Early Estimation of the Size of VHDL Projects

William Fornaciari, Fabio Salice, Daniele Paolo Scarpazza,
Politecnico di Milano - DEI
Piazza Leonardo da Vinci, 32, Milano, Italy
{fornacia, salice, scarpazz}@elet.polimi.it

ABSTRACT

The analysis of the amount of human resources required to complete a project is felt as a critical issue in any company of the electronics industry. In particular, early estimating the effort involved in a development process is a key requirement for any cost-driven system-level design decision.

In this paper, we present a methodology to predict the final size of a VHDL project on the basis of a high-level description, obtaining a significant indication about the development effort. The methodology is the composition of a number of specialized models, tailored to estimate the size of specific component types. Models were trained and tested on two disjoint and large sets of real VHDL projects. Quality-of-result indicators show that the methodology is both accurate and robust.

Categories and Subject Descriptors

B.6.3 [**Hardware**]: Design Aids—*Hardware description languages*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures*

General Terms

Management, Design, Languages

Keywords

VHDL analysis, embedded systems, design metrics, cost estimation, system-level design

1. INTRODUCTION

In the last years we assisted to the growth of design reuse initiatives, such as the VSIA [1], and to the spread of third-party suppliers of intellectual property cells. There is a consistent number of cost models, which pay particular attention both to the advanced concept study phase (like in [5] for the automotive market) and to the management of the design cycle (like in [3]).

Technical managers face new scenarios, where the driving forces are *time to market* and flexibility, together with the capability of controlling costs. The success of a strategy often depends strongly on coarse-grained decisions taken during the early phases. For example, solving the *make or buy* problem requires such an accurate estimate of resource consumption and reuse cost-effectiveness.

There exist accurate estimation techniques and flows for performance, area and power consumption, some working at high level of abstraction too, but when it comes to human time (currently the most valuable and scarcest resource), a well-assessed theory to development effort estimation is still a long way to come.

In this paper we present a method for estimating the number of lines of VHDL code in which an embedded system design project will evolve, given its specification, which is recognized to be one of the factors affecting its development effort, e.g. through CoCoMo-style models [2] like $E = a \cdot S^b$, where E is the effort, S is the code size and a and b are parameters accounting for multiplicative and scale phenomena (see [7] for more details). In our methodology, without loss of generality, specifications are not provided in a distinct language but in VHDL itself: in fact any incomplete VHDL project can be considered as an intermediate step towards the final product. Our methodology can be applied at any stage of a development-by-refinement design process, and as the draft approaches the completed project, size estimates will converge to the actual final value.

The paper structure is as follows: section 2 provides the conceptual framework of the methodology, while section 3 introduces a convenient formal representation of VHDL designs on which the entire methodology is based. The core of the methodology is presented in section 4, where models for each of the basic elements composing the designs are presented, together with a constructive strategy to provide estimates for the entire design.

To apply our methodology, we implemented a complete evaluation flow (including a VHDL-93 compliant parser), able to perform model training, test and application. Details are described in section 5. Section 6 discusses the achievements of the methodology and possible improvements.

2. GENERAL APPROACH

Designing systems in VHDL consists in designing an appropriate set of interconnected entities, each accompanied by one or more implementations (called architectures) and their respective internals (processes, signals, variables, functions and procedures; see [8] and [4]). Usually, these entities are not developed concurrently at the same time, instead the whole project is first defined as a top-level entity with no internal details, then decomposed in more subsystems, developed as independent entities, and so on, in a top-down fashion.

When a project is complete, all of its components and relationships can be hierarchically represented with a graph, on which the calculation of the total project size from each part's size is trivial.

In incomplete projects (that is, projects considered at an arbitrary

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

intermediate development stage), parts are not all known equally: some parts are finished (*completely known*), for some only their external interface is known, typically because they were identified in a top-down decomposition but not yet decomposed in further subsystems (*externally known*), some are *completely unknown*, typically because they will be identified in the next decomposition steps of externally known parts.

For incomplete projects, we also found a formal and convenient way to represent the information available on completed parts, their completion status and the remaining of the project, still in the form of a graph. We will provide a set of models and application rules, that allow to estimate the overall resulting size of the finished project (expressed in lines of code, LOC) on the basis of the above graph.

Since a specification is nothing more than an incomplete project, considered at an arbitrary (usually early) stage of a top-down, development-by-refinement design process, it is correct to say that our method is capable of estimating the final size of a VHDL project on the basis of its specifications. Then, the development effort cost from the project size can be estimated, by following the strategy proposed in (as shown in [7]).

3. FORMAL ASPECTS

In this section we introduce some definitions required to describe the method in a formal and concise way. Intuitive explanations are reported in place of formalisms whenever the second would be unnecessarily tedious; for all the formalities, see [10].

3.1 Syntax objects and graphs

A *syntax object* (SO, for short) is any of the following: a project, an entity, an architecture, a process, a subprogram, a component declaration or instantiation. As anticipated, all the SOs of a complete project can be hierarchically arranged in a graph, called *syntax object graph* (*SOG* for short), which depicts their relationships. A SOG is a graph containing nodes of 7 types (one for each SO type) and edges of 2 types: *contains*-type and *references*-type edges. Given a project, building its SOG means:

- creating a node for each SO appearing in the project;
- connecting node A to node B with a *contains*-type edge ($A \supset B$) whenever the SO named A contains the SO named B (that is, B is defined inside the A, like a process can be defined inside an architecture);
- connecting node A to node B with a *references*-type edge ($A \rightarrow B$) when SO A references SO B (informally, inside A there is a call or an instantiation to B).

A sample SOG, generated by our tools and depicting the structure of a Xilinx ADS7870 8-channel voltmeter Springboard module, implemented on a CoolRunner XPLA3 CPLD, is reported in fig. 1. Dashed lines represent $\supset$-type edges and solid lines represent $\rightarrow$-type edges.

In incomplete projects, SO are usually known partially. To handle these cases, we introduce the KSOG, which is a decorated SOG, where a tag is added to each node, indicating its knowledge condition; in detail, a SO is marked with:

- (C) *completely known*, if it is complete and finished in all its constituents;
- (E) *externally known*, if its external interface is known (ports for an architecture, signature for a function, etc.); no internals need to be provided;

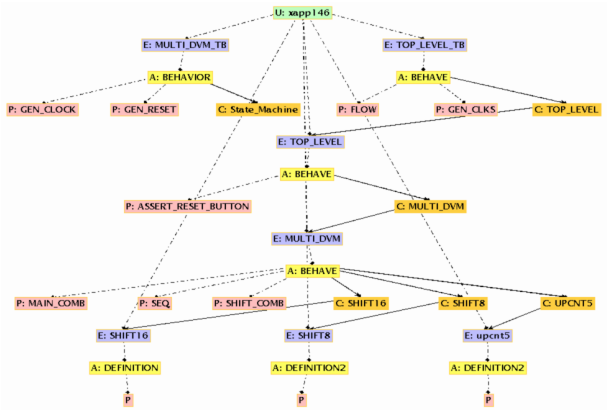

Figure 1: The SOG for a real VHDL project.

- (I) *internally known*, if it is externally known and all its directly contained element are at least externally known;
- (V) *virtually completely known*, when its cost is already known (e.g. it was reused, or bought externally), even though its details are not provided; dealing with these SO's is trivial, since they are treated as an additive cost constant;
- (U) *completely unknown*, when it is not declared at all: size estimates cannot be generated; instead, estimates for container objects must take into account the possible existence of unknown objects.

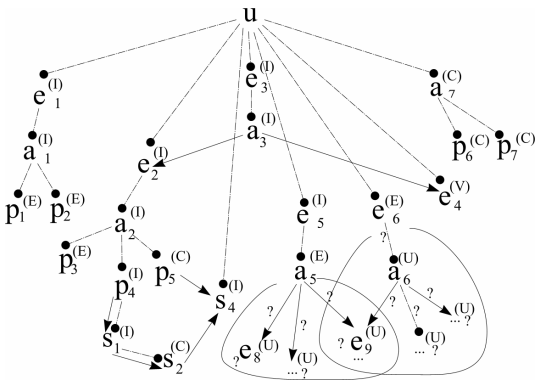

Figure 2: A sample KSOG.

Figure 2 shows a sample KSOG and illustrates some of the possible combinations of nodes and edges.

If we regard a project specification as an incomplete project (thus a KSOG), we can reduce the estimation of its development cost starting from its specifications to the evaluation of the cost of its KSOG.

3.2 Bunches

Most KSOGs have much higher complexity than samples in figures 1 and 2, possibly containing thousands of nodes and edges. Therefore, designing an appropriate evaluation algorithm running on KSOGs could be impractical. Instead, we introduce an object of intermediate granularity, the bunch, that simplifies the task of KSOG evaluation.

Roughly speaking, a bunch is a structure composed by a VHDL entity and whatever belongs to it. More formally, a bunch is a tree rooted at an entity E, containing all those nodes reached by the

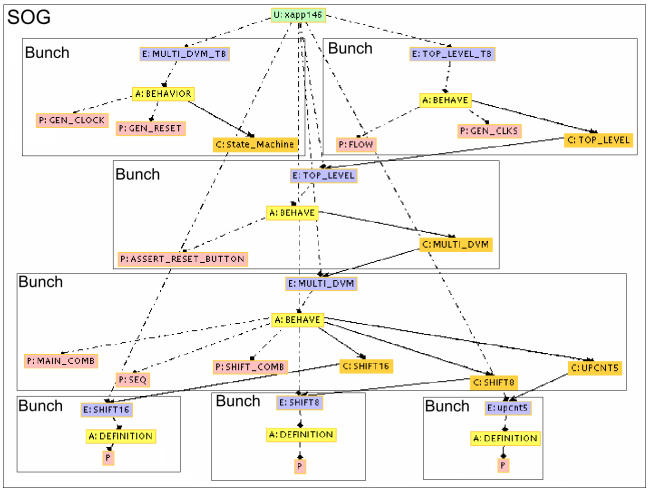

Figure 3: Bunch decomposition of the sample SOG from figure 1.

transitive closure of $\supset$ starting from node E. Bunches exhibit the following useful properties:

1. all the information required to evaluate the size of a bunch is associated to its nodes, therefore each bunch size can be evaluated independently;

2. any SOG and KSOG can be exactly partitioned into one or more bunches[1], as in figure 3, therefore the size of a SOG is given by the sum of its bunches' size. For KSOGs things are more complicated, since some bunches could be completely unknown and their size will be estimated (details in the next section).

3.3 Levels

It is convenient to arrange bunches in levels, thus obtaining a structure that matches the top-down decomposition process; the usefulness of this operation will become clear in the next section. To do that, we observe $\rightarrow$-type edges that cross bunch boundaries. Starting with all the bunches at the top level, and moving bunch R one level beneath Q's level whenever a $Q \rightarrow R$ cross-bunch edge is found, the desired arrangement is obtained.

As we did for partially known SOs, we introduce the additional hypothesis that specifications are not unbalanced, in the sense that either all bunches in a level are at least externally known, or all are completely unknown. Our methodology is not able to deal effectively with promiscuous levels, containing known and unknown bunches at the same time. Good sense suggests that incomplete projects can be unbalanced in the above sense, but specifications (a strict subset of incomplete projects) cannot, therefore the above condition is not really restrictive.

4. THE CORE OF THE METHODOLOGY

The application of our methodology consists of the following steps:

1. a KSOG is built starting from specifications;

2. the KSOG is decomposed into bunches;

3. bunches are split among levels according to their mutual structural $\rightarrow$-type edges;

4. for each bunch, its size is calculated by applying the most appropriate model aggregate;

5. for each known level, its size is calculated as the sum of sizes of all bunches belonging to that level;

6. the estimated cost of the KSOG (comprising unknown parts) is obtained from the sum of the size of the first n^{th} levels by applying an appropriate SOG model.

Our methodology works at different granularities, obtaining the size of a KSOG as a sum of the sizes of its levels, the size of each level as the sum of the sizes of its bunches, the size of each bunch as the sum of the sizes of its syntax objects. Since, at each granularity, one or more constituents can be incompletely known, the above summations would be impossible without a number of specialized models that allow to estimate the size of unknown parts from their available information.

4.1 Models

Accordingly with the methodology so far defined, we introduce three categories of models: SOG models, Bunch models and SO models.

SO models estimate the final size of a SO given the set of available information on it, not counting contained objects (which are subject to respective SO models, if at least externally known). For completely unknown SOs, their number and size are estimated at bunch level, thanks to bunch models. Through bunch models, an estimate of the final size of each at-least-externally-known bunch is obtained; then, SOG models are applied to obtain the size of the whole project as a function of the size of the at-least-externally-known levels.

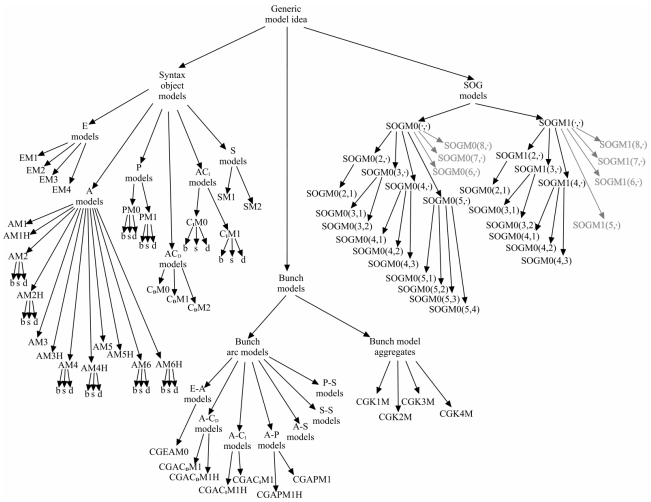

Figure 4: All the models used in this methodology.

The whole set of models used in our methodology is illustrated in figure 4; it is possible to classify them in a tree view.

4.1.1 *Syntax object models*

SO models return an estimate of the core size of a object for which some information is given; they are specialized on the basis of object type (entity, architecture, process, ...), object *mode* (behavioral, structural, data-flow) if applicable, and amount of information required. A list containing the most interesting variables and quantities subject to estimation is reported in table 1.

Similar models were developed for the other SO types; their full details and statistical performance evaluation can be found in [10].

[1] with negligible exceptions, shown in [10].

Quantity	Symbol
Total number of ports	n_p
Number of `in`, `out`, `inout`, other ports	$n_{ip}, n_{op}, n_{iop}, n_{xp}$
Total number of generics	n_g
Total number of internal signals	n_s, h_s
Number of component declarations, instantiations	n_{cd}, n_{ci}
Number of processes in an architecture	n_{pr}
Number of sensitivity signals in the i-th process	n_{ps_i}
Number of variables in the i-th process	n_{pv_i}
Length of entity declaration in LOC	L_e
Length of architecture core in LOC	L_{ac}
Length of the i-th component declaration in LOC	L_{cd_i}
Length of the i-th component instantiation in LOC	L_{ci_i}
Length of the i-th process in lines of code	L_{pr_i}

Table 1: Variables and estimates used in models.

	Last known-size level							
n	L1	L2	L3	L4	L5	L6	L7	L8
1	1	·	·	·	·	·	·	·
2	0.486	1	·	·	·	·	·	·
3	0.128	0.505	1	·	·	·	·	·
4	0.141	0.294	0.721	1	·	·	·	·
5	0.083	0.149	0.425	0.658	1	·	·	·
6	0.053	0.214	0.808	0.972	0.985	1	·	·
7	0.023	0.070	0.308	0.415	0.505	0.925	1	·
8	0.113	0.167	0.408	0.706	0.860	0.921	0.979	1

Table 2: Size of the first i levels as a fraction of the whole project.

4.1.2 Bunch models

Bunch models estimate the cardinality of a set of SOs, directly contained in a given SO, all of which are completely unknown and of the same type. For example, there are models to estimate the number of processes inside a given architecture, or the number of subprograms directly declared inside a process, and so on. Bunch models are classified on the basis of the type of nodes involved in the $\supset$-type edge.

In [10], we conducted an extensive study in order to identify all the possible combinations of node types involved in a $\supset$ relationship, then we counted the number occurrence for each type in our project base, realizing that only a small amount of them were statistically significant, and developed an appropriate set of models for each of them.

4.1.3 SOG models

Once the size of the not-completely-unknown bunch levels has been estimated, the last task to perform is to estimate the size of all the KSOG, on the basis of the above result. This is the purpose of SOG models. The creation of appropriate SOG models was a difficult task, since it was not clear which ones, among all the properties of the KSOG representing the incomplete project and other possible available data, were significant in order to estimate the full final KSOG size. In order to understand that, we designed a rich set of different hypotheses and tested them against our project base. The hypothesis with the highest predictive power turned out to be the following: given a KSOG of depth n, where the size of all levels from 1 to k is known, it should be possible to find an appropriate value, representing the following ratio:

$$\frac{L}{\sum_{i \leqslant k} L_i} = \frac{\sum_{i \leqslant n} L_i}{\sum_{i \leqslant k} L_i}$$

This ratio expresses how many times the whole project is larger than levels $1..k$. In each cell (row i, column j) of table 2 we reported the average values of such ratios collected on our project base. For example in projects with exactly 6 levels (row 6), the number of lines of code belonging to levels 3 and above (column L3) represents the 80.8% of all the lines of code of the project.

	K1	K2	K3	K4
Entity interface	✓	✓	✓	✓
Entity mode	✓	✓	✓	✓
Number of declared components	✓	✓	✓	·
Declared component interface	✓	✓	·	·
Number of instanced components	✓	✓	✓	·
Instanced component interface	✓	✓	·	·
Number of architecture signals	✓	·	·	·
Number of processes	✓	✓	·	·
Process variables	✓	·	·	·

Table 3: Required known information for each model aggregate.

4.2 Model Aggregates

As said before, in order to estimate the size of a bunch, a strict cooperation between SO and bunch models is required. Given a bunch populated with a reasonable number of nodes (one architecture, several processes and components, several signals and variables), the number of possible different knowledge conditions that could occur is remarkable. It is therefore impractical (and of dubious usefulness) to validate models in any possible condition; Instead, we established four discrete conditions, associated with respective model sets, ready to be applied to assess whether a bunch in a given refinement state qualifies or not for a given knowledge state. Such states, called K1, K2, K3 and K4, and the associated rules are illustrated in table 3. For each cell, the presence of a tick mark (✓) means that variables indicated in that row must be known in order to qualify for the knowledge state indicated in that column.

The corresponding model aggregate for condition K2 is reported below as an example.

$$\begin{array}{lcll} (K2)\ \hat{L} & = & \hat{L}_e(n_p, n_g) & \text{(EM3)} \\ & + & \hat{L}_{ac}(h_{ip}, h_{op}, h_{iop}, h_{xp}) & \text{(AM2H)} \\ & + & \sum_{i=1}^{n_{cd}} \hat{L}_{cd_i}(n_{p_i}, n_{g_i}) & (\text{C}_\text{D}\text{M2}) \\ & + & \sum_{i=1}^{n_{ci}} \hat{L}_{ci_i} & (\text{C}_\text{I}\text{M1}) \\ & + & n_{pr} \cdot \hat{L}_{pr} & \text{(PM0)} \end{array}$$

5. EXPERIMENTAL RESULTS

To assess the accuracy of our statistical modeling, we performed a number of experiments. First, to constitute a suitable database of projects, we collected 60 publicly-available fully-developed VHDL projects[2]. Their application scope covers general purpose processors, digital signal processors, basic building blocks like FFTs, microcontrollers, neural networks and so on. This base of project was split in two sets: one used to tune the methodology and the other used for validation purpose. Relevant statistical data characterizing of such projects are summarized in the following table:

	Tuning	Validation
Number of projects:	41	19
Number of VHDL files:	573	469
Number of VHDL code lines:	388,790	222,188
Cumulative size:	16.5M	12.0M
Number of entities:	945	571
Number of architectures:	967	570
Number of component declarations:	952	634
Number of component instantiations:	46,653	35,478
Number of subprogram declarations:	587	298
Number of ports:	9,276	4,984
Number of signals:	58,836	39,017
Number of variables in processes:	1,747	2,449
Number of variables in subprograms:	387	299

[2]the full contents of this archive is available at our website [9]

To automatically collect and process project base data, and to tune, validate and apply models, we developed a set of tools comprising a lexical filter to remove comments, a dedicated VHDL parser to read source files, extract syntax-related information and store it in a specifically-designed SQL database, a set of Tcl scripts to apply all the models and a rich, Tcl/Tk-based, graphical front-end to the above tools. The tool stack is represented in figure 5.

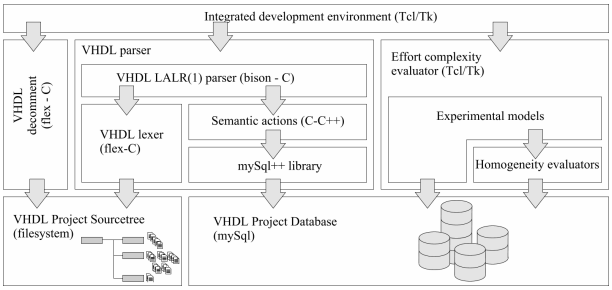

Figure 5: Automatic tools developed for this research.

An exhaustive evaluation of our methodology would consider any possible subset of each project SOG of our project base, thus obtaining a KSOG, then submitting it to the models and comparing the result with the real size of the SOG. But since the number of all possible partial knowledge conditions in a project is extremely high, a test like the one described above is impractical.

Instead, for each of our projects, we set as unknown all information apart those required by model aggregate K2. Then, for all possible k, we used as input the first k^{th} levels of the KSOG obtained at the previous step. Results follow:

Internal validation (based on tuning projects):

	L	$\hat{L}$	$\hat{L}-L$
Average value	2582.071	1826.582	427.492
Standard deviation	3868.919	2950.914	1400.412
Correlation coefficient between L and $\hat{L}$			0.8627

External validation (based on validation projects) :

	L	$\hat{L}$	$\hat{L}-L$
Average value	4016.750	2029.705	-489.674
Standard deviation	5514.420	3121.928	3034.134
Correlation coefficient between L and $\hat{L}$			0.8713

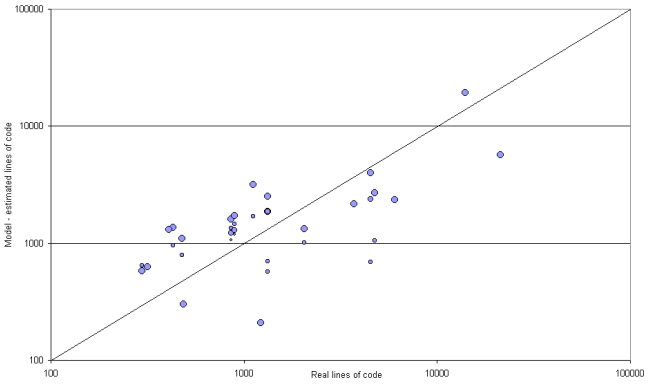

Figure 6: Test set: actual vs. estimated lines of code

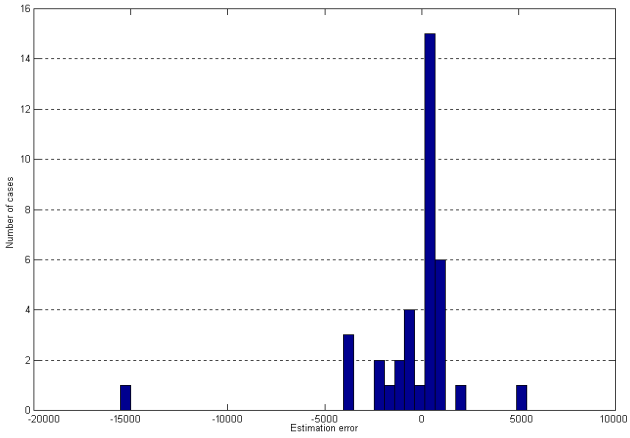

Figure 7: Test set: error density distribution.

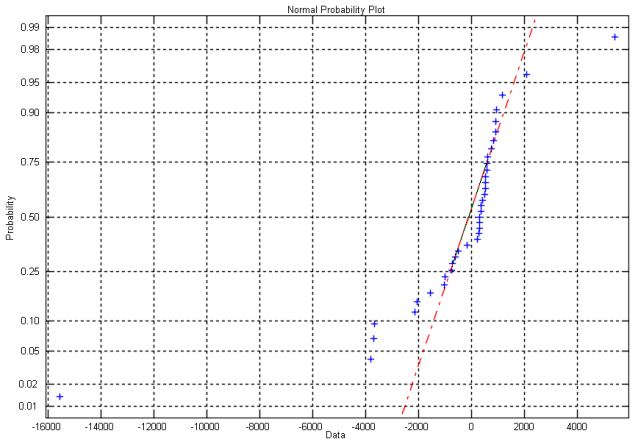

Figure 8: Test set: error cumulative distribution.

6. CONCLUSIONS

After collecting a relevant amount of code ($\geq$ 28 MB), belonging to real industry projects, and implementing automatic analysis and modeling tools, we delivered a methodology, able to estimate the size of each project, with a degree of accuracy depending on the amount of available knowledge. The methodology proves to be both accurate and robust.

Accuracy is proved by a high ρ between real and estimated data (0.8627 and 0.8713 respectively, for internal and external validation) and by acceptable σ_e (1400.412 and 3034.134 lines of code respectively; int more than 80% of the cases, estimation error falls in $\pm\sigma_e$; as represented in fig. 6, 7 and 8). Robustness is confirmed by null degradation of ρ and a tolerable degradation of σ_e when validation is switched from training to test set.

Error compensation occurred whenever models were aggregated: models resulting from composition of finer granularity sub-models exhibit better performances than their constituents (e.g. when syntax object models were integrated to form bunch models, and bunch models coalesced to constitute SOG models).

Our project base contains the vast majority of the public VHDL models on the Internet and is superabundant for SO model tuning, sufficient for bunch models, but scarce when it comes to SOG models. The current effort is to achieve better results with SOG models, by increasing the project base size, and to refine the back-end strategy to derive development effort from project sizes (as anticipated in [7]).

$\rho(\cdot, L)$	n_p	n_{ip}	n_{op}	n_{iop}	n_{xp}	n_g	n_{ps}
Entities	0.7875 (0.6536)	0.4277 (0.5544)	0.5687 (0.5670)	0.8740 (0.2495)	0.1509 (0.3066)	0.9059 (0.5737)	
Architectures	0.4622 (0.1625)	0.2535 (0.1090)	0.3174 (0.1615)	0.6192 (0.0932)	0.0844 (0.0923)	0.5640 (0.0127)	
Processes	0.2291 (-0.0372)					0.2867 (0.0771)	0.3239 (0.0849)

Table 4: Impact of reformatting on $\rho(\cdot, L)$.

7. CURRENT DEVELOPMENTS

Our research group is currently committed to enhancing the methodology in a number of ways: extending the number of projects in the training and test sets, removing the influence of different coding styles, and moving towards more complex models with higher contents in semantics.

7.1 Influence of coding-style

The next version of our estimation flow includes a VHDL reformatter (see [6]), which proved to be dramatically effective in reducing the source code length variability due to different coding styles. Preliminary studies show that the coefficients of correlation between the length of a given syntax object and each of its available variables exhibit significant increases after the code reformatting is performed, as shown in table 4 (values before reformatting are in related in paretheses).

7.2 Risks of overfitting

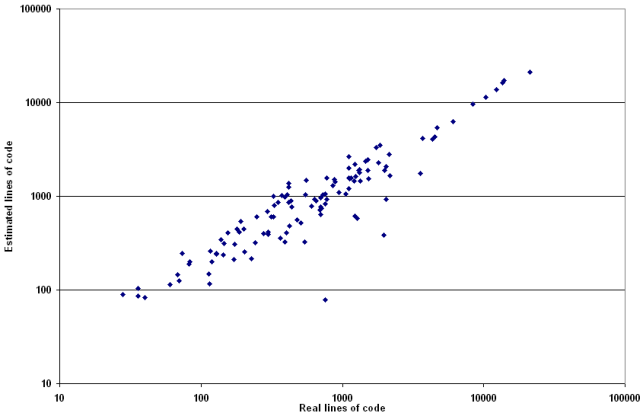

Figure 9: Training set: actual vs. estimated lines of code

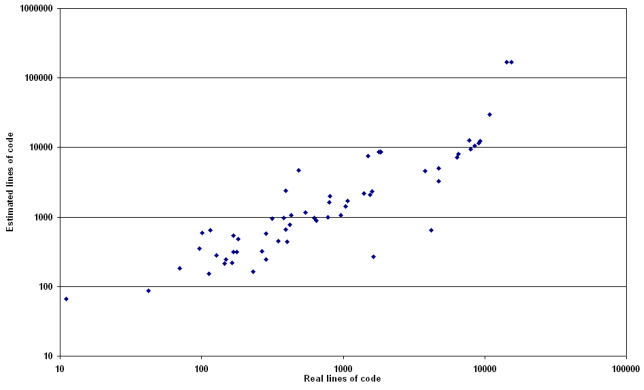

Figure 10: Test set: actual vs. estimated lines of code

The vast majority of the models introduced in section 4.1.1 are simple linear models, using only variables with high correlation coefficients with the length. In several cases, the influence of a given variable (e.g. number of ports) over an object's length is known to be linear, thus, a linear model is known to be correct *a priori*. For all the other cases, we are currently evaluating the use of higher-order models, using all the available variables.

In Figure 9 and 10 we report the results the internal and external validation respectively of an estimation flow in which most linear models have been replaced with second order models.

Internal validation:

	L	$\hat{L}$	$\hat{L} - L$
Average value	1543.269	1843.131	299.861
Standard deviation	3096.056	3375.566	622.811
Correlation coefficient between L and $\hat{L}$			0.9852

External validation:

	L	$\hat{L}$	$\hat{L} - L$
Average value	2376.034	8848.160	6472.126
Standard deviation	3659.174	30521.040	27922.626
Correlation coefficient between L and $\hat{L}$			0.7398

We believe that an improper use of high-order models using also low-correlation variables could lead to overfitting, as above, with very good accuracy in the internal validation and decreased accuracy in the external. These considerations deserve a deeper study and are currently our work goals.

8. REFERENCES

[1] Virtual Socket Interface Alliance, `www.vsia.org`.

[2] *COCOMO 2.0, Model definition manual, v.1.2*. 1997.

[3] Numetrics, *Measuring IC and ASIC design productivity*, white paper, `www.numetrics.com`, May 2002.

[4] P. J. Ashenden, *The Designer's Guide to VHDL*. Morgan Kauffmann Publishers, San Francisco, 1995.

[5] J. Axelsson, Cost model for electronic architecture trade studies. In *Proc. 6th Intl. Conf. on Engineering of Complex Computer Systems*, 2000.

[6] P. S. Elliot and M. Gumm, *MVP v1.1*, Institute for Parallel and Distributed Systems, Stuttgart, Germany, 1994.

[7] William Fornaciari et al., Development cost and size estimation starting from high-level specifications. In *Proc. 9th Intl. Symposium on Hw/sw Codesign*, pages 86–91, ACM Press, 2001.

[8] IEEE Standards Board, *ANSI/IEEE Std 1076-1993, IEEE Standard VHDL Language Reference Manual*.

[9] D. P. Scarpazza, VHDL effort estimation on-line resources at CEFRIEL, `www.cefriel.it/vhdl-estimate`.

[10] D. P. Scarpazza, Development effort and size estimation for partially specified VHDL projects. CEFRIEL Internal report, 2002.