Refinement and Property Checking in High-Level Synthesis Using Attribute Grammars

George Economakos and George Papakonstantinou

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

Abstract. Recent advances in fabrication technology have pushed the digital designers' perspective towards higher levels of abstraction. Previous work has shown that attribute grammars, used in traditional compiler construction, can also be effectively adopted to describe in a formal and uniform way high-level hardware compilation heuristics, their main advantages being modularity and declarative notation. In this paper, a more abstract form of attribute grammars, relational attribute grammars, are further applied as a framework over which formal hardware verification is performed along with synthesis. The overall hardware design methodology proposed is a novel idea that supports provable correct designs.

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, 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. All such processes need some kind of validation for their results. This validation can be performed by *formal verification* [11], mainly to prove that a transformation from one state to another is correct (*refinement checking*), that two states, initial and transformed, are equivalent (*equivalence checking*) or that the transformed state satisfies certain conditions (*property checking*).

High-level synthesis [7,9,10,13], 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. Even though it has been introduced over twenty years ago, it has recently gained acceptance because the lower level tools have matured enough to support it. However, a lot of problems are still open.

Attribute grammars (AGs), were devised by Knuth [8] 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. The main advantage of AGs over other formal specification methods is that they can also be used as an executable method, for the automatic construction of programs to implement the specified mapping [12].

Attempting to overcome the inefficiencies of conventional high-level synthesis and propose a unifying formal framework, an AG formalism describing scheduling heuristics was proposed in [4], which operates by decorating the parse tree of a behavioral circuit description with appropriate attributes. Recently [3,6,5], this methodology was realized into the AGENDA integrated design environment that supports top-down implementation of behavioral descriptions using the VHDL hardware description language [2]. Overall, the main advantages of AGs as a formal specification and implementation high-level synthesis formalism are modularity and the declarative notation used for implementation.

In this paper, an extended grammar based methodology is given, which can support two kinds of formal verification, refinement and property checking. It is based on the definition of a more abstract form of AGs, the *Relational Attribute Grammars* (RAGs) [1]. This methodology is a great improvement since it supports provable correct high-level synthesis transformations using a simple, formal and uniform specification and implementation formalism.

2 Attribute Grammars in Synthesis

High-level synthesis transformations can be performed during semantic analysis using AGs. For example, scheduling 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 each operation will be performed. By altering the semantics, the evaluation rules are altered and thus, different heuristics are implemented. For example, consider the ASAP scheduling algorithm. Using AGs, ASAP scheduling is performed by attaching special attributes to all primitive operator parsing syntactic rules, like the following:

$$operation \rightarrow operand_1 \; operator \; operand_2 \tag{1}$$

ASAP scheduling requires that each output must be scheduled in the next control step after all its inputs have been scheduled. This can be accomplished by using an attribute to pass scheduling information (the control step at which the operator is scheduled) from inputs to outputs, with the following semantic rule attached to (1):

$$operation.ASAPcs = MAX(operand_1.ASAPcs, operand_2.ASAPcs) + 1$$

The scheduling information, along with all other information about each primitive operation, is inserted at each such rule, into a special, list type attribute, that passes information from all leaves of the parse tree to the root. So evaluation of the whole AG, results in a root attribute that accumulates the scheduled CDFG of the behavioral description.

3 Attribute Grammars in Refinement Checking

After evaluating the synthesis AG of the previous section, a special attribute of the root of the parse tree contains the scheduled CDFG of the given behavioral description. However, the resulting architectural implementation may or may not be correct. To prove its correctness, a *specification* [1] must be constructed, which can verify certain conditions of the synthesis or refinement process. Generally speaking, a specification is a set of formulas for each nonterminal symbol of the underlying grammar, where all free variables are attributes of that symbol. Each formula may be true or false. A specification is said to be *inductive* if, for any production rule $p = X_0 \rightarrow X_1 \ldots X_n$, when the specifications of all $X_i, i = 1 \ldots n$ are true and all attributes of the rule have been evaluated, the specification of X_0 can be proven to be true. When a specification is inductive, the AG is correct with respect to it.

For the circuit implementation of a behavioral description to produce correct outputs, one condition must hold. For each input of operator o_i found in the scheduled CDFG and assigned a value in a previous statement in the behavioral description, if this assignment has been scheduled at some control step s_i, o_i must be scheduled later than s_i. In other words, variable dependencies of the original description have not been violated. This can be proven to hold by proving that the synthesis AG is correct with respect to a corresponding specification.

4 Attribute Grammars in Property Checking

After evaluating the synthesis AG of the first section, the scheduled CDFG, which can be seen as a rough architectural implementation (assuming a greedy allocation), is contained in the special attribute of the root of the parse tree. Even though the refinement process may be correct, checked with inductive specifications as described in the previous section, the implementation may not satisfy some design constraints, because inappropriate synthesis algorithms have been used. In that case, design constraints can be tested using *semantic conditions*. Semantic conditions are relations that the attributes of some production must satisfy for attribute evaluation to be valid. Semantic conditions differ from inductive specifications because they check only local (within a single production) conditions. Since, attributes contain implementation details, attribute relations directly reflect implementation properties. If a relation is true, the corresponding property holds and so, semantic conditions can be used for property checking.

5 Experimental Results and Conclusion

Experiments have been conducted, synthesizing provable correct hardware modules. The traditional way to validate the results are through simulation, behavioral or presynthesis as well as postsynthesis. For each example at least a few hours were needed to produce output waveforms for different inputs and validate them. For large circuit implementations, with large number of gates, simulation must be very thorough in order to find errors. On the contrary, using AGENDA all examples were synthesized with a single iteration through the design process, requiring milliseconds on a Sun Ultra SPARC 140Mhz. Moreover, the final implementations were proven correct following the proof method presented in this paper. For each different AG, the proof is needed only once, while, when simulation is used, each example must be validated. The number of AG code to be proven correct is much smaller than the number of gates in the implemenation, so the problems seem to be less complicated, and most of them are trivial (no need for any proof). All these advantages can dramatically increase the designer's productivity.

References

1. P. Deransart and J. Maluszynski. *A Grammatical View of Logic Programming.* MIT Press, 1993. 331, 332
2. 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. 331
3. 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. 331
4. 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. 331
5. 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. 331
6. 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. 331
7. D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis.* Kluwer Academic Publishers, 1992. 330
8. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. 330
9. Y-L. Lin. Recent development in high level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 2(1):2–21, 1997. 330
10. 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. 330
11. K. L. McMillan. Fitting formal methods into the design cycle. In *31st Design Automation Conference*, pages 314–319. ACM/IEEE, 1994. 330
12. J. Paaki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995. 331
13. R. A. Walker and S. Chaudhuri. High-level synthesis: Introduction to the scheduling problem. *IEEE Design & Test of Computers*, 12(2):60–69, 1995. 330