VHDL Quality: Synthesizability, Complexity and Efficiency Evaluation

M. Mastretti
Italtel SIT− 20019 Settimo Milanese (MI)
M.L. Busi, R. Sarvello, M. Sturlesi, S. Tomasello
University of Milano − Computer Science Department

Abstract

With VHDL models increasing their size, it becomes more important to assure the quality of these descriptions in order to improve simulation performances, to make project maintainability easier and to create an efficient link with hardware synthesis results. Goal of this paper is to summarize the activities carried out within the SAVE project, leading to the development of a collection of static analysis tools in order to reduce the time spent in the design verification phase, to improve modifiability, reusability and readability of models and focusing on the different aspects related to hardware semantics (synthesizability analysis).

1: Introduction

Because of the important role played by VHDL in design methodologies and the increasing complexity and size of descriptions, assuring that VHDL code is developed following some well–founded guidelines may have a relevant impact on the quality of the overall design process. As many designers simultaneously work on the same project, defining a standard coding style assumes a considerable importance.
It is possible to analyze the quality of VHDL code under different aspects: software complexity, good synthesis results, testability and high simulation performances can be considered the most important attributes in quality evaluation of a VHDL model.
A good improvement derives from the definition of suitable metrics and guidelines from software complexity, synthesizability and testability points of view. Starting from software engineering approaches, some metrics and guidelines have been adapted to VHDL to make it more readable and maintainable, while, regarding synthesizability and testability, they have been experimentally determined and assessed starting from the existing literature. This kind of analysis is appreciated by designers and, in this way, it satisfies their demands for standardizing code and producing it in a more synthesizable style.
The goal of the SAVE project is to create tools that, in an automatic way, measure the quality of descriptions and help designers to improve them with a set of suggestions.
Using these tools, code that has unacceptable quality is identified early reducing the cost of finding and correcting errors which grows rapidly with the life cycle.
This paper will report how the VQC (Verify Quality Check) prototype, contained in SAVE, evaluates VHDL descriptions quality and helps developers to create better models. Session 2 presents the SAVE project. An analysis of synthesizer results is described in Session 3. In Session 4 an analysis in order to evaluate the readability and the maintainability of VHDL models is reported. Session 5 describes the analysis from the simulation performances point of view.

2: The SAVE Project

The goal of the SAVE project is to find rules and guidelines (theoretical analysis) oriented to the improvement of VHDL model quality as well as to the

implementation of a set of tools to improve readability and maintainability of models, to manage hardware semantics and to reduce the time spent in the simulation phase.

A set of guidelines, to evaluate and possibly increase the quality of synthesis results and simulation speed of VHDL descriptions, have been determined on an experimental basis while, in order to measure and improve code readability and maintainability, existing software complexity metrics have been analyzed to verify their applicability to VHDL models. Since these studies have pointed out that complexity analysis of VHDL source code is a completely unexplored research field and VHDL specific aspects have no direct counterpart in software design, new metrics have been developed.

Moreover, synthesizability analysis allows to show in advance if code can be synthesized verifying its non ambiguous hardware semantics and if it is optimized in order to speed up the synthesis process and to improve the quality of the results achieved. SAVE tools can give the designer some suggestions about a more quickly synthesizable coding style (currently related to synthesizer AutoLogic of Mentor Graphics) or perform some checks to avoid synthesis of bad descriptions. The designer can choose to replace code in an automatic way.

The LVS (Leda VHDL System) supports the parsing step of VHDL source files and also builds an intermediate representation within an object–oriented database according to VIF (VHDL Intermediate Format) specifications. Starting from the results of the parsing step, a custom tool (Preprocessor) builds a new representation more suitable for further processing by exploiting LVS support for user–defined extensions to the basic VHDL schema.

Such an enriched representation collects all data needed for the computation of simulation efficiency, complexity and synthesizability analysis. The designer can choose to evaluate the project in term of these aspects and the analysis is performed activating different tools that compute metrics embedded in the rule base.

A graphical interface module (Presentation Manager) enables the display of the above characteristics by using graphs, tables and diagrams.

3: Synthesizability

VHDL is a language essentially used to perform simulation, but only some of its constructs are accepted by currently available synthesizers. To assure quality results after the synthesis phase and to reduce the time spent in this phase of the project life cycle, a complexity analysis from a hardware point of view has been developed. The main goal of this kind of analysis is to suggest a more efficient coding style in some particular situations or perform some checks to avoid synthesis of non–correct descriptions.

A common situation is the presence of undesired latches: when a signal is not assigned under all possible execution paths through a process, latches will be inserted into the design in order to store the value of the signal during all the unspecified states.

A function that checks the presence of latches has been implemented: for every signal that causes a latch, the designer can choose between adding an assignment and leaving the structure as it is.

For the first choice, two possibilities are provided: assigning the value in the previous branch (or path) or assigning the default value (the value in the initialization).

The assignments are inserted in the source code in an automatic way. For any particular circuit there are many ways to express the function in VHDL. Even if the functionality is the same, two different coding styles can produce very different implementations. Sometimes the most intuitive method for describing a circuit will not necessarily result in the best circuit.

Typical examples are large CASE and IF structures where the same signals are assigned in all the branches with values that changes for few bits. If code is modified so that the only assignments to the bits that are changing are made, the system will run much faster and with less memory. For instance, the following code will generate, with AutoLogic, an intermediate result of 28 generic gates:

```
p1:PROCESS (sel)
     BEGIN
       CASE sel IS
           WHEN ”00” => y <= ”0001”;
           WHEN ”01” => y <= ”0010”;
           WHEN ”10” => y <= ”0100”;
           WHEN ”11” => y <= ”1000”;
       END CASE;
   END PROCESS;
```

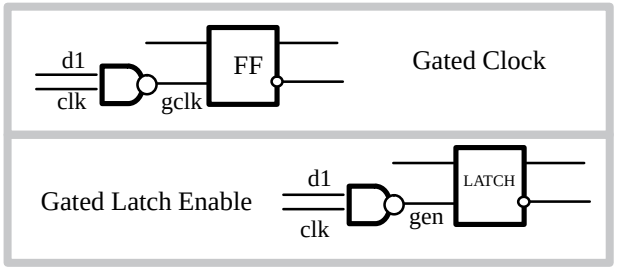
that can be reduced to about 6 gates in the following manner:

```
p1:PROCESS (sel)
     BEGIN
       y <= ”0000”;
       CASE sel IS
           WHEN ”00” => y(0) <= ’1’;
           WHEN ”01” => y(1) <= ’1’;
           WHEN ”10” => y(2) <= ’1’;
           WHEN ”11” => y(3) <= ’1’;
       END CASE;
   END PROCESS;
```

These two different ways of writing CASE statements will have the same result, but the second one does not need logic optimization.

Relating to the quality improvement of the results in terms of chip area, it has been noted that particular care has to be paid in order to assure that a specified code portion produces, for example, the smallest number of gates. Among many experiments made in this field, it is possible to mention the recognition of code portions in which the ’X’ state is used. As AutoLogic does not treat assignments to the ’X’ state as an unknown state, this can lead to some inefficient structures being generated. Choosing the default value for don’t care states can make a big difference during area optimization.

SAVE is also based on the analysis of synthesis guidelines and design rules resulting from Italtel designers expertise. For example, the tool is able to recognize Gated Clock, Gated Latch Enable Signal and Tristate.



SAVE allows to automatically evaluate the VHDL code giving measures and advises about it and the possibility to replace code portions.

Other problems related to efficiency in the synthesis phase are:

– the use of unconstrained integers that would be implemented as 32 bit numbers; if all the 32 bits are not required, they would be pruned from the circuit during logic optimization with a certain waste of time. The preferred method is to constrain all integers;

– the use of LOOP statements.

Another situation considered by the analyzer is the verification of correctness in component declarations: in fact, some compilers do not verify that the mode, type and order exactly correspond to the entity declaration of the component. This situation can cause very serious problems during the synthesis phase that are very difficult to find out. All these checks are notified by the analyzer through a textual report.

4: Complexity

The relevance of complexity analysis is particularly related to the problem of maintenance of source code which involves different aspects such as modifiability, reusability, readability, and so on.

The SAVE project investigates existing complexity metrics and adapts them in order to satisfy VHDL requirements.

In the following, metrics measuring the complexity of sequential statements and guidelines making source code more readable will be introduced.

4.1: Maintainability Analysis

The first kind of metrics applied represents a combination of three suitably adapted traditional metrics : Mc Cabe’s cyclomatic number [Hen92], nesting level and information flow. They measure the statements complexity (i.e. low maintainability, low testability ...). These metrics are applied to single VHDL sequential blocks (processes, functions and procedures) and then the complexity of the whole VHDL description is estimated with some cost functions.

Mc Cabe’s cyclomatic number is one of the most popular metrics in Software Engineering [McCab76]. It grows with the program complexity and can be considered a measure of maintainability and testability.

The nesting level metric represents the maximum depth of nested statements, that is each statement contained in another one. A large nesting level is often an indication of bad programming style: subprograms can help to break code into manageable parts.

Information flow has been adapted to VHDL

communication mechanism and measures the information exchange between processes. A high value of this measure can indicate confused processes without a well defined functionality. The definition studied is :

$$IF = a_1 \cdot Nsgl + a_2 \cdot Nsgs$$

where *Nsgl* and *Nsgs* stand for the number of signals read and the number of signals written respectively with $a_1 < a_2$ (it has been given more importance to signals written because they wake up all processes sensitive to them).
The complexity evaluation of a process has been calculated by the following formula:

$$C_{Process} = C\_Size \cdot (k_1 \cdot C\_Cabe + k_2 \cdot C\_Nest + k_3 \cdot C\_IF)$$

where coefficients *C_Cabe*, *C_Nest* and *C_IF* depend on evaluation of cyclomatic number, nesting level and information flow respectively and *C_Size* depends on dimension of the process measured in non–comment source lines of code. A possible argument that *C_Size* should be a significant factor, rests on theory that, since the developer has more code to understand, maintaining it would be more difficult.
The average on complexity values of the single processes determines the description evaluation. The results are presented in a histogram which represent each process with its evaluation, making easy and quick to localize any process difficult to maintain.
Some checks have been made in order to report suggestions for a good programming style oriented to maintainability, although they did not influence the evaluation.

4.2: Readability Analysis

Another approach to complexity analysis consists in creating a readable and easily modifiable code. In fact designers coding style determines the readability of their descriptions, therefore they can improve it following some general guidelines.
A standard form of the code could increase its readability: only one statement per line, uniform indentation and uniform casing for VHDL reserved words are verified and automatically applied in the SAVE project. If the header does not exist or does not complain to a standard template, it is possible to automatically insert all the fields: Title, Engineer, Company, Project, Filename, Purpose, Simulator, Synthesis, Revision and optionally Limitations and Note. If some of these fields are not present, the tool inserts the informations available from the VHDL code file as default (Engineer and Filename) and asks for the others to the user.
It is possible to distinguish between information comments and separator ones. Information is inversely proportional to the repetition of a set of alphanumeric characters (the maximum amount of information per symbol is provided by an alphabet whose symbols occur with an equal probability [Har92]). So a regular composition of repeated characters (low information) is named ”separator” while in all the other cases the line is properly named ”comment”.
Others checks are carried out on the identifiers used for files, signals and variables. Some of these informations, for example using of a suffix related to the dimension or to the ports and signal goal (vectors: ’ v_ ’, tristate: ’ z_ ’, present state: ’ ac ’, future state: ’ px ’), are also utilized in the synthesis analysis to verify the correspondence between the designers specified target and the actual hardware implementation.
Another kind of control is related to the name of the architecture: it should have a suffix which represents the kind of modeling style used (BEHAVIORAL, STRUCTURAL, DATAFLOW).
Furthermore, there is the possibility to check if types, procedures, functions and components are defined into the VHDL file. Because it should be better to avoid this programming style, a suggestion is given in order to separate this portion of code making a package.
To measure all code attributes, the basic idea is to divide a description in single modules (entity and architecture declarations, blocks, processes, functions and procedures) and then evaluate the readability degree of each one on a scale of values obtained by heuristic methods. So the following formula can be used to evaluate the entity readability:

$$R_{Entity} = k_1 \cdot DPC + k_2 \cdot CLMI + k_3 \cdot CQNP + k_4 \cdot DGC$$

where DPC is the density of the commented ports in the source code, CLMI is a coefficient that represents identifiers average length, CQNP is related to the quality of the name of the identifiers (e.g. an IN port should be named with identifiers like *port_IN*) and SGC is the density of

commented generics. The weights $k_1 ... k_4$ and the coefficients above are empirically obtained.
The measure of readability of blocks, processes, functions, procedures can be calculated with the following:

$$R_{Module} = t_1 \cdot CS + t_2 \cdot CLMI + t_3 \cdot CL + t_4 \cdot DCI + t_5 \cdot DVS + t_6 \cdot CHEAD$$

where CS depends on the presence and the quality of comments with the functionality to separate, in a visual way, the module from other parts of code; CLMI is the coefficient of the identifiers average length while CL is related to the presence of a label of the module; DCI and DVS represent the density of inside comments and the density of the commented variables and signals respectively. Finally CHEAD is a coefficient related to a possible header comment at the beginning of the module. The weights $t_1 ... t_6$ and coefficients are calculated in a heuristic way and they have different values if the module is a block, a process, a function or a procedure.
Moreover the architecture declaration is evaluated with the following formula:

$$R_{Arch\_Decl} = h_1 \cdot CLMI + h_2 \cdot CL + h_3 \cdot DVS$$

Finally the readability of the whole project is calculated as follows:

$$R_{TOT} = w_1 \cdot R_{Entity} + w_2 \cdot (1 / N) \cdot \Sigma_{i=1...N} R_{Module\_i} + w_3 \cdot R_{Arch\_Decl}$$

where N is the modules number.
The result of the application of these kind of metrics is not only a degree of readability but also several guidelines that help the designer to improve the project.

5: Efficiency

The purpose of this kind of analysis is to find guidelines in order to increase the simulation speed of VHDL descriptions. The improvements obtained following the proposed guidelines are still useful (although at a lower degree) even if new generation simulators are available in the market. Such guidelines have been discovered on an experimental basis making several tests on various commercial simulators such as Vantage of ViewLogic, QuickSim II and QuickVhdl of Mentor Graphics and searching for constructs which are semantically equivalent but with different simulation performances.
It has been noted the higher efficiency of sequential VHDL descriptions versus concurrent ones: so merging together processes with the same sensitivity list, replacing signals with variables whenever possible, joining together processes sharing one signal as communication channel to get rid of it and so on, are some of the guidelines suggested. Tests have shown that static sensitivity lists are more efficient than dynamic ones at the bottom of the process and without conditions. Moreover, in order to create fast code, it is suggested to the designer to avoid resolved signals whenever possible, limit the use of attributes returning signals, reduce the number of functions, procedures and generics and so on.
Besides, if in some examples with QuickSim II and Vantage simulators applying suggestions has led to a gain up to 50% in simulation time, the gain obtained with the more efficient QuickVhdl is not so evident. For more detailed information about this task see [Bal94].

6: Conclusions

The goal of the SAVE project is the definition and the implementation of a set of tools to support designers to improve the quality of VHDL descriptions in the synthesizability, complexity and efficiency area.
It should be pointed out that quality attributes have different weights according to different project targets. The presence of historical archives could help in determining more precisely these weights. Even if some attributes may be in contrast with others, for example using procedures improves readability but makes simulation performances worse, in future developments a trade–off analysis will be welcome in order to support a full integration of the different analyzers managing conflicting goals.
SAVE produces not only a numerical quality measure but mostly gives to the designer a set of textual and graphical suggestions to improve his knowledge and the description quality. Currently, Italtel designers utilize SAVE not to obtain a mere hardware designers ability evaluation, which is often unappreciated, but to learn to write good quality code.

References

[Bal94] A.Balboni, M.Mastretti, M.Stefanoni : ” Static Analysis for VHDL model Evaluation ”, EURO VHDL, 1994 IEEE

[Bon92] A.Bonomo, P.Garino, G. Ghigo, A. Balboni, M.Mastretti ” VHDL optimisation techniques for coding and simulation ”, Rapporto Tecnico CSELT

[Cha93] S.Cha, I.S.Chung, Y.R.Kwon, ” Complexity measures for concurrent programs based on information–theoretic metrics ”

[Fen91] N.E. Fenton ” Software metrics : a rigorous approach ”, Chapman & Hall 1991

[Fen94] N.Fenton ” Software Measurement: A necessary Software Scientific Basis ”,IEEE Transaction on Software Engineering,vol. 20,no. 3, march 1994

[Gan86] J.D. Gannon, E. Katz, V Basili, ” Metrics for ADA packages: an initial study ”, Communication of the ACM – July 1986

[Gill91] G.K.Gill C.F.Kemerer ” Cyclomatic Complexity Density and Software Maintenance Productivity ”, IEEE Transaction on Software Engineering, vol.17 n.12,december 1991

[Har 92] W. Harrison ” An Entropy Based Measure of Software Complexity ”, IEEE Transaction on Software Engineering, vol. 18, n. 11, november 1992

[Hue91] M.Hueber ” VHDL experiments on Performance ” Euro–VHDL 1991

[Hen92] B. Henderson Sellers, ” Modularization and McCabe’s cyclomatic complexity ”, Communication of the ACM 1992

[Kit90a] B.Kitchenham, L. Pickard, S. Linkman ” An evaluation of some design metrics ”, Software Engineering Journal, january 1990

[Kit90b] B.A. Kitchenham, S.J. Linkman ” Design metrics in practice ”, in Information and software technology 1990

[Lev91] O. Levia ” Writing High Performance VHDL Models ”, Euro–VHDL 1991

[McCab76] T.J.McCabe, ” A complexity measure ”, IEEE Trans. Software Engineering, vol. SE–2, pp. 308–320, 1976.

[Mid90] S.Midkiff , D.Padua ” Issues in the optimization of parallel programs ”, International Conference on Parallel Processing, 1990

[Oma92] P.Oman, J.Hagemeister ” Metrics for assessing a Software system’s maintenability ”, IEEE Transaction on software engineering, 1992

[Ott92] L.Ott, J.Bieman ” Effects of software changes on module cohesion ”, IEEE Transaction on software engineering, 1992

[Pau91] B.Paulsen O.Levia ” Techniques for Writing High Performance and High Quality VHDL Models ”, Euro VHDL 1992

[Ram85] J.Ramamoorthy, W.Tsai, T. Yamaura, A.Bhide ”Metrics guided methodology”, Proc. 9th Computer Software and Application Conf. Oct.1985 p 11

[Rob91] P.N.Robillard, D.Coupal, F.Coallier ” Profiling Software Through the Use of Metrics ”, Software–Practice and Experience, Vol. 21(5), 1991

[Sch92] N.F.Schneidewind ” Methodology For Validating Software Metrics ”,IEEE Transaction on Software Engineering,vol. 18,n. 5,may 1992

[Sha88] S.Shatz ” Towards Complexity Metrics for ADA tasking ”, IEEE Transaction on software engineering, august 1988

[She90] M. Shepperd ” Design metrics : an empirical analysis ”, Software Engineering Journal, january 1990

[Wood81] S.N.Woodfield, H.E.Dunsmore, V.Y. Shen ” The effect of modularization and comments on program comprehension ”, 5th International Conference on Software Engineering March 1981