Behavioral Synthesis with SystemC*

George Economakos, Petros Oikonomakos, Ioannis Panagopoulos,
Ioannis Poulakis and George Papakonstantinou
National Technical University of Athens
Department of Electrical and Computer Engineering
Zographou Campus, GR-15773 Athens, Greece

E-mail: george@cslab.ece.ntua.gr

Abstract

Having to cope with the continuously increasing complexity of modern digital systems, hardware designers are considering more and more seriously language based methodologies for parts of their designs. Last year, the introduction of a new language for hardware descriptions, the SystemC C++ class library, initiated a closer relationship between software and hardware descriptions and development tools. This paper presents a synthesis environment and the corresponding synthesis methodology, based on traditional compiler generation techniques, which incorporate SystemC, VHDL and Verilog to transform existing algorithmic software models into hardware system implementations. Following this approach, reusability of software components is introduced in the hardware world and time-to-market is decreased, as shown by experimental results.

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 and language based design descriptions. However, each design must be described, eventually, at the lowest level (e.g. layout masks), in order to be fabricated, through various synthesis processes. This has motivated the *Electronic Design Automation* (EDA) industry to produce software tools, which accept language based design specifications (most of the times, schematics can also be used if the user is more familiar) and perform synthesis.

The most widely used *Hardware Description Languages* (HDLs) today are VHDL [2] and Verilog [1]. Since their adoption as IEEE standards, they have been enthusiastically adopted by they EDA industry also. Today, having overcome initial maturity problems, they are used in many design houses all over the world. Last year, a new competitor came into the market, the SystemC C++ class library [14]. Even though commercial synthesis environments based on SystemC are not available yet, this language promises a higher level of design abstraction. Since it is a C++ class library, it can operate on software, and thus algorithmic, system models and use software development tools (C++ compilation environments) for simulation.

Algorithmic or behavioral hardware modeling introduce a higher level of design abstraction for the EDA industry. *High-level or behavioral synthesis* [3, 9, 17], 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.

When language based design entry is used, high-level synthesis presents many similarities with traditional compiler construction (at least during the initial transformation stages). 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, some problems have to be solved before it is widely accepted by both industry and academia. Among them, high-level synthesis lacks a theoretical framework (like Boolean algebra for logic design) that would further accelerate research. Examples of hardware design environments based on compilation techniques can be found in [5, 6, 7, 10, 12].

This paper presents a high-level hardware compiler that takes SystemC behavioral input specifications and gener-

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

ates VHDL, Verilog and SystemC RTL output specifications, after performing high-level synthesis. Utilizing this environment, a whole new hardware design methodology is presented, which can start by writing new or reusing already tested software models. The basic building block for the new environment is a robust and flexible compiler construction system called Eli [16], which offers declarative, and thus more abstract, ways to describe the problems of high-level synthesis and their solutions. Declarative notations along with modularity form an abstraction layer, a meta-level between hardware transformations and their implementation. The performance of the overall environment in both execution speed and quality of results is very promising, as shown with experimental results.

2. Hardware Compilation Environment

The design environment used to build hardware models out of algorithmic specifications is based, as stated above, on the Eli compiler construction system. Eli makes extensive use of the *Attribute Grammar* (AG) computational model, originally proposed in [8]. AGs consist of a set of syntactic rules and a set of domain specific values called attributes. Each syntactic rule is associated with a number of attributes and equations, called semantic rules, which define each attribute in terms of other attributes of the syntactic rule (or even of remote syntactic rules in the case of Eli). Large computations based on a syntactically defined input set can be performed with AGs. Their advantage is that the programmer defines the relations between attributes, which most of the time represent characteristic values of the input set, and not the computation steps needed to calculate them (loops, conditions, etc). Attribute dependencies determine the order of attribute computations. Attributes can hold complex data types, even text templates (which is extensively used the current work to produce output in different languages).

2.1. Hardware Transformations Using AGs

When language based design is applied, behavioral circuit transformations can be performed during a compilation phase, using AGs. This happens because, compilation is based on the parse tree of a behavioral description, which is in fact a superset of its dataflow graph, on which behavioral transformations are applied. In this context, scheduling for example is performed, by decorating the nonterminal symbols of the parse subtree corresponding to primitive operations, with an attribute that is evaluated as the control step at which the operation will be performed. By altering the semantics, the evaluation rules are altered and thus, different heuristics are implemented.

For simple scheduling heuristics, like ASAP and ALAP, evaluation rules are very easy to implement since decision about the time when each operation will be performed, depends on the immediate inputs and outputs of the operation. By generating local dependencies between input and output attributes, whole operator chains are scheduled. Using an automated compiler writing system based on AGs, this formalism works as an executable specification also and thus, a hardware compiler performing ASAP or ALAP scheduling to every input behavioral description is automatically generated.

However, the ASAP and ALAP scheduling examples are rather restricted and of no practical use. Modern scheduling and allocation heuristics require complicated computations. To support them, an automated compiler construction system must be rich in expressive power and provide computational constructs that, along with simple attribute evaluation rules, can describe any kind of dataflow graph computation. Such constructs are provided by the Eli compiler construction system.

In brief, four basic advanced constructs of Eli can be applied to define advanced high-level synthesis transformations. The first is support for iterative attribute evaluations, which leads to generalized loop computations through attribute dependencies (all attributes that depend on an iterative attribute are also iteratively evaluated). The second construct is remote attribute dependency operators, which lead to a multi-pass and global attribute evaluation algorithm, transparent to the user (the user writes dependencies and the system determines the correct visit sequence which will satisfy them). The third construct is the chain dependency operator, which evaluates and propagates the value of an attribute at all nodes of the parse tree, during a left-to-right depth-first traversal. The chain dependency may be used to force multiple passes through all nodes of the parse tree. The final advanced construct is the dependency operator, used to describe dependent computations in time. That is, the computation at the left of the operator, usually an attribute evaluation, will be executed after evaluating a list of other attributes, found at the right, regardless of their values (more details about Eli can be found in [16]).

These constructs can be put to use for the design of executable and formal descriptions of advanced transformations, like resource constrained list scheduling [3]. For each operator type, ready operators are inserted in a different priority list, using the operator's modality (ALAP-ASAP value) as its priority. Iterating through the available control steps, operators are scheduled as long as resources are available. This algorithm can be expressed using Eli advanced specification constructs, in order to be performed during a compilation phase. This specification, in pseudocode, is given below.

```
At each operator node:
```

```
 compute ASAP
 compute ALAP
 compute modality
At root of the parse tree:
 Cstep=1
ITERATE UNTIL all operators are scheduled
 With a chain:
  for each ready operator put its modality
  into a root list attribute (one for each
  operator type)
 At each operator node:
  if ready and modality has a position in
  list such that resources are available,
  then schedule it at root.Cstep and make
  scheduled=true
 At root of the parse tree:
  Cstep=Cstep+1
END ITERATE
```

As a second representative example, consider the problem of optimum register allocation and the left-edge algorithm [3] used to solve it. For each variable of the behavioral description, the 2-tuple `(StartTime,EndTime)` represents its lifetime interval. Variables not yet mapped to registers are inserted in a list in ascending order with their start times as the primary key, and in descending order with their end times as the secondary key. Iterating through available registers, compatible variables are detected and mapped to the same register. As in the case of list scheduling, this algorithm can also be expressed to work during a compilation phase, using advanced specification constructs. This specification in pseudocode, implemented using Eli syntax in a straightforward manner, is given below.

```
At each variable node:
 compute start
 compute end
At root of the parse tree:
 reg=1
ITERATE UNTIL all operators are scheduled
 At root of the parse tree:
  last=0
 With a chain:
  put each not mapped variable into a root
  list attribute
 At each variable node:
  if not mapped, has start≥root.last
  and all previous opera-
tors in list can not
  be mapped, map it to register root.reg,
  make root.last=end, delete it from list
  and make mapped=true
 At root of the parse tree:
  reg=reg+1
END ITERATE
```

In this way, the basic operations of high-level synthesis are performed in a compiler generator environment. However, further functionality is required. Resource constraints are maintained using a symbol table type of construct, like in [6]. Timing constraints and interface specifications are given following a special syntax, and play the role of initial values for scheduling attributes [11]. User interaction is through Tcl/Tk scripts, which present a graphical view of the synthesized dataflow graph of the algorithmic description along with its textual specification.

2.2. Language Interfaces

With the methodology presented in the previous subsection, the parse tree of an input behavioral specification is transformed into a structural RTL description. However its effectiveness depends on the input and output language interfaces, which integrate the proposed system with other components in the design automation process. The presented system includes one input language interface, for SystemC, and three output language interfaces, for VHDL, Verilog and SystemC.

The input language interface corresponds to the syntax of the input behavioral specification and is given in a separate file, as a set of productions in Eli. SystemC has been chosen as the input language because it is based on a traditional programming language and may look more familiar for writing behavioral models.

The output language interfaces produce synthesizable VHDL and Verilog architectural descriptions, as produced after high-level synthesis, which can be used at lower levels of the synthesis process. Furthermore the same architectural details and the same architectural description style is used to generate architectural SystemC descriptions also. With this output, pre-synthesis and post-synthesis simulation results can be obtained from the same test pattern generator and in the future, if SystemC synthesizers become available, RTL synthesis will also be performed.

To generate architectural descriptions, each output language interface generate language constructs that correspond to registers and functional units. Registers are described in VHDL with a process that includes the `if clk='1' and clk'event` construct and in Verilog with the always block `always @ (posedge clk)`. A similar construct in SystemC is to declare a member function as being `sensitive_pos(clk)`. Functional units are straightforward to describe. They correspond to operators in expressions, provided the correct data types and operator functions are available (sometimes, they can be found in special purpose library units).

Under the Eli environment, output for all languages is produced using the *Pattern-based Text Generator* (PTG) tool. A PTG specification is a set of named patterns describing the structure and textual components of the output description. Each pattern corresponds to a function,

which yields an internal representation of a pattern application. These functions are called at appropriate nodes of the parse tree. Their arguments are either attributes calculated during high-level synthesis, or specific, syntax based information about the particular tree node (like the functionality of an operator - addition, subtraction, multiplication, division, etc.). The result of these function calls is a pointer attribute, which points to ready-to-be-output text patterns. All ready patterns are actually output at a later phase by the system, after all transformations have been performed. So, the whole synthesis process can be split into three parts: high-level synthesis transformations through attribute evaluations, pattern preparation with PTG and finally system initiated pattern output. The concept of text patterns attached to tree nodes makes output coding very flexible, because it supports modular and reusable coding techniques.

As an example, consider VHDL coding of a functional unit that is required to work at a specific control step. The following code fragment is required.

```
if (present_state=state1) then
 a1<=a2+a3;
endif;
```

Such coding is generated using the following general text pattern, called `ifframe` (`\n` and `\t` are the newline and tab characters).

```
ifframe : "\t\t\tif present_state=state"
                                  $1 " then\n"
          "\t\t\t\t" $2 "<=" $3 $4 $5
          ";\n\t\t\tendif;"
```

The `ifframe` pattern instructs Eli to generate the function *PTGiframe*, which takes five arguments (`$1`, `$2`, `$3`, `$4` and `$5` in the above text) and when called, returns a pointer to a piece of code with all arguments placed as the pattern dictates. This function is called at all nodes of the parse tree that have the form *operation* $\rightarrow$ $operand_1$ *operator* $operand_2$ as following.

$$PTGiframe(operation.cs, operation.place, operand_1.place, operator.place, operand_2.place)$$

where *cs* is the control step when operation is to be executed and *place* is a string attribute that holds the lexical values of the corresponding tokens.

3. Design Methodology

The design environment presented in the previous section support a new algorithmic level design methodology that can transform software into hardware system models.

Under this methodology, a design starts by writing a new or reusing a pre-existing software implementation of the algorithm under implementation using the C++ language. The software model is tested with the corresponding software development environment. Next C++ is changed into SystemC in a straightforward way and timing is introduced to the design. With the same software development environment this initial hardware model is tested against the software model. Next, each member function of the behavioral SystemC implementation is passed through the synthesis environment presented above. From the VHDL and Verilog outputs, synthesis goes on until the final implementation is reached. The SystemC output replaces parts of the initial hardware model and through simulation, it can validate the results of high-level synthesis with the same test vector generator.

The effectiveness of the proposed methodology will be shown with experimental results.

4. Experimental Results

The presented synthesis environment has been found to provide notable advantages, especially for researchers. This is due to the fact that the 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.

Another advantage is that 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. Relevant to this is the fact that the Eli system includes a library of specifications, for some common subproblems, which are easily available.

However, a question that had to be answered was the efficiency of the proposed methodology. For this reason tests were conducted with a number of randomly generated benchmark circuits, a number of benchmark circuits found in [4] and a complete example of a medical application found in [15].

From the randomly generated benchmarks, the execution speed of the environment was measured. Table 1 shows execution times for experiments with different scheduling heuristics, using a Pentium 166MHz Linux based workstation. It is shown that the new environment can handle both small and large experiments in considerable time.

	10 nodes	50 nodes	100 nodes
ASAP	0.02 sec	0.09 sec	0.17 sec
ALAP	0.02 sec	0.09 sec	0.18 sec
LIST	0.03 sec	0.10 sec	0.20 sec

Table 1. Execution times for randomly generated circuits

From the benchmarks taken from [4], the final results were compared with results obtained from equivalent behavioral specifications, passed through the Synopsys Behavioral Compiler [13]. The results of the proposed environment used resources that ranged from 16% less than the corresponding result from Behavioral Compiler to 5% more than the corresponding result from the Behavioral Compiler.

Finally, the example found in [15] implements a feature detection algorithm, which consists of five computational components, a low-pass filter, a high-pass filter, a derivation, a squaring and a moving window integration. A software model for each component is given in [15]. All five components were written in SystemC and passed through the proposed environment separately. At the same time the software models were manually translated into VHDL, without changing coding style. Since the specific software models used common and simple constructs, it turned out that the manually generated VHDL code was synthesizable by commercial RTL synthesizers. So both the automatically generated output of the proposed environment and the manual design were passed through the Xilinx's Foundation Express [18] synthesizer and implementation environment, using different synthesis constraints (bit width of operands, target library, etc). The results of the proposed environment used resources that ranged from 7% less than the corresponding result from Foundation Express to 6% more than the corresponding result from Foundation Express.

5. Conclusion

This paper has presented a new design environment for high-level hardware synthesis, involving VHDL, Verilog and the recently introduced SystemC. The corresponding design methodology utilizes a traditional compiler generator, to implement behavioral transformations and automatically translate existing software projects into hardware. Experiments have shown that this approach offers advantages in design space exploration, without compromising either execution times or quality of results. Moreover, the presented environment makes extensive use of declarative programming constructs and thus, it stands as a meta-level between hardware transformations and their implementation. Such toolsets can be proven valuable in fast evaluation of new research ideas and techniques in the field.

References

[1] J. Bhasker. *A Verilog HDL Primer, Second Edition.* Star Galaxy Publishing, 1999.

[2] J. Bhasker. *A VHDL Primer, Third Edition.* Prentice Hall, 1999.

[3] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[4] N. Dutt and C. Ramachandran. Benchmarks for the 1992 high-level synthesis workshop. Technical Report 92-108, UCI, 1992.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] I. Poulakis, G. Economakos, and P. Tsanakas. Interaction in language based system level design using an advanced compiler generator environment. In *Workshop on VLSI*, pages 97–102. IEEE/CS, 2000.

[12] 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.

[13] Synopsys. *Behavioral Compiler User Guide Version 1999.10*, 1999.

[14] Synopsys, CoWare, Frontier Design. *SystemC Version 1.0 User's Guide*, 2000.

[15] W. J. Tompkins. *Biomedical Digital Signal Processing: C-Language Examples and Laboratory Experiments for the IBM PC.* Prentice Hall, 1995.

[16] W. M. Waite. An executable language definition. *ACM SIGPLAN Notices*, 28(2):21–40, 1993.

[17] R. A. Walker and S. Chaudhuri. High-level synthesis: Introduction to the scheduling problem. *IEEE Design & Test of Computers*, 12(2):60–69, 1995.

[18] Xilinx. *Foundation Series 2.1i User Guide*, 1999.