Hardware Compilation Using Attribute Grammars

Economakos, G., Papakonstantinou, G., Pekmestzi, K. and Tsanakas, P.
National Technical University of Athens
Department of Electrical and Computer Engineering
Zografou Campus, GR-15773 Athens, Greece
george@dsclab.ece.ntua.gr

Abstract

Attribute grammars have been used extensively in every phase of traditional compiler construction and, therefore, practical attribute grammar evaluators have been developed to automate the task. The similar task of hardware compilation has not however taken advantage of these developments yet. Previous work has shown that attribute grammars can be effectively adopted to handle high-level hardware synthesis. In this paper, these past results are further elaborated and integrated in the construction of a prototype for an attribute grammar driven hardware compiler from behavioral descriptions to VHDL. Due to its flexibility and rapid design, such a compiler can be used as a workbench for testing various synthesis algorithms and optimization criteria. This novel approach can be proven valuable for evaluating new algorithms and techniques in the field.

Keywords

Hardware compilation, high-level synthesis, automated hardware synthesis, attribute grammars, formal synthesis methods, attribute grammar evaluators

©IFIP 1997 Published by Chapman & Hall

1 INTRODUCTION

Attribute grammars (AGs) were devised by Knuth (Knuth, 1968) as a tool for the formal specification of programming languages. However, in the general case, an AG can be seen as a mapping from the language described by a *context free grammar* (CFG) into a user defined domain. Since their introduction, they have been a subject of intensive research, both from a conceptual and from a practical point of view. The conceptual work has produced several subclasses of attribute grammars (Paaki, 1995) with advanced implementation algorithms. The closely coupled pragmatic efforts have created a large number of automated systems based on attribute grammars. These systems, usually called *compiler-compilers*, *compiler writing systems*, or *translator writing systems*, generate different kinds of language processors from their high-level specifications.

The development of such systems is the main advantage of AGs over other formal specification methods; that is, they can also be used as an executable method, for the automatic construction of a program, which will implement the specified mapping. This advantage has made AGs one of the most widely applied semantic formalisms.

Traditionally, AGs have been extensively used in compiler construction (Aho, 1986), (Deransart, 1988). Recently they have also been adopted in many other areas such as knowledge representation (Papakonstantinou, 1986a), (Papakonstantinou, 1986b), logic programming (Arbab, 1986), (Deransart, 1985), and as a dataflow language (Papakonstantinou, 1988). In the field of AG-driven compiler construction, a lot of work is presented by Aho et al (Aho, 1986) and Waite and Goos (Waite, 1984). In the field of AG-driven dataflow computing, Farrow (Farrow, 1983) first showed that an attribute evaluator can be viewed as a dataflow program that computes the translation of a source string, when its parse tree is given as input and Papakonstantinou and Tsanakas (Papakonstantinou, 1988) presented a method for the dataflow extraction of computational algorithms.

A recent extension of these two fields is the high-level automated hardware synthesis of special purpose architectures (Camposano, 1991), (Gajski, 1992), (Hafer, 1983), (Lin, 1997), (McFarland, 1990), (Paulin, 1989), (Tanaka, 1989), (Trickey, 1987), (Walker, 1991), (Walker, 1995). It is defined as the transformation of behavioral circuit descriptions into *register-transfer level* (RTL) structural descriptions that implement the given behavior while satisfying user defined constraints, and can be seen as either a compilation process, or a dataflow computation over a loosely defined hardware architecture. The result of this transformation is the exact definition of the optimal (or suboptimal) architecture for each given behavior.

High-level synthesis is an evolving research topic in the field of design automation, with a lot of recent work being published. However, concerning its acceptance in the industrial world, we must recognize that a lot of problems are still open (Gajski, 1992). The reasons are manifold. Over the last 10 years, many systems were introduced increasing the complexity of hardware description languages, design capture methods and technologies that each new approach has to consider. In the contrary with the lower levels of design abstraction, high-level synthesis lacks the existence of a theoretical framework (like Boolean algebra for logic design) that would further accelerate research. The majority of optimization problems faced are NP-complete, thus heuristics are mandatory.

All these problems motivated the search for formal methods to describe and perform high-level synthesis. One of the first proposals (Hafer, 1983), integer linear programming, is still considered as a widely used methodology (Lin, 1997). However, its computational complexity limits its application to very small problems.

Attempting to overcome inefficiencies and propose a unifying formal framework for high-level synthesis, Economakos *et al* (Economakos, 1995) proposed an attribute grammar based approach. Earlier, other formal methods for the automated synthesis of special purpose architectures had been investigated, like FP (Tsanakas, 1989), (Tsanakas, 1992) and PROLOG (Tsanakas, 1991).

This, to the best of our knowledge, was the first attempt to fully describe the whole process of hardware synthesis using AGs. Earlier, Naini (Naini, 1989) presented a dataflow based solution, Farrow *et al* (Farrow, 1989) presented an AG driven compiler of the VHDL hardware description language (Bhasker, 1992), (Lipsett, 1993) and Jones *et al* (Jones, 1986) presented an AG based solution to the incremental evaluation of properties and conditions in VLSI circuits. This last work was very thorough but rather complicated, involving circular AGs, and has not been tested in practical design systems. Circular AGs were involved because the authors used an internal representation of the VLSI circuit (generally composed of circles) as the design tree, to which attributes were attached. On the contrary, the approach presented in (Economakos, 1995), was much simpler and attached non-circular attributes to the parse tree of the behavioral specification language. Jones aimed at the development of interactive design editors while our approach was aimed at the process of ***hardware compilation***. Such a tool, was the syntax directed system developed by Keutzer *et al* (Keutzer, 1988). However, this work was aimed at a lower level of abstraction. It faced the problem of register-transfer level realization, that is, the optimal transformation of an FSM architecture into netlists of digital gates, and used more than one language processors.

In this paper the results of both (Economakos, 1995) and (Economakos, 1997) are further elaborated for the implementation of an AG driven high-level behavioral hardware compiler. The internal representation of the design consists of a *Control/Data Flow Graph*, which directly designates the structure of the design. To accomplish this, control structures were investigated and the proposed scheduling algorithms were expanded to cross control boundaries while preserving the overall behavior (global scheduling). Also, the language used for behavioral description has been extended to handle declarative as well as procedural semantics. The later can describe behavior efficiently. However, the former is also needed to describe real world entities and interfaces that will be synthesized. Finally, implementation problems were solved and a VHDL translator tool has been realized to be used as a preprocessor for VHDL based simulation and synthesis tools.

Even though many hardware compilation tools have been presented in the past (Biesenack, 1993), (Keutzer, 1988), (Thomas, 1990), (Walker, 1991), our approach has the advantage of being flexible, because it can easily incorporate different algorithms by changing the appropriate attributes. Consequently, it can be used as a testbench for the evaluation of new design algorithms, thus facilitating the rapid development of hardware compilers.

The rest of this paper is organized as follows. Section II presents some basic ideas about AGs and high-level hardware synthesis. Section III gives a detailed description of a hardware compiler based on the ideas presented in (Economakos, 1995) using common compiler construction tools. Section IV presents experimental results and, finally, section V gives the conclusions of the presented work and proposes possible extensions.

2 PROBLEM DEFINITION

2.1 Attribute grammars (AGs)

An *attribute grammar* (AG) (Knuth, 1968) is based upon a *context free grammar* (CFG) G=(N,T,P,Z), where N is the set of nonterminal symbols, T is the set of terminal symbols, P is the set of productions (syntactic rules) and Z ($Z \in N$) is the start symbol.

Each symbol in the vocabulary V ($V = N \cup T$) of G has an associated set of attributes A(X). Each attribute represents a specific context-sensitive property of the corresponding symbol. The notation X.a is used to indicate that attribute a is an element of A(X). A(X) is partitioned into two disjoint sets; the set of *synthesized attributes* AS(X) and the set of *inherited attributes* AI(X). *Synthesized attributes X.s* are those whose values are defined in terms of attributes at descendant nodes of node X of the corresponding semantic tree. *Inherited attributes X.i* are those whose values are defined in terms of attributes at the parent and (possibly) the sibling nodes of node X of the corresponding semantic tree.

Each of the productions $p \in P$ ($p=X_0:X_1...X_n$) of the CFG is augmented by a *semantic condition* SC(p) and a set of *semantic rules* SR(p). A semantic condition is a constraint on the values of certain attributes that are elements of the set $\cup_{i=0..n}A(X_i)$. The semantic condition SC(p) must be satisfied in every application of the production (syntactic rule) p. A semantic rule defines an attribute in terms of other attributes of terminals and nonterminals appearing in the same production. The semantic rules associated with production p define all the synthesized attributes of the nonterminal symbol X_0 (on the left-hand side of p), as well as all the inherited attributes of symbols $X_1,...,X_n$ (on the right-hand side of p).

An attribute grammar is, therefore, defined by the five-tuple AG=(G,A,D,SR,SC), where:

- G is a reduced CFG
- $A=\cup_{X \in V}A(X)$ is a finite set of attributes
- D is the set of domains of all attribute values
- $SR=\cup_{p \in P}SR(p)$ is a finite set of semantic rules
- $SC=\cup_{p \in P}SC(p)$ is a finite set of semantic conditions.

The analysis of an input string by an AG interpreter proceeds in two phases. In the first, called *syntax analysis*, a context-free parser is used to construct a parse tree of the input string. In the second, called *semantic analysis*, the values of the attributes at the nodes of the parse tree are evaluated and the semantic conditions are tested.

AGs are using a nonprocedural formalism. Therefore, they do not impose any sequencing order in the process of parsing or in the process of evaluating the attributes. Instead, they just describe the dependencies among the syntactic structures and among the attribute values. Consequently, they can be adopted to define a sequencing order for the subcomponents of any language based description, inferred from attribute dependencies.

2.2 High-level automated hardware synthesis

Hardware synthesis is the task of searching for a set of interconnected components (*structure*), which implements a certain way of component and environment interaction (*behavior*), while satisfying a set of goals and constraints. Eventually, the structure must be mapped into a *physical design*, i.e., a specification of how the system is actually to be built. Behavioral, structural and physical are distinguished as the three*domains* in which hardware can be described.

Just as designs can be described in different domains, they can also be described at various levels of abstraction in each domain. Traditionally, abstraction levels are presented as concentric circles on an Y-chart (Gajski, 1992). On top of the design hierarchy is the so-called *system* level, where computer systems are described as algorithms, interconnected sets of processors, chips and boards in the different domains. The next level is called *microarchitectural* or *register-transfer* level with focus on register transfers, netlists of ALUs, MUXs and registers and module floorplans. Next comes the *logic* level, where the system is described with Boolean equations, as a network of gates and flip-flops or as geometrically placed modules. Last comes the *circuit* level, which views the design in terms of transistor functions, transistor netlists or wire segments and contacts.

We define *synthesis* the process of translating a behavioral description into a structural description, similar to the compilation of conventional programs into assembly language. *High-level synthesis*, as we use the term, means going from a system level behavioral specification of a digital system, to a register-transfer level structural description that implements that behavior. Obviously, there are many different structures that can be used to realize a given behavior. Consequently, one major task of high-level synthesis is to find the best structure that carries out the required computations and meets user defined constraints, such as limitations on cycle time, area, power, etc.

Design entry in a high-level synthesis system is an algorithmic description written in a common programming language (like PASCAL or FORTRAN), or by a special purpose *hardware description language* (HDL), such as ISPS, DSL or VHDL.

The first step in high-level synthesis is usually the compilation of the formal HDL specification into an internal representation. Most approaches use graph-based representations that contain both the data and the control flow implied by the specification. Such representations are called *Control/Data Flow Graphs* (CDFGs). Operations in the behavioral descriptions are mapped as nodes in the CDFG and values as edges. Additionally, the CDFG can also represent conditional branches, loops, etc., hence the name control/data flow graph.

Once the CDFG has been constructed, the three central synthesis tasks in a typical high-level synthesis system are the following:

- *Scheduling* – determining the sequence in which the operations are executed provided a sequence of discrete time slices called *control steps.*
- *Allocation* – selecting the appropriate number of functional units, storage units and interconnection units from available component libraries.
- *Binding* – assigning operations to functional units, assigning values to storage units and interconnecting these components to cover the entire data path.

Many high-level synthesis systems combine allocation and binding into the same task and call this combined task allocation. Also, the solution to any of the three major tasks under user defined constraints, is strongly related to the solutions to the others. Most of the problems arising in this combined optimization case are NP-complete, so heuristics are mandatory.

3 AG-DRIVEN HARDWARE COMPILER

The design complexity of VLSI architectures has grown exponentially making the traditional *capture-and-simulate* design methodology obsolete in many cases. A new *describe-and-synthesize* methodology has become necessary. The first step in this methodology is the high-level synthesis of a behavioral description performed by hardware compilation tools. This section is a detailed description of an AG-driven hardware compiler based on ideas presented in (Economakos, 1995).

3.1 Input specification

Design entry in a high-level synthesis system is an algorithmic description written in a conventional or special purpose HDL. All HDLs exhibit some common programming language features, including constructs like data types, operators, assignment statements and control statements, supporting behavioral abstractions in different levels. Supplementary, hardware specific properties are also supported by modern HDLs with constructs like interface declarations, structural declarations, register-transfer and logic operators, asynchronous operations and constraint declarations. Finally, all HDLs define an execution ordering, with sequential and parallel threads of operation.

The experimental HDL used in (Economakos, 1995) as the underlying CFG that was decorated by a scheduling AG, was a subset of PASCAL, presented in (Aho, 1986). This language had only procedural semantics to describe behavior. However, for hardware design entry, declarative semantics are also needed to define entities and interfaces between entities, i.e. the I/O ports of the design and their mode of operation, as mentioned above. An HDL containing both procedural and declarative semantics, constructed as an extension of a common procedural language, is HardwareC (Ku, 1990). A subset of HardwareC is used in the hardware compiler presented here. Its syntax is defined in figure 1.

This HDL includes the same procedural semantics as the HDL of (Economakos, 1995) as well as entity, port, and signal declarations. The execution ordering imposed on every behavioral description is strictly sequential. The descriptions produced are, therefore, closer to the way a human designer conceives the abstract functionality of the desired structure. The hardware compilation process can evaluate all possible parallel orderings of the sequential behavior and evaluate all possible architectural tradeoffs, by applying appropriate algorithms.

3.2 Internal representation

As stated above, the first step in high-level synthesis is the compilation of the input specification to a dataflow type internal representation. This step has many similarities with dataflow computing, for which, an AG formalism has been given in (Papakonstantinou, 1988). These similarities were exploited in (Economakos, 1995), where a first AG formalism for hardware compilation was given.

```
design → block id ( port_declarations_list) compound_statement.
port_declarations_list→ port_declarations |port_declaration_list; port_declarations
port_declarations→ mode port declaration_list
mode → in | out | inout
declaration_list → declaration | declaration_list , declaration
declaration → id | id [ num ]
compound_statement→ begin optional_declarations optional_statementsend
optional_declarations → id_declarations_list; | e
id_declarations_list → identifier_declrations| id_declarations_list; identifier_declarations
identifier_declarations→ type declaration_list
type → boolean | static | int
optional_statements→ statement_list| e
statement_list→ statement| statemet_list; statement
statement→ variable assignop expression| compound_statement
            | if expression then statementelse statement| while expression do statement
variable → id
expression → simple_expression| simple_expression relop simple_expression
simple_expression → term | sign term | simple_expression addop term
term → factor| term mulop factor
factor→ id  | num | ( expression) | not factor
sign → + | -
```

Figure 1 HDL syntax

This formalism was used to produce a sequential form for the dataflow graph of the algorithm described by an input file. The process was similar to the intermediate code generation phase of traditional compilers. The sequential form consisted of a quadruple for each graph node, containing the corresponding operation, references to its inputs and outputs and scheduling information. The input and output references were used to designate the edges of the graph. Scheduling was supported by an attribute instance in each nonterminal that corresponded to a node. The evaluation of this instance followed a widely used scheduling heuristic algorithm with attribute dependencies used to convey information and define the order in which each node would be scheduled. This effort has been presented in detail in (Economakos, 1995). However, for the development of a hardware compiler, this sequential form is not adequate.

One disadvantage of the sequential dataflow representation is that the edges are not explicitly defined, but are implied by the input and output references. However, explicit definition of edges is preferred for a hardware compiler tool, since it will need to access them many times for optimization and final mapping into actual hardware. Also, an efficient hardware compiler implementation requires control flow information, to construct a finite state machine that generates the control signals to drive the datapath. For this reason, a modified internal representation of the one presented in (Thomas, 1990) has been adopted. All operator nodes are described by the set $X=\{x_a\}$ where each operator is indexed by a subscript a. When describing operator inputs and outputs, it is necessary to distinguish them from each other. This is accomplished using the sets $I=\{i_{a,b}\}$ and $O=\{o_{a,c}\}$ for all operator inputs and outputs, respectively. Each operator input is indexed by the operator index a and a second index b that numerically identifies the inputs to the operator. Similarly, each operator output is specified by the operator index a and a second index c that numerically identifies the operator's output. All three sets are implemented as single linked lists with additional links from each operator in X to its corresponding inputs in I and outputs in O.

Control flow is described by grouping operators into basic blocks. Each basic block represents a group of operators translated from a sequence of statements (a block of statements containing no branches) and has Boolean conditions that guard the entry and exit from the basic block. Operators within a basic block may execute in any order that does not violate data dependencies. Control flow between basic blocks is determined by the guard conditions. The basic blocks are implemented by attaching special fields to the elements of X defining the conditions that must hold for the corresponding operation to be executed.

As an example, consider the following code fragment:

```
IF A>0 THEN I:=I+1;   ELSE I:=I-1;
```

The corresponding CDFG and the basic block structure are given in figure 2, where solid lines are used to denote data flow and dashed to denote control flow.

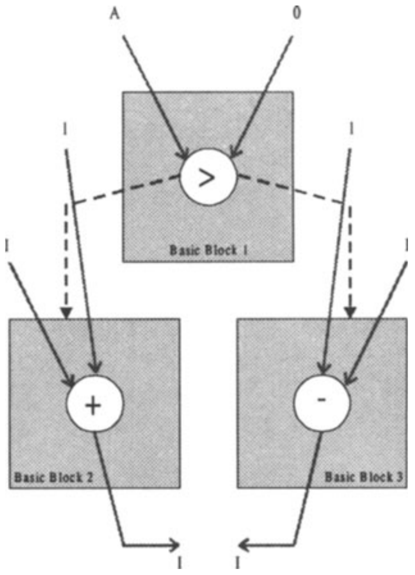

Figure 2 An example CDFG and basic block structure

3.3 AG-driven scheduling

A crucial task in high-level synthesis is scheduling. A scheduled CDFG is a complete implementation of the specified behavior, if enough resources are provided. Generally, four scheduling problems exist in high-level synthesis. Given a set of operations X, a set of functional unit types K, a type function $\tau : X \rightarrow K$, a time constraint (deadline) D on the overall schedule length, and resource constraints m_k, $1 \leq k \leq K$ for each functional unit type, the four problems can be defined as:

- *Unconstrained scheduling* (*UCS*): Find a feasible (or optimal) schedule for X that obeys the precedence constraints.
- *Time-constrained scheduling* (*TCS*): Find a feasible (or optimal) schedule for X that obeys the precedence constraints and meets the deadline D.
- *Resource-constrained scheduling* (*RTS*): Find a feasible (or optimal) schedule for X that obeys the precedence constraints and meets the resource constraints for each functional unit type.
- *Time and resource-constrained scheduling* (*TRCS*): Find a feasible (or optimal) schedule for X that obeys the precedence constraints, meets the deadline D and meets the resource constraints for each functional unit type.

For each of the four problems, different algorithms have been proposed over the past years. Each hardware compiler system must implement one or more of these. In (Economakos, 1995), two algorithms for UCS were expressed in an AG formalism, *As Early As Possible* (ASAP) scheduling and *As Late As Possible* (ALAP) scheduling. These two algorithms are also involved in upper and lower bound calculations for other scheduling algorithms. Their description can be found in figure 3.

ASAP scheduling

```
for each node ui∈V do
  if Predui=∅ then
    Ei=1; V=V-{ui};
  else
    Ei=0;
  endif
endfor
while V≠∅ do
  for each node ui∈V do
    if ALL_NDS_SCHD(Predui,E) then
      Ei=MAX(Predui,E)+1; V=V-{ui};
    endif
  endfor
endwhile
```

ALAP scheduling

```
for each node ui∈V do
  if Succui=∅ then
    Li=T; V=V-{ui};
  else
    Li=0;
  endif
endfor
while V≠∅ do
  for each node ui∈V do
    if ALL_NDS_SCHD(Succui,L) then
      Li=MIN(Succui,L)-1; V=V-{ui};
    endif
  endfor
endwhile
```

Figure 3 ASAP and ALAP scheduling algorithms

In the above, Ei (Li) is the ASAP (ALAP) control-step index calculated for every node in a CDFG V and E (L) the set of all indexes. Predui (Succui) denotes all the nodes in the CDFG that are immediate predecessors (successors) of node ui. The function ALL_NDS_SCHD returns true if all the nodes in the set passed as its first parameter are scheduled, i.e., have a non-zero label. Finally the function MAX (MIN) return the control-step index with the maximum (minimum) value from the set passed as its first parameter.

The main difference between the two algorithms is their evaluation order. ASAP moves from the root of the CDFG to the leaves while ALAP moves the other way. This was reflected in the AGs presented in (Economakos, 1995). Since all attributes were attached to the parse tree of the given behavior, by examining the HDL syntax given in figure 1, one can easily see that ASAP required attribute instance dependencies with direction from the leaves of each subtree to its root and ALAP the opposite. In AG terminology, ASAP required synthesized attributes while ALAP required inherited ones.

Since the CDFG nodes are operators, the attributes used for scheduling are attached to the syntactic rules that deal with operators. The general case can be written as:

$$\text{operation} \rightarrow \text{operand}_1 \text{ operator operand}_2. \qquad (1)$$

By examining the HDL syntax of figure 1, one can easily see that many rules like (1) are used.

For AG driven ASAP scheduling, a synthesized attribute called control_step is used and the semantic rule corresponding to (1) is:

$$\text{operation.control\_step}= \\ =\text{MAX}(\text{operand}_1\text{.control\_step},\text{operand}_2\text{.control\_step})+1. \quad (2)$$

The initial condition needed to calculate all control step assignments is that every constant can be considered as scheduled in control step 0. Also, the control step when each variable is last generated must be kept in a symbol table so that every operator that uses it will be scheduled after that. An AG implementation of a symbol table can be found in (Economakos, 1995).

For AG driven ALAP scheduling, an inherited attribute called again control_step is used, evaluated by the following semantic rule attached to (1):

$$\text{operand}_i\text{.control\_step}=\text{operation.control\_step}-1, i=1,2. \qquad (3)$$

The initial condition needed in (3) is that the final operator in each subgraph of the CDFG must be scheduled in the last control step, except when there is a data dependence for the variable that is generated by that operator. In this case, it must be scheduled a number of control steps earlier, equal to the length of the dependence. So, before scheduling any final operator node, all data dependencies must be found. To perform this in an AG driven environment, the input behavioral description must be passed two times: first to extract dependencies and next to perform scheduling. A detailed description of this technique can be found in (Economakos, 1997). Full AGs for both ASAP and ALAP scheduling can be found in (Economakos, 1995) and (Economakos, 1997).

3.4 VHDL preprocessor

VHDL has been proposed and adopted as a standard language to describe digital designs in various levels of abstraction. Currently, it is widely and uniformly used for simulation purposes. For synthesis, vendor specific subsets of the language are supported.

VHDL can describe designs following three basic styles:

- *Behavioral style*: All common procedural programming language constructs and abstract data types are supported; the design is expressed as a set of concurrent processes. This style has very poor performance for synthesis.
- *Dataflow style*: The design is expressed as concurrent signal assignments and guard conditions that perform a partition of all assignments into control states. The underlying architecture is that of a finite state machine driven datapath and can be automatically synthesized with quite satisfactory results.
- *Structural style*: The design is expressed as a netlist of basic blocks and can be generally synthesized with no problems. The quality of the produced results depend on the quality of the input description.

For the hardware compiler project, a tool has been developed that translates the internal representation into dataflow style VHDL that can be used as a preprocessor in modern CAD simulation or synthesis environments. Dataflow style has been chosen because it is a straightforward translation of the scheduled CDFG.

The translation produces one concurrent signal assignment for each node of the CDFG. For example, for operator OP with inputs IN1 and IN2 and output OUT, the following line of VHDL code will be generated:

```
OUT <= IN1 OP IN2;
```

However, only with this code, all operations will be performed simultaneously. No scheduling information is used. To support scheduling, the assignments of each control step are grouped together in a block statement. This block has a guard condition that permits the execution of all enclosed assignments only at the correct control step, by the use of a signal that holds the current control step. Also, the guard condition includes the well known (CLK'EVENT and CLK='1') condition, that is used to denote an edge triggered register. The block also includes an assignment for updating the state signal. For example, if the above operation is scheduled at control step CS1, with a clock cycle of N ns and CS1 is followed by CS2, the following code will be generated:

```
CS1: BLOCK ((CLK'EVENT AND CLK='1') AND (STATE=CS1))
      BEGIN
            OUT <= GUARDED IN1  OP  IN2;
            STATE <= GUARDED CS2 AFTER N NS;
      END BLOCK;
```

This basic construct is used for all CDFG operations. In the case of operations that will be performed only if some condition holds (that is, inside conditional basic blocks), these are translated into conditional assignments.

A representative example of the whole compilation process and the VHDL output is presented in the following section.

4 EXPERIMENTAL RESULTS

The ideas presented so far have been implemented in a hardware compiler tool based on AGs. The basic difference of the two presented algorithms (ASAP requires synthesized attributes while ALAP requires inherited) played an important role in the development of the compiler. The current version is based on the YACC (Aho, 1986) compiler construction tool and implements only ASAP, since YACC can evaluate only synthesized attributes.

To evaluate the AG driven hardware compiler, a representative example of an automatically synthesized design is presented. In figure 4, the behavioral description of the differential equation solver presented in (Dutt, 1992) is given. The description passes through YACC for scheduling and then through the VHDL preprocessor. The output is given in figure 5.

```
block diffeq(inout port x[16],u[16],y[16];
             in port dx[16],a[16])
begin
  boolean x1[16],u1[16],y1[16];
  while x<a do
    begin
      x1:=x+dx;
      u1:=u-(3*x*u*dx)-(3*y*dx);
      y1:=y+(u*dx);
      x:=x1;
      u:=u1;
      y:=y1
    end
end.
```

Figure 4 Differential equation solver

```
entity vhdl is
   port (a:in integer;
         dx:in integer;
         y:inout integer;
         u:inout integer;
         x:inout integer);
end vhdl;

architecture data_flow of vhdl is
   signal yl:integer;
   signal T10:integer;
   signal ul:integer;
   signal T8:integer;
   signal T7:integer;
   signal T6:integer;
   signal T5:integer;
   signal T4:integer;
   signal T3:integer;
   signal xl:integer;
   signal Tl:boolean;
   signal state:integer:=1;
   signal CLK:bit:='0';
begin
    CLK<=not CLK after 10 ns;
S1: block ((CLK'EVENT and CLK='1') and (state=1))
      begin
         Tl<=guarded x<a;
         xl<=x+dx when (Tl and Tl'ACTIVE) else
xl;
         T3<=3*x when (Tl and Tl'ACTIVE) else T3;
         T7<=3*y when (Tl and Tl'ACTIVE) else T7;
         T10<=u*dx when (Tl and Tl'ACTIVE) else
T10;
         process
           begin
             wait on xl;
             x<=xl;
           end process;
         state<=guarded 2 after 2 ns;
      end block;
```

```
S2: block ((CLK'EVENT and CLK='1') and (state=2))
      begin
         T4<=guarded T3*u when Tl else T4;
         T8<=guarded T7*dx when Tl else T8;
         yl<=guarded y+T10 when Tl else yl;
         process
           begin
             wait on yl;
             y<=yl;
           end process;
         state<=guarded 3 after 2 ns;
      end block;
S3: block ((CLK'EVENT and CLK='1') and (state=3))
      begin
         T5<=guarded T4*dx when Tl else T5;
         state<=guarded 4 after 2 ns;
      end block;
S4: block ((CLK'EVENT and CLK='1') and (state=4))
      begin
         T6<=guarded u-T5 when Tl else T6;
         state<=guarded 5 after 2 ns;
      end block;
S5: block ((CLK'EVENT and CLK='1') and (state=5))
      begin
         ul<=guarded T6-T8 when Tl else ul;
         process
           begin
             wait on ul;
             u<=ul;
           end process;
         state<=guarded 1 after 2 ns;
      end block;
end data_flow;
```

Figure 5 Dataflow style VHDL description of differential equation solver

The code fragment of figure 5 was used as input to the Accolade PeakVHDL simulator. Figure 6 presents the resulting waveforms for three different test cases. It must be stated that all calculated values for x and y are significant parts of the result and not only the final values (just before the circuit reaches steady state).

For the ALAP case, in order to evaluate the two pass AG with inherited attributes, a new implementation is under development using an AG evaluation tool that handles any case of non-circular AGs (Sideri, 1989).

5 CONCLUSIONS

A novel AG-driven approach to the implementation of a flexible hardware compiler has been presented in this paper.

The results obtained and presented show that this combination is promising. Its main advantages include the extensive use of existing tools and techniques (for attribute evaluation) and the incorporation of the AG formalism as a very high level meta-language describing high-level synthesis algorithms. The proposed implementation is a flexible hardware compiler, which can be seen as a testbench or as a fast prototyping platform.

Currently we are working on the expansion of the proposed formalism, mainly to include other scheduling and allocation algorithms, in order to tackle the whole problem of hardware compilation under an AG formalism.

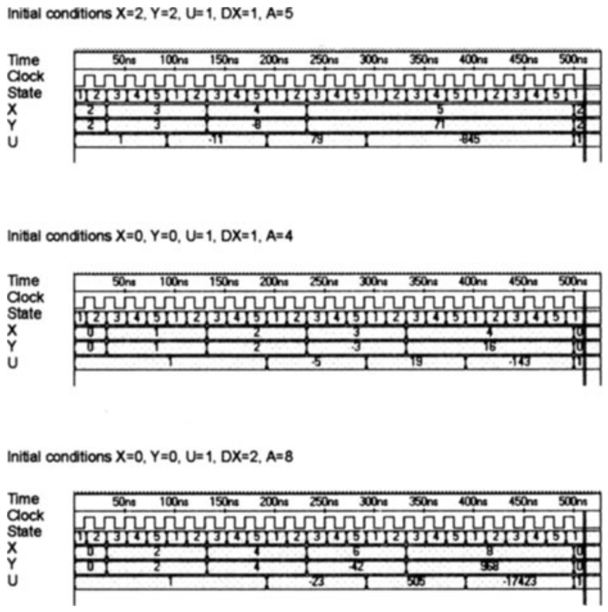

Figure 6 Simulated waveforms for 3 test cases of the differential equation solver

6 ACKNOWLEDGMENT

We wish to thank Petros Economakos for his valuable help in the design of the VHDL preprocessor and the presentation of the differential equation solver circuit.

7 REFERENCES

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) Compilers: Principles, Techniques and Tools. Addison-Wesley.

Arbab, B. (1986) Compiling Circular Attribute Grammars into Prolog. *IBM J. Res. & Development*, Vol 30, No 3.

Bhasker, J. (1992) A VHDL Primer. Prentice Hall.

Biesenack, J., Koster, M., Langmaier, A., Ledeux, S., Marz, S., Payer, M., Pilsl, M., Rumler, S., Soukup, H., When, N. and Duzy, P. (1993) The Siemens High-Level Synthesis System CALLAS. *IEEE Transactions on Very Large Scale Integration Systems* , Vol 1, No 3, pp 244-253.

Camposano, R. (1991) Path-Based Scheduling for Synthesis. *IEEE Transactions on Computer-Aided Design* , Vol 10, No 1, pp 85-93.

Deransart, P. and Aluszynski, J. (1985) Relating Logic Programs and Attribute Grammars*J. Logic Programming* , Vol 2, pp 119-155.

Deransart, P. et al. (1988) Attribute Grammars, in *Lecture Notes in Computer Science*, Spinger-Verlag.

Dutt, N. and Ramachandran, C. (1992) Benchmarks for the 1992 High-Level Synthesis Workshop. *UCI Technical Report #92-108*

Economakos, G., Papakonstantinou, G. and Tsanakas, P. (1995) An Attribute Grammar Approach to High-Level Automated Hardware Synthesis. *Information and Software Technology* , Vol 37, No 9, pp 493-502.

Economakos, G., Papakonstantinou, G. and Tsanakas, P. (1997) Attribute Grammar Driven Scheduling for the High-Level Synthesis of ASICs. *submitted to 1997 IEEE/ACM International Conference on Computer Aided Design.*

Farrow, R. (1983) Attribute Grammars and Dataflow Languages. *ACM SIGPLAN Symposium on Programming Language Issues in Software Systems* , pp 28-40.

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

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

Hafer, L. J. and Parker, A. C. (1983) A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic. *IEEE Transacttions on Computer-Aided Design*, Vol 2, No 1, pp 4-18.

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

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

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

Ku, D. and De Micheli, G. (1990) HardwareC: A Language for Hardware Design. *Stanford University Technical Report CSL-TR-90-419*, Version 2.0.

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

Lipsett, R., Schaefer, C. F. and Ussery, C. (1993) VHDL: Hardware Description and Design. Kluwer Academic Publishers.

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

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

Paaki, J. (1995) Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys* , Vol 27, No 2.

Papakonstantinou, G. and Kontos J. (1986a) Knowledge Representation with Attribute Grammars. *The Computer Journal*, Vol 29, No 3, pp 241-246.

Papakonstantinou, G. *et al.* (1986b) An Attribute Grammar Interpreter as a Knowledge Engineering Tool. *Angew. Inf.*, Vol 9, pp 382-388.

Papakonstantinou, G. and Tsanakas, P. (1988) Attribute Grammars and Dataflow Computing. *Information and Software Technology* ,Vol 30, No 5, pp 306-313.

Paulin, P. G. and Knight, J. P. (1989) Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Transactions on Computer-Aided Design*, Vol 8, No 6, pp 661-679.

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

Tanaka, T., Kobayashi, T. and Karatsu, O. (1989) HARP: Fortran to Silicon. *IEEE Transactions on Computer-Aided Design* , Vol 8, No 6, pp 649-660.

Thomas, D. E., Lagnese, E. D., Walker, R.A., Nestor, J. A., Rajan, J. V. and Blackburn, R. L. (1990) Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench. Kluwer Academic Publishers.

Trickey, H. (1987) Flamel: A High-Level Hardware Compiler. *IEEE Transactions on Computer-Aided Design*, Vol 6, No 2, pp 259-269.

Tsanakas, P., Alexandridis, N. and Papakonstantinou, G. (1989) An FP-based Design Methdology for Problem-oriented Architectures. *The Computer Journal*, Vol 32, No 5, pp 453-460.

Tsanakas, P., Papakonstantinou, G. and Kaxiras, S. (1991) A PROLOG-based Design Environment for the High-Level Synthesis of Application-Specific Architectures.*Microprocessing and Microprogramming* Vol 32, pp 307-313.

Tsanakas, P., Papakonstantinou, G. and Bilalis, N. (1992) Systematic Synthesis of Parallel VLSI Architectures from FP Specifications and its Application to Scene Matching. *Microprocessing and Microprogramming*, Vol 35, pp 579-586.

Waite, W. M. and Goos, G. (1984) Compiler Construction. Springer-Verlag.

Walker, R. A. and Camposano, R. (1991) A Survey of High-Level Synthesis Systems. Kluwer Academic Publishers.

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

8 BIOGRAPHY

G. Economakos received his Diploma in Electrical and Computer Engineering from the National Technical University of Athens (1992). Currently, he is a Ph.D. student in the National Technical University of Athens. His research interests include hardware design automation, combinational synthesis, sequential synthesis, high-level synthesis and language based design automation.

G. Papakonstantinou received his Diploma in Electrical Engineering from the National Technical University of Athens in 1964, the P.I.I. Diploma in Electronic Engineering from Philips Int. Inst. In 1966, and his M.Sc. in Electronic Engineering from the N.U.F.F.I.C. Netherlands in 1967. In 1971 he received his Ph.D. in Computer Engineering from the National Technical University of Athens. He has worked as a Research Scientist at the Greek Atomic Energy Commision / Computer Division (1969-1984), as Director of the Computer Division at the Greek Atomic Energy Commision (1981-1984). From 1984 he serves as a Professor of Computer Science at the National Technical University of Athens. His research interests include knowledge engineering, hardware design automation, as well as parallel architectures and languages.

K. Pekmestzi received his Diploma in Electrical Engineering from the National Technical University of Athens (1975). From 1975 to 1981 he was a research fellow in the Electronics Department of the Nuclear Research Centre "Demokritos". He received his Ph.D. in Electrical Engineering from the University of Patras (1981). From 1983 to 1985 he was a professor at Higher School of Electronics in Athens. Since 1985 he has been with the National Technical University of Athens, where he is currently an Associate Professor. His research interests include computer arithmetic, VLSI digital filters and VLSI design automation.

P. Tsanakas received his Diploma in Electrical Engineering from the University of Thessaloniki (1982), his M.Sc. in Computer Engineering from Ohio University (1985), and his Ph.D. in Computer Engineering from the National Technical University of Athens (1988). He is now serving as Associate Professor at the National Technical University of Athens. His research interests include parallel / distributed computing systems and applications, automated synthesis of VLSI architectures, image / signal processing, and intelligent learning systems.