Advances in Attribute Grammar Driven Hardware Compilation*

George Economakos, Ioannis Panagopoulos and George Papakonstantinou

National Technical University of Athens,
Department of Electrical and Computer Engineering
Zographou Campus, GR-15773 Athens, Greece
george@cslab.ece.ntua.gr

Abstract

High-level or behavioral synthesis of digital circuits 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. Since most of the times the algorithmic description is given in textual form, high-level synthesis transformations share common aspects with traditional compiler transformations (the process is also called hardware compilation). As a consequence, hardware compilation can be performed using attribute grammars. However, since modern high-level synthesis transformations are of high complexity, advanced attribute evaluation techniques are required. This paper presents recent advances in hardware compilation under the FNC-2 attribute grammar system. It follows two different approaches. One that decorates the parse tree of the behavioral specification with implementation details, taking advantage of the use of a definition table, and one that constructs and decorates a new tree, which follows the structure of the corresponding data flow graph. Both approaches have been found to be flexible and efficient in supporting future research efforts in the field.

1. Introduction

Over the last twenty years, advances in circuit fabrication technology have increased device densities and as a consequence, they have increased design complexity. To manage continuously emerging tasks, designers have moved towards higher levels of abstraction [31] and language based design descriptions [14, 16, 27, 28, 21], which are closer to the way they conceive their work. However, each design must be described, eventually, at the lowest level (e.g. layout masks), in order to be fabricated. The transformation from one level of abstraction to the next is performed by various synthesis processes.

High-level or behavioral synthesis [11, 18, 19, 30, 31], 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. Today, with the adoption of standard *Hardware Description Languages* (HDLs) like VHDL [2, 3] or Verilog [25], such RTL descriptions can be automatically transformed to final circuit implementations, through an iterative refining application of ECAD tools. For behavioral descriptions, given in procedural, high-level language, high-level synthesis is also called hardware compilation and presents many similarities with traditional compiler

*This work was partially funded by the Greek Ministry of Development, General Secretariat for Research and Technology, project PENED 99$E\Delta$521

construction [1]. Therefore, tools and techniques applied to the latter, may also be applied to the former, if advantageous.

The reason that such application may be favorable is that, even though high-level synthesis has been introduced over twenty years ago, a lot of problems has to be solved before it is widely accepted. Among them, high-level synthesis lacks a theoretical framework (like Boolean algebra for logic design) that would further accelerate research. Attempting to overcome this inefficiency and propose a unifying formal framework for language based design, an *attribute grammar* (AG) [17] formalism, describing scheduling heuristics was proposed in [8]. It was based on the fact that the parse tree of a behavioral description is a superset of its data flow graph and so, behavioral transformations can be implemented by decorating the parse tree with appropriate attributes, which apply their functionality. Overall, the main advantages of AGs as a formal specification and implementation formalism (through automated compiler construction systems) are modularity and the declarative notations used. Recently [10, 9, 6], the methodology of [8] was realized into the AGENDA integrated design environment that supports top-down implementation of behavioral descriptions using VHDL [5].

This paper further exploits the advantages of AGENDA under a robust and flexible compiler construction system, FNC-2 [15] (AGENDA was based on the rich in theory but poorly implemented SDP [29] environment. The objective is again the same, that is, to give a formal executable specification for high-level hardware transformations using AGs and support the whole design process through HDL RTL output. Specifically, two different approaches are followed, both fully supported by FNC-2. The first decorates the parse tree of a behavioral description with implementation detail holding attributes, while the second constructs a new tree, which follows the structure of the corresponding data flow graph, and decorates this new tree with the same attributes. Downwards compatibility with AGENDA, for RTL output code generation, is supported through similar data structures. The underlying methodology automatically produces high-quality hardware compilers, from declarative notations of the transformations that they support.

The content of the paper is organized as follows. After this introduction, section 2 briefly presents theoretical material while section 3 summarizes previous related research efforts. Next, section 4 gives a brief discussion of the overall methodology. Section 5 gives parsing details under FNC-2. After parsing is presented, the two alternative approaches followed, for the design of a hardware compiler, are given in sections 6 and 7. Finally, section 8 gives the conclusions of applying the proposed methodology.

2. Theoretical background

2.1. Hardware compilation

High-level synthesis (HLS) [11] takes a specification of the behavior of a digital system along with a set of constraints and goals on the resulting hardware, and finds a structure that realizes the given behavior while satisfying the given goals and constraints. The behavior is usually described as an *algorithm*, similar to programming language descriptions. The structure is a register-transfer level implementation that includes a *data-path* portion and a *control* portion. The data-path contains a network of functional units, registers, and their interconnection. The control activates components of the data-path to realize the required behavior. One of the objectives of synthesis is to find a structure that satisfies the constraints, such as requirements on area, latency or cycle-time, while minimizing other costs. For example, the goal might be to minimize the area while meeting timing requirements.

Due to its complexity, high-level synthesis is divided into a number of distinct yet inter-dependent tasks [19]. First, a hardware description language specification is parsed into an internal representation that models both the control-flow and data-flow of the input behavior. The internal representation is optimized by compiler-like transformations such as dead-code elimination, common subexpression elimination and constant propagation. *Scheduling* and *binding* are then performed to map the behavior into structure. Scheduling assigns operations to control steps, where a control step is the fundamental unit of sequencing in synchronous systems and corresponds to a clock cycle. Binding assigns operations to specific allocated hardware resources. They are closely related and inter-dependent. For example, scheduling attempts to minimize the number of required control steps subject to the amount of available hardware, which depends on the results of the binding. Likewise, binding exploit concurrency among operations to allow sharing of hardware resources, where the degree of concurrency is determined by scheduling.

For a given schedule and data-path, control logic must be synthesized to activate components in the data-path according to this schedule. There are many different ways of synthesizing the control, ranging from hardwired control, such as *finite-state machines* (FSMs), to microcoded control, where each control step in the schedule corresponds to a microprogram instruction.

The major difficulty in high-level synthesis is the large number of design alternatives that must be examined in order to select the design that best meets the design goals and still satisfies the constraints. The computational complexity of even one of the above subtasks can be intractable in the presence of constraints. For example, the problem of scheduling under resource constraints is provably NP-complete [12]. Compounding the problem is the tight inter-dependencies that exist between scheduling and binding that make the *design space* of possibilities multi-dimensional and irregular. For high-level synthesis to be practical, it is necessary to find an acceptable compromise between the degrees of freedom in exercising design choices and the complexity of the synthesis computation.

Also, high-level synthesis still lacks a formalism of the type already developed for synthesis at lower abstraction levels, that is layout and logic synthesis. Layout models have a good formal foundation in set and graph theory since the algorithms deal with placement and connectivity of two-dimensional objects. Logic synthesis is based on well-known formalisms of Boolean algebra because the algorithms deal predominantly with minimization and transformation of Boolean expressions. High-level synthesis uses formalisms based on several different areas, varying from the programming-language paradigm for the behavioral description to layout models for cost and performance estimation. However, it does not have an identifiable formal theory of its own. Most frequently, a mixture of finite-state machine and register-transfer notations serves as a high-level synthesis formalism.

2.2. Attribute grammars

An *attribute grammar* (AG) [4, 17, 24] 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 $p \in P$ ($p = X_0 \rightarrow X_1 \dots X_n$) of the CFG is augmented by a set of *semantic rules* $SR(p)$ that define attributes in terms of other attributes of terminals and nonterminals appearing in the same production.

Since their introduction, AGs have been a subject of intensive research, both from a conceptual and from a practical point of view. This has lead to the creation of 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 programs to implement the specified mapping.

2.3. The FNC-2 attribute grammar system

One of the main characteristics of the FNC-2 attribute grammar environment is its modular nature. Every phase of the compilation process, from parsing the input text to producing output textual results, can be implemented within a module. This feature enhances reusability of pre-created modules and simplifies compiler development, by providing a structured way of implementing each distinct phase.

Syntactic and lexical analysis of the input text are performed by the SYNTAX analyzer, which is fully supported by the FNC-2 attribute grammar environment. Along with the token and grammar descriptions of the language, the user provides tree construction mechanisms, which are attached to each production of the concrete syntax description. These tree construction functions, referred as "operators" in FNC-2, create the abstract syntax of the compiler, which is a tree grammar expressing the structure of notions of the language being described rather than the textual appearance of those notions. Attribute decoration and evaluation is later performed on this abstract syntax tree. This frees the grammar developer from syntactic constraints which arise due to the chosen parsing method and enables to modify the concrete syntax description without affecting the structure of the language itself.

Attribute grammar computations are then performed on modules implementing attribute grammars. These modules have the form of functions, taking an abstract syntax tree as their input and producing an identical or modified decorated tree, with all desired attributes evaluated. With these functions, the compiler developer implements the rules for attribute computations of the input tree, which are executed the one after the other on the evaluation phase of the compiler.

The first abstract syntax tree is constructed during the parsing phase of the compiler and is fed to the first attribute evaluating function. The output tree of this function (new or the same decorated) is fed to another function for further evaluation. After the final function, the output tree is destructured by a special module implemented with the PPAT sub-system, which provides the output text.

The whole implementation of the grammar described in this paper is divided into as many modules as possible. Each function performs a single action, corresponding to a specific attribute evaluation, and outputs the resulting decorated tree. This way the function can be used in different computations on the same compilation process, whenever the corresponding action is needed.

3. Related research

The design complexity of modern digital circuits has motivated researchers to move towards higher abstraction levels and to utilize language based processors. As far as synthesis is concerned, language based processors followed the idea that by expressing the design in a higher level formalism which is bound to a specific computational model, one can use an executable version of the model to perform synthesis. As the design specification passes through the model, constructs that are crucial for synthesis are recognized and special procedures are conducted.

This idea was adopted by Seawright *et al* [26, 27] to develop Clairvoyant, a system to perform hardware compilation using production-based specifications. However, only control path synthesis is performed (datapath synthesis is not considered at all). The presented examples are very limited, the largest being a part of the 8251 controller benchmark.

As an extension to Clairvoyant, Oberg *et al* [21, 22] have presented PRO-GRAM, a YACC-like grammar-based synthesis environment for data communication protocols. In this work, a higher level of abstraction (to define the circuit behavior) is used and the architecture of the final design is fixed. The whole process is bound to a specific application domain (i.e., data communication protocol implementations) and although larger examples than in Clairvoyant are presented [13], the tool is not adequate for other application domains. With a recent extension [20], PRO-GRAM has been used for the automatic generation of VHDL test benches for communication interfaces, used for design validation through simulation.

Similar to PRO-GRAM was the extension of Clairvoyant presented by Seawright *et al* [28], the system called Dali. Dali uses a graphical hierarchical representation to describe systems that process structured data streams or handle structured control protocols, i.e. telecommunications protocols or compression algorithms. Compilation techniques are used to transform this specification into an FSM circuit (control path synthesis). The strong point of Dali is that it can reflect errors back onto the source specification and thus ease debugging.

AGs were introduced to the field of design automation by Jones *et al* [14], 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. The authors used an internal representation of the VLSI circuit (generally composed of circles) as the design tree, to which attributes were attached.

A tool for hardware compilation, was the syntax-directed system developed by Keutzer *et al* [16]. It faced the problem of register-transfer level realization, that is, the optimal transformation of a *finite-state machine with data-path* (FSMD) architecture into netlists of digital gates, and used more than one language processors.

In [8], a first attempt to fully describe the whole process of high-level hardware synthesis using AGs was given. The approach presented was much simpler than [14]. Datapath synthesis was performed using well known heuristics, during the semantic analysis of a behavioral specification. Control path synthesis was driven by the parsing process of control constructs. Implementations of the formalism into a full synthesis system were presented in [7, 10, 9].

This paper further exploits the advantages of [8] under a robust and flexible compiler construction system, FNC-2 [15]. Two different approaches are followed, both fully supported by FNC-2. The first decorates the parse tree of a behavioral description with implementation detail holding attributes, while the second constructs a new tree, which follows the structure of the corresponding data flow graph, and decorates this new tree with the same attributes. The underlying methodology automatically produces high-quality hardware compilers, from declarative notations of the transformations that they support.

4. Problem specifications

The AG computational model is based on the decoration of parse trees with domain specific values, called attributes. For traditional compiler construction, attributes have been involved in every phase. They are used to express modular and declarative local transformations, attached to syntactic rules of the input grammar. Moreover, modern compiler generators allow attributes to express more global relations and computations, with respect to the final compiler output, under the same declarative environment. Following these ideas, the FNC-2 attribute grammar environment [15], is based on functional modules, which take as input a parse tree, generate and evaluate attributes, and produce a decorated tree as output. Each decorated tree can be input to another module for further attribute decoration or output generation (using the PPAT subsystem of the FNC-2 environment). Moreover, each FNC-2 attribute can hold, apart from the typical numeric or alphanumeric values, structured data types such as lists or records or even new constructed trees. In the latter case, these new trees are constructed by invoking tree construction functions on the nodes of the original abstract syntax

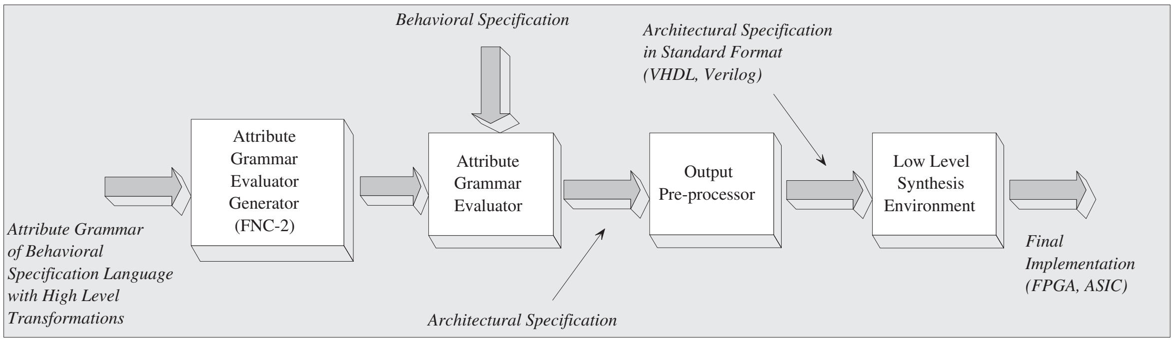

Figure 1: Proposed methodology overview

tree.

For hardware compilation, attributes can be used to hold implementation details of the generated circuit implementation (like the control step when an operation is scheduled or the register that will hold the output of an operation). They may also be used for other "house-keeping" chores, like the implementation of symbol tables for different entities in the behavioral description (operators, operations, variables). The first solution to the hardware compilation process given in this paper is based on these aspects and operates upon the parse tree of a behavioral description.

However, synthesis transformations originally operate upon a data flow graph. The parse tree produced during a compilation process is a superset of the data flow graph of the corresponding computation (contains all data flow graph nodes plus others, language specific). The second solution to the hardware compilation process given in this paper, uses attributes to construct a new tree, which consists of entities appended with the correct order, to a tree-type attribute of the root of the original parse tree. This tree is the data flow graph, which is decorated with implementation attributes during subsequent attribute evaluation phases.

In both solutions, results are presented in both textual (RTL code) and graphical form (after executing a Tcl/Tk [23] script, which is generated during the compilation process).

The overall methodology proposed in this paper is given in figure 1. Attribute grammars are used as a meta-level, supporting executable high-level synthesis transformation specifications. These specifications are implemented in a synthesis environment, taking behavioral specifications and producing architectural descriptions, after passing through an attribute grammar evaluator generator, like FNC-2. The architectural descriptions can be coded in standard formats (synthesizable VHDL or Verilog code) by a pre-processor tool and then, become real hardware, under different technologies (FPGA or ASIC), using efficient, low level synthesis tools.

This new methodology has been found to provide notable advantages, especially for researchers. This is due to the fact that the high-level transformation specifications needed are declarative and thus, very close to the actual description of the heuristic they implement. This makes them flexible and easy to manipulate and cause minor modifications, which is crucial for new research ideas. Moreover, all specifications are modular, so a problem can easily be partitioned into subproblems with separate specifications. When common subproblems are found, reusable specifications may be written.

5. The input tree constructed by the parser

The FNC-2 attribute grammar system uses an abstract form of the parse tree as its input. This means that besides the description of the concrete syntax of the input language, we have to provide

```
grammar scheduler is

root is PROGRAM ;
PROGRAM = program ;
program -> DEFS ;
DEFS = defs ;
defs -> DEF + ;
DEF     = def ;
def -> VAR_DEF
       EXP ;
EXP     = expression ;
expression -> OPS ;
OPS     = op ;
op -> TOKEN
      TOKEN ;
TOKEN   = num var_use newop ;
newop -> OPS ;
VAR_DEF = var_def ;
var_def ->  ;
var_use -> VAR_USE ;
VAR_USE = var_use_term ;
var_use_term ->  ;
num -> NUM ;
NUM     = num_term ;
num_term ->  ;

$var_id_def (VAR_DEF)   : token ;
$var_id_use (VAR_USE)   : token ;
$int_value (NUM)        : token ;
$op_id (OPS)    : token ;

end grammar ;
```

Figure 2: Abstract syntax of the input behavioral specification language

an abstract syntax tree that will be constructed during the parsing phase of the compilation process. This feature is very useful since we can actually change the appearance of the input grammar without affecting the actual tree that will be used later for attribute evaluation. For simplicity, consider a language which consists only of expressions and variable definitions. The abstract syntax for this language is given in figure 2 and part of the graphical representation of the abstract syntax tree for a sequence of three variable defining expressions in figure 3. The symbol "|" is used to denote alternative configurations. Words in lowercase express productions from a node to its children. These productions are called operators in the FNC-2 system and are used for tree construction mechanisms. Words in uppercase are the names of the nodes. The (T) symbol defines a terminal leaf. A leaf with no children and without the symbol (T) is assumed to have the same children as its immediately left sibling node.

The parse tree defines that there is a set of expressions in the input text, in the form of ex. x=4+5+5. Each child of a node named DEFS, holds one such variable defining expression. In other words, each subtree with a root named DEF, is created during parsing one expression of the form described above. The leaf named VARDEF holds the name of the variable to be defined. The node named EXP holds the expression that defines the variable. Each operator of the expression corresponds to a new node, named OPS and its operands are held in the two children of OPS, named

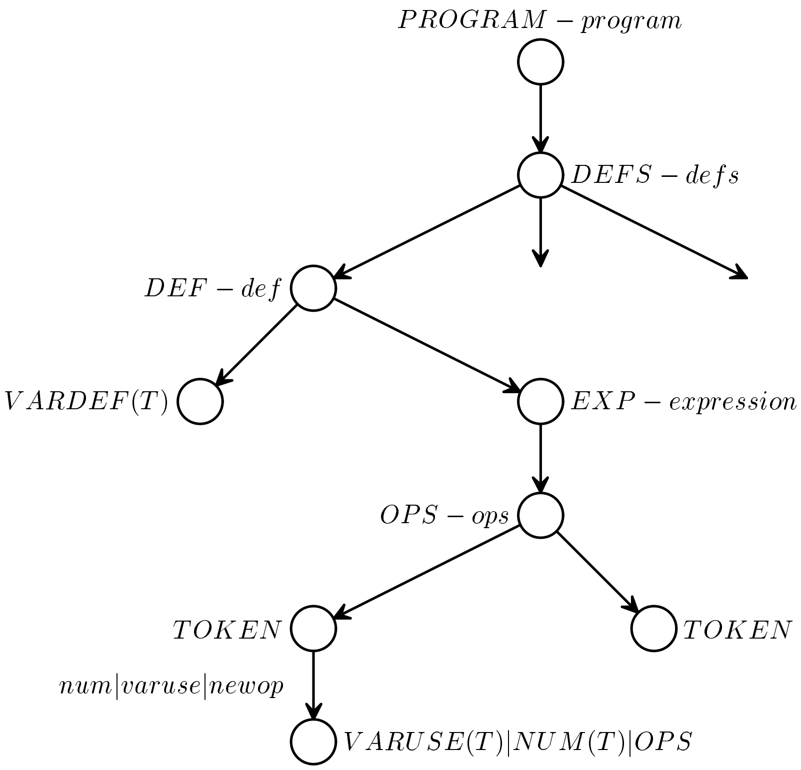

Figure 3: Representation of the input abstract syntax tree

TOKEN. According to the type of the TOKEN (constant number, variable , new operator), the children of TOKEN may be a NUM leaf, a VARUSE leaf (which are terminal leaves) or a new OPS, leaf constructing the rest of the expression's tree.

Applying FNC-2 to perform hardware compilation, the first step in both approaches taken, is to generate the abstract syntax tree. This tree is decorated during a first common phase with attributes that hold the values of the numbers and the names of the variables used as operands. The constructed tree is hereafter referred as the *Attributed Abstract Syntax* (AAS) tree.

6. First approach. Decorating the parse tree

6.1. Dependency related attributes

A crucial problem in parallelizing and hardware compilers is dependency analysis. That is, in order to move operations in time, as required for scheduling operations in high-level synthesis, each variable may only be used as input after it has been defined. If a defining expression for a variable is present, there exist a dependency between it and any expression using the variable. Our approach of respecting dependencies between variables is the construction and use of a definition table. This table needs information about the expression where every variable is defined. It is implemented in FNC-2 with a record type that holds the names of the variables, the expression where they are defined as well as implementation details (discussed further on). The definition table is a a list of such records. This list is constructed with a first pass through the AAS tree and is attached as a root attribute in the output tree. When the output tree will be considered as input for implementation transformations, easy access to the definition table is provided (for all required dependency checks).

The first pass corresponds to the construction of the first grammar, which takes the AAS tree as input, and returns the same tree, decorated with the following attributes:

vardefs The list containing the table described above. This attribute will be declared global in all grammars, so that it is accessible from all nodes of the tree.

dependsfrom This attribute is used for those leaves in the tree that correspond to variables uses. Note that, when a variable has to be looked-up in the definition table for a certain dependency, its name may not be able to identify it uniquely. This happens when we have multiple variable definitions for the same variable (more details later).

pos An attribute that holds the position (line number) of the variable in the input text.

During evaluation of the first grammar, the tree is traversed in bottom-up, left to right order and the definition table list is created. The pos attribute gets the value of the automatically maintained pos attribute of the FNC-2 system. This attribute holds the number of the child of DEFS that holds each variable definition. In this way we guarantee that it is a unique number and reflects the line number of the definition in the input text.

The list is created by traversing only the DEF leaves of the AAS tree. The record's id field gets the name of the variable and the position field gets the value of the pos attribute described above. The list is created bottom-up meaning that the first element in the list is that corresponding to the last definition of the input text, the right-most sibling of DEFS.

In order to evaluate the dependsfrom attribute, a specific list search function is implemented. This recursive function has the following form (in pseudocode):

```
FUNCTION findvardef(id : VAR NAME ; l : List of variable Definitions)
 IF list is empty THEN
```

```
    RETURN error
  ELSE IF
    top_record_in_list.id=id AND position_used>top_record_in_list.definedin
    RETURN top_record_in_list
  ELSE
    CALL findvardef(id,rest_of_list)
  END IF
END FUNCTION
```

The FNC-2 system offers two functions for list manipulation, head(l) and tail(l), which return the head and the tail of list l correspondingly. These functions are involved for the implementation of the above recursive computation.

As it has been mentioned above, the definition table list is generated as we traverse bottom-up the input text. This ordering is required for the correct evaluation of the dependsfrom attribute, as highlighted with the following example. Consider a source code with these definitions:

```
1. x=4+5+6;
2. y=7+8+9;
3. z=5+x;
4. x=6+7;
5. w=4+x;
```

The definition table list will look like this:

Entry	id	definedin
1	w	5
2	x	4
3	z	3
4	y	2
5	x	1

Consider variable x. In line 3 of the code, x is used in an expression. Beginning the search in the list we look for the name of the variable, found in the id column. The first record found is record 2, which records the definition of x in line 4. If we examine the recursive function used to evaluate the dependsfrom attribute of the use of x in line 3, we see that the second condition (position_used>top_record_in_list.definedin) does not hold for record 2. Indeed, the use of x in line 3 does not depend on the definition of x in line 4. If we move on, dependsfrom will be correctly evaluated at record 5, since the use in line 3 depends on the definition in line 1. The fact that the list is generated with a bottom-up traversal of the input text guarantees that at each variable use, searching the list from top to bottom, the dependsfrom attribute will be correctly evaluated at the first record defining the variable, above the use.

Having evaluated these three attributes, the new decorated output tree is ready to be fed to another module, which will add more implementation details.

6.2. Scheduling attributes

As mentioned earlier, FNC-2 can perform calculations progressively, by passing one AAS tree through more than one grammar evaluators in a row (this in fact is achieved with a special C file, by writing the row with all grammars through which all attributes will be computed). This programming style greatly enhances modularity and thus reusability of the corresponding specifications.

Following this idea, after having evaluated dependency attributes in the AAS tree, more implementation details may be calculated. Consider scheduling, which is the first transformation that can produce architectural descriptions, if no resource sharing is performed. Consider a simple scheduling heuristic, which generates maximally parallel implementations, the *As Soon As Possible* (ASAP) scheduling heuristic. ASAP requires that each operation is scheduled one control step after its latest input becomes available. The AAS tree with dependency attributes is ready for ASAP calculation. The new output tree will be decorated with all previously constructed and computed attributes plus an attribute with the name asapvalue, which will hold the controls step into which each operator will be scheduled, according to ASAP scheduling.

The default traversal of the tree is bottom-up and left to right for a synthesized attribute. This traversal reflects a route in the input text from top to bottom, which is enough to compute ASAP scheduling values. We declare the asapvalue attribute as synthesized. Its value for the input constant numbers will be defined as 0 and for the input variable uses will be looked-up in the definition table. Starting from the first child of the DEFS node, it is sure that no variables will be used because there are no definitions yet. So the bottom leaves of the subtree are all constant numbers and get the asapvalue of 0 (in fact, they may also be externally initialized variables, which may also be regarded as constants). Whenever an operator is traversed, its asapvalue is evaluated with the following script:

```
asapvalue(OPS)=IF asapvalue(TOKEN1)>asapvalue(TOKEN2) THEN
                 asapvalue(TOKEN1)+1
               ELSE
                 asapvalue(TOKEN2)+1
               END
```

When the variable definition is reached (leaf DEF), a function that assigns the corresponding asapvalue to a new field (called asap) of the list of variable definitions is called. This function uses a trick with the let...in construct of the FNC-2 system, because this is the only way to define a field of a record type attribute twice.

When a leaf corresponding to a variable use is reached (VARUSE), an attribute evaluation rule collects the already defined asapvalue of the variable's definition and passes it to its father (for calculating its asapvalue too, according to the above script). Although this technique works in most cases, there is a certain setting that gives false results. That setting is described by the input text below. Assume that one writes the following:

```
x=7+8;
y=x+7+5;
x=7+8;
```

If the only issue was to store the asapvalue of each variable definition and use it later to its uses, we would get the following schedule (the numbers in parentheses are the ASAP values):

```
x=7(0) +(1) 8(0), so x is defined at control step 1
y=x(1) +(2) 7(0) +(3) 5(0), so y is defined at control step 2
x=7(0) +(1) 8(0), so x is re-defined at control step 1
```

This is an error. The x variable is used in line, 2 at control step 2, after its first definition in line 1 and control step 1. It is also re-defined in line 3 at time step 1. This final definition must be scheduled after the use, since it is found in a greater code line. Generally speaking, we need a way to postpone a new definition of an already defined variable, until all its uses between the two definitions are scheduled. Following ASAP scheduling, this means that we have to schedule each new definition one control step later than the last use of each already defined variable, even if it is possible to schedule it earlier. With respect to the above ASAP scheduling script, for every operator, we should not only check which of the input TOKENS has the greatest asapvalue but, we should also check whether the greater of these values is smaller than the asapvalue of the last use of the variable it is defining. In the last case, the operator must be scheduled one control step later than this last use.

To solve this problem we have to add a new field to the record type of the variable definition list. This field is named uses and holds the ASAP value of the last use of each variable. We also need to add an identification attribute to each TOKEN node in the AAS tree, in order to be able to track the existence and use of a variable in the local environment of an operation parsing rule (this happens because in the FNC-2 system an attribute evaluation rule in a production can not access attributes that belong to its children). So, in the OPS node, apart from the computation of the ASAP scheduling control step, we change the use field of any variable in the operation to computed ASAP value. Of course, ASAP computation itself should now be taking into consideration this modification. Therefore, the above mentioned scheduling script is modified as below:

```
asapvalue(OPS)=IF asapvalue(TOKEN1)>asapvalue(TOKEN2) THEN
                 IF thisvariabledefinition.use<asapvalue(TOKEN1)
                   asapvalue(TOKEN1)+1
                 ELSE
                   thisvariabledefinition.use+1
                 END IF
               ELSE
                 IF thisvariabledefinition.use<asapvalue(TOKEN2)
                   asapvalue(TOKEN2)+1
                 ELSE
                   thisvariabledefinition.use+1
                 END IF
               END IF

IF TOKEN1=variable THEN thisvariable.use=asapvalue(OPS)
IF TOKEN2=variable THEN thisvariable.use=asapvalue(OPS)
```

6.3. Experimentation

After passing the AAS tree through implementation grammars, like the ASAP scheduling grammar of the previous subsection, an architectural description can be written, trough an output preprocessor tool. FNC-2 provides an easy way to do that, with the PPAT module. By implementing a program with the PPAT syntax system, we destructure the tree and rewrite the whole algorithm taking into account all implementation details. For illustrative reasons, a simple descriptive preprocessor has been implemented, which presents the definition table list first and then all calculations of the algorithm with their corresponding control step values. Using the whole compilation environment, suppose that we submit the following input text:

```
x=6+7+4;
y=8+x+10+11+12+14;
z=y+x;
y=x+6;
x=5+7;
w=x+y+z;
```

The output we get will be like the following (considering ASAP scheduling and greedy allocation - no resource sharing):

```
-------------------------- List vardefs ----------------------
```

```
Name:  x Defined at : 1 ASAP cs : 2 Last Used at step: 9
Name:  y Defined at : 2 ASAP cs : 7 Last Used at step: 8
Name:  z Defined at : 3 ASAP cs : 8 Last Used at step: 12
Name:  y Defined at : 4 ASAP cs : 9 Last Used at step: 11
Name:  x Defined at : 5 ASAP cs : 10 Last Used at step: 11
Name:  w Defined at : 6 ASAP cs : 12 Last Used at step: 0

--------------------------- Calculations ----------------------


x ( 1 ) = 6 7 Operator: + Asap value: 1
4 Operator: + Asap value: 2

y ( 2 ) = 8 x ( 1 ) Operator: + Asap value: 3
10 Operator: + Asap value: 4
11 Operator: + Asap value: 5
12 Operator: + Asap value: 6
14 Operator: + Asap value: 7

z ( 3 ) = y ( 2 ) x ( 1 ) Operator: + Asap value: 8

y ( 4 ) = x ( 1 ) 6 Operator: + Asap value: 9

x ( 5 ) = 5 7 Operator: + Asap value: 10

w ( 6 ) = x ( 5 ) y ( 4 ) Operator: + Asap value: 11
z ( 3 ) Operator: + Asap value: 12
```

Concluding, we can say that the first approach in performing attribute grammar based hardware compilation gives correct results but it is quite complicated (consider the discussion of the previous subsection). The attempt to maintain global variables, to store dependencies between code lines produces a complicated specification, which is in contrast to the fundamental idea of the current work (propose a specification methodology which should be abstract and easy to understand by hardware designers). However, this inefficiency is due to the fact that the parse tree is used as the internal representation of the circuit implementation, instead of the data flow graph used in most traditional (non attribute grammar based) hardware compilation environments. This is the basic idea for the second approach for attribute grammar based hardware compilation, described in the next section.

7. Second approach. Construction of a data flow graph

FNC-2 provides a very useful mechanism for constructing new trees. Attributes may carry trees themselves, that are constructed during the attribute evaluation phase of the parse tree. The tree construction mechanism is implemented with the use of functions, implicitly declared by the use of operators (in FNC-2, an operator is used to declare new children in a tree node) in the AAS tree definition. After invoking the required tree construction functions, a new tree constructed as an attribute, can be passed to other grammar evaluations for further computations.

This tree construction mechanism enables us to create a whole new tree from the input source code that comes closer to a data flow graph. To overcome the differences between trees and graphs described earlier, we have to add joint points in the tree, which actually add implicit connections

```
grammar output is

root is OUTPUT ;
OUTPUT = points ;
points -> POINTS ;
POINTS = joints ;
joints -> JOINT * ;
JOINT = joint ;
joint -> OPERATOR ;
OPERATOR = operator ;
operator -> OPERAND OPERAND ;
OPERAND = opinp opop opreg ;
opinp -> INPUT ;
INPUT = input ;
input -> ;
opop -> OPERATOR ;
opreg -> JOINT ;

$id (JOINT,INPUT) : token ;
$op (OPERATOR) : token ;
$joints (POINTS) : ejoints ;

end grammar ;
```

Figure 4: Abstract syntax to describe the data flow graph

between nodes not possible to connect in a tree.

7.1. The new AAS tree (data flow tree)

The construction of the data flow graph is based on an abstract syntax, given in figure 4. A part of the graphical representation of the data flow abstract syntax tree for a sequence of three variable defining expressions is given in figure 5. Again, the symbol "|" is used to denote alternative configurations. Words in lowercase define operators and words in uppercase node names. Symbol (T) defines a terminal node. When a node does not have either children or symbol (T), it is assumed to have the same children as its immediately left sibling.

In this implementation, we assume that a variable in input text is not only declaring a register for storing data. It also declares a joint point in the data flow AAS tree. In other words, it is considered as an easy and comprehensive way for the programmer to break the whole operation into pieces and store intermediate results. So the first step, before any implementation detail is added, is to transform the AAS tree into a series of joint points and computational nodes, described as children of JOINT nodes. The only attributes needed for this transformation are:

id Used to identify joint points and input.

op Holds operator types (+,-,* e.t.c).

joints This attribute holds a list of the joints corresponding to variable definitions, to be used later for register allocation.

The new constructed tree holds each expression in a subtree (one of the root JOINT children). Each variable from the input text corresponds to a joint point (a JOINT node) and its definition is

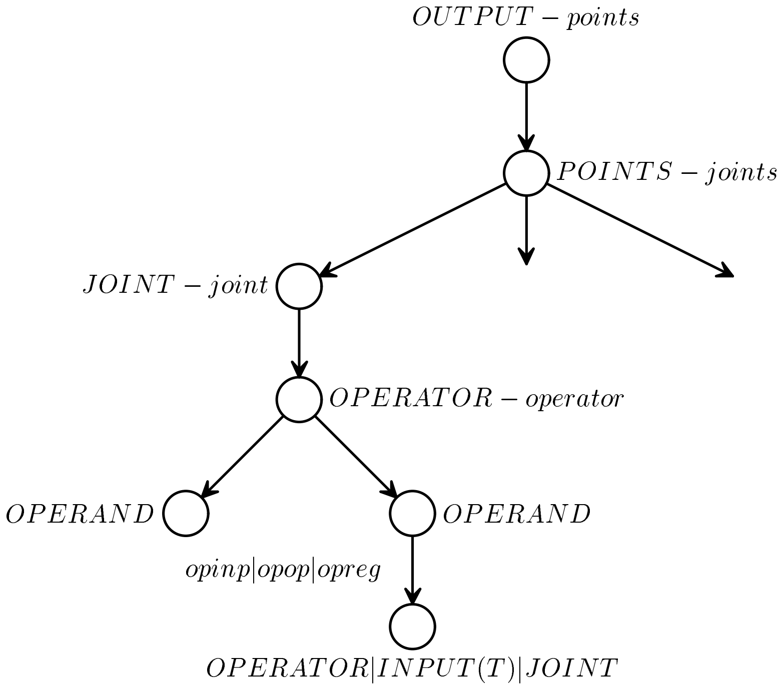

Figure 5: Representation of the data flow abstract syntax tree

the operations needed to be calculated, to reach this joint point. So the only child of a JOINT node is an OPERATOR node, holding the last operation that should be computed before reaching the joint point. The OPERATOR node has two OPERAND children. Each one of them may lead to an INPUT node (if the operand is a constant number), or a new OPERATOR for the rest of the expression, or a JOINT node (if one of the OPERANDS needs the value of another computation performed in another children of the POINTS root node).

7.2. Some useful list types

For implementation reasons, special purpose data structures are required to design the data flow AAS tree. First of all, we need a list of records that hold the correspondence between joints and variables. It is needed to know which variable is converted into which node, for proper joint connection. So, this list contains records with two fields. One holding the variable id and the other holding the joint identification number, to which it corresponds. For list manipulation, two functions have to be implemented. One that searches the list for a specific variable id record and one that inserts the joint identification number to this record. Both of them follow the same implementation pattern, as the definition table of the previous section. Second, we need a list that stores joints along with all their implementation details (scheduling step, register allocated, e.t.c). This list must also have a search function and a new function for each implementation detail attribute attached. Assignment of the values to all details is done in a second grammar pass. During tree construction, this list is only filled with the joint identification numbers.

7.3. The tree construction grammar

This grammar constructs the new AAS tree, consisting of operations and joint points. First, only children of the POINTS node and considered and the list of the joints is constructed. Each child refers to a joint and is assigned an identification number equal to the automatically created pos attribute of the FNC-2 system. This way we are sure that no two joints will have the same id.

A second, left to right traversal, creates the list for the variable references. Each record in this list is filled with the id name of the variable and the position of the POINTS node corresponding child (the identity of the joint). In this way we already have the array of variable references with respect to the joints.

Each terminal of the input grammar carries a tree construction function. The attributes that carry the intermediate results of tree computations are all synthesized. So the new tree is constructed bottom-up and left to right.

num⇒NUM Constructs an INPUT below an OPERAND node and attaches to its id attribute the value of the constant.

var_use⇒VARUSE Constructs a JOINT below an OPERAND node with no children and attaches to its id attribute the number of the joint it is implicitly connected. This number is provided by a search in the variable reference list created before.

newop⇒OPS Attaches a whole subtree of computations.

op⇒OPS Constructs an OPERATOR node with two OPERAND children, the two previously constructed leaves from NUM or VAR_USE or OPS. The attribute op is assigned the current operator's id.

def⇒DEF The newly constructed subtree of computations is added as a new child to the POINTS node. The id attribute of that JOINT takes the value of the automatically created attribute pos of FNC-2.

After pass the input text through this grammar, the output tree is closer to the actual data flow graph. Next, circuit implementation details are attached as attributes to nodes of this tree, instead of the whole parse tree.

7.4. Scheduling attributes

After having constructed the new AAS tree, more implementation details may be calculated. Considering ASAP scheduling again, only one new attribute will decorate the output tree of the data flow graph construction grammar. Like in the first approach, it will be the asapvalue attribute holding the ASAP value for each operator used. However, the implementation of the attribute evaluation scheme is easier than before, because it is not needed to care about define-use dependencies. Traversing from left to right and bottom-up ,each input constant is initialized with an ASAP value of 0. At each OPERATOR node, its ASAP value is computed by the following script:

```
asapvalue(OPERATOR)=IF asapvalue(OPERAND1)>asapvalue(OPERAND2) THEN
                        asapvalue(OPERAND1)+1
                      ELSE
                        asapvalue(OPERAND2)+1
                      END
```

When the root of the subtree of the first JOINT child of the POINTS node is reached, the ASAP value of the last OPERATOR becomes the ASAP value of the joint and is assigned to the corresponding field in the joint list. Traversing from left to right each use of that joint will have to just search in the joint list and find its ASAP value.

Without any other coding, with this grammar we get as output the ASAP scheduled data flow graph.

7.5. Advanced computations

More involved computations may be implemented following the second approach. Consider another scheduling heuristic, the dual of ASAP, the *As Late As Possible* (ALAP) scheduling heuristic. ALAP requires that each operation is scheduled one control step before the earliest use of its output. ALAP scheduling implementation is a little bit more complicated. The restriction for only a left to right bottom-up traversal of the tree does not allow to schedule the uses of the joints first, before their definitions, which is required to evaluate correctly ALAP scheduling values. To overcome this problem the ALAP implementing grammar must be executed multiple times. In fact, this grammar is called within a loop construct n times, where n is the number of joints the tree has.

For implementing ALAP scheduling, another field is required in the joint list. This alap field is initially assigned the maximum allowed control step value (ex.10). An alapvalue attribute is declared as inherited. This means that its value propagates top-down. At the joint->OPERATOR production the alapvalue attribute gets the value stored in the corresponding joint record. The alapvalue attributes of the operators at its subtree are defined by this simple script:

```
alapvalue(OPERAND1)= alapvalue(OPERATOR)-1;
alapvalue(OPERAND2)= alapvalue(OPERATOR)-1;
```

Whenever a JOINT use is reached, the attribute grammar evaluator stores the OPERAND's alapvalue attribute to the alap field of the joint in the joint list. An iterative process is initiated, in order to fill all these fields with the alapvalue attribute of the last use of each joint. In the end, a final tree traversal assigns alapvalues to the definition subtrees of each these joints.

During the first iteration, none of these definitions is scheduled correctly except the last (last joint). This last definition assigns the correct alapvalue to all the joints it uses. At the second iteration, the last definition remains unaffected but now the new alapvalues of the joints it uses are defined correctly and are used to schedule their definition subtrees. At this point, new joint uses are defined correctly. Continuing like that, when the loop iteration counter reaches the number of the JOINT children of the root, all definitions have the correct alapvalue.

7.6. Experimentation

After passing the AAS data flow graph through all implementation grammars, like the ASAP and ALAP scheduling examples, the architectural description can be written, in the same way as the first approach. For example, consider the following set of operations and the corresponding output, which reports all ASAP and ALAP calculated values for all joint points:

Input text:

```
x=7+8;
y=8+9+x;
z=x+y+6+7;
```

Output text:

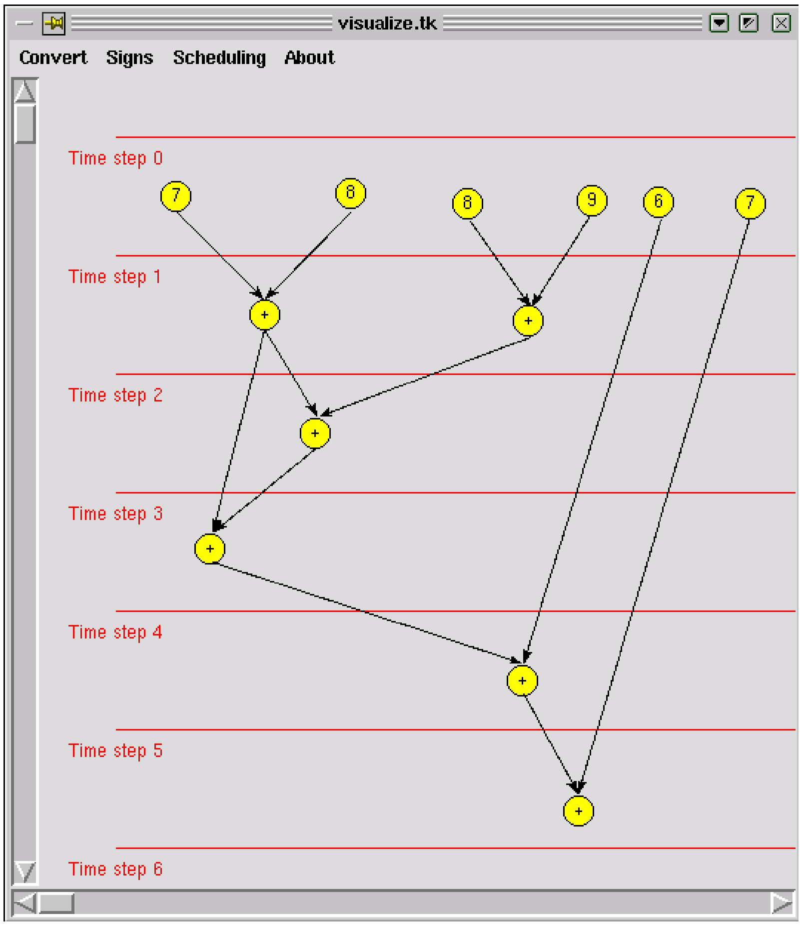

Figure 6: Output presentation using Tcl/Tk

Joint	ASAP	ALAP
2	2	7
1	1	7
3	5	10

```
J# 1 (in 7 ) [ + 1 6 ] (in 8 )
J# 2 (in 8 ) [ + 1 6 ] (in 9 ) [ + 2 7 ] (J# 1 )
J# 3 (J# 2 ) [ + 3 8 ] (J# 1 ) [ + 4 9 ] (in 6 ) [ + 5 10 ] (in 7 )
```

Also, the final grammar produces Tcl/Tk code, with which a graphical representation of the scheduled data flow graph is generated. This is a valuable addition for hardware designers, who have an illustrative way of examining the results of all automated transformations. For the above example, the ASAP scheduled data flow graph is presented as shown in figure 6.

8. Conclusion

Two different version of a full featured hardware compiler have been implemented using the FNC-2 attribute grammar system. They both present different advantages and disadvantages. Overall, they both show that advanced compiler construction tools can be effectively adopted to handle complex transformations for high-level hardware design. Attribute grammars offer an efficient and unifying infrastructure for different problems of high-level languages based design. This methodology is a great improvement since it supports fast prototyping of digital systems using a simple and uniform declarative specification and implementation formalism. The resulting design environment stands as a meta-level between hardware transformations and their implementation, and can be proven valuable in fast evaluation of new research ideas and techniques.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] J. Bhasker. *A VHDL Primer.* Prentice Hall, 1992.

[3] K. C. Chang. *Digital Design and Modeling with VHDL and Synthesis.* IEEE Press, 1997.

[4] P. Deransart and J. Maluszynski. *A Grammatical View of Logic Programming.* MIT Press, 1993.

[5] G. Economakos and G. Papakonstantinou. Exploiting the use of VHDL specifications in the AGENDA high-level synthesis environment. In *24th EUROMICRO Conference, Workshop on Digital System Design*, pages 91–98. EUROMICRO, 1998.

[6] G. Economakos and G. Papakonstantinou. A formal method for hardware design using attribute grammars. In *6th International Conference on Electronics, Circuits and Systems*, pages 233–236. IEEE, 1999.

[7] G. Economakos, G. Papakonstantinou, K. Pekmestzi, and P. Tsanakas. Hardware compilation using attribute grammars. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 273–290. IFIP WG 10.5, 1997.

[8] G. Economakos, G. Papakonstantinou, and P. Tsanakas. An attribute grammar approach to high-level automated hardware synthesis. *Information and Software Technology*, 37(9):493–502, 1995.

[9] G. Economakos, G. Papakonstantinou, and P. Tsanakas. AGENDA: An attribute grammar driven environment for the design automation of digital systems. In *Design Automation and Test in Europe Conference and Exhibition*, pages 933–934. ACM/IEEE, 1998.

[10] G. Economakos, G. Papakonstantinou, and P. Tsanakas. Incorporating multi-pass attribute grammars for the high-level synthesis of ASICs. In *Symposium on Applied Computing*, pages 45–49. ACM, 1998.

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

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[13] A. Hemani, J. Oberg, and D. Lindqvist. System level virtual prototyping of DSP ASICs using grammar based approach. In *International Workshop on Rapid System Prototyping*, pages 166–171. IEEE, 1999.

[14] L. G. Jones and J. Simon. Hierarchical VLSI design systems based on attribute grammars. In *13th Symposium on Principles of Programming Languages*, pages 58–69. ACM, 1986.

[15] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Language Design and Implementation*, pages 209–222. ACM SIGPLAN, 1990.

[16] K. Keutzer and W. Wolf. Anatomy of a hardware compiler. In *Conference on Programming Language Design and Implementation*, pages 95–104. ACM SIGPLAN, 1988.

[17] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[18] Y-L. Lin. Recent development in high level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2(1):2–21, 1997.

[19] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[20] J. Oberg, A. Jantsch, and A. Hemani. Validation of interface protocols using grammar-based models. In *International High Level Design Validation and Test Workshop*, pages 40–46. IEEE, 1998.

[21] J. Oberg, A. Kumar, and A. Hemani. Grammar-based hardware synthesis of data communication protocols. In *9th International Symposium on System Synthesis*, pages 14–19. ACM/IEEE, 1996.

[22] J. Oberg, A. Kumar, and A. Hemani. Grammar-based hardware synthesis from port-size independent specifications. *IEEE Transactions on Very Large Scale Integration Systems*, 8(2):184–194, 2000.

[23] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[24] J. Paaki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[25] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Prentice Hall, 1996.

[26] A. Seawright and F. Brewer. High-level symbolic construction techniques for high performance sequential synthesis. In *30th Design Automation Conference*, pages 424–428. ACM/IEEE, 1993.

[27] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Transactions on Very Large Scale Integration Systems*, 2(2):172–185, 1994.

[28] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck. A system for compiling and debugging structured data processing controllers. In *European Design Automation Conference with EURO-VHDL*, pages 86–91. ACM/IEEE, 1996.

[29] M. Sideri, S. Efraimidis, and G. Papakonstantinou. Semantically driven parsing of context-free languages. *The Computer Journal*, 32(1):91–93, 1989.

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

[31] R. A. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, 1991.