Static Analysis of VHDL Code: Simulation Efficiency And Complexity

Mirella Mastrctti
ITALTEL-SIT
Central Research Labs
Settimi Milanese (MI) – ITALY

Maurizio Sturlesi, Sergio Tomasello
Universita’ Degli Studi di Milano
Computer Science Dept. – Milano – ITALY

Abstract

Automatic VHDL static analysis can be a valuable approach to develop, measure and compare models generated by hardware designers or by high-level specification tools. Moreover VHDL code should be developed according to some well founded guidelines to improve the quality of the overall design process.

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 to manage hardware semantics.

SAVE activities include theoretical analysis tasks as well as practical experimentation and implementation of prototypes tools.

Furthermore, some high-level design tools are able to generate VHDL source code in an automated way, but in some cases the resulting code may be too large and complex for the human reader. Therefore, automated source code analysis may be a valuable approach to develop, measure and compare models managed by all the above methodologies and tools in order to assure the correctness of VHDL descriptions before adding them to model libraries.

If enhanced with advising capabilities, a static analyzer may also assist the user in the challenging task of introducing significant modifications and improvements into source code.

So, it becomes very useful to integrate the development process with some quality assessment activities in order to assure that code is developed according to some well-founded guidelines.

Introduction

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. Moreover in the typical VHDL-based environment no support is given in order to cope with the increasing complexity of VHDL descriptions and the widespread demand for their quality evaluation and improvement.

In Software Engineering literature the term quality means the degree to which software possesses a desired combination of attributes.

A distinction between internal and external attributes can be made:

– Internal attributes of a model are those which are related only to the features of the model itself. For example the number of possible execution paths, structure, size, number of comments are some examples of
internal attributes.

External attributes of a model are those which depend on how the model relates to its environment. Examples of external attributes are readability, maintainability (that are strictly related to the development team) and simulation performances (that depend on the particular simulation tool).

The presence of an attribute in a product can be measured with the application of some metrics. A metric is a function whose inputs are code data (i.e. elementary software measurements such as the number of edges and number of nodes in a directed graph that represents the program control flow) and whose output is a single numerical value that can be interpreted as the degree to which product possesses a given attribute.

External attributes tend to be the ones that managers would most like to evaluate and predict in order to know the cost-effectiveness of some process or the productivity of their personnel.

Unfortunately external attributes, by their very nature, cannot be measured directly as internal ones: for instance, maintainability costs depend on different factors as number of errors or designers experience and so on, while the model size is evaluated simply counting lines of instruction code.

Nevertheless, there is a wide consensus that good external quality depends on good internal structure.

Therefore VHDL model quality features like high simulation performances, low maintainability costs and good synthesis results can be estimated by measure of some internal attributes inside descriptions.

This is the approach followed in the SAVE project, which goal is to create tools that, in an automatic way, measure the quality of descriptions and help the designers to improve them with a set of suggestions.

These tools permit to reduce time costs of quality controls, help designers to create standard and easily modifiable descriptions and to improve their ability to create models. It is important that designers use these tools during the development process as they use tools as compilers, simulators and so on. Actually code that has unacceptable quality must be identified sufficiently early because the cost of finding and correcting errors grows rapidly with the life cycle.

This paper will report how the SAVE tools evaluate VHDL descriptions quality and help the developers to create better models: in particular it will focus on simulation efficiency and complexity analysis. Section 1 presents the SAVE project architecture. Section 2 describes the analysis from the simulation performances point of view. In Section 3 and in Section 4 analyses on the internal attributes of VHDL models and a FSM consistency verification are presented respectively.

I. The SAVE project architecture.

The SAVE project includes theoretical analysis tasks as well as more concrete activities including implementation of prototype tools in order to reduce the time spent in the simulation phase, to improve readability and maintainability of models and to manage hardware semantics.

An architectural scheme of the prototype environment is depicted in Figure 1. The LVS (Leda VHDL System) support the parsing step of VHDL source files, including semantic analysis concerning the standard language definition. LVS is also able to build 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 the 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 and complexity analysis.

The designer can choose to evaluate the project in term of these aspects and the analyzes are performed activating different tools that compute the 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.

2. Simulation efficiency

The goal of this kind of analysis is to find rules and guidelines in order to evaluate and possibly increase the simulation speed of VHDL descriptions.

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.

The purpose of these experiments was to search for constructs that are semantically equivalent but with different simulation performances.

It has been noted, in [Bal94], 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, replacing nesting of guarded blocks with equivalent sequential processes in which the guarded condition is tested in a WAIT ON – UNTIL, joining together processes sharing one signal as communication channel to get rid of it, 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 a fast code, it is suggested to the designer to avoid resolved signals whenever possible and to limit the use of attributes returning signals. Subprograms are very costly.

In figure 2 code written using procedure calls is compared with code with utilizing
unrolled "inline" procedures. Sometimes function subprograms can be replaced with equivalent procedures. Actually it is sufficient to declare a procedure with an OUT parameter as a return value. It has been noted, with various tests on QuickVhdl simulator, that functions that return an integer value are not less efficient than procedures, but function subprograms that return an integer array are less efficient than equivalent procedures with an OUT integer array parameter (Fig. 3). Moreover limiting the use of loop statements can improve simulation performance.

The cost of package size was investigated by simulating the same description using packages of different size but no performance improvements have been noted. Similar tests have been carried out about overloading: overloaded function calls have been compared with the same functions renamed and also in this case the simulation times didn’t change.

3. Complexity analysis

Internal attributes of descriptions such as structural complexity and coding style can determine low maintainability costs. The term 'structural complexity' derives from the field of software engineering and it is related to the facility to read, understand and possibly modify source code. A significant effort has been devoted in the past in order to provide some kind of estimation for software internal attributes. However, it should be pointed out that the goal of defining suitable complexity measures for hardware description languages such as VHDL introduces specific aspects that may have no direct counterpart in the more assessed field of software design (mixed behavioral/structural paradigms, event-driven behavior,...).

So, the result of this study is the application of existing software metrics which have been chosen and modified to adapt them to the particular nature of VHDL.

In the following two classes of metrics are described: initially metrics that measure the complexity of sequential statements will be introduced, followed by metrics related to guidelines to make source code more readable.

The first kind of metrics applied represents a combination of three suitably adapted traditional metrics: Mc Cabe's cyclomatic number, nesting level and information flow. These metrics are applied to single processes and then the complexity of the whole VHDL description is estimated with some cost functions.
The Mc Cabe's cyclomatic number is one of the most famous metrics in Software Engineering [Cab76]. If \( G \) is a directed graph representing the control flow of a sequential program (e.g. code within VHDL process) the cyclomatic number \( V(G) \) can be obtained with the following:

\[
V(G) = e - n + 2
\]

where \( e \) is the number of edges and \( n \) is the number of nodes.

McCabe showed that \( V(G) \) is also equal to the number of binary decision nodes in \( G \) plus one, equivalent to the maximum number of the linearly independent circuits in a program control graph. Therefore the cyclomatic number grows with the program complexity and can be considered a measure related to the maintainability and testability.

The nesting level metric represents the maximum depth of nested statement, 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 the code into manageable parts.

Information flow has been adapted to VHDL communication way. Its goal, in this context, is to measure the information exchange between processes. A high value of this measure can indicate confused processes without a well defined functionality.

The definition studied is the following

\[
\text{Inf. Flow} = a_1 \cdot R_{sgn} + a_2 \cdot W_{sgn}
\]

where \( R_{sgn} \) and \( W_{sgn} \) stand for the number of signals read and the number of signals written.

Goal of the weights \( a_1 \) and \( a_2 \) is to give a different importance to signals written and read.

In this case it is in force the relation \( a_1 < a_2 \) (it has been given more importance to signals written because they wake up all processes that are sensitive to them).

The complexity evaluation of a process has been calculated by the following formula:

\[
C_{PF} = k_1 \cdot C_{Ch} + k_2 \cdot C_{Nt} + k_3 \cdot C_{IF} + k_4 \cdot C_{Sz}
\]

where coefficients \( C_{Ch}, C_{Nt} \) and \( C_{IF} \) depend on evaluation of cyclomatic number, nesting level and information flow respectively and \( C_{Sz} \) depends on dimension of the process measured in non-comment lines of source code. A possible argument that \( C_{Sz} \) should be a significant factor rests on theory that since the developer has more code to understand, maintaining it would be more difficult.

In order to link the value \( C_{PF} \) to maintainability costs of projects in a particular design team environment, the weights \( k_1, \ldots, k_4 \) can be chosen by the user according to previous projects data.

The evaluation of the whole description is obtained making the average on complexity values of the single processes.

The results are presented in a graphic way through a histogram that represents every process with its evaluation.

To reduce maintainance costs designers should try to reduce the number of errors in the product but also create a readable and easily modifiable code; so the second kind of metrics are related to VHDL coding style: it is how designers use VHDL constructs that determines the readability of their descriptions; they can improve it following some general guidelines.

To give an indicative evaluation of the coding style, a description is divided in single modules and then the readability degree of each one is evaluated on a scale of values obtained by heuristic methods.

The modules are entity and architecture declarations, blocks, processes, functions and procedures.

The descriptions are evaluated through the application of some metrics that express how many guidelines have been followed.

Some of these guidelines verified by the analyzer are:

- Ports, generics, signals and variables should be declared one per line with a brief
comment, on the same line, explaining its functionality.

- Little prefix or suffix could be useful to express port directions (e.g. name_port_IN, name_portIN or IN_name_port and so on..).

- Identifiers names should be not too short neither too long (e.g. from 3 to 16 characters could be a good range); on the one hand, names should express objects features; on the other, names too long could create problems with indentation.

- It might be useful to separate modules from the other code portions, in a visual way, by the use of some empty lines or lines of characters as "--".

- At the beginning of a module a brief header comment could be used to specify the behavior of the module itself.

- Processes, concurrent signal assignment statements, concurrent procedure calls, etc. could be named in a descriptive way with the optional label.

- Code within modules should be suitably commented.

- Lines length should not exceed 80 characters and it should be better to put only one statement per line.

- File names should be similar to entity ones: this simplifies the research of instantiated components.

---

![Image](image.jpg)

**Fig.4** The SAVE tools interface
4. FSM analysis

A tool has been made to analyze VHDL Finite State Machines in order to give the designer the ability to verify the conformity between FSMs specifications and their implementations.

Results of this analysis are diagrams of the modelled FSMs and a report about the conditions that control state transitions: if conditions that can not become true are found, they are displayed in dashed lines and possible trap states or unreachable states are highlight [Bal94].

These checks are performed by a static analysis, so only conditions not related to timing are considered.

To resolve problems related to static analysis approach limitations (e.g. signals can not be evaluated statically!) a post simulation analysis is provided.

After state recognition, through the analysis of log-file generated during the simulation, is possible to detect the states reached and the transitions covered.

This tool permits to substitute huge and incomprehensible simulation log file with simple textual report or graphical representation of the state diagram, helping in this way to create a complete and comprehensive documentation (Fig. 4).

5. Conclusions

Goal of the SAVE project is the definition and the implementation of a set of tools to
support the designers to improve the quality of VHDL descriptions in the complexity, efficiency and synthesizability area. It should be pointed out that quality attributes have different weights according to different project goals. In fact synthesizability properties are not so important if the model must be solely simulated. Moreover some attributes may be in contrast each other: for example using procedures improves readability but make simulation performances worse. However, speed bottlenecks are usually restricted to small specific section of the whole program: for example it is important to improve performance of processes with a large sensitivity list that are resumed by a large number of events. Therefore in future developments a tradeoff analysis will be welcome in order to support a full integration of the different analyzers managing conflicting goals.

References

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

[Bon92]
A.Bonono, P.Garino, G. Ghigo, A. Balboni, M.Mastretti "VHDL optimization techniques for coding and simulation" Rapporto Tecnico CSET

[Cab76]

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

[Fen91]

[Fen94]

[Gan86]

[Gil91]

[Hen92]
B. Henderson Sellers, ”Modularization and McCabe's cyclomatic complexity”. Communication of the ACM1992

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

[Kit90]

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

[Lin90]
B.A. Kitchenham, S.J. Linkman "Design metrics in practice", in Informationand software technology 1990

[Mid90]

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

[Rob91]

[Sch92]

[Sha88]

[She90]

[Wood81]