11

Static Analysis of VHDL Source Code: the SAVE Project

M. Mastretti
ITALTEL–SIT– Central Research Labs
Settimo Milanese (MI) – ITALY
phone: +39.2 43888582
fax: +39.2 43888593
e–mail: mastrett@settimo.italtel.it

M. L. Busi, R. Sarvello,
M. Sturlesi, S. Tomasello
Universita' degli Studi di Milano – Computer Science Dept.
Milano – ITALY
e–mail:
busi@ghost.sm.dsi.unimi.it, sarvello@ghost.sm.dsi.unimi.it
sturlesi@ghost.sm.dsi.unimi.it, tomasell@ghost.sm.dsi.unimi.it

Abstract

VHDL (Very High Speed Integrated Circuits Hardware Description Language) is one of the most popular languages (IEEE standard) for building software models of hardware systems. While the typical VHDL–based design environment provides tools for code simulation and logic synthesis, 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.

Automated source code analysis is a valuable approach to develop, measure and compare models in order to assure the satisfaction of quality requirements of VHDL descriptions before adding them to model libraries. Therefore a static analyzer may assist the user in the challenging task of introducing significant modifications and improvements into source code so that, assuring that VHDL code is developed according to some well–founded guidelines, a relevant impact on the quality of the overall design process may be achieved.

The goal of this paper is to summarize the activities carried out within the SAVE project, leading to the development of a collection of quality analysis tools in order to improve modifiability, reusability, readability of models reducing the VHDL descriptions complexity.

S. Bologna et al. (eds.), *Achieving Quality in Software*

Keywords
complexity metrics, quality, CAD/CASE environment, static analysis, hardware design flow.

1 INTRODUCTION

The design methodology for hardware systems (in particular integrated circuits) is migrated from interactive capture of electrical schematics to software modeling. This is the reason why software engineering techniques begin to become strategic also within hardware design centers.

The VHSIC (Very High Speed Integrated Circuits) Hardware Description Language is an industry standard language (IEEE 1076) used to describe hardware from the abstract to the concrete level. Computer–aided engineering workstation vendors throughout the industry are standardizing on VHDL as input and output for their tools which include simulation and automatic synthesis (Perry).

VHDL allows to describe the functionalities of design components (like memory cells, chips, logic ports and so on) in an algorithmic way without assumptions about the technology of a device or the methodology used to design it and, by the synthesis phase, it is possible.

It is based on ADA language principles and it supports some characteristics such as information hiding, components instantiation (like objects instantiation) and the possibility of declaring new data structures and functions enclosed in a package. Currently some committees are trying to extend it towards an object oriented methodology. Moreover, its concurrent computational paradigm is based on processes and signals concepts.

VHDL simulation is the modeling and behavior analysis of an electronic design in order to verify the design functionality. This type of analysis, when performed before the logic design phase, ensures design quality earlier in the engineering process, where errors are easier and cheaper to fix. With a simulator it is possible to analyze the design on a test bench, with stimulus, probes, and waveform displays. With a top–down design method employing VHDL synthesis, functional changes can be made rapidly and verified through simulation.

Synthesis is the process of automatically generating a logic level representation, i.e. the traditional circuit representation where basic components (logic gates, flip–flops, etc.) are connected together with wires and buses, from an algorithmic description. According to different constraints introduced in the VHDL description, the synthesis process can produce various implementation alternatives. A typical design flow is represented in Figure 1.

While the typical VHDL–based design environment provides tools for code simulation and logic synthesis , 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. Furthermore, some high–level design tools (similar to CASE environments) 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 is a valuable approach to develop, measure and compare models managed by all the above methodologies and tools in order to satisfy quality requirements 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 into source code to improve simulation performances to make project maintainability easier and to create an efficient link with hardware synthesis results.

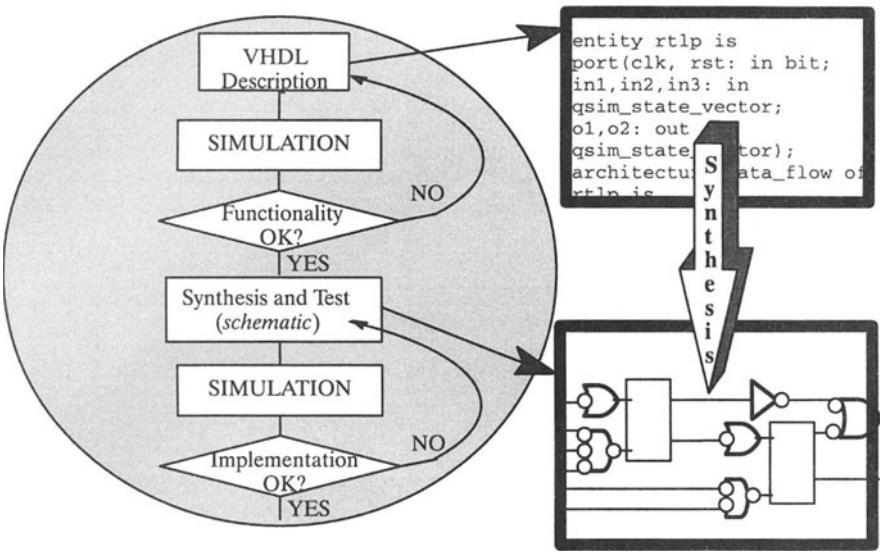

Figure 1 Design Flow.

This paper describes the activities carried out within the SAVE project. Starting from software engineering principles, SAVE consists in some tools which assist designers in writing high quality VHDL code, that means easily readable, modifiable and reusable code.

Qualitative analysis of VHDL source code, in fact, is a completely unexplored research field and results from software engineering can provide some interesting suggestions that need to be extended to some particular aspects strictly related to hardware design concepts (simulation and synthesis). Because of the intrinsic complexity of the problem, heuristic techniques (giving approximate results) look very promising.

If code is written following some guidelines enclosed in the SAVE expert system, the time spent in functional verification can be reduced. Furthermore synthesis evaluation shows in advance if code can be synthesized and if it is optimized to make synthesis more efficient.

These tools give additional advantages like advising designers about models not conforming to corporate standards and the possibility of improving code style and project documentation.

From the theoretical point of view, existing software metrics have been analyzed in order to apply the most suitable ones to a single VHDL module (process, procedure, function) and develop higher level metrics based on cost functions. Moreover, some new metrics and guidelines have been discovered on experimental basis.

The goal of the approach followed in the SAVE project is to produce not only a numerical quality measure, but mostly give to the designer a set of textual and graphical suggestions to improve the description quality itself.

2 SAVE PROJECT ANALYSIS

2.1 VHDL code complexity analysis

In Software Engineering literature the term *quality* means the degree to which software satifies a selected combination of attributes. A distinction between internal and external attributes can be made like in (Fenton, 1991): internal attributes are related only to the features of the model itself (number of execution paths, data size, etc.); external attributes depend on how the model relates to its environment (readability, maintainability, etc.).

In quality evaluation, external attributes tend to be the ones that managers would most like to evaluate and predict in order to estabilish the cost–effectiveness of some processes or the productivity of their personnel.

Unfortunately, by their own nature, external attributes cannot be measured as directly as internal ones: for instance, maintainability costs depend on different factors like number of errors or designer expertise 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.

For example, (Robillard, 1991) shows an interesting method for evaluating software based on metrics. Users may define a range of values for each metric. The method consists in the collection of metric measurements, determining the metrics distribution for each module of the project. The groupings of the various metrics constitute quality factors like testability, that can be evaluated

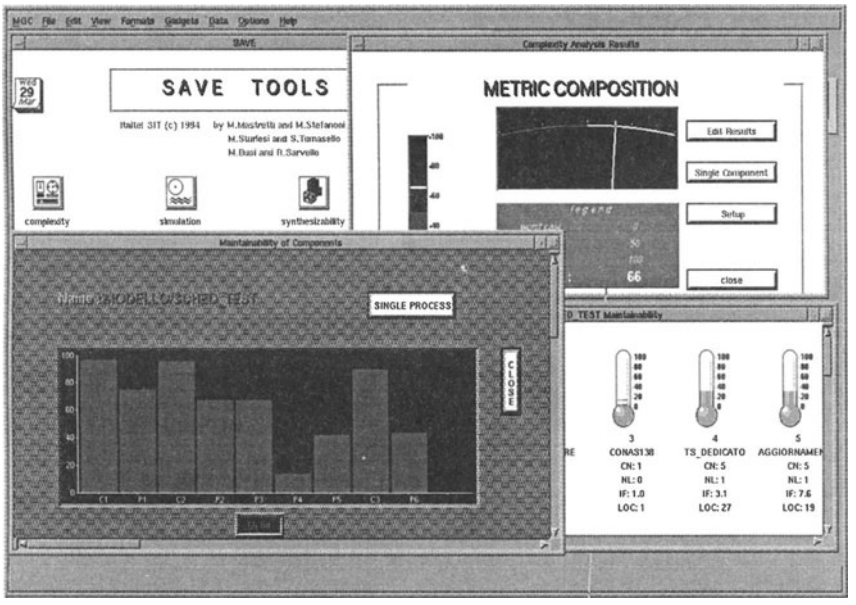

Figure 2 Example of some SAVE tools.

using some metrics (number of independent paths, number of loops, etc.) within the given ranges. The percentage of modules meeting the metrics ranges represents a good indication of this quality factor. The resulting profile provides a quick visualization of the overall project. High level management can easily interpret such a profile.

Therefore, VHDL model features like low maintainability costs, high simulation performances and good synthesis results can be estimated measuring some internal attributes inside descriptions and respective metrics can evaluate the presence of these indicators. However, it should be pointed out that defining suitable complexity measures for hardware description languages such as VHDL, involves specific aspects which may have no direct counterpart in the more assessed field of software design (mixed behavioral/structural paradigms, event–driven behavior, ...).

While static analysis techniques for concurrent programs are emerging (Gannon, 1986), (Ramamoorthy, 1985), (Shatz, 1988), (Cha, 1993), several theoretical and practical issues still make VHDL analysis a very complex task. In fact the concurrent model of VHDL is different at a large degree from the ADA model (from which many VHDL language constructs have been derived).

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.

Some metrics represent a combination of some adapted traditional ones like Mc Cabe' s cyclomatic number (McCabe, 1976), nesting level and information flow. Because of VHDL relevant complexity, a main problem is to determine the most suitable metrics.

Finding only a single evaluation standard may not be a correct approach; in fact VHDL language, with respect to other programming languages, provides many different description styles (imperative, data–flow and structural).

The SAVE solution is to compose different metrics by means of a weighted sum, adjustable by the user, following the team typical design style. Users can easily change metrics coefficients to obtain a custom evaluation of the project: different profiles will result mainly based, for example, on nesting level and cyclomatic number or process size and information flow (Figure 3).

Of course, the traditional metrics have undergone adaptations for the particular language to analyze. For example, information flow has been adapted to the VHDL communication mechanism. Its goal, in this context, is to measure the information exchange between processes. A high value in this measure can indicate a non optimal process partitioning without a well defined functionality. The definition adopted is the following:

$$Inf\_Flow = a_1 \cdot R_{sgn} + a_2 \cdot W_{sgn}$$

where R_{sgn} and W_{sgn} stand for the number of signals read and the number of signals written respectively with $a_1 < a_2$ (more importance has been given to written signals as they wake up all processes sensitive to them).

A further metric adopted in VHDL code complexity evaluation is the concurrency level. In a VHDL description, concurrent processes have a sensitivity list, i.e. a list of signals whose changes activate processes; the concurrency level is defined as the number of times signals in a sensitivity list are written in other processes. This metric gives an estimation of the number of times a process is activated. This measure is only an estimation, because in the SAVE approach the analysis is made by a static analyzer, while the actual value can be obtained only by dynamic analysis.

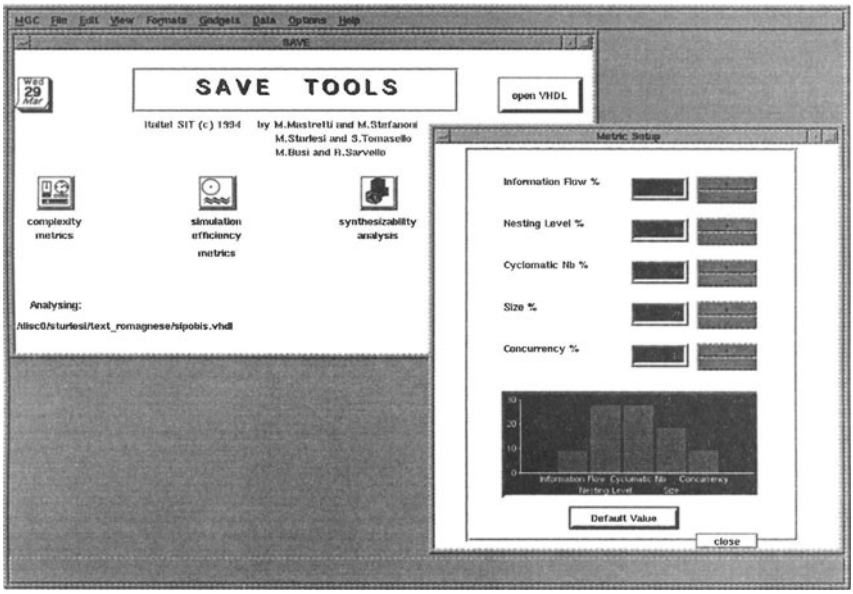

Figure 3 Users can easily change metrics coefficients to obtain a custom evaluation.

The global evaluation of the whole VHDL description is then obtained by averaging the complexity values of single processes. The results are presented graphically through a histogram that represents each process with its evaluation.

It is very difficult to determine the relation between VHDL code complexity and maintainability effort, because of the lack of historical archives for this kind of software.

To reduce maintenance costs, designers should try not only to decrease the number of errors in the design but also to create a readable and easily modifiable source code; so another kind of metrics are related to VHDL coding style, i.e. how designers use VHDL statements to achieve readability in their descriptions. They can improve code by following some general guidelines.

In order to give a general evaluation of the coding style, a description is divided into single modules and the readability degree of each one is measured on a value scale obtained by heuristic methods. The descriptions are evaluated through the application of metrics expressing the number of guidelines followed.

This kind of analysis is twofold: first of all it gives an evaluation (from 0 to 100) of the metrics composition discussed above; second it permits to evaluate readability of the VHDL source code giving some useful guidelines regarding identifier names, lack of comments, module separators and so on and obtaining a rank (from 0 to 100) according to the followed guidelines. The output of these analysis is both textual and graphical (Figure 1), enabling the designer to focus own work on critical modules.

A standard format of the source code assumes a strong relevance in an industrial context because of the need for maintaining control on a hardware design in spite of designers' turn–over.

This approach could increase the VHDL description readability.

If the header does not exist or does not complain to a standard template, it is possible to interactively insert the missing 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 automatically inserts the informations available from the VHDL file as default and asks for the others to the user. Only one statement per line, separators and comments presence, uniform indentation and uniform casing for VHDL reserved words are verified and automatically applied.

Additional algorithms under development include the extraction of particular code sections translatable into procedures, by means of pattern recognition techniques.

2.2 Synthesis and simulation efficiency analysis

A set of guidelines to possibly increase the simulation speed of VHDL descriptions, has been discovered on an experimental basis (with tests on various commercial simulators such as Vantage of ViewLogic, QuickSim II and QuickVhdl of Mentor Graphics) searching for constructs which are semantically equivalent but with different simulation performances. One of such guidelines advises that replacing many concurrent processes with a sequential one loads to a compression of the simulation time up to 20% (Stefanoni, 1994). These results agree with (Hueber, 1991) and partially with (Levia, 1991).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, to limit the use of attributes returning signals, to reduce the number of functions, procedures and generics and so on. Besides if in some examples with simulators QuickSim II and Vantage, applying suggestions has led to a gain up to 50% in simulation time (Figure 4) while the gains obtained with the more efficient QuickVhdl are not so evident.

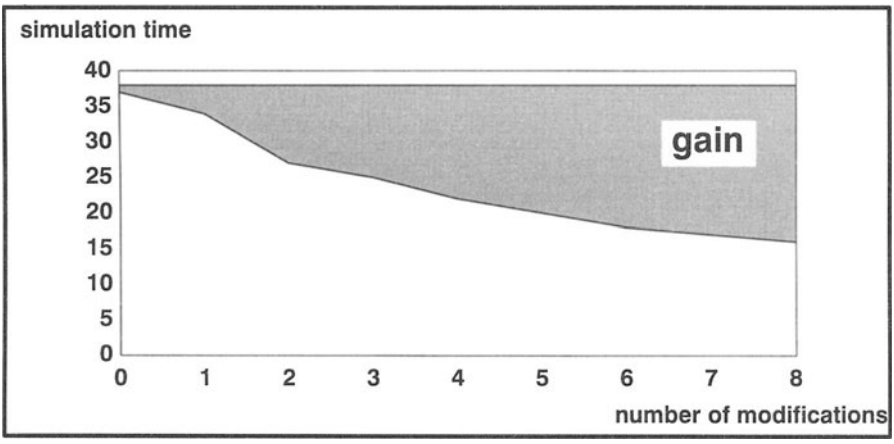

Figure 4 Benefits achieved following the SAVE suggestions.

Moreover, an analysis from the synthesizer point of view, has been developed in order to reduce the time spent in the synthesis phase and to guarantee better results. SAVE tools can also give

the designer some suggestions about a more quickly synthesizable coding style (currently related to AutoLogic synthesizer of Mentor Graphics) or perform some checks to avoid synthesis of bad descriptions. All the warnings obtained from these tools suggest some code replacements that designers can choose to apply in an automatic way.

An example of the synthesizability guidelines is large CASE structures where the same signals are assigned in all the branches changing only for few bits. If code is written so that the only assignments to the changing bits are made, the synthesis tool will run much faster and with less memory. For instance, the following code will generate, with the synthesis tool, 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 way:

```
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.

The SAVE usefulness is to advise designers of this (and many others) situations before the synthesis phase saving a lot of time; synthesis takes a long time and SAVE can predict some results letting designers to modify code (sometimes automatically) avoiding unwanted synthesis results.

In a structured design, SAVE is also able to recognize Gated Clock and Gated Latch Enabled Signals (Figure 5) and this capability is extended recoursively in the overall design structure. First of all SAVE recognizes all the clock implemented in the design, then there is an automatic verification about the existence of assignment of the clock value to another clock (Gated Clock) or to a signal (Gated Latch Enabled Signals).

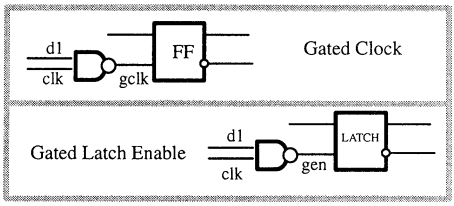

Figure 5 Gated Clock and Gated Latch Enable Signals.

This verification is made, in the beginning, analyzing the same file in which the clock has been implemented and then analyzing the others files always respecting the design hierarchy.

3 THE SAVE IMPLEMENTATION

The metrics and methodologies discussed in the previous sections have been implemented in the SAVE tools: complexity analysis, simulation efficiency analysis and feasibility of the synthesis process. Moreover an integrated expert system provides the user with suggestions to improve source code. An architectural scheme of the prototype environment is depicted in Figure 6.

The developed tools assist 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

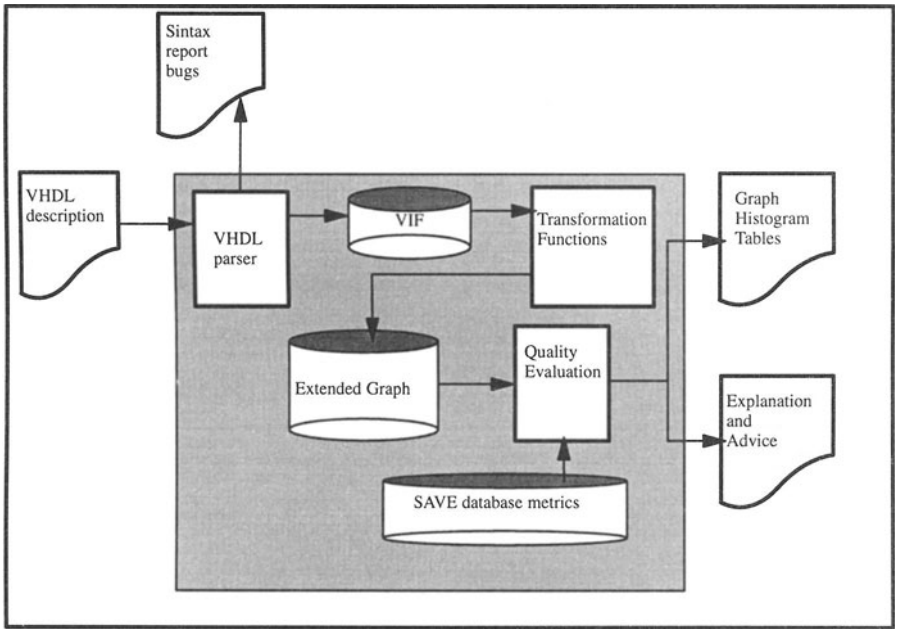

Figure 6 The SAVE Project Architecture.

development process as they use tools as compilers, simulators and so on. In fact, starting from the suggestion reports and graphs which improve the capabilities of the designer to write efficient code, he can evaluate the code, possibly deciding for appropriate modifications.

The LVS (Leda VHDL System) tool supports 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, 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 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.

4 CONCLUSION

The SAVE goal consists in creating support tools to measure, check and generate suggestions with an automatic replacement of pieces of code in order to improve the quality of VHDL models.

In fact, designers should utilize SAVE not to obtain a mere hardware ability evaluation, which is often unappreciated, but to learn to write good quality code.

Users do not need only to know results about software or statistical metrics, they have to recognize the existing relationsships among attributes measure, written code and the possible pratical effect on final design quality.

As much metrics are difficult to understand, as more they will be improbably utilized. That is the reason why simple measures are better than complex ones and why starting with a small set of metrics, increasable during the design development, is the best approach.

Designers will verify the real activity control usefulness and will be more persuaded to analyze their products. All the measures have to be applied in an automatic way using suitable tools and this approach results in minimizing the necessary resources.

5 REFERENCES

A.Balboni, M.Mastretti, M.Stefanoni (1994) Static Analysis for VHDL model Evaluation, EURO VHDL, Grenoble.

A.Balboni, P.Cavalloro, M.Mastretti, A.Bonomo, E.Paschetta, G.Buonanno, D.Sciuto (1994) A set of tools for VHDL–Code Quality Evaluation, VHDL–FORUM for CAD in EUROPE IEEE, Tremezzo.

A.Balboni, M.Mastretti, A.Bonomo, E.Paschetta, G.Buonanno, D.Sciuto (1994) Tool–supported quality evaluation in VHDL–based design, *EDAC 94*, Paris.

A.Bonomo, P.Garino, G. Ghigo, A. Balboni, M.Mastretti (1993) VHDL optimization techniques for coding and simulation. *Rapporto Tecnico CSELT.*

T.J.McCabe (1976) A complexity measure, in *IEEE Trans. Software Engineering,* **vol. SE–2,** 308–12.
S.Cha, I.S.Chung, Y.R.Kwon (1993) *Complexity measures for concurrent programs based on information–theoretic metrics.*
N.E. Fenton (1991) *Software metrics: a rigorous approach,* Chapman & Hall, Norwich.
J.D. Gannon, E. Katz, V Basili (1986) Metrics for ADA packages: an initial study. *Communication of the ACM.*
M.Hueber (1991) VHDL experiments on performance. *Euro VHDL.*
M.Mastretti, M.Sturlesi, S.Tomasello (1995) Quality Measures and Analysis: a way to improve VHDL models. *Workshop on Libraries, Component Modeling, Model Verification and Quality Assurance,* Nantes.
M.Mastretti, M.Sturlesi, S.Tomasello (1995) Static Analysis of VHDL Model Evaluation: Simulation Efficiency and Complexity. *VHDL International User's Forum,* San Diego.
B.A. Kitchenham, S.J. Linkman (1990) Design metrics in practice, in *Information and software technology.*
O.Levia (1991) Writing high performance VHDL models. *Euro VHDL.*
S.Midkiff , D.Padua (1990) Issues in the optimization of parallel programs. *International Conference on Parallel Processing.*
P.Oman, J.Hagemeister (1992) Metrics for assessing a Software system's maintenability. *IEEE Transaction on software engineering.*
L.Ott, J.Bieman (1992) Effects of software changes on module cohesion. *IEEE Transaction on software engineering.*
B.Paulsen O.Levia (1992) Techniques for Writing High Performance and High Quality VHDL Models. *Euro VHDL.*
D.L.Perry *VHDL.* McGraw–Hill, Inc.
J.Ramamoorthy, W.Tsai, T. Yamaura, A.Bhide (1985) Metrics guided methodology. *Proc. 9th Computer Software and Application Conf.,* **11.**
P.N.Robillard, D.Coupal, F.Coallier (1991) Profiling Software Through the Use of Metrics. *Software–Practice and Experience vol.21(5),* **507.**
S.Shatz (1988) Towards Complexity Metrics for ADA tasking. *IEEE Transaction on software engineering.*
M. Shepperd (1990) Design metrics : an empirical analysis , *Software Engineering Journal.*
M. Stefanoni (1994) *SAVE: analizzatore statico di codice VHDL.*
S.N.Woodfield, H.E.Dunsmore, V.Y. Shen (1981) The effect of modularization and comments on program comprehension. *5th International Conference on Software Engineering.*

6 AUTHORS

Mirella Mastretti received the doctoral degree in Computer Science from the University of Milano in 1989.
She joined Italtel in 1989 as member of the technical staff of Central Research Labs. She has worked since then on electronic design automation, developing methodologies and tools to support hardware design flow.
Her main current interests are in the fields of HDL languages, high level synthesis and object–oriented programming.

Maria Laura Busi Since 1989 she joined the Computer Science Department of University of Milano as a computer science degree student. She is currently in stage at the Central Research Labs of Italtel.
Her main interests concern synthesizability and testability of VHDL designs.

Roberto Sarvello After the military service he joined the Computer Science Department of University of Milano as a computer science degree student. He is currently in stage at the Central Research Labs of Italtel.
His main interests concern software quality and artificial intelligence.

Maurizio Sturlesi received the doctoral degree in Computer Science from the University of Milano, in 1995, working in the electronic design automation area at the Central Research Labs of Italtel.
His main interests concern software quality, object–oriented programming and HDL languages.

Sergio Tomasello received the doctoral degree in Computer Science from the University of Milano, in 1995, working in the electronic design automation area at the Central Research Labs of Italtel.
His main interests concern software quality, object–oriented programming and HDL languages.