Software quality metrics and their impact on embedded software

Marcio F. S. Oliveira, Ricardo Miotto Redin, Luigi Carro, Luís da Cunha Lamb, Flávio Rech Wagner
Informatics Institute, UFRGS, Brazil
{mfsoliveira, rmredin, carro, lamb, flavio}@inf.ufrgs.br

Abstract

Although many improvements for software development are proposed by software engineers, the embedded system community faces a hard task in applying these improvements to software development, due to the strong dependence between software and hardware in embedded systems, which raises a trade-off between software quality, measured by traditional metrics, and optimization for a specific platform. Traditional software quality concerns reuse, abstraction, coupling, and coherence. Such concepts are usually in conflict with performance, memory footprint, and other physical metrics used for design evaluation in embedded systems. The purpose of this work is to evaluate the relationship between quality metrics for software products and physical metrics for embedded systems, in order to guide an embedded designer in selecting the best alternative design during design space exploration already at the model level. After the experiments, we could analyze the correlation between these metrics and highlight the behavior for quality and physical metrics, which help us to better understand the trade-off between reuse and optimization.

1 Introduction

The ever growing hardware capabilities, following the Moore’s law, allow performance improvements and addition of new functionality into current devices and systems. In order to reduce the productivity gap, which is the difficulty of a designer in using the available hardware capability, new hardware design methodologies improve abstraction and reuse in order to allow a designer to expend most of the development time in the effective design of new products, instead of wasting resources with implementation details or rework of pre-designed functions. It has been reported that the hardware reuse factor grew from 10% in 1991 to 90% in 2001 [21].

Meanwhile, software engineers have been improving the software design process, and new methods were proposed for all software development steps, from requirements specification until test. New programming paradigms have arisen, such as Object-Orientation and Aspect-Orientation, as well as new development methods such as Model-driven Engineering. As a key factor of any engineering process is the measurement of its characteristics, different metrics to gauge and improve the quality of software products have been also proposed [1]. Such metrics are concerned to concepts such as reuse, abstraction, cohesion, coupling, complexity, and other software attributes.

Exploiting the technology advances of today’s hardware, the functionality of embedded systems are increasingly implemented in software. The amount of embedded software in a product increased from 1 or 2 Mbytes 8 years ago to 16 up to 128 Mbytes in today’s embedded systems, with tens of applications and hundreds of software components [20].

Unfortunately, the embedded system community does not benefit from the new advances of software methodologies as it does for hardware development. As a result, the most critical challenge for SoC design is the software development, which now accounts for 80% of embedded system development [11]. Even applying many software engineering methods in the embedded software development, it is considered in industry that the current practice of embedded software development is unsatisfactory [8].

Currently, the main metrics used in embedded system design are physical ones, such as performance, memory, energy, power, size, and weight, guided by design constraints. Other important and related metrics are reuse, time-to-market, and price. Although many methodologies approach the physical metrics by proposing estimation or simulation tools, the reuse and time-to-market factors are approached only through design methods without direct or indirect evaluation. However, quality metrics provided by software engineers for general-purpose systems have been successfully applied to improve the software quality, obtaining as result improvements in reuse and time-to-market.

The purpose of this work is to identify the correlation of quality metrics for traditional software products with the so important physical metrics for embedded systems. Our goal is to: i) verify the applicability of these metrics during the evaluation of alternative designs, for design space exploration purposes; ii) improve the abstraction and reuse in embedded system design; and iii) improve the design space exploration by keeping the original design flexible and abstract, even though performing optimization steps.
After the experiments, we could identify the strong correlation of quality metrics for traditional software products and physical metrics for embedded systems. By using the both set of metrics to evaluate alternative solution during the design space exploration step, we conclude that although a penalty for using object orientation and software engineering best-practices is inevitable, it can be worthwhile in terms of quality and reuse of components if software quality metrics are applied together with physical ones looking for the best balance between then.

The remaining of this paper is organized as follows. Section 2 introduces basic concepts. Section 3 discusses related work. The experimental setup used in this work is presented in Section 4, which extends a design space exploration scenario, published in [18], by extracting quality metrics for software products and comparing these ones to physical metrics extracted from alternative design solutions of the same application. Section 5 illustrates the evaluation results, highlighting the correlation of these metrics on embedded system physical properties. Section 6 draws main conclusions and introduces future work.

2 Background

As the key of any engineering process is the measurement, research efforts have provided many measures and metrics in order to evaluate processes, software products, and projects and to guide design decisions. A set of important metrics was selected from [9] and [15]. Since there is no well-defined or widely accepted metrics classification, we grouped these metrics by the attribute in which the metric refers to, in order to facilitate the presentation. The applied classification and each metric description are as follow.

**Coupling:** It measures the relationship between components, including calls, and number of instances. These metrics regards to the reduced encapsulation, potential reuse, and the understanding and maintainability. Higher values of these metrics lead to an application that is poor in encapsulation, reuse and maintainability:

- **Afferent Coupling (Ca):** The number of classes outside a package that depend on classes inside the package;
- **Efferent Coupling (Ce):** The number of classes inside a package that depend on classes outside the package;
- **Instability (I):** Ce / (Ca + Ce). This metric indicates the need of changes in any scope, if there was a change in other system scope. A value of zero indicates a completely stable package;

**Cohesion:** Measure the degree to which the elements of a scope are functionally related. A strongly cohesive module implements functionality that is related to one feature of the software and requires little or no interaction with other modules. High values of cohesion were desirable (and by consequence, small values of Lack of Cohesion of Methods) because it means that components are architecturally and logically well defined:

**Lack of Cohesion of Methods (LCOM):** A measure for the Cohesiveness of a class, which is calculated with the Henderson-Sellers method. If \( m(A) \) is the number of methods accessing an attribute A, calculate the average of accesses for all attributes, subtract the number of methods m, and divide the result by \( (1-m) \). A low value indicates a cohesive class, while a value close to 1 indicates a lack of cohesion and suggests that the class might better be split into a number of (sub)classes;

**Extendibility and reuse:** these metrics evaluate the possible reuse of a scope and the capacity of it to be extended:

- **Abstractness (A):** Indicates the capacity of the system or component to be extended. It is calculated by the number of abstract classes (and interfaces) divided by the total number of types in a package. A high value of this metric lead to more reusable components and a lower value mean a concrete solution;
- **Normalized Distance from Main Sequence (Dn):** Calculated by \( | A + 1 - 1 | \), this number should be small, close to zero, for a good packaging design;
- **Depth of Inheritance Tree (DIT):** Distance from class Object in the inheritance hierarchy. A high values of this metric means very specialized inheritance hierarchy in which components have a small and well defined function. Small values mean that the solution is flat and less reusable;
- **Number of Overridden Methods (NOVM):** Total number of methods in the selected scope that are overridden from an ancestor class. High values translate into a more flexible solution (with interfaces and abstract classes) or reengineering of a component;

**Population (or size) metrics:** It is the metrics measure the system in terms of attributes, methods, classes. These metrics are also associated to complexity. These were primary metrics in the sense that their values were used to calculate other metrics. In general, higher values of these metrics means an increase in memory footprint, lower performance and a more complex solution. Although, the distribution between higher and lower values between them impact much more in the dynamic behavior of the application:

- **Number of Attributes (NOA):** Total number of attributes (non-static) defined in the selected scope. High values of this metric lead to a better application locality;
- **Number of Classes (NOC):** Total number of classes defined in the selected scope. High values mean high memory footprint, higher complexity but high
modularity too. Lower values can lead to poor application design but better system physical proprieties;

- **Number of Interfaces (NOI):** Total number of interfaces defined in the selected scope. High number of interfaces lead to a more flexible and adaptable solution and more reuse;

- **Number of Methods (NOM):** Total number of methods defined in the selected scope. Higher number of methods means more modularization (assuming two solutions with the same Method of Lines of Code) and this lead to a more readable solution but also mean more method calls (that can greatly reduce performance);

- **Number of Packages (NOPK):** Total number of packages defined in the selected scope. This metric raise the same concerns about the application than Number of Classes;

- **Number of Parameters (NOP):** Total number of parameters in the selected scope. Raising the number of parameters as the number of attributes increases the locality of the application because these type of variables are allocated on the stack and it is easier to access and update. Trading-off short and heavy used method calls for parameters or fields can lead to a great performance improvement;

- **Number of Static Attributes (NOSA):** Total number of static attributes in the selected scope. Raising the number of Static Attributes translates into memory footprint increase and more complexity on the application;

- **Number of Static Methods (NOSM):** Total number of static methods in the selected scope. Static calls are faster than dynamic ones, translating into a performance increase. But the abuse of static methods lead to a brittle solution that does not improve the reuse factor;

- **Total Lines of Code (TLOC):** Total lines of code in the selected scope. It only counts non-blank and non-comment lines inside method bodies. Raising this metric means the solutions is becoming more complex, have a larger memory footprint among other disadvantages;

**Complexity:** These metric measures the hardness to understand or express the problem/algorithm. These are related to alternative execution flow, element granularity/hierarchy and nested execution.

- **McCabe Cyclomatic Complexity (McCabeCC):** It counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the?: ternary operator, as well as the && and || conditional logic operators in expressions), this metric is incremented by one. It is calculated for methods only. High values of this metric means that the application is very complex or at least that it have a large number of alternative flows;

- **Method Lines of Code (MLOC):** This metric indicates the number of non-blank and non-comment lines inside method bodies. Method lines of code are directly proportional to the program memory. Higher values of this metric lead to more memory footprint but also translate into more complex solution;

- **Nested Block Depth:** The depth of nested blocks of code. More nested blocks lead to worse readability and more complex solutions;

- **Weighted Methods per Class (WMC):** Sum of the McCabe Cyclomatic Complexity, for all methods in a class. High value of this metric translates into a more complex solution in the local scope.

There are wide numbers of software metrics, then we select these set of metrics, because they are ones of more traditional and there are several tools to automatically extract these metrics from source code and UML models. Surely, other metrics could be applied in this study, and important metrics are planned to be applied in future experiments.

### 3 Related Work

There are many methodologies for evaluating alternative designs for the purpose of design space exploration. These methodologies perform system evaluation at different abstraction levels. Examples of methodologies especially suited for embedded and real-time systems are the Metropolis [6], which performs system evaluation using UML models, and SPADE [13], which extracts physical metrics from C/C++ source code. At the lower levels, Cinderella [12] estimates performance from compiled C code. For different abstraction levels, MILAN [2] uses different tools such as HiPerE [17] and specialized simulators. These tools are used to support design space exploration, whose main focus is the mapping from the application into the platform.

Other approaches use physical metrics to evaluate alternative designs for platform tuning, such as Platune [7] and [3]. There are other specific approaches to support software optimization. SPE/PASA provides evaluation support for optimization of physical metrics analyzing UML models. In [14], a tool is presented that can automatically change the application code in order to increase its performance with a minimal increase in memory footprint. The tool called DESEJOS [14], tries to transform as many dynamic objects into static ones, in order to enhance performance and decrease memory footprint.
In [5], the object-oriented programming style is evaluated in terms of both performance and power for embedded applications. A set of benchmark kernels, written in C and C++, is compiled and executed on an embedded processor simulator. The work shows that object-oriented programming can significantly increase both execution time and power consumption.

In the software engineering literature there are many proposals of metric frameworks to evaluate quality of the software products. Traditional information about metrics can be obtained in [9] and [15]. A survey of object-oriented metrics was published in [26]. A recent empirical study of object-oriented metrics is presented in [1]. The work applies quality metrics, proposed by different researches, to three different projects and studies the relationship between these metrics.

Few works regarding quality for software products and physical metrics for embedded systems is found. In [27] the authors describe the results of an experiment where four different mobile devices running Role Playing Games applications were analyzed in terms of software quality metrics and performance. The study shows that through reuse of platform and/or software components the development effort can be greatly reduced without compromising the performance.

Our work focuses on measuring the impact of “design for reuse” by using a more readable and maintainable design, without forgetting the physical constraints of embedded systems. We are trying to demonstrate that although a penalty for using object orientation and software engineering best-practices is inevitable, it can be worthwhile in terms of quality and reuse of components if software quality metrics are applied together with physical ones looking for the best balance between them.

4 Experimental setup

This section presents the experimental setup applied in this study, where four alternative modeling solutions for the same application are proposed in order to explore different design alternatives. These models were specified using UML and, after that, implemented using Java for the target platform. For every alternative design, a synthesis tool [10] was used to obtain the hardware description and the Java byte codes for the application. From the final implementation, in Java byte codes, we extracted the physical metrics by using a cycle-accurate simulation [4], while from the Java source code we extracted the quality metrics for software products by using the Eclipse Metrics plug-in [16]. Finally, the results for physical metrics and software quality metrics were compared in order to evaluate the relationship between them under the embedded system development view. Figure 1 illustrates the experiment flow.

This experiment extends the ones presented in [18] by adding information to guide the designer during the design space exploration process. With the added information we intend to assess the challenge between reuse and optimization, improve the embedded software quality, and also reduce the time-to-market.

The case study consists of a real-time embedded system dedicated to the automation and control of an intelligent wheelchair that helps people with special needs. This wheelchair has several functions, such as movement control, collision avoidance, navigation, target pursuit, battery control, system supervision, task scheduling, and automatic movement. This experiment focuses only on the wheelchair movement control use case, which is essential to the system and incorporates critical hard real-time constraints.

To implement the movement control, several components provided by the target platform can be reused, such as a mathematical library to solve the control equations, a real-time Java API, and RTOS components [25]. Consequently, the platform characteristics were taken into account by the costs related to the real-time scheduler and other platform services. Furthermore, the application was mapped to a single processing unit, a multi-cycle-RTC Femtojava [25] processor, in order to concentrate the experiments on the design space that can be explored when a designer proposes different UML modeling solutions.

The movement controller function has two responsibilities: sensing the movement, thus getting information about speed and angle, and actuating on the wheelchair motors. To obtain the values for speed and
angle, two sensors are used, one for each wheel. The sensing function reads the values from the sensors and converts them into speed and angle, which are required for other components, such as the display output interface and the navigation system. To actuate, the system receives values from a joystick or a vocal command interface and writes the values to the actuator interface. Values read from the joystick can be directly written to the actuator; or they may be converted into a standardized representation in order to use the movement controller component together with a standard input interface. These two alternatives will be explored by different modeling solutions. Another point to be explored is the model flexibility in terms of number of threads, object encapsulation, and interaction between objects.

The first modeling solution converts the joystick values into the standard representation. This solution uses fine-grain capsules, i.e., the system responsibilities are distributed among a larger number of threads and objects. This modeling approach is flexible and allows high software reuse. In this way, it is easy to extend and insert new components into the system. Moreover, the provided flexibility could exploit parallelism if a multiprocessor architecture was used. This solution uses four threads - two threads implement the movement actuating function, while the other ones implement the movement sensing function. Figure 2 illustrates the first’s solution structure. In this figure, the blocks out side of thread are static objects and the blocks inside of Thread#4 – MovementController are passive objects.

The second modeling solution was developed as a lightweight one. It uses only static objects and does not use threads to implement the movement control. Instead, it directly uses the interrupt control system. The movement-actuating scenario in this second solution has a lower encapsulation of functions than the first one. The class ‘Navigator’ has been removed, and its responsibilities were incorporated into the main class “MovementController”. The ‘SensorDriver’ class is now responsible for reading both wheel sensors, while in the first solution we had two objects of the same class, one for each wheel, so that the number of objects has been reduced. Moreover, the second solution also converts the joystick values into a standard representation. Figure 3 illustrate the second solution’s structure. In this figure, all blocks represent static objects. Note the block which represent a class that implements the responsibility of two different classes (MovementController and Navigator). This block represents the designer’s choice of raise the complexity in order to improve performance.

As the first solution has higher costs and the second one is static and inflexible, we designed a third solution to obtain intermediate results in terms of performance and flexibility. This third solution uses the same number of classes and objects as the first one. However, it uses only one thread for executing both the actuating and sensing functions. A reduced encapsulation allows some objects to directly interact with other ones, as in the second solution. We also kept the conversion of the joystick values into a standard representation.

Figure 2: First solution structure.

Figure 3: Second solution structure.
We obtained better performance results in the third solution than in the first one, due to the smaller number of threads and objects, thus reducing the platform scheduling overhead and the interaction between objects. But the second solution is still much better than the third one, due to the scheduling overhead and dynamic memory allocation, as shown in Table 1. However, in the third solution new components can be deployed more easily than in the second one, because it uses dynamic allocation of objects and a scheduler, making it easier to expand the system or to change the configuration of the components.

In this context, the fourth solution was designed using an architecture that is similar to the third one. However, differently from the three previous alternatives, joystick values are not converted into a standard representation in the actuating scenario. The fourth solution presents a similar encapsulation as the third solution, with only one thread, and the same extended “SensorDriver” class as in the second solution. Moreover, in the movement sensing scenario a more efficient algorithm was used, avoiding the computation of the angle and speed values when this is not required. With these changes, some of the flexibility presented in the first and third solutions was lost, in order to reach a performance closer to the second solution.

5 Results

We start extracting the physical metrics for every alternative solution. Table 1 presents the results. In this table memory values are represented in bytes, and performance ones in cycles. Energy consumption is evaluated in gate switching activity, as in [4] and [24]. Both performance and energy were measured in terms of Best Case (BC) and Worst Case (WC) executions. The BC execution case occurs when the wheelchair system is in inertia – stopped or without acceleration. In this case none computation of new values or actuation are required. In other case, computation of new values and iterative actuation until the stable values are required, demand more cycles and energy.

<table>
<thead>
<tr>
<th>Property</th>
<th>Sol. 1</th>
<th>Sol.2</th>
<th>Sol. 3</th>
<th>Sol. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory</td>
<td>6,248</td>
<td>2,063</td>
<td>5,208</td>
<td>5,094</td>
</tr>
<tr>
<td>BC Data memory</td>
<td>582</td>
<td>372</td>
<td>431</td>
<td>421</td>
</tr>
<tr>
<td>WC Data memory</td>
<td>582</td>
<td>372</td>
<td>431</td>
<td>421</td>
</tr>
<tr>
<td>BC Performance</td>
<td>28,588</td>
<td>1,898</td>
<td>9,104</td>
<td>7,776</td>
</tr>
<tr>
<td>WC Performance</td>
<td>41,591</td>
<td>14,423</td>
<td>21,673</td>
<td>14,510</td>
</tr>
<tr>
<td>BC Energy</td>
<td>40,569,570</td>
<td>2,714,132</td>
<td>12,916,023</td>
<td>11,026,748</td>
</tr>
<tr>
<td>WC Energy</td>
<td>58,863,463</td>
<td>20,302,894</td>
<td>30,535,449</td>
<td>20,511,557</td>
</tr>
</tbody>
</table>

Table 1: Extracted physical metrics.

The BC and WC measurement for data memory were the same, since the maximum value of dynamic allocation was found in the scheduler that must be considered in both cases. This happened for every alternative solution.
Figure 6: Physical metrics for alternative design

Figure 6 presents the relative difference between alternative designs (normalized for the values in the first solution). This figure does not show the results for BC, but it presents the same behavior as WC. The light-weight and static solution (second solution) presents the best results, regarding all physical metrics. The gain of this alternative for BC is even better.

A designer could select the second alternative due to the best results for physical metrics. However, observing also the results for quality metrics of software product, one could change his/her decision. Table 2 presents the results for these metrics.

<table>
<thead>
<tr>
<th>Property</th>
<th>Sol. 1</th>
<th>Sol. 2</th>
<th>Sol. 3</th>
<th>Sol. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstractness (A)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Afferent Coupling (Ca)</td>
<td>2.4</td>
<td>1.25</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>DIT</td>
<td>1.444</td>
<td>1</td>
<td>1.111</td>
<td>1.111</td>
</tr>
<tr>
<td>Efferent Coupling (Ce)</td>
<td>1.2</td>
<td>0.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Instability (I)</td>
<td>0.347</td>
<td>0.417</td>
<td>0.333</td>
<td>0.333</td>
</tr>
<tr>
<td>LCOM</td>
<td>0.524</td>
<td>0.602</td>
<td>0.471</td>
<td>0.384</td>
</tr>
<tr>
<td>McCabeCC</td>
<td>1.814</td>
<td>1.938</td>
<td>1.912</td>
<td>1.968</td>
</tr>
<tr>
<td>MLOC</td>
<td>214</td>
<td>178</td>
<td>182</td>
<td>169</td>
</tr>
<tr>
<td>Nested Block Depth</td>
<td>1.372</td>
<td>1.344</td>
<td>1.353</td>
<td>1.355</td>
</tr>
<tr>
<td>Dn</td>
<td>0.653</td>
<td>0.583</td>
<td>0.667</td>
<td>0.667</td>
</tr>
<tr>
<td>Number of Attributes (NOA)</td>
<td>22</td>
<td>0</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>Number of Classes (NOC)</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Number of Interfaces (NOI)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of Methods (NOM)</td>
<td>29</td>
<td>2</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>NOVM</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of Packages (NOPK)</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>NOP</td>
<td>0.791</td>
<td>1.125</td>
<td>1.088</td>
<td>1</td>
</tr>
<tr>
<td>NOSA</td>
<td>31</td>
<td>23</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>NOM</td>
<td>14</td>
<td>30</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>Total Lines of Code (TLOC)</td>
<td>616</td>
<td>480</td>
<td>523</td>
<td>503</td>
</tr>
<tr>
<td>WMC</td>
<td>78</td>
<td>62</td>
<td>65</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 2: Extracted software quality metrics.

The first observation about these metrics is that population metrics are proportional to physical ones, except the NOSM and NOP metrics. As an example, Figure 7 presents two physical metrics (Cycles WC and Data Memory) and two population metrics (NOM and TLOC), while for clarity the other metrics are not shown. We can see that all four metrics follow the same trend in all alternative designs.

Figure 8: Population metrics vs. physical metrics

Figure 8 shows the relative values for Cycles WC, NOSM, and NOP. The Number of Static Methods (NOSM) and Number of Parameters (NOP) are “almost” inversely proportional to the physical metrics. The NOSM values for the last two alternative solutions remain constant, as differences between these alternatives are smaller than for the others. The NOP value for the fourth alternative was reduced together with the physical metrics due to the removal of classes and methods, thus reducing also the lines of code.

In the second alternative more static methods and static attributes were applied, thus it presents high value for NOSM. The call of static methods is more efficient than the call of dynamic ones, thus reducing the data memory allocated in the frame and the instructions needed to be executed. However, [22] states that local variables and method argument variables are the fastest variables to
access and update. Local variables are allocated on the stack and can be manipulated directly by the JVM and underlying machine. Besides that, instance and static variables can be an order of magnitude slower to manipulate and these variables are allocated on the heap. Heap variables are manipulated through JVM special byte codes, and there are special byte codes to manipulate the first four stack slots.

Moving simple method calls to parameters of a method can improve performance without greatly compromising readability and avoiding method calls. As an example, we can consider the size() or length() methods that are called simply to know the actual size of the Vector, Array, or Collection that is being manipulated. We can greatly improve the performance of a method that calls many times these methods by simply storing the required value in a variable and accessing this variable instead of calling the method. Note that adding method parameters can improve the configurability of the method leading to a more reusable component.

Figure 9 shows that the McCabe Cyclomatic Complexity (McCabeCC) is inversely proportional to the physical metrics. By observing the figure, one can notice that the MLOC and WMC values are inversely proportional to the McCabeCC. Therefore, the global complexity, measured by the McCabeCC metric, is inversely proportional to the local complexity, measured by MLOC and WMC.

However, the global and local complexities are related to instability, cohesion, and coupling of the application components. These are important metrics and are indirectly related to reuse and time-to-market. Figure 10 illustrates the relationship between these metrics.

The instability and the coupling regard the effect of changes in a given component and the impact of these changes on the other components. The cohesion regards the importance of the attributes for the problem. The inheritance from the structured methodologies preaches that a quality system should present high cohesion and less coupling. That improves the reuse, reduces design problems, and hence reduces time-to-market.

Figure 10: Global vs. local complexity

It is important to notice that the application of an extreme fine granularity, by using numerous classes, parameters, and methods, does not necessarily increase the software product quality. Then there is a trade-off between global and local complexity in order to fulfill the system requirements/constraints and keep the components’ reusability and maintainability.

Finally, the observation of the differences between the alternative designs suggests a very large design space to be explored, when the designer considers different alternatives for encapsulating functions into threads and deciding on the number of threads and on the way they interact. Accordingly, a design space exploration approach, which takes into account the impact of these choices on the final system implementation, should be provided to the designer. Furthermore, new quality metrics aware methodologies and tools should be integrated into the embedded system design process.

6 Conclusions and future work

This work presents a comparison between quality metrics for software products and the physical properties of embedded systems. The purpose of this study is to analyze the correlation of the selected quality metrics in the
conventional metrics used to evaluate embedded system design.

Currently, the quality metrics are widely applied during the software design for non-embedded systems. However, the embedded system community is not willing to use new Software Engineering techniques, thus losing the great evolution of software design methodologies.

In this work, we found that the embedded software design cycle can greatly take advantage from the use of software product quality metrics. These metrics can help to raise the reuse factor and as consequence shrink time-to-market. Moreover, this work highlights the behavior for quality and physical metrics, which help us to better understand the trade-off between reuse and optimization.

Selecting the best design alternative in the early design stages might result in significant gains in terms of physical characteristics. Moreover, high-level evaluation and exploration of alternative designs may avoid software architectural problems, since at the firsts design steps the cost to fix design problems is much smaller. In this sense, software product quality metrics can highlight which solution is more flexible, reusable and maintainable. Further, analyzing together the physical proprieties of each solution we can check if the solution meets the requirements.

By using quality metrics for software product during the design evaluation at several levels, one can choose a solution that meet the requirements and has a better architecture, improve reuse and is more flexible. Benefits will reflect at the always sought time-to-market and component reuse. Moreover, maintainability is also improved. This is especially important, as the embedded system is increasingly dominated by software.

Future work will first address the integration of the quality metrics for software product in an automated tool, which performs the selected optimization action. These metrics can also be integrated in our design space exploration tool [19]. In this fashion, an automatic process can manage the optimization, software product quality, and reuse factors with designer assistance. Moreover, a set of specific metrics can be extracted already from UML models. Then, this measurement could be performed together with physical metrics estimation in the early development stage as provided by our estimation tool [18].

References


