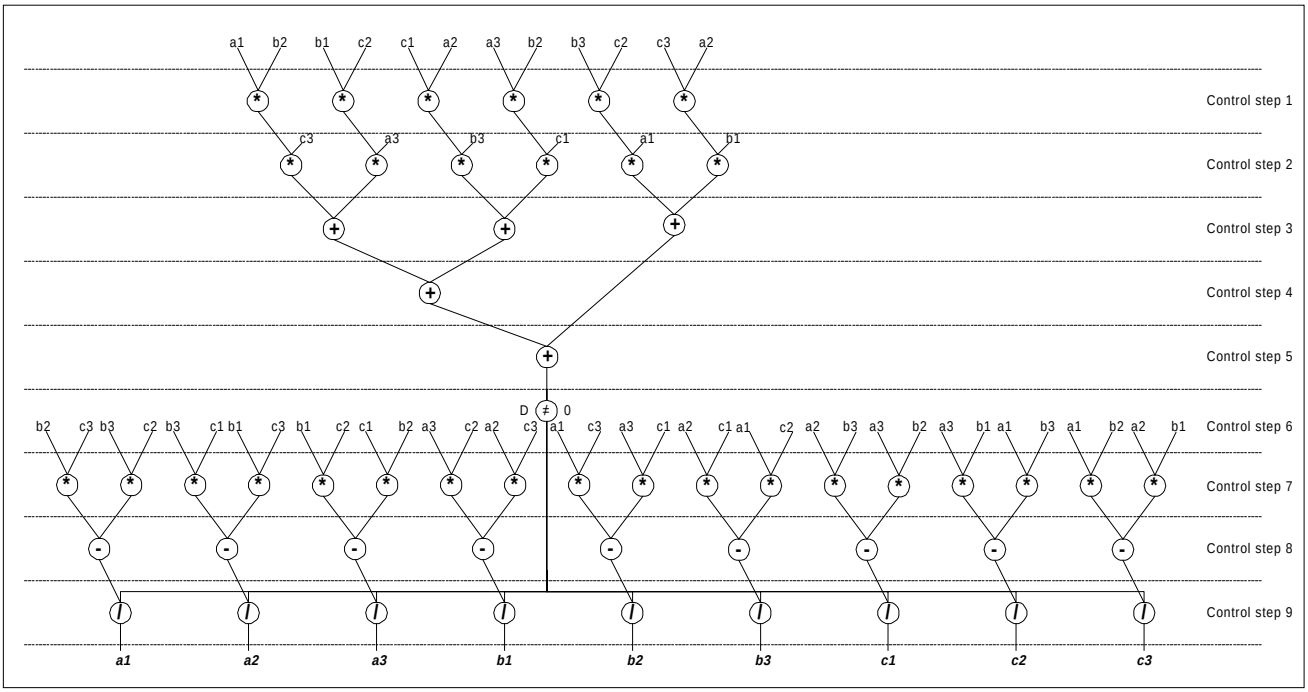

Fig.4. Normal scheduling tree using ASAP scheduling algorithm

using global scheduling from different basic blocks. Specifically we can divide the calculation of the minor determinants in three groups. The first group will start in the 3rd control step, the second in the 4th and the last group in the 5th control step. In this way we managed to reduce the total number of multipliers in 6. The new behavioral description is shown in fig.5 and the corresponding schedule in fig.6.

```
Program inverse_A;
Begin
D:=a1*b2*c3+b1*c2*a3+c1*a2*b3-a3*b2*c1-
b3*c2*a1-c3*a2*b1;

if (D>0)
Begin
A1:=b2*[&3][g 1]c3-b3*[g 1]c2; A2:= b3*[g 1]c1-b1*[g 1]c3;
A3:=b1*[g 1]c2-b2*[g 1]c1; B1:= a3*[&4][g 2]c1-a2*[g 2]c3;
B2:=a1*[g 2]c3-a3*[g 2]c1; B3:= a3*[g 2]c1-a1*[g 2]c3;
C1:=a2*[&5][g 3]b3-a3*[g 3]b2; C2:= a3*[g 3]b1-a1*[g 3]b3;
C3:=a1*[g 3]b2-a2*[g 3]b1;
a11:=A1/D; a21:=B1/D; a31:=C1/D;
b11:=A2/D; b21:=B2/D; b31:=C3/D;
c11:=A3/D; c21:=B3/D; c31:=C3/D;
End
End.
```

Fig.5 Estimation of inverse matrix A^{-1} using attributed behavior ASAP scheduling algorithm.

Comparing the schedules in figs 4 and 6, it is obvious that careful use of the step, delay and group relationships has reduced the resources (6 multipliers instead of 18 or 24 and 3 subtractors instead of 9) needed to calculate A^{-1}. At the same time, less computation latency is achieved (2 control steps less). As it can be seen, the expression has been rescheduled as if a different (resource constrained) scheduling heuristic has been used. Alternatively we could reduce even more the resources with cost in time. We can reduce the dividers from 9 to 5 or 3 increasing the execution time 1 or 2 control steps respectively.

4. Conclusion

An interactive HLS synthesis environment, following the attributed-behavior specification paradigm has been presented in this paper. It takes advantage of the capabilities supported by the AG computational model, that is, declarative and modular design specifications, and allows users to supplement scheduling heuristics with their implementation preferences. Moreover, using the step, c_step, delay and group attributes it allows the designer to interfere with the scheduling algorithm in two ways, using the attributes within the algorithm or after the scheduling algorithm is performed. Step c_step, delay and group can support both minor and major modifications to the scheduling algorithm. Specifically, the more skilled the designer is the more powerful the scheduling algorithm becomes.

This idea can be very helpful in design space exploration and with the implementation flexibility offered by AGs, can support a new paradigm for efficient system level design.

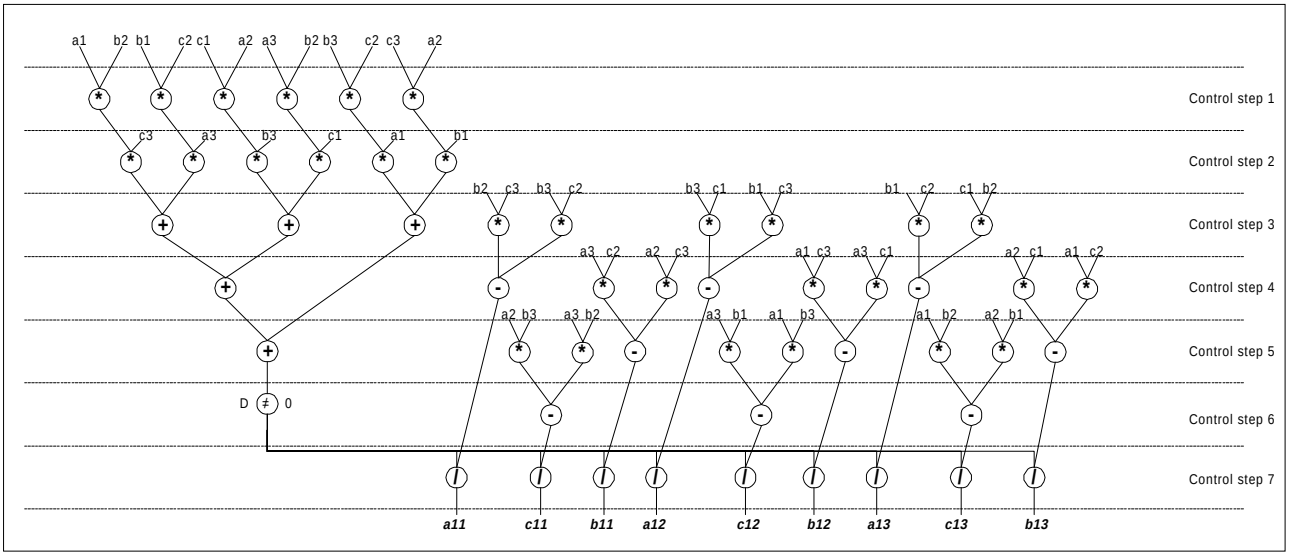

Fig.6. Attributed behavior scheduling tree.

References:

[1] L. F. Arnstein and D. Thomas, "The Attributed-Behavior Abstraction and Synthesis Tools", *ACM/IEEE Design Automation Conference DAC94*, (1994), 557-561.

[2] G. Economakos, G. Papakonstantinou and P. Tsanakas, "AGENDA: An Attribute Grammar driven Environment for the Design Automation of Digital Systems", *ACM/IEEE Design Automation and Test in Europe Conference and Exhibition DATE98*, (1998) 933-934.

[3]G.Economakos, I. Poulakis G. Papakonstantinou and P. Tsanakas,"An Attribute Grammar Based Interactive High Level Synthesis Tool", *International Workshop on Logic and Architecture Synthesis IWLAS98, (1998)*

[4] R. Farrow and A. G. Stanculescu, "A VHDL Compiler Based on Attribute Grammar Methodology", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (1989) 120-130.

[5] D. Gajski, N. Dutt, A. Wu and S. Lin., *High-Level Synthesis* (Kluwer Academic Publishers, 1992).

[6] L. G. Jones J. Simon, "Hierarchical VLSI Design Systems Based on Attribute Grammars", *13th ACM Symposium on Principles of Programming Languages POPL86*, (1986) 58-69.

[7] K. Keutzer and W. Wolf, "Anatomy of a Hardware Compiler", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (1988) 95-104.

[8] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory*, Vol 2, No 2 (1968) 127-145.

[9] Y. L. Lin, "Recent Development in High Level Synthesis", *ACM Transactions on Design Automation of Electronic Systems*, Vol 2, No 1 (1997) 2-21.

[10] M. C. McFarland, A. C. Parker and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, Vol 78, No 2 (1990) 301-318.

[11] M. Naini, "A Dedicated Dataflow Architecture for Hardware Compilation", *22nd Annual Hawaii International Conference on System Sciences*, (1989).

[12] J. Oberg, A. Kumar and A. Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols" *9th ACM/IEEE International Symposium on System Synthesis ISSS96*, (1996) 14-19.

[13] A. Seawright and F. Brewer, "Clairvoyant: A Synthesis System for Production-Based Specification", *IEEE Transactions on Very Large Scale Integration Systems*, Vol 2, No 2, (1994) 172-185.

[14] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe J. and Buck, "A System for Compiling and Debugging Structured Data Processing Controllers", *ACM/IEEE European Design Automation Conference with EURO-VHDL*, (1996) 86-91.

[15] M. Sideri, S. Efraimidis and G. Papakonstantinou, "Semantically Driven Parsing of Context-Free Languages", *The Computer Journal*, Vol 32, No 1 (1989).

[16] R. A. Walker and S. Chaudhuri, "High-Level Synthesis: Introduction to the Scheduling Problem", *IEEE Design & Test of Computers*, Vol 12, No 2 (1995).