Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 280347, 12 pages
doi:10.1155/2008/280347

Research Article

A Priori Implementation Effort Estimation for Hardware Design Based on Independent Path Analysis

Rasmus Abildgren,[1] Jean-Philippe Diguet,[2] Pierre Bomel,[2] Guy Gogniat,[2] Peter Koch,[3] and Yannick Le Moullec[3]

[1] *CISS, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark*
[2] *Lab-STICC (UMR CNRS 3192), Université de Bretagne Sud, Centre de recherche, BP 92116, 56321 Lorient Cedex, France*
[3] *CSDR, Aalborg University, Fredriks Bajers Vej 7, 9220 Aalborg East, Denmark*

Correspondence should be addressed to Rasmus Abildgren, rab@es.aau.dk

Received 15 March 2008; Revised 30 June 2008; Accepted 18 September 2008

Recommended by Markus Rupp

This paper presents a metric-based approach for estimating the hardware implementation effort (in terms of time) for an application in relation to the number of linear-independent paths of its algorithms. We exploit the relation between the number of edges and linear-independent paths in an algorithm and the corresponding implementation effort. We propose an adaptation of the concept of cyclomatic complexity, complemented with a correction function to take designers' learning curve and experience into account. Our experimental results, composed of a training and a validation phase, show that with the proposed approach it is possible to estimate the hardware implementation effort. This approach, part of our light design space exploration concept, is implemented in our framework "Design-Trotter" and offers a new type of tool that can help designers and managers to reduce the time-to-market factor by better estimating the required implementation effort.

Copyright © 2008 Rasmus Abildgren et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

1.1. Discussion of the problem

Companies developing embedded systems based on high-end technology in areas such as telecommunication, defence, consumer products, healthcare equipment are evolving in an extremely competitive globalised market. In order to preserve their competitiveness, they have to deal with several contradicting objectives: on one hand, they have to face the ever-increasing need for shorter time-to-market; and on the other hand, they have to develop and produce low-cost, high-quality, and innovative products.

This raises major challenges for most companies, especially for small- and medium-sized enterprises (SMEs). Although SMEs are under pressure due to the above-mentioned factors, they are either not applying the latest design methodologies or cannot afford the modern electronic system level (ESL) design tools. By limiting themselves to traditional design methodologies, SMEs make themselves more vulnerable to unforeseen problems in the development process, making the time-to-market factor one of the most critical challenges they have to deal with. A survey released at the Embedded Systems Conference (ESC 2006) [1] indicated that more than 50% of embedded design projects are running behind schedule (i.e., 25% are 1-2 months late, 18% 3–6 months). In the 2008 version of the survey [2], it is again shown that meeting the schedule is the greatest concern for design teams.

Moreover, a workshop [3] held for Danish SMEs working in the domain of embedded systems clearly indicates that there is a need for changing and improving their design trajectories in order to stay in front of the global market. More specifically, this calls for setting modern design, that is, hardware/software (HW/SW) codesign, and ESL design into actual practice in SMEs, so that they can reduce their time-to-market factor and keep up with their competitors by being more efficient in producing embedded systems.

Although HW/SW codesign and ESL design tools (both commercial and academic) have been available for several years, there are several barriers that, so far, have prevented their wide adoption such as the following:

(i) difficulty in transferring the methods and tools developed by academia into industry, because they are mostly developed for experimenting, validating, and proving new concepts rather than for being used in companies; therefore adapting and transferring these methods and tools require additional and tedious efforts, delaying their adoption;

(ii) financial cost in terms of tool licenses, training, and so forth that many SMEs cannot afford, since the cost of a complete commercial tool chain can exceed in excess of 150 k€ per year;

(iii) training cost and knowledge management issues, meaning that switching to a new design trajectory also involves the risk of loosing momentum, that is, loosing time and efficiency because of the training needed to master the new methods and tools;

(iv) many modern design flows are not mature enough to generate efficient and automatic real-time code, and combined with the previous item, cause potential adopters to wait until it is safe to switch.

Considerable research has been undertaken to estimate implementation factors such as area, power, and speed up that are subsequently used in HW/SW partitioning tools with different focuses related to granularity, architecture model, communication topology, and so on. All of these research projects do not include the man-power cost which is the most critical one for many companies, and especially SMEs. This work takes its outset in a research framework facilitating the HW/SW partitioning step for SMEs. It focuses on a light design space exploration approach called "DSE-light" that combines the advances in terms of design methodologies found in academia and the ease of integration required by SMEs, that is, lowering the above-mentioned barriers.

The contribution presented in this paper is the development of a method for estimating the man-power cost (i.e., development time) for implementing hardware components and the integration of this method into our framework, so that HW/SW partitioning decisions can be wiser. A method that used iteratively and systematic will form the engine for precise development schedules. The following subsections present the rationale for this work and the idea enabling this contribution.

1.2. Parameters that influence the implementation effort

A common problem in both SMEs and larger companies is that of estimating the amount of time required to map and implement an algorithm onto an architecture given parameters such as [4, 5] the following :

(i) manpower, that is, the available development team(s) and their size(s),

(ii) quality of the social interactions between the team members and the teams,

(iii) experience of the developers (e.g., years of experience, previously developed projects, novelty of the current project, etc.),

(iv) skills of the developers, that is, their ability to solve problems (this is not the same as experience, which only reflects how often one has tried before),

(v) availability of suitable and efficient tools and how easy they are to learn and use,

(vi) availability of SW/HW IP code/cores,

(vii) involvement of the designers, that is, are they working on other projects simultaneously?

(viii) design constraints, that is, real-time requirements,

This work addresses the issue of adding man-power cost parameter into the cost function and thereby guiding the HW/SW partitioning. More specifically we concentrate on the mapping process, that is, the process of mapping a given algorithm onto a given architecture and the implementation effort (i.e., time) related to the complexity of that algorithm. Our framework also addresses other issues of HW/SW partitioning, for example, [6].

1.3. Idea

In order to understand what makes an algorithm difficult to implement, five semistructured interviews have been conducted with engineers (hardware developers) with very little to 20 years of experience. (Semistructured interview is an information-gathering method of qualitative research. It is also an adequate tool to capture how a person thinks of a particular domain [7].)

From the interviews, it was deduced that several parameters influence on the hardware design difficulty. The hardware developers stated that available knowledge about worst cases, dependencies between variables, and the completeness of the design description of the entire system including all communications are important for the design time. However, according to them, the major parameter influencing a hardware design is the number of connections and signals between the internal components. This should be viewed in the way in which every time a signal enters a component, it means that the component needs to act on it. More signals bring more parameters into the component and that very often leads to an increased complexity.

Based on the interviews, we form our hypothesis, which is that a strong relation exists between what renders an algorithm complex to implement and the number of components as well as the number of signals/paths in the algorithm.

To ensure that not only the number of paths are counted but also that a high number of components is present, we choose to only measure the number of linear-independent paths. Furthermore, this insures that components occuring several times during the execution are counted only once, which better reflects the actual implementation efforts.

The remainder of the paper is organised as follows: Section 2 gives an overview of the state-of-the-art methods for estimating the implementation effort both for software and hardware designs and indicates the need for further work for hardware design. In Section 3 a new metric for estimating the development time is defined and combined with our

research tool "Design-Trotter." Section 4 presents some test cases used to investigate the validity of the above-mentioned hypothesis and of the proposed metric. Furthermore, the experimental results are analysed. Finally we conclude in Section 5.

2. STATE OF THE ART

2.1. Software

Most research about estimating implementation effort is found in the software domain, especially within the COCOMO project [8]. The problem of estimating the implementation effort is twofold. First, a reasonable measure needs to be developed for being able to quantify the algorithm. Second, a model needs to be developed, describing a rational relation between the measure and the implementation effort.

2.1.1. COCOMO

To start with the model, a typical power model has been proposed inside the COCOMO experiment [8, 9]:

$$\text{Effort} = A \times \text{Size}^{b}, \tag{1}$$

where Size is an estimate of the project size, and A and b are adjustable parameters. These parameters are influenced by many external factors which we previously discussed in Section 1.2, but can be trained, based on previous project data.

To use this COCOMO measure, there is a need for expressing the size of the project. Inside the software domain, the dominating metric is lines of code (LOC). Using LOC is not without difficulties, for example, how is a code line defined? Reference [10] discusses this issue and states that LOC is not consistent enough for that use; this is also supported by [11]. Using the LOC metric also has several difficulties, for example, it is not a language independent metric. Furthermore, hardware developers also tend to disapprove this measure, since they do not feel that it is a representative measure for hardware designs.

However, we do not claim that there is no relation between LOC and the implementation effort. It is impossible to write 10 k lines in one day, but for VHDL the relation is not always straightforward. In the experiments that we have performed (data shown in Table 1) there is no unambiguous relation between the LOC in VHDL and the development time.

Reference [11] describes that making "a priori" determination of the size of a software project is difficult especially when using the traditional lines of code measure; instead function points-based estimation seems to be more robust.

2.1.2. Function points analysis

The function points metric was first introduced by Albrecht [12] and consists of two main stages: The first stage is counting and classifying the function types for the software. The identified functions need to be weighted reflecting their complexity, that is determined on the basis of the developers' perception. The second stage is the adjustment of the function points according to the application and environment, based on 14 parameters. The function points can then be converted into an LOC measure, based on an implementation language-dependent factor, and, for example, [11] reports that the function points metric can be used as an implementation effort estimation metric. The function points analysis has been criticised of being too heuristic and [10] has proposed the SPQR/20 function points metric as an alternative. Reference [13] has compared the SPQR/20 and the function points analysis and found their accuracy comparable even though the SPQR/20 metric is simpler to estimate.

2.2. VHDL function points

To the knowledge of the authors, limited research has been carried out in the field of estimating the implementation difficulty of hardware designs.

Fornaciari et al. [14] have taken up the idea from the function points analysis and modified it to fit VHDL. By counting the number of internal I/O signals and components, and classifying these counts into levels, they extract a function point value related to VHDL. They have related their measure to the number of source lines in the LEON-1 processor project, and their predictions are within 20% of the real size. However, as stated previously, estimating the size does not always give an accurate indication of the implementation difficulty, and the necessary implementation time.

By measuring the number of internal I/O signals and components, their work goes along the same road as our initial observations indicate. However, our approach is pointing towards estimating the implementation effort, based on a behavioural description of the algorithm in the C-language. Furthermore, it also takes the designer's experience into account.

3. METHODOLOGY

The proposed flow for estimating the implementation effort is illustrated in Figure 1. It takes its outset in a behavioural description of the algorithm, in C-language (including library function source code), which is intended to be implemented in hardware. From this description, we use the design-Trotter framework to generate a hierarchical control data flow graph (HCDFG) which is then measured to identify the number of independent paths. The resulting measure, combined with the experience of the developers, gives an estimate of the required implementation effort. The method is self-learning in the sense that after each successful implementation, new knowledge about the developers involved can be integrated, and improve the accuracy of the estimates. The HCDFG and the approach for modelling the developers experience are covered later in this section but initially we investigate how the number of paths can be measured.

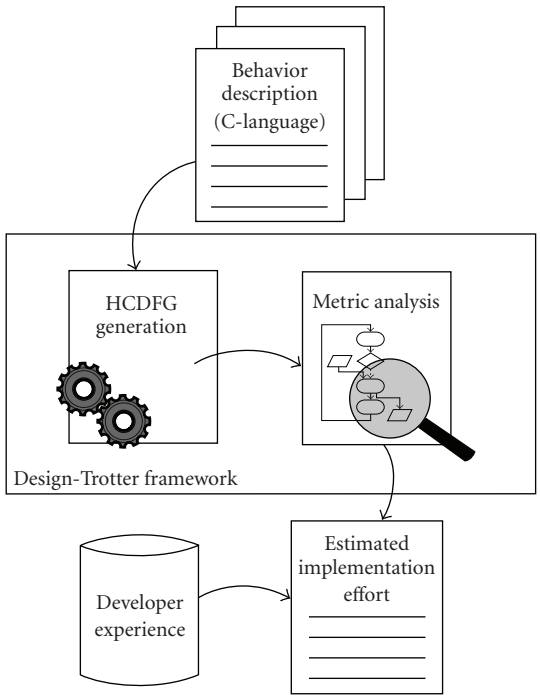
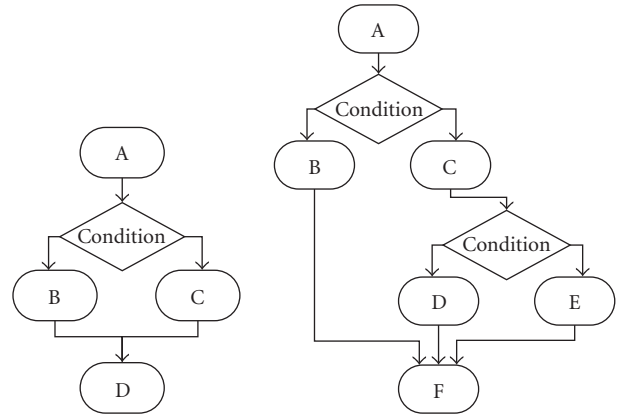


Figure 1: The flow of estimating the required implementation effort. The starting point is a behavioural description in C of the algorithm to be implemented in hardware (e.g., via VHDL). From this description, an HCDFG is generated and measured to identify the number of independent paths in the algorithm. This measure, combined with the experience of the developers, gives an estimate of the required implementation effort (expressed in time).



(a) $P(G) = 2$ (b) $P(G) = 3$

Figure 2: Two examples of graphs for which the cyclomatic complexities have been calculated.

3.1. Cyclomatic complexity

As described in Section 1.3, the number of independent paths is expected to correlate with the complexity that the engineers are facing when working on the implementation. Therefore, finding a method to measure the number of independent paths in an algorithm could help us investigating this issue. A metric measuring is the cyclomatic complexity measure proposed by McCabe [15] which measures the number of linear-independent paths in the algorithm.

The cyclomatic complexity was originally invented as a way to intuitively quantify the complexity of algorithms, but has later found use for other purposes especially in the software domain. The cyclomatic complexity has been used for evaluating the quality of code in companies [16], where quality covers aspects from understandability over testability to maintainability. It has also been shown [17] that algorithms with a high cyclomatic complexity more frequently have errors than algorithms with lower cyclomatic complexity. The cyclomatic complexity has furthermore been used for evaluating programming languages for parallel computing [18], where languages that encapsulate control statement in instructions are receiving higher scores. All use the cyclomatic complexity measure under the assumptions that the complexity has significant influence on the number of paths the developers need to inspect, its correlation to the number of paths that needs to be tested, or a combination of the two.

In the domain of hardware, the cyclomatic complexity has also found use, judging the readability and maintainability in the SAVE project [19]. It is worth noticing that they use a misinterpreted [20] definition of the cyclomatic complexity [21].

All these projects utilise the cyclomatic complexity's ability to measure the number of independent paths and relate them to their individual cases:

$$P(G) = \pi + 1, \tag{2}$$

where π represents the number of condition nodes in the graph G representing the algorithm being analysed. Figure 2 shows two examples of graphs and the corresponding cyclomatic complexity.

In this work, we propose an adapted version of the cyclomatic complexity definition to estimate, a priori, the number of independent paths on a hierarchical control data flow graph (HCDFG), defined in the following section. The cyclomatic complexity for an HCDFG is obtained by examining its subgraphs as explained in Section 3.3.

3.2. HCDFG

For this work we use the hierarchical control data flow graphs (HCDFGs), which are introduced in [22, 23]. The HCDFGs are used to represent an algorithm with a graph-based model so the examination task of the algorithm is eased. Control/Data Flow Graphs (CDFGs) are well accepted by designers as a representation of an algorithm where data flow graphs represent the data flow between different processes/operations, and the control flow layer, encapsulating these data flows and adding control structures to the graphical notation. The hierarchy layered structure is added to help representing large algorithms as well as to enable the analysis mechanism to identify functions/blocks in the graph. Such an identified block can then be seen as a single HCDFG that can be instantiated several times.

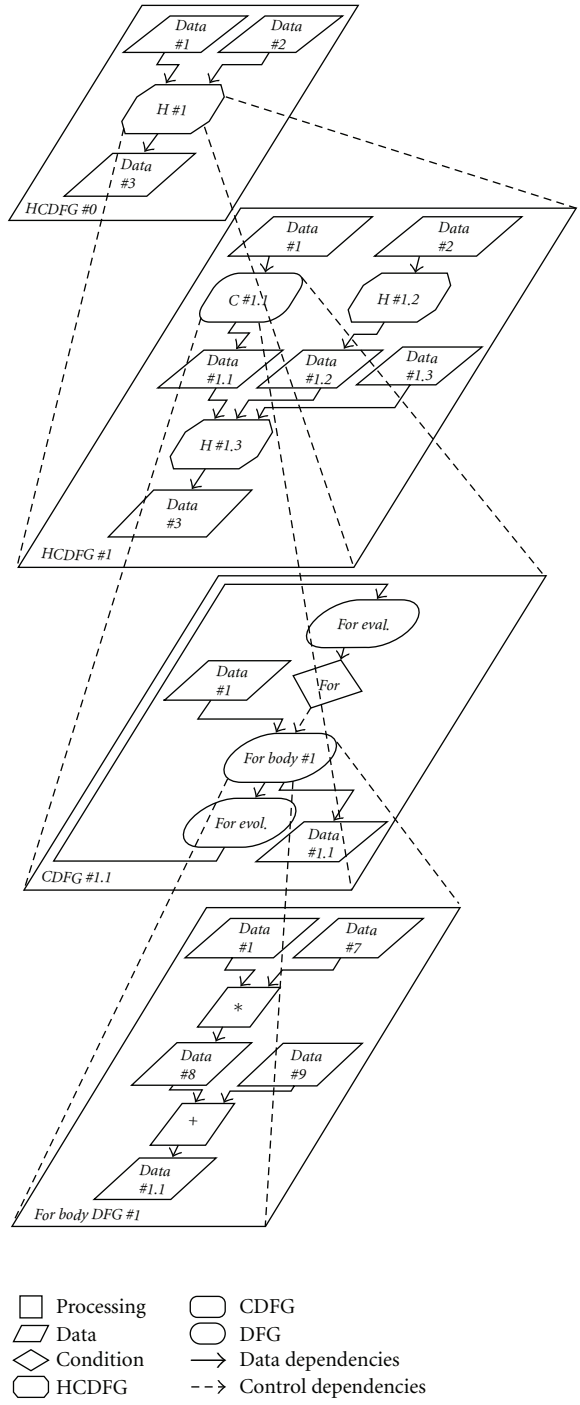


Figure 3: An overview of how the hierarchy in an HCDFG allows analysis of an algorithm on different levels and how the levels are related.

Figure 3 shows an example of a hierarchical control data flow graph.

In this work the design space exploration tool "Design-Trotter" is used as an engine for analysing the algorithms. The HCDFG model is used as "Design-Trotter's" internal representation.

The hierarchy of an HCDFG is shown in Figure 3. An HCDFG can consist of other HCDFGs, Control/Data flow graphs (CDFGs) and data flow graphs (DFGs) as well as elementary nodes (processing, memory, and control nodes). An HCDFG is connected via dependency edges. In this work we only explore the graph at levels above the DFGs, and therefore only concentrate on these when we define the graph types in what follows.

Let us consider the hierarchical control data flow graph, $G_{\text{HCDFG}} = (N_{\text{HCDFG}}, E_{\text{HCDFG}})$, where N_{HCDFG} arethe nodes denoted by $N_{\text{HCDFG}} = \{n_{\text{HCDFG}_1}, \ldots, n_{\text{HCDFG}_m}\}$ and the nodes are $N_{\text{HCDFG}} \in \{G_{\text{HCDFG}} | G_{\text{CDFG}} | G_{\text{DFG}} | \text{Data}\}$, meaning that the nodes in the G_{HCDFG} can be instances of its own type, encapsulated control data flow graphs, G_{CDFG}, encapsulated data flow graphs G_{DFG}, or data transfer nodes, *Data*. The last one is introduced to avoid the duplication of data representations in the hierarchy, when data is exchanged between the graphs. Thereby, data are only represented by their nodes and not by edges as it is common in many other types of DFGs.

The edges, E_{HCDFG}, connect the nodes such that $E_{\text{HCDFG}} = \{e_{n_{\text{HCDFG}_i}, n_{\text{HCDFG}_j}}\}$, where $i \neq j$ and represent the indexes of the nodes, $E_{\text{HCDFG}} \in \{\text{DD}\}$ and where every node can have multiple input and/or output edges. For the G_{HCDFG}, only data dependencies, DD, are allowed, and no control dependencies, CD.

In this way the HCDFG forms a hierarchy of encapsulated HCDFGs, CDFGs, and DFGs, connected via exchanging data nodes. The HCDFG can be seen as a container graph for other graph types such as the CDFG.

We can define the CDFG as $G_{\text{CDFG}} = (N_{\text{CDFG}}, E_{\text{CDFG}})$, where N_{CDFG} are the nodes denoted by $N_{\text{CDFG}} = \{n_{\text{CDFG}_1}, \ldots, n_{\text{CDFG}_m}\}$ and the nodes are $N_{\text{CDFG}} \in \{\text{CC} | G_{\text{HCDFG}} | G_{\text{DFG}} | \text{Data}\}$, where $\text{CC} \in \{\text{if}|\text{switch}|\text{for}|\text{while}|\text{do-while}\}$. In this way the G_{CDFG} is able to describe common control structures, where the actual data processing is encapsulated in either DFGs or HCDFGs. Again, the data exchange nodes are used to exchange data between the other nodes.

The edges, E_{CDFG}, connect the nodes such that $E_{\text{CDFG}} = \{e_{n_{\text{CDFG}_i}, n_{\text{CDFG}_j}}\}$, where $i \neq j$ and represent the indexes of the nodes. If $n_{\text{CDFG}_i} \in \text{CC}$ and $n_{\text{CDFG}_j} \in \{G_{\text{HCDFG}} | G_{\text{DFG}}\}$, then $\{e_{n_{\text{CDFG}_i}, n_{\text{CDFG}_j}}\} \in \{\text{CD}\}$, else $\{e_{n_{\text{CDFG}_i}, n_{\text{CDFG}_j}}\} \in \{\text{DD}\}$.

Beneath the control data flow graphs G_{CDFG}, the data flow graphs G_{DFG} exist but they are of no use in this work so we will not define them further here.

3.3. Calculating the cyclomatic complexity on CDFGs

Now that the HCDFG has been defined, we explain our proposed method for measuring the cyclomatic complexity on the CDFGs.

Since the cyclomatic complexity only considers the control structure in finding the number of independent paths in the algorithm, the DFG part of the algorithm is, as mentioned earlier, of no interest for this task because it only gives a single path. On the other hand, what is of interest is how the cyclomatic complexity is measured on the CDFGs

and HCDFGs which are built by the tool Design-Trotter. This leaves us with the following cases which are described in detail afterwards:

(i) If constructs,

(ii) Switch constructs,

(iii) For-loop,

(iv) While/do-while loops,

(v) Functions,

(vi) HCDFGs in parallel,

(vii) HCDFGs in serial sequence.

3.3.1. If constructs

"If constructs" case is represented as CDFGs, G_{CDFG}, where one node is a control node of type *if* (see Figure 4(a)). Before arriving at the control node, a condition evaluation node $n_{\text{eval}} \in \{G_{\text{HCDFG}} | G_{\text{DFG}}\}$ is traversed to calculate the boolean variable stored in n_{Data} (to maintain simplicity, these are not shown in Figure 4(a)) that is used in the condition node. If the variable is true, the algorithm follows the path through the true body node, $n_{\text{true}} \in \{G_{\text{HCDFG}} | G_{\text{DFG}} | \varnothing\}$. Else it goes to the false body node $n_{\text{false}} \in \{G_{\text{HCDFG}} | G_{\text{DFG}} | \varnothing\}$. Note that in some cases, either the true body or the false body does not exist, but it still gives a path. In this case, according to the cyclomatic complexity measure, the number of independent paths is

$$P(n_{\text{if}}) = P(n_{\text{true}}) + P(n_{\text{false}}) + P(n_{\text{eval}}) - 1. \quad (3)$$

The last part of (3), $+P(n_{\text{eval}}) - 1$ is included in case the evaluation graph is an HCDFG node.

3.3.2. Switch constructs

"Switch constructs" case is represented as CDFGs, G_{CDFG}, and is almost the same flow as the "if constructs" case discussed above. One node is a control node of switch type. Before arriving to the control node, a condition evaluation node $n_{\text{eval}} \in \{G_{\text{HCDFG}} | G_{\text{DFG}}\}$ is traversed. Depending on the output, the switch node leads the algorithm flow to the selected case node: $n_{\text{case}_i} \in \{G_{\text{HCDFG}} | G_{\text{DFG}}\}$. An example is shown in Figure 4(b). According to the cyclomatic complexity measure, the number of independent paths is as follows :

$$P(n_{\text{switch}}) = P(n_{\text{eval}}) - 1 + \sum_{i=1}^{N} P(n_{\text{case}_i}), \quad (4)$$

where N represents the number of cases, i the index to the corresponding node on which the paths are measured.

The same argument goes for the $P(n_{\text{eval}}) - 1$ part of (4); it is included in case the evaluation graph is an HCDFG node, but else it is omitted.

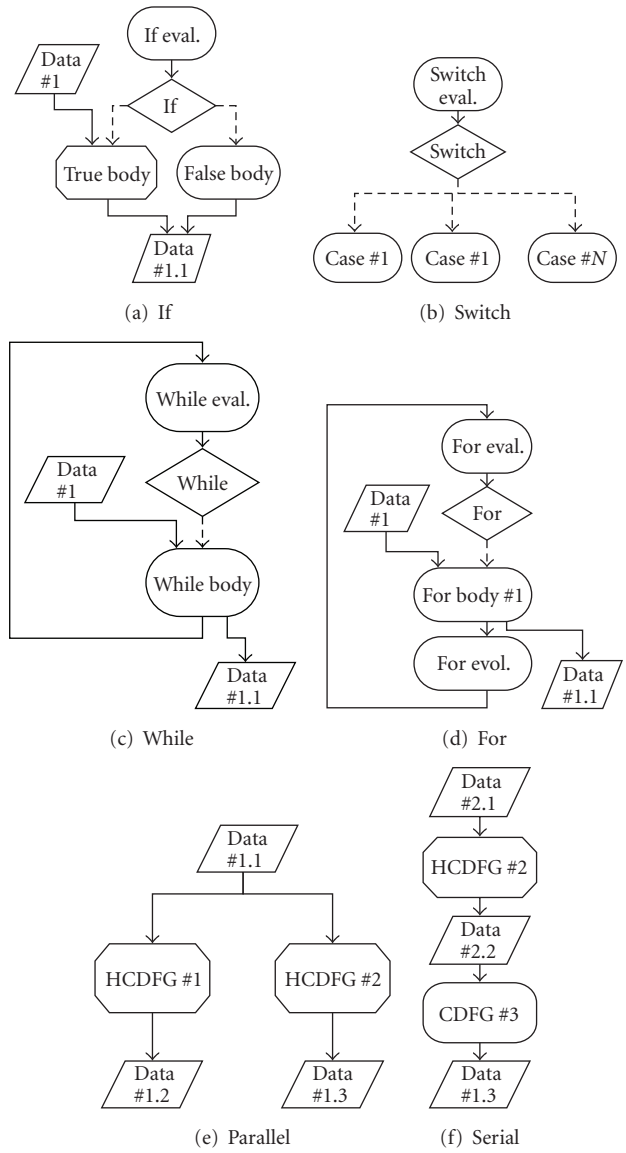


Figure 4: Overview of the different CDFGs and combined HCDFGs, on which the cyclomatic complexity values are measured. Between the (HC)DFGs there is a set of data exchange nodes which are here left out for simplicity. The symbols are similar to those presented in Figure 3.

3.3.3. For-loop

"For-loop" case is the most complex of the control structures. Strictly speaking, a "for loop" consists of three different parts: the evaluation body, the evolution body, and the for body, n_{eval}, n_{evol}, and $n_{\text{for-body}}$, respectively. The control node nfor, determines, based on the output from the evaluation graph, whether the flow should go into the "for loop" or leave it. The evolution node updates the indexes. Since each iteration of the graph needs to pass through the evaluation and evolution nodes, the number of independent paths is calculated as

$$P(n_{\text{for}}) = P(n_{\text{for-body}}) + P(n_{\text{eval}}) - 1 + P(n_{\text{evol}}) - 1. \quad (5)$$

In many cases, the evaluation and evolution part of the "for loop" are quite simple indexing functions, meaning that $n_{\text{eval}} \in \{G_{\text{DFG}}\}$, $n_{\text{evol}} \in \{G_{\text{DFG}}\}$, will leave $P(n_{\text{for}}) = P(n_{\text{for-body}})$. The "for loop" is illustrated in Figure 4(d).

3.3.4. While loops and do-while loops

"While loops" and "do-while loops" cases are described jointly since it is only the entry to the loop structure that separates them and their cyclomatic complexity are equivalent. The "while loops" consist of two main parts: the while body $n_{\text{while-body}} \in \{G_{\text{HCDFG}}|G_{\text{DFG}}\}$, and the while evaluation $n_{\text{eval}} \in \{G_{\text{HCDFG}}|G_{\text{DFG}}\}$. This is illustrated in Figure 4(c). Deciding whether to continue looping is decided by the control node $n_{\text{while}} \in \{\text{while}\}$ based on the output of the n_{eval}. Similarly to the "for loop," each iteration of the graph needs to pass through the evaluation nodes, so the number of independent paths can be calculated as

$$P(n_{\text{while}}) = P(n_{\text{while-body}}) + P(n_{\text{eval}}) - 1. \tag{6}$$

In many cases, the evaluation part of the while loop is a set of simple test functions, meaning that $n_{\text{eval}} \in \{G_{\text{DFG}}\}$, which leaves the $P(n_{\text{while}}) = P(n_{\text{while-body}})$.

3.3.5. Functions

The goal is to identify the number of independent paths in the algorithm/system. For this, reuse in terms of functions/blocks of code is important. When all independent paths through a function are known, reuse of this function does not change the number of independent paths in the system. From an implementation point of view, such functions represent an entity where the paths only need to be implemented once. In HCDFGs, a function/block can be seen as an encapsulated G_{HCDFG}. Therefore, the number of independent paths in function/blocks of reused code should only count once. The paths can be calculated as

$$P(n_{\text{HCDFG}_{\text{function}}}) = \begin{cases} 0 & \text{if reuse,} \\ P(n_{\text{HCDFG}}) & \text{else.} \end{cases} \tag{7}$$

3.3.6. HCDFGs in parallel and serial

Knowing how to handle all the HCDFGs that are identified for reuse (function), together with all the CDFGs, does not give it all. How the hierarchy of graphs should be combined is also of interest. For a parallel combination of two or more HCDFGs/CDFGs, as shown in Figure 4(e), the increase in the number of independent paths is then additive. The number of paths can be calculated as

$$P(n_{\text{HCDFG}_{\text{Parallel}}}) = \sum_{i=1}^{N} P(n_{\text{HCDFG}_i}), \tag{8}$$

where N represents the number of nodes in parallel, i the index to the corresponding node where the paths are measured.

For serial combination of two or more HCDFGs and/or CDFGs, the number of independent paths is a combination of the independent paths of the involved HCDFGs/CDFGs. Remembering that there always needs to be one path through the system, the number of independent paths in a serial combination, is given as

$$P(n_{\text{HCDFG}_{\text{Serial}}}) = \sum_{i=1}^{N} P(n_{\text{HCDFG}_i}) - (N-1), \tag{9}$$

where N represents the number of nodes in serial, i the index to the corresponding node where the paths are measured.

An example of serial combination is shown in Figure 4(f). The number of independent paths for the entire algorithm, $(P(n_{\text{HCDFG}_{\text{Alg}}}))$, is equivalent to the top HCDFG node which includes all the independent paths of its subgraphs.

3.4. Experience impact

The experience of the designer has an impact on the challenge that he/she is facing when developing a system. A radical example is when a beginner and a developer with ten years of experience are asked to solve the same task. They will not see equal difficulty in the same task, and thereby do not need to put the same effort into the development.

Experience is influenced by many parameters but in this work we only focus on the time the developer has worked with the implementation language and the target architecture.

The impact of experience is a factor that slowly decreases over time: consider a new developer, the experience that he/she obtains in the first months working with the language, and architecture improves his/her skills significantly. On the other hand, a developer who has worked with the language and architecture for five years, for example, will not improve her/his skills at the same rate by working an extra year. The impact from the experience is therefore not linear but tends to have a negative acceleration or inverse logarithmic nature, with dramatic change in impact in the beginning, progressing towards little or no change as time increases.

In literature, for example, [24], many studies try to fit historical data to models. An example of a model is a power function with negative slope or a negative exponential function. From the vast variety of models that has been proposed over the years, the only conclusion that can be drawn is that there are multiple curvatures, but they all appear to have a negative accelerating slope, which tends to be exponential/logarithmic.

In order to get the best possible outset for predicting the implementation effort, it is of vital importance to obtain some data of the developers' experiences, and also how they performed in the past. The parameters involved in the experience curve can then be trimmed to create the best possible fit. However, it has not been the purpose of this work to select the perfect nature for a learning curve nor to evaluate the accuracy of such one. The learning curve will be adapted to the individual developers, and as the model is used in subsequent projects, its accuracy will progressively improve. As a consequence, the experience here is only

intended as an element in modelling the complexity and thereby a means for more accurate estimates.

For the experiments in this study we have chosen to use the following model:

$$\eta_{\text{experience}}(\text{Dev}) = \frac{1}{\alpha \log(\text{Experience}(\text{Dev}) + \beta)}, \quad (10)$$

where α and β are trim parameters which can be used to optimise the curve to fit reality, Experience is the number of weeks which the developer, Dev, has worked with the language and architecture. Figure 5 depicts the shape of the experience model.

In this work,our initial experiments have shown that setting $\alpha = 1$ and $\beta = 1$ makes our model sufficiently general, and therefore we have not further investigated the tuning of these two parameters.

4. RESULTS

In order to verify the hypothesis, a classical test has been conducted. The test is dual phased and consists of (i) a training phase using a first set of real-life data, during which the hypothesis is said to be true, and (ii) a validation phase during which a second set of real-life data is used to evaluate whether the hypothesis holds true or not.

4.1. *Phase one—training*

The real-life data used as training data originate from two different application types that are both developed as academic projects in universities in France. The first application is composed of five different video processing algorithms for an intelligent camera, which is able to track moving objects in a video sequence. The second application is a cryptographic system, able to encrypt data with different cryptographic/hashing algorithms, that is, MD5, AES and SHA-1. The system consists of one combined engine [25] as well as individual implementations. These projects were selected since they all follow the methodology of using a behavioural specification in C, as a starting point for the VHDL implementation. Common to this data is that none of the developers has made the behavioural specification in C. For the cryptographic algorithms the behavioural specification comes from the standards, and the video algorithms were based on a previous project.

Using the behavioural description as the starting point of the experiment, the exercise consists of studying the relationship between the complexity of the algorithms (as defined in Section 3) and the implementation effort (i.e., time) required to implement them in VHDL (including testbed and heuristic tests).

The developers involved in these projects have all been Master and Ph.D. students with electrical engineering backgrounds but no VHDL background other than what they obtained during their studies, see Table 2. All developers were taught VHDL by other instructors than the authors, but at our university. Table 3 summaries the training data.

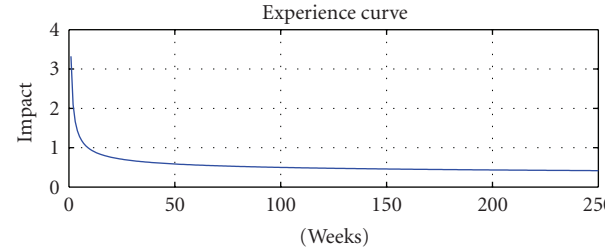


Figure 5: An example of how the lack of experience impacts the difficulty the engineers are facing.

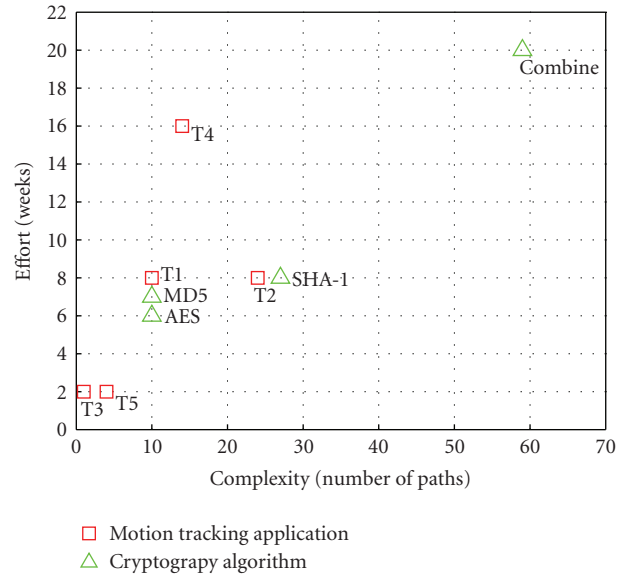


Figure 6: Relation between the implementation effort (number of weeks) and the not corrected complexity (as defined in Section 3).

Figure 6 shows the relation between the implementation effort and the measured complexity for the individual algorithms. Please note that in this graph the complexity values are not yet corrected for the designers' experience.

A first examination of the data points indicates a possible relation between some of them. However many other points are located far away from any relation. These data are not corrected for the designers' experience and, as earlier mentioned, we strongly believe that the experience of the individual designer has a nonnegligible influence on the development time. If we inspect the data more thoroughly, it is clear that the points of greatest divergence are those implementations where the developers have very limited knowledge and experience with the VHDL language.

Applying the proposed equation (10) (nonlinear) experience transform onto the data, results in a significantly different picture as depicted in Figure 7. A clear trend toward a relation is now visible in the plotted data. From the COCOMO II project [8], it is known that the relationship between the implementation time and the complexity measure (in their case lines of code, LOC) can be expressed as a

Table 1: Line of code, area, and time constraints for the validation data.

Algorithm	SS1	SS2	SS3	SS4	SS5	SS6	Ethernet	App 4
Dev. Time (weeks)	3.6	6.4	2.4	16.4	12	17.2	16	2
LOC-VHDL	994	1195	776	1695	760	2088	3973	232
Slices	564	2212	382	888	372	2171	3372	750
FlipFlops	913	2921	1290	1366	1208	2077	6149	942
LUTs	997	3157	6453	1569	6443	3458	18255	567
Time Constraint. (ns)	112	128	360	112	360	248	696	56

Table 2: Facts about the developers. Developers for training data (top) and validation data (bottom).

Developer	Education	Years in the domain
Dev 1	Ph.D. stud.	0
Dev 2	Stud. (EE)	0
Dev 3	Stud. (EE)	0
Dev 4	Stud. (EE)	0
Dev 5	BSc.EE.	9
Dev 6	MSc.EE.	15
Dev 7	MSc.EE.	9
Dev 8	MSc.EE.	8
Dev 9	MSc.EE.	8

Table 3: Training data (top) and validation data (bottom). Algorithms are related to the developers and their experience at the given time. Complexity is not corrected.

Algorithm	Complexity	Developer	Dev. Exp.
T1	10	Dev 1	2
T2	24	Dev 1	10
T3	12	Dev 1	18
T4	14	Dev 2	1
T5	4	Dev 1	20
MD5	10	Dev 3	1
MD5	10	Dev 4	1
AES	10	Dev 4	8
SHA-1	27	Dev 4	14
Combined	59	Dev 4	14
SS1	25	Dev 6, 7	150
SS2	35	Dev 5	150
SS3	17	Dev 5, 6, 7, 8	150
SS4	50	Dev 6	6
SS5	29	Dev 7	3
SS6	25	Dev 5, 6, 7	3
Ethernet app	60	Dev 5, 6, 7, 8, 9	150
App 4	9	Dev 6	150

power function with a weak slope. We showed its nature in (1), and with correction for experience it becomes

$$\text{Effort} = A \times \eta_{\text{experience}}(\text{Dev}) \times P\left(n_{\text{HCDFG}_{\text{Alg}}}\right)^{b}. \quad (11)$$

The parameters A and b are found, via a least square (LS) fit on our training data, to be $A = 0.226$ and $b = 1.103$. In

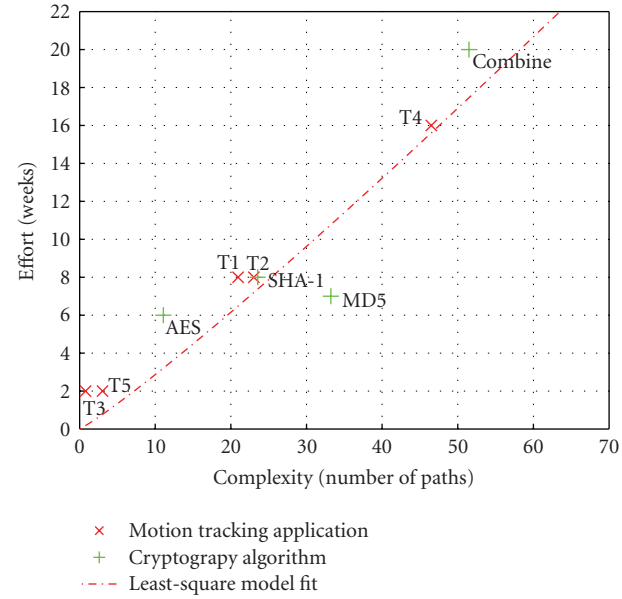


Figure 7: Relation between the implementation effort (number of weeks) and the complexity corrected according to the designers' experience model as shown in Figure 5.

Figure 7 the dashed line illustrates the relationship, with the parameters given above.

4.2. *Phase two—validation*

After having elaborated on a model based on the training data, we proceeded with the validation of its correctness. For this, a new set of data provided by ETI A/S, a Danish SME, is used. The dataset originates from a networking system and consists of Ethernet applications that have been implemented on an FPGA, as well as corresponding testbeds. This Ethernet application is part of an existing system with which it requires interaction. Table 1 shows additional implementation information with regards to these applications. The system is a real-time system with hard-time constraints and all algorithms were implemented as to meet these constraints. Similar to the training data, the development flow for this application has been as follows: a behavioural C++ model of the application has been constructed before the implementation on the FPGA architecture. The behavioural model has been developed by developers separate to those undertaking the implementation. The developers responsible

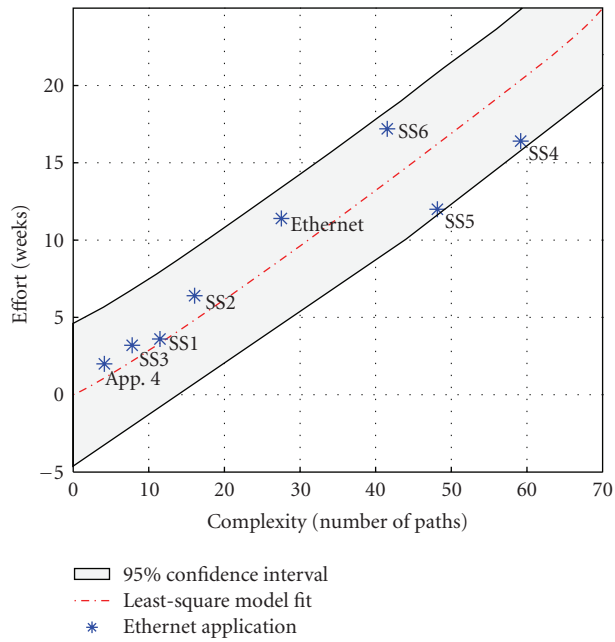


Figure 8: Validation data plot: relation between implementation effort (number of weeks) and complexity, corrected according to the designers' experience model.

for the implementation have obtained their skills in VHDL from a professional course with no relation to our university in Denmark.

The time spent on the implementation process covers: the design and implementation of the VHDL code of the functionalities and testbed as well as the tests of the different modules in the applications. This data is shown in the lower part of Table 3. The time data originate from the company's internal registration for the project, and correspond therefore to the effective time used.

The relation between implementation effort and complexity is plotted in Figure 8. It can be seen that this data, corrected for the designers' experience ($*$) closely follows the model derived from the training data (dashed line). Figure 8 also shows the 95% confidence interval, indicating that, with 95% confidence, future predictions of implementation effort will lie within this interval, given that the model holds true.

Comparing the predicted effort (dashed line) to the real effort ($*$), indicates that there is an estimation error. The values are also shown in Table 4, The average estimation error is 0.2 week with a variance of 8. In the next section, we discuss the validity of the model.

4.3. Validity discussion

Estimating the effort required in implementing an algorithm into hardware involves many parameters. We discussed a number of these parameters in Section 1.2, but could not include them all in this study. The proposed model is therefore devised from the idea of the relation between implementation effort and number of linear-independent paths.

Table 4: Development time and estimated development time measured in weeks together with the error.

Algorithm	Dev. time	Est. dev. time	Error
SS1	3.6	3.3	0.3
SS2	6.4	4.8	1.6
SS3	3.2	2.2	1
SS4	16.4	20.3	−3.9
SS5	12	16.2	−4.2
SS6	17.2	13.8	3.4
Ethernet app	11.4	8.8	2.6
App 4	2	1.1	0.9
Mean (variance):			0.2 (8)

To validate the model, a classical two-phased hypothesis test has been performed and the validity of this test depends on the following important factors: (i) the independence between training and validation data; (ii) the volume and variety of the experiments.

In the first instance, not only different applications were used for training and validation data, but in addition the developers had no relation in terms of education, nationality, work, and so forth. Moreover, the validation data has not been measured before the model was trained. All this strengthens the validity of the results. The only potential connection is that some of the developers who have been involved in the implementation of the training and validation data have also been included within those interviewed. However, this accounts for a minority and we see this as a minimal risk.

Secondly, we should ideally have had a large volume and variety of experimental data for training and validation. However, our set of data originates from a single company and a few developers. So strictly speaking we can only conclude that this model applies to the specific SME setup involved in the study and partially to the academic environment studied.

In order to generalise our model, more cases of validation are needed. However, obtaining all the statistical data for this new methodology is time consuming. We would therefore like to remind the reader that this paper proposes a methodology for estimating implementation effort and the validation of the model concentrates on illustrating its usefulness. Looking at the graphs, we can determine a clear trend in the results. The curve identified in the training data is sustained for the validation data as well: they both fall in line with the underlying rationale, and we are quite confident in the strength of the proposed model.

The results clearly show the necessity for the proposed correction function; the proposed logarithmic nature works well, even though the correction function has not been trimmed to fit the individual developers due to the lack of available data. In this light, our approach must be seen as the engine of a global methodology for the management of design projects, that impose a systematic registration of man-power. With such a registration, a database of the developers' experience can easily be constructed and the

correction function can be trimmed to fit the companies' individual designers. Several iterations of this process would provide convergence towards a more precise estimation of the implementation effort.

The limited data set on which the model is constructed also limits the complexity window to which this model can be applied: having no algorithm with a corrected complexity value larger than 51, extrapolating the model further would weaken the current conclusion. More training data, from larger and more varied projects would allow for a more refined model.

Nevertheless, the results described in this paper are very encouraging with all the real-life cases that we have examined and we are reasonably confident that this model can easily be applied to other types of applications.

5. CONCLUSION

The contribution presented in this paper is a metric-based approach for estimating the time needed for hardware implementation in relation to the complexity of an algorithm. We have deduced that a relationship exists between the number of linear-independent paths in the algorithm and the corresponding implementation effort. We have proposed an original solution for estimating implementation effort that extends the concept of the cyclomatic complexity.

To further improve our solution, we developed a more realistic estimation model that includes a correction function to take into account the designer's experience.

We have implemented this solution in our tool design Trotter of which the input is a behavioural description in C language and the output is the number of independent paths. Based on this output and the proposed model, we are able to predict the required implementation effort. Our experimental results, using industrial Ethernet applications, confirmed that the data, corrected for the designers' experience, follows the derived model closely and that all data falls inside its 95% confidence interval. Using this method iteratively paves the way for an implementation effort estimator of which the accuracy improves continuously after each project.

REFERENCES

[1] D. Blaza, "Embedded systems design state of embedded market survey," Tech. Rep., CMP Media, New York, NY, USA, 2006.

[2] R. Nass, "An insider's view of the 2008 embedded market study," Tech. Rep., CMP Media, New York, NY, USA, 2008.

[3] "Workshop for Danish smes developing embedded systems co-organized by the Danish technological institute, the center for software defined radio (csdr) and the center for embedded software systems (ciss)," Nyhedsmagasinet Elektronik & Data, Nr.1 2008, Aarhus, Denmark, 2008.

[4] O.-H. Kwon, "Keynote speaker: perspective of the future semiconductor industry: challenges and solutions," in *Proceedings of the 44th Design Automation Conference (DAC '07)*, San Diego, Calif, USA, June 2007.

[5] S. McConnell, *Software Estimation: Demystifying the Black Art*, Microsoft Press, Wasthington, DC, USA, 2006.

[6] R. Abildgren, A. Saramentovas, P. Ruzgys, P. Koch, and Y. Le Moullec, "Algorithm-architecture affinity—parallelism changes the picture," in *Proceedings on the Design and Architectures for Signal and Image Processing*, Grenoble, France, November 2007.

[7] M. A. Honey, "The interview as text: hermeneutics considered as a model for analysing the clinically informed research interview," *Human Development*, vol. 30, pp. 69–82, 1987.

[8] B. W. Boehm, C. Abts, A. W. Brown, et al., *Software Cost Estimation with COCOMO II*, Prentice-Hall, Upper Saddle River, NJ, USA, 2000.

[9] B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

[10] T. Jones, *Programming Productivity*, McGraw-Hill, New York, NY, USA, 1986.

[11] G. C. Low and D. R. Jeffery, "Function points in the estimation and evaluation of the software process," *IEEE Transactions on Software Engineering*, vol. 16, no. 1, pp. 64–71, 1990.

[12] A. J. Albrecht, "Measuring application development productivity," in *Proceedings of the IBM Application Development Symposium*, pp. 83–92, Monterey, Calif, USA, October 1979.

[13] D. R. Jeffery, G. C. Low, and M. Barnes, "Comparison of function point counting techniques," *IEEE Transactions on Software Engineering*, vol. 19, no. 5, pp. 529–532, 1993.

[14] W. Fornaciari, F. Salice, U. Bondi, and E. Magini, "Development cost and size estimation starting from high-level specifications," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign*, pp. 86–91, Copenhagen, Denmark, April 2001.

[15] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[16] D. L. Lanning and T. M. Khoshgoftaar, "Modeling the relationship between source code complexity and maintenance difficulty," *Computer*, vol. 27, no. 9, pp. 35–40, 1994.

[17] T. J. Walsh, "Software reliability study using a complexity measure," in *Proceedings of the National Computer Conference (NCC '79)*, vol. 48, pp. 761–768, AFIPS Press, New York, NY, USA, June 1979.

[18] S. P. VanderWiel, D. Nathanson, and D. J. Lilja, "Complexity and performance in parallel programming languages," in *Proceedings of the 2nd International Workshop on High-Level Programming Models and Supportive Environments (HIPS '97)*, pp. 3–12, Geneva, Switzerland, April 1997.

[19] M. Mastretti, M. L. Busi, R. Sarvello, M. Sturlesi, and S. Tomasello, "VHDL quality: synthesizability, complexity and efficiency evaluation," in *Proceedings of the European Design Automation Conference with EURO-VHDL (EURO-DAC '95)*, pp. 482–487, IEEE Computer Society Press, Brighton, UK, September 1995.

[20] I. Feghali and A. H. Watson, "Clarification concerning modularization and mccabe's cyclomatic complexity," *Communication of the ACM*, vol. 37, no. 4, pp. 91–94, 1994.

[21] B. Henderson-Sellers, "Modularization and mccabe's cyclomatic complexity," *Communication of the ACM*, vol. 35, no. 12, pp. 17–19, 1992.

[22] Y. Le Moullec, N. B. Amor, J.-P. Diguet, M. Abid, and J.-L. Philippe, "Multi-granularity metrics for the era of strongly personalized SOCs," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, pp. 674–679, Munich, Germany, March 2003.

[23] Y. Le Moullec, J.-P. Diguet, N. B. Amor, T. Gourdeaux, and J.-L. Philippe, "Algorithmic-level specification and characterization of embedded multimedia applications with design

trotter," *The Journal of VLSI Signal Processing*, vol. 42, no. 2, pp. 185–208, 2006.

[24] A. Heathcote, S. Brown, and D. J. Mewhort, "The power law repealed: the case for an exponential law of practice," *Psychonomic Bulletin & Review*, vol. 7, no. 2, pp. 185–207, 2000.

[25] S. Ducloyer, R. Vaslin, G. Gogniat, and E. Wanderley, "Hardware implementation of a multi-mode hash architecture for MD5, SHA-1 and SHA-2," in *Proceedings on the Design and Architectures for Signal and Image Processing Workshop (DASIP '07)*, Grenoble, France, November 2007.