A First Step Towards Hw/Sw Partitioning of UML Specifications

W.Fornaciari (1,2), P.Micheli (1), F.Salice (1,2), L.Zampella (1)

(1) Politecnico di Milano, D.E.I., P.zza L.Da Vinci, 32 – 20133 Milano, Italy
{fornacia,salice}@elet.polimi.it
(2) CEFRIEL, Via Fucini, 2 – 20133 Milano, Italy

Abstract

This paper proposes a novel methodology tailored to design embedded systems, taking into account the emerging market needs, such as hw/sw partitioning, object-oriented specifications, overall design costs and early analysis of design alternatives. The proposal tackles the problem by considering UML as the starting point for system-level description and uses a customization of Function Point analysis and COCOMO to provide cost metrics both for hardware and software. Finally, a genetic algorithm is used to select the best candidate architecture. The paper also reports some results, obtained from a case studies, showing the viability of the proposed approach.

1. Introduction

Embedded Systems is a common term to represent a wide class of devices, submitted to strict requirements in terms of performance, architecture flexibility, operating conditions, cost and development time. However, the designer's challenge is manifest especially whenever the implementation technology is not *a priori* committed and many alternatives should be compared to embrace the best one suiting both functional and implementation goals. Mixed Hardware-Software architectures and concurrent management of all the aspects of the design process, nowadays represent the cornerstones of the so-called *Codesign* discipline.

Within this context, time-to-market pressure is exacerbating the requirements, forcing the designers to consider predictive models (*virtual prototyping*) as soon as possible along the design flow, possibly built on top of executable specifications aiming at capturing the system-level perspective (e.g., C++, VHDL, SystemC and UML).

Design with (for) reuse techniques, can also be adopted to achieve valuable shortening of the design turnaround time, sometimes in detriment of the final implementation cost. The tradeoff is between the potential market loss, due to delayed delivering of the product and the bare implementation cost. Customization of flexible architectures (*platform based design*) has also been adopted with a certain success for specific application fields [1]. However, in many industrial scenarios, like that summarized in [2] for the automotive market, the cost model pays particular attention to the *advanced concept study phase*, where coarse grain decisions have to be taken, such as: number, type and location of the control units (ECUs) composing the system, partitioning of the functionality over the existing ECUs and, selection of the proper communication schemas among the functionality/ECUs. Since embedded systems typically exploit Hw/Sw synergy, this phase help to freeze the amount of resources (Hw, Sw and communication) and the mapping of the functionality. The missing of a significant commitment in reducing the cost during this phase in a systematic manner, not only based on the designer experience, can result in a critical mismatch of the final budget with respect to the forecast. As shown in figure 1, the cost of exploring alternatives is affordable only during the *concept* study, whose main *value added* is the identification of the boundaries containing the solutions candidate for further detailed investigation.

A common agreement on the standards for system-level representation is still a long way to come, even if it seems to be clear the increasing popularity of object oriented (OO) paradigms for both hardware and software, especially if design reuse is envisioned [3] [1]. We adopted UML (Unified Modeling Language) since it is an important standard *de-facto*, and many extensions are going to be added, useful also to capture the peculiarities of embedded systems.

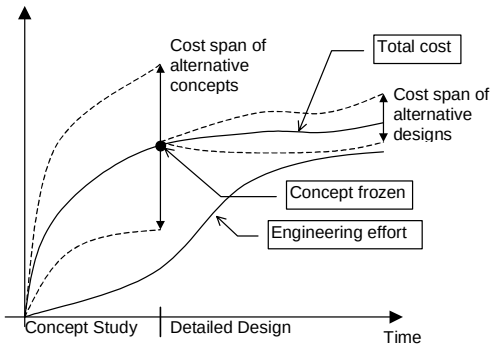

Figure 1. Evolution of the effort and cost during the development time.

Our goal is to argue the effectiveness of a concept study, by providing the designer a methodology to: specify a uncommitted system-level behavior, predict through metrics the global cost for realizing both hardware and software module and, finally, propose an efficient strategy to select a suitable solution within the design space, based on a set of constraints such as cost and reusability. For space reasons, the focus of this paper will be on the partitioning strategy, even if the evaluation metrics will be sketched together with proper references for interested readers.

1530-1591/03 $17.00 © 2003 IEEE

The paper is organized as follows. Section 2 describes the proposed design flow. Section 3 presents the cost functions used for the estimation of both area and cost. Such functions constitute the elements used by the partitioning algorithm to determine the best hw/sw implementation of the system (section 4). Experimental results, considering a design of a Board Computer used in the automotive field, are presented in section 5. Finally, section 6 reports the conclusions of the presented work.

2. The Design Flow

The starting point for developing our tool has been a UML description compliant the Rationale Rose 2000 format The designer feeds the tool with a UML description (PTL file) of the system. Class diagrams are used to provide a general outlook and to analyze the properties of single classes, while Sequence Diagrams specify the temporal interactions among the object to realize the system functionality. Such a textual format is then parsed and improved with additional entries for each class (module), that are processed by our partitioning tool. This additional information reflects the designer needs/constraints plus estimates of crucial system properties. In summary, the information specified by the designer consist of: module already existing, module to be acquired, module to be implemented via reuse, timing constraints, weights for the goal function, and, driving parameters for the partitioning algorithm.

The remaining information, that will be estimated, are: cost of a generic module (hw /sw), cost of a module with reuse (hw/sw), area for Hw-bound modules, equivalent Area for Sw-bound modules (memory). The role of the partitioning is to compute a vector representing the hw vs sw bounding for each class composing the system. This result will be optimal in the sense of optimizing the goal function while fulfilling the user constraints (like reuse) and avoiding full search of the design space.

Many authors afforded the problem of system partitioning. A first step to map UML specs onto hw/sw architectures has been proposed in [4], based on communication refinement; however, the problem of considering design alternatives is not the main issue. Our proposal is to generate and select the design alternatives by using a micro-genetic approach [5] to reduce the computation time dramatically, while maintaining a significant degree of flexibility in adopting user-oriented goal functions.

3. Cost estimation

Probably one of the *tricky* tasks of any manager is to compute reliable forecast of the cost (basically manpower and time to market) of a system starting from top level, possibly incomplete or not very detailed, specifications. Besides, the presence of hardware and software makes harder achieving acceptable accuracy.

Our long-term goal is to propose a unified strategy working at system level, to take into account both the implementing cost and the cost related to the organization of the activities within a design team. For the first stages of the typical design flows (Hw and Sw), there exists a significant parallelism with models conceived for the software development [6]. In particular we considered the COCOMO 2 approach to compute the global *development effort* (Eff), measured in person/month (pm), to realize a given system, and the time T (measured in months) to develop the project assuming a full time commitment of a properly composed group of R designers. For the sake of completeness, main concepts of COCOMO 2 are here recalled; more details can be found in [7] [8].

In general, the cost C will be proportional to the effort $C = K*Eff$, with $Eff= A* S^B$ and $T = A_2 * Eff^{B2}$, so that $R = Eff/T$. The parameters are the *project size* S (Klines of code, KLOC), the coefficient A, A_2 considering possible *multiplicative factors* on the effort and the *scale factors* B, B_2 accounting for economy/diseconomy originated in developing projects of different sizes. It is possible to determine the values of the parameters, according to the modality of developing the project, which is also influenced by the severity of the design constraints and the *novelty* of the application. The typical values derived from a statistical analysis carried out over a significant variety of designs [7] [8] are summarized in table 1, ranging from small and simple projects (organic) to large size ones (embedded) requiring the fulfillment of stringent constraints and thus, a careful control of the development process.

Mode	A	B	A2	B2
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.2	2.5	0.32

Table 1. Values of the model parameters.

As it appears evident from the above relations, the keypoint influencing the quality of the results is the ability to supply values (LOC) for the project size S, both for the hardware and software domains.

Direct determination and use of LOC is a controversial issue since its definition is pretty vague; LOC radically depends on the programming language and its prediction during the preliminary steps of the design produces unacceptable errors. Most of the experts, in fact, tend to underestimate (from 50% to 150%) the size of the project with catastrophic impacts on the design management.

To cope with these problems, getting harder for the presence of Hw and Sw, we use *functional* metrics, instead of trying to guess the project size (see figure 2).

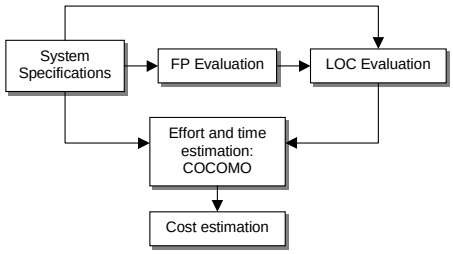

Figure 2. From uncommitted specification to Global Cost.

We adopted an analysis path resembling Function Point (FP) analysis [9] [10], as an intermediate step towards LOC and cost. This strategy provides a measure of the complexity of realizing software applications, by considering the required characteristics, so that it should be independent of the technology and the language used for the implementation. It has been originally proposed by Albrecht [10], and considers characteristics like External Inputs and Outputs; User interaction; External interfaces and Files used by the systems. Each of these items can be

determined from the requirement/design specification (or program code if available) and then individually assessed for complexity and credited a weight ranging typically from 3 to 15. Currently, there exist some versions of the function point analysis, enlarging the scope and solving some weakness, such as that of the IFPUG [11], Feature Point version [12] and our customization to account for the peculiarity of a final hardware implementation based on VHDL (for more details see [6]).

For our purposes, the computation of FP starting UML specification followed the guideline outlined in [13], using Class and Sequence Diagrams. The translation of FP into the corresponding LOC is based on the conversion factors reported in [14], considering statistics derived from the analysis of about thousand projects. For instance, 19 lines of VHDL code are required to implement one FP of the system specification, while for C++ the correspondence is 29 lines per FP and for C this value grows up to 128. The top is the assembly language, with an average of 320 lines/FP. The accuracy of estimating VHDL LOC from FP analysis has been shown [6] to be in the range of 20%.

Due to their wide diffusion in real projects, we restricted our attention only to VHDL, C and C++, but the approach and the analysis tool can be easily retargeted. We also considered other novel figures of merit, as user-defined *directive*, depending on both the *percentage of reuse* and the *degree of reusability* of a certain class composing the specification. The *percentage of reuse*, called π_R, is an estimate of which parts of a module could be conveniently reused, so that $\pi_R = 0$ is the value for any component designed "from scratch". The *degree of reusability* (δ_R) is a factor raging in [1.5...4], representing the additional effort necessary to make reusable a module (internal end external documentation, module parameterization, functional decomposition, functional independence, test benches,…) [6].

The total implementation cost of a module, whose expression is reported in eq.1, comprises the design and material costs, where the design cost, in summary, encompasses the design costs "for reuse" (WR) and "from scratch" (FS):

$$Cost_{imp} = Cost_{des} + Cost_{mat} = (Cost_{FS} + Cost_{WR}) + Cost_{mat} \quad \text{eq.1}$$

The first term of the design cost represents the cost of the design sections $(1- \pi_R)$ unaffected by reuse while the second term accounts for the parts related to reuse. This formula, introducing the aforementioned design for reuse factors δ_R and π_R becomes:

$$Cost_{des} = Cost_{FS} + Cost_{WR} = (1- \pi_R)*K*A*S^B + \pi_R*\delta_R*K*A*S^B =$$

$$=((1- \pi_R)+\pi_R*\delta_R)*K*A*S^B \quad \text{eq.2}$$

Such expression is adopted to compute the design cost of a class of the system specification, starting from the estimates of the project size S (Klines of code, KLOC), from the degree of reusability directive and from the percentage of reuse.

Note that a module with a so good implementation to be reusable could probably be actually considered in other future projects. However, successive uses of the module require some adaptation: the *integration factor* (ι_R), whose value ranges from 0.2 to 0.7, accounts for such effort to incorporate a module designed to be reused in another project. Typically, high levels of reusability correspond to a significant effort to make reusable the module (high value of δ_R), namely more information (testbenches, internal and external documentation) and parameters need to be introduced so that the module functionality can be easily retargeted and understood. For these reasons, ι_R has been considered proportional to $1/\delta_R$ by a factor ε_R, $(4 \geq \varepsilon_R \geq 1)$ where $\varepsilon_R = 1$ means that the integration is performed by the same designer of the considered module (or a skilled designer). Such coefficient takes into account the designer expertise as well as the designer experiences on the considered module: $\iota_R = \varepsilon_R/\delta_R$. Consequently, any successive reference to a reusable element *x* as a basic component to implement a given functionality *y* implies the following design cost:

$$Cost_{des\_y} = (1- \pi_{R\_y})*K*A*S_y^B + \pi_{R\_y}*\iota_R*K*A*S_y^B = ((1- \pi_{R\_y}) +$$

$$+ \pi_{R\_y}*\iota_R)*K*A*S_y^B \quad \text{eq.3}$$

where $\pi_{R\_y}*S_y^B = S_x^B$, $S_x^B = \pi_R * S^B$ and S is project size in eq.2.

By gathering all of the above formulas and by defining with N_{uses} the number of estimated reuses, the general expression for the design cost of the *first* implementation is:

$$Cost_{des} = ((1- \pi_R)+(\pi_R*\delta_R+(\pi_R*(\varepsilon_R/\delta_R)*(N_{uses}-1))/N_{uses})*K*A*S^B = CostCoef*K*A*S^B \quad \text{eq.4}$$

where $(\pi_R*\delta_R+\pi_R*(\varepsilon_R/\delta_R)*(N_{uses}-1))/N_{uses}$ is related with the *Return Of the Investment* (ROI) since every successive integration of a reusable module inside a new project affects the initial cost. Due to space limitation, further considerations about other important time dependent factors (time to market, CAD tool productivity, money cost, estimated time between reuses, functionality decay due to technology evolution, designer competence evolution and turn over influence) cannot be included in this paper.

Table 2 reports a compact analysis of such proposal for a first order model of the design costs. It depicts the breakeven under different environmental conditions, i.e. the number of uses making valuable designing for reuse (*CostCoef*=1).

π_R	δ_R	ε_R	N_{uses}	π_R	δ_R	ε_R	N_{uses}
0	-	-	-				
0,5	1,5	1	2.5	0,5	1,5	3	NS
1	1,5	1	2.5	1	1,5	3	NS
0,5	2,75	1	3.7	0,5	2,75	3	NS
1	2,75	1	3.7	1	2,75	3	NS
0,5	4	1	5.0	0,5	4	3	13.0
1	4	1	5.0	1	4	3	13.0
0,5	1,5	2	NS	0,5	1,5	4	NS
1	1,5	2	NS	1	1,5	4	NS
0,5	2,75	2	7.5	0,5	2,75	4	NS
1	2,75	2	7.5	1	2,75	4	NS
0,5	4	2	7.0	0,5	4	4	1539.0
1	4	2	7.0	1	4	4	4047.0

Table 2. Analysis of the Breakeven conditions, NS stands for no-Solution.

It is worth noting that eq.4 can be easily extended to consider the possibility of acquiring IPs externally, from a third-part supplier. In this case, the *degree of reusability* (δ_R) can be set close to 4 (best reusability effort) and the coefficient ε_R (capturing the designer expertise and the designer experience

with the considered module) has to be grater then 1. Under these assumptions, the design cost model becomes:

$$Cost_{des} = Cost_{IP}*a+((1- \pi_R)+(\pi_R*\delta_R*(1-a) +$$

$$+ \pi_R*(\varepsilon_R/\delta_R)*(N_{uses}-1))/N_{uses})*K*A*S^B \quad \text{eq.5}$$

where a=0 means that the functionality is an internal IP.

Going back to the material cost, the influence of the selling volumes can be represented by decomposing its expression in the following way:

$$Cost_{mat} = Cost_{fixed}+Cost_{variable} = Cost_{fixed}+Cost_{var\_unit}*Volume \quad \text{eq.6}$$

Where the fixed cost is independent of the number of products developed while the variable cost strictly depends on the selling volume.

As far as the variable cost per unit is concerned, a cost function should gather both hardware and software. In the latter case, the cost of the processor must be summed with a term accounting for program storage, obtained by multiplying the equivalent gate cost of a basic memory cell with the software size. As an example, in the following it is assumed that the RAM cost per gate is equal to the *variable cost per gate* of the Cell Based ASIC (CBIC). Similarly, the material cost for the hardware part of the system is computed by multiplying the area estimated for the hardware and the hardware *variable costs per gate* (the model for area estimation is presented later).

By considering the fixed cost, if the solution space is analysed comparing different implementations based on the same technology, this term does not influences the result; on the contrary, costs reported in table 4 could be used.

Processor	Number of Gates	Costs
Sparc	100000	80
V6502	4000	3.2
VZ80	8000	6.4
V8-µRISC	3000	2.4
V8086	18000	14.4
Turbo86	20000	16
V186	28000	22.4
Turbo186	30000	24
Sparc	100000	80
V6502	4000	3.2

Table 3. Processors characteristics

Technology	Cost per gate
FPGA	0.39
MGA	0.1
CBIC	0.08

Technology	Fixed Cost
FPGA	20000
MGA	80000
CBIC	150000

Table 4. Technology costs

Unfortunately, it is hard to get reliable information related to costs from vendors and companies, so that we based the analysis on a single but trustworthy source of data [15]. The costs reported in table 4 and table 3 are expressed in Euros as of 1997. In the case of only the cost is relevant, the goal function (GF) to be minimized, for a system composed of *k* classes, either hardware or software, is:

$$GF_{COST} = \sum_{i=1}^{k} (Cost_i^{sw}*b_i*\phi + Cost_i^{hw}*(1-b_i)) \quad \text{eq.7}$$

Where the Costs (hw or sw) are computed according to eq.1[1], b_i is a binary value representing the hw (b_i=0) or sw (b_i=1) bound of the i-th class and ϕ ($0<\phi\leq 1$) is the software flexibility whose effect is to reduce the influence of the cost. This goal function is biased toward a fully software implementation, since it is typically characterized by lower costs and maximum flexibility with respect to hardware.

Regarding area, a second goal function to be minimized has been assembled, following a strategy similar to the previous one, i.e. the area is estimated from the LOC computed via FP analysis [6].

In a fist order approximation, we can assume a linear dependence between the area (in terms of equivalent gates) of a hardware implementation and its complexity. By analyzing a number of existing projects with different complexity and application fields, a range of 1-10 equivalent gates (EG) per VHDL line of code has been identified (a typical value for *structural* description is around 2 gates/VHDL line). The conversion of FP to VHDL LOC has been performed by considering the factor suggested in [14], that is 19, so that:

$$Area_{EG}^{hw} = FP * 19 * f(\text{application, description mode})$$

where the application is a combination of *FSM*, *CombinationalGenericModule*, *ROMs*, *RAMs*, *ArithmeticOperators* (single cost functions has been identified for each element) and the description mode is *Structural*, *DataFlow* and *Behavioural*. The current version of the model has been implemented using 2 as the typical value. It has been identified by using a set of benchmarks whose VHDL descriptions are a mix between *Structural* and *DataFlow* and statistically assessed by evaluating the FPs and implementing the devices on a VirtexII-1000 technology by using *Leonardo* and Xilinx ISA4.1.

Concerning the software, the concept of area is less obvious and it has been computed by adding up two contributions: the memory and the processor. As far as the processor is concern, the number of gates have been extracted from the data sheets of the considered processors; conversely, the memory occupation has been computed referring to assembly LOC and 32 bits instructions, as typical for RISC architectures. Note that such parameter can be tailored to account for other Instruction Set Architecture (ISA) peculiarities. For the Intel processor family, the average instruction length has been computed considering a number of benchmarks. As an example, in the following we refer to a RISC architecture with a fixed instruction size of 32 bits, so that the area becomes 9.6=(0.3*32) equivalent gates per assembly line, since a 1-bit cell typically requires 0.3 gates, due to the high regularity of the memory structures (this value has been estimated using a set of data provided by Siemens).

[1] LOC is calculated considering the target language for the hw and sw implementation of the classes. Both costs, can involve reuse.

It is worth noting that the area for data storing is not part of the model; this approximation has been introduced since the influence of small amounts of data can be neglected while significant amounts of data require memory for both hardware and software; under this assumptions, the data size is an invariant with respect to the partitioning problem.

For the software, the area (evaluated in gates) will thus be:

$$Area_{EG}^{sw} = Area_{RAM\_code} + Area_{Processor} = \\ = FP * 320 * 9.6 + Area_{Processor} \quad \text{eq.8}$$

And the global goal function tailored to consider only area becomes:

$$GF_{AREA} = \sum_{i=1}^{k} Area_i^{sw} * b_i + Area_i^{hw} * (1-b_i) \quad \text{eq.9}$$

Dually, this goal function is biased toward a fully hardware implementation, since it is typically characterized by lower area with respect to software.

In order to perform the partitioning, area and cost have been normalized considering that the maximum values are associated with the area of the fully software implementation and the cost of the fully hardware implementation while the minimum values are associated with the area of the fully hardware implementation and the cost of the fully software implementation. Hence, the global goal function is:

$$GF = \left(\frac{Area_{MAX} - GF_{area}}{Area_{MAX} - Area_{min}}\right)^{(1-A)} * \left(\frac{Cost_{MAX} - GF_{cost}}{Cost_{MAX} - Cost_{min}}\right)^{A} \quad \text{eq.10}$$

Combinations of both goal functions can be considered to better adhere the designer's needs, as shown in Section 5 (parameter A).

4. System partitioning

The variability of design alternatives allows the user to take into account a number of possible characteristics like reusability (including the additional effort for making reusable the modules), cost and size of both hw and sw, the possibility of using third-part components (COTS) and so on. Due to the wide extension of the design space to be explored, full search or even a simple Branch&Bound strategy have been discarded, in favor of more computationally effective heuristics, able to discover acceptable sub-optimal solutions, e.g. simulated annealing or genetic algorithms.

We selected a strategy based on a variation of Microgenetic algorithms, tailored to optimized multi-goal functions [5]. The basic difference with respect to classical genetic strategies, is the peculiarity of the considered populations, that are restricted and the presence of external memory where to record the best candidate solutions. A proper replacing strategy of the stored solutions with new ones is used to limit the memory requirements. The algorithm exploits clustering and exhibit *elitism*, i.e. the capability to span uniformly the entire solution space, following not only random paths.

The operations executed within a micro-cycle are the classical ones: generation of the initial population, reproduction, crossover and mutation. The initial parameters of the algorithm are the total number of iterations and the probability of mutation and crossover. The tool implementing the algorithm allows the operating modes: *single* and *multi*. Single mode executes only one elaboration of the partitioning algorithm, whose result are stored in a reports, while multi mode produces a (user-defined) set of executions (partitionings) from the same input file, so to make possible for the user to compare similar results.

As sketched in previous sections, the partitioning algorithm operates at the granularity of classes, since considering functionality is too coarse. Finer grain is not considered since it is impossible starting from UML schemas: the methods specifiable during the phase of concept study are not very accurate.

The tool analyzes one functionality of the system at a time, each involving several classes. This means that, to obtain a solution for the overall system, the execution of the tools must be invoked multiple times, to process all the existing functionality. Finally, the best solutions identified for each functionality are gathered to constitute the global solution. The current version of the tool implicitly assumes that the functionalities are always disjoint, not considering the cost and area of the integration additional components. In practice, such overhead can be -in the average- neglected or considered as a multiplicative factor [7][6][8], not influencing the *structure* of the methodology and the kernel code of the algorithm. Also the management of UML schemas adopted by Rational Rose force in this direction: for each USE CASE, representing a functionality, the corresponding Class and Sequence diagrams are designed.

5. Experimental Results

The tool implementing the methodology has been implemented in C++, Kdevelop 1.4 (Linux Mandrake 8.0), using RCS for configuration management, Rational Rose 2000 and ZTC for syntax checking of formal specifications. The code has been validated through black-box and white-box testing using small and toy benchmarks, as well as by using real-world examples.

In general we can observe that in the case of small class diagrams (less than 10 classes) and with more than 1500 iterations, the results are always those expected. As the number of classes increases, the amount of iterations to achieve 100% of matching rises up more than linearly. For example, for a 20 classes schema and considering as goal function the GF_{cost} in order to obtain a fully sw solution, more than 5000 iterations are required. As an example to point out the practical use of the methodology and of the tool, in the following we consider the design of a Board Computer used in Automotive. The class diagram is composed of six classes, controlling all the car functionality: brakes, engine, air conditioning, windows and alarms. The computation of the FP, followed the suggestions of [13]. Table 5 reports a summary of the obtained results, showing for each class, the contributions for each of the five characteristics: Internal Logical File (ILF), External Initerface File (EIF), External Input (EI), External Inquiry (EQ) and External Output (EO), before introducing the adjustment factor, as suggested by the FP methodology.

The generation and evaluation of alternative partitions has been performed considering a simple while flexible composite goal functions, in order to easily explore the outputs produced by varying the importance of area and cost, modifying only one parameter "A" ranging in [0..1]: A is the weight for the cost and

(1-A) that for the area. In this example, equation 10 is the considered cost function.

Class name	ILF	EIF	EI	EQ	EO	FP
Board comp.	1×7	0×5	1×3	3×3	10×4	**59**
Brakes	0×7	1×5	0×3	1×3	1×4	**16**
Engine	0×7	1×5	2×3	0×3	1×4	**15**
Windows	0×7	1×5	2×3	0×3	0×4	**11**
Front wind.	0×7	1×5	2×3	0×3	0×4	**11**
Air cond.	0×7	1×5	3×3	0×3	0×4	**14**

Table 5. FP calculation (summary) for the board computer.

The border solutions considering A=1 or A=0, represent the cases where only cost or area are relevant, respectively. These solutions also correspond to fully software or hardware implementation. Intermediate value of A, depending of course on the user needs, allows to obtain mixed hw/sw architectures considering both area and cost goals. For each of elaboration performed, corresponding to a different value of A, i.e. a different goal function, 100 iterations of the algorithm have been considered and five attempts for each values of the parameter A (the runtimes are always of few minutes).

Table 6 reports the optimal hw/sw partition performed by the implemented procedure. In particular, each configuration corresponds to the minimal value of the proposed goal function produced in five runs of the partitioning algorithm.

Usr needs A	Brake abs	Board Computer	Front Windows	Windows	Air Cond	Engine Manag
1.0-0.6	SW	SW	SW	SW	SW	SW
0.5	HW	SW	SW	SW	SW	SW
0.4	SW	SW	SW	SW	SW	HW
0.0	HW	HW	HW	HW	HW	HW

Table 6. Final system partitioning with respect to some different user needs (A) without reuse.

As a final analysis, the presence of hardware reuse has been considered; for the sake of conciseness table 7 reports only the result concerning one component.

A	π_R	δ_R	ε_R	N_{uses}	ABS	Board Comput	Front Windows	Windows	Air Cond	Engine Manag
0.5	1	1,5	1	6	HW	SW	SW	SW	HW	SW
0.5	1	2,75	1	6	HW	**HW**	SW	SW	HW	SW
0.5	1	4	1	7	SW	**HW**	SW	SW	SW	SW
0.5	1	1,5	2	NS	HW	SW	SW	SW	HW	SW
0.5	1	2,75	2	20	SW	**HW**	SW	SW	SW	SW
0.5	1	4	2	11	SW	**HW**	SW	SW	SW	SW
0.5	1	2,75	3	NS	HW	SW	HW	HW	HW	HW
0.5	1	4	3	40	HW	**HW**	HW	HW	HW	HW
0.5	1	4	4	NS	HW	SW	HW	HW	HW	HW

Tabella 7. Final system partitioning with hardware reuse and user needs A=0.5. The parameters π_R, δ_R and ε_R are imposed and the N_{uses} is calculated such that it is convenient the hw implementation with reuse of the board computer component.

The goal has been to identify under which conditions the presence of reuse can carry to fully hardware implementations (bold text). For example, for a project with a poor documentation and a designer without significant experience (δ_R=1.5), that is the case of the fourth row, there is no convenience (NS, no solution) to introduce a hw implementation with reuse, so that it will be sw.

6. Concluding remarks

The paper presented a methodology to afford the problem of freezing up a suitable hw/sw partitioning for an embedded application, starting from a top-level description of the architecture, in this case UML, although for different OO paradigms the proposal still maintain its applicability.

The analysis is based on a novel extension of function point analysis to cover also the peculiarity of hardware-bound systems in a unified manner. Appropriate metrics to predict implementation costs and designer goals have been identified, working at a coarse grain so to be used during the earlier stages of the design. The validity of the methodology and in particular of the proposed partitioning strategy based on a suitable customization of the genetic algorithms has been assessed considering the design of a board controller for automotive application. Other analyses have been performed to point out the impact of component reuse within a project as well as the presence in the system of pre-designed parts coming from third-part suppliers.

Work is in progress to extend the population of sample projects to better tune the parameters of the methodology and the estimates of the implementation cost.

7. References

[1] A.S.Vincentelli, G.Martin, *Platform-Based Design and Software Design Methodology for Embedded* Systems, IEEE Design & Test of Computers, vol.18, n.6, Nov-Dec '01, pp 23-33.

[2] J. Axelsson, *Cost Model for Electronic Architecture Trade Studies*, Proc. Sixth Int. Conf. on Engineering of Complex Computer Systems, Tokyo, Japan, 2000.

[3] R.Pasko, S.Vernalde, P.Schaumont, *Techniques to Evolve a C++ Based System Design language*, Proc. of Design Automation and Test in Europe, DATE 2002,, Paris, France, March 4-8, 2002. pp. 302-309.

[4] G.Martin, L.Lavagno, J.L.Gurein, *Embedded UML: a merger of Real-Time UML and Codesign*, 9th Int. Symp. on Hw/Sw Codesign (CODES '01), Copenhagen, Denmark, April, 2001.

[5] C. A. Coello, G. T. Pulido, *A Micro-Genetic Algorithm for Multiobjective Optimization*, Lania-RI-2000-06, Laboratorio Nacional de Informática Avanzada, 2000.

[6] U. Bondi, W.Fornaciari, E. Magini and F. Salice, *Development Cost and Size Estimation Starting from High-Level*, 9th Int. Symp. on Hw/Sw Codesign (CODES '01), Copenhagen, Denmark, 2001.

[7] COCOMO 2.0 *Model Definition manual*, ver 1.2, 1997.

[8] Bohem, *Software Engineering Economics*, Prentice Hall, 1981

[9] Carper Jones, © 1997, Software productivity Research Inc., *What are Function Point?* www.spr.com/library/0funcmet.htm.

[10] A.J.Albrecht, *Function Point Analysis*, Encyclopedia of Software Engineering, vol. 1, Jhon Wiley & Sons, 1994.

[11] *Function Point Counting Practice Manual, Release 4*. IFPUG International Function Points Users Group, http://www.ifpug.org.

[12] C.Jones,, *Applied Software Measurements*, McGraw-Hill, 1996.

[13] T.Uemura, S.Kusumoto, K.Inoue, *Function Point Measurement Tool for UML Design Specification*, Proc. of the Sixth IEEE International Symposium on Software Metrics, November, 1999.

[14] Carper Jones, © 1997, Software productivity Research Inc., *Programming Language Table*, Release 8.2, March 1996. http:// www.spr.com/library/0langtbl.htm.

[15] www.dacafe.com/DACafe/EDATools/EDAbooks/ASIC/ASICs.htm.