# An Agile Development Methodology Applied to Embedded Control Software under Stringent Hardware Constraints

Lucas Cordeiro[1,2], Carlos Mar[1], Eduardo Valentin[1,4], Fabiano Cruz[1,4]
Daniel Patrick[1], Raimundo Barreto[1], and Vicente Lucena[3]

[1]Departamento de Ciência da Computação - Universidade Federal do Amazonas (UFAM), Brazil
{caam, dpo, rbarreto}@dcc.ufam.edu.br

[2]Centro de Ciências, Tecnologia e Inovação do Pólo Industrial de Manaus (CTPIM), Brazil
lucas@ctpim.org.br

[3]Centro de P&D em Tecnologia Eletrônica e da Informação (CETELI/UFAM), Brazil
vicente@ufam.edu.br

[4]Instituto Nokia de Tecnologia (INdT), Brazil
{eduardo.valentin, fabiano.cruz}@indt.org.br

## ABSTRACT

In recent years, discrete control systems play an important role in the development and advancement of modern civilization and technology. Practically every aspect of our life is affected by some type of control systems. This kind of system maybe classified as an embedded real-time system and requires rigorous methodologies to develop the software that is under stringent hardware constraints. Therefore, the proposed development methodology adapts agile principles and patterns in order to build embedded control systems focusing on the issues related to the system's constraints and safety. Strong unit testing is the foundation of the proposed methodology for ensuring timeliness and correctness. Moreover, platform-based design approach is used to balance costs and time-to-market in view of performance and functionality constraints. We conclude that the proposed methodology reduces significantly the design time and cost as well as leads to better software modularity and reliability.

## Categories and Subject Descriptors

J.7 [**Computers in Ohter Systems**]: Industrial control, Process control, Real time; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics, testing tools.*

## Keywords

Agile methodologies, Health Care, Embedded Agile Development, Organizational Patterns, Platform-Based Design, Real-time Software.



## 1. INTRODUCTION

The micro-controllers becoming cheaper, smaller and more reliable make them economically attractive to be used as computer systems in several appliances. Approximately 3 billion of micro-controllers ($\mu$C) are sold each year and smaller $\mu$C (4-,8-, and 16-bit) are dominating the market and adding value to products [10]. Embedded computer systems are used in a wide range of system from machine condition monitoring to airbag control systems. As the system complexity increases, its development lifecycle is also affected. Because of that, system development methodologies must be applied in order to manage the team size, the product requirement *(scope)*, and meet the project's constraints *(time-to-market and costs)*.

Nevertheless, many development methodologies that are used to produce software that runs on the personal computers (PC's) are not appropriate for developing discrete control systems. These devices share common characteristics with typical embedded real-time systems, i.e. they have a data acquisition stage, the application of a complex control algorithm, followed by output of a result. Therefore, this kind of system contains very different characteristics such as dedicated hardware and software, and constraints that are not common to PC's based systems (e.g. energy consumption, execution time, memory footprint). Another important point is that some embedded control systems may put lives at risk *(mission criticality)* if some failure occurs. Therefore, these systems should be treated differently from the case where the only cost of failure is the project's investment.

Based on this context, we propose a development methodology named as TXM (The neXt Methodology) based on the agile principles such as *adaptive planning, flexibility, iterative and incremental approach* in order to make the development of embedded control software easier. To achieve that, this methodology is composed by practices from Software Engineering and Agile methods (Scrum and XP) which aim at minimizing the main problems present on the software de-

velopment context *(i.e. requirement volatility and risk management)*, and by others practices that are needed to achieve hardware and software development *(i.e. platform-based design* [20]). With this goal in mind, we focus our attention on hardware-bound embedded software that imposes several challenges to the software design methodology.

From the point of view of embedded software design methodologies, the proposed work aims to: *(i)* tradeoff flexibility and performance by adopting highly programmable platforms, *(ii)* adopt processes and practices to develop embedded software (ESW) that is under stringent hardware constraints, *(iii)* support a software driven hardware development approach through a comprehensive flow from specification to implementation, *(iv)* propose test techniques in a combination we have not seen before, *(v)* make use of the iterative and incremental approach in order to offer clearly an iterative process where the designer can validate the system specification, and *(vi)* provide experimental results of the application of the proposed methodology in the embedded control systems domain.

The remainder of this paper is organized as follows: Section 2 summarizes the related works. Section 3 overviews the agile methods and patterns that were integrated into the proposed methodology. Section 4 is concerned with describing the proposed agile development methodology and its main componentes (processes, lifecycle, roles and responsibilities). Section 5 shows the application of the proposed methodology by focusing on the processes that were applied to the digital soft-starter and induction motor simulator prototypes. Section 6 shows the experimental results of our proposed methodology. Finally, section 7 summarizes this paper and identifies the next steps of this research.

## 2. RELATED WORKS

There are few works available in the literature about agile development methodologies for embedded systems. However, there is an interesting paper that describes the experience of applying Agile approaches to the development of firmware for the Intel Itanium processor family [8]. In this paper, they identified the agile practices that the Intel team successfully applied, but they did not take into account the hardware related development. Moreover, this work did not mention how to address the non-functional requirements (e.g., code size and real-time) and did not provide any experimental results of their work.

Manhart and Schneider [11] also related a successful industrial experience when partially adopting agile methods in the production of software for embedded systems. Indeed they made slight modifications in a well established software development process for the automotive branch adopting some agile elements in order to adequate their process to new needs as flexibility and high speed software production. As pointed out in the paper many other application areas may benefit from their experiments, nevertheless the authors did not presented any measurement results that could prove their expectations.

The conceptual framework proposed by Ronkainen e Abrahamsson, evaluate the possibility to use agile development practices in embedded software environment [13]. Therefore, they define requirements for new agile methods with the purpose of making the embedded software development process easier. These requirements include *(i)* special emphasis on hardware/software architecture, *(ii)* refactoring must be integrated into the configuration management system, *(iii)* techniques to measure the code mature in different development phase, and *(iv)* techniques to design test cases that take into account not only the correctness but also the timeliness of the application. Although this paper is totally conceptual, the requirements for new agile methods served as basis for our proposed methodology.

Vicentelli and Martin propose a rigorous methodology that aims to *(i)* deal with integration problems among intellectual property (IP) creators, semiconductor vendors, and design house, *(ii)* consider metrics to measure embedded system design, *(iii)* work from conception to software implementation, and *(iv)* favor reuse by identifying requirements for real plug-and-play operation [20]. Nevertheless, they did not provide any concrete guidance and they rely on abstract rules of thumb only. Although the methodology proposed by them is totally conceptual, it also served as basis for the development of our proposed methodology.

The hardware/software co-design methodology proposed by Gajski [7] aims to develop embedded systems by formally describing the system's functionalities in an executable language rather than a natural language. The executable specification is refined through the system-design tasks of *allocation, partition,* and *refinement*. Estimators are also used in order to explore design alternatives. However, this methodology does not provide any project management activity and assumes that most of the requirements are captured before applying the partitioning algorithms. The next section describes the Scrum and XP agile methods as well as the organizational patterns that were integrated into the proposed methodology.

## 3. A BRIEF LOOK AT THE AGILE METHODS AND PATTERNS

In this section, a brief look at the agile principles, methods, and patterns that were used in the proposed methodology is presented. It identifies the main product development and management practices of the XP and Scrum methods respectively.

### 3.1 Extreme Programming

The most recognizable agile method is eXtreme Programming (XP) which is very communication-oriented and team-oriented [2]. XP is composed of 12 core practices and some of its main characteristics that were integrated into the proposed methodology include: Refactoring practice *(i)* which is the process of changing a software system in such a way that it does not alter the external behavior of the code and at the same time improves its internal structure.

In the Continuous Integration practice *(ii)*, the code is compiled and tested in an automated process every time it is checked-in. Test-driven development practice *(iii)* means that the unit tests are written by the developers before coding. These unit tests are automated tests that test the functionality of pieces of the code. In the Coding Standard practice *(iv)*, everyone involved in the project needs to follow the same code style. It specifies a consistent format for source code, within the chosen programming language.

XP promotes an evolutionary approach to design the system by using the first three practices described above. The main benefit of this approach is that the system grows in an incremental way and it aims to reduce project's risk and un-

certainty too early *(risk management)*. Section 4 describes how these XP practices were adapted into the proposed methodology.

## 3.2 Scrum

Scrum is a simple and straightforward approach to manage the software development process based on the assumption that environmental (i.e. people) and technical (i.e. technologies) variables are likely to change during the process [15]. Scrum is composed of 14 practices and some of its main characteristics that were integrated into the proposed methodology include: Sprint practice *(i)* is the iteration organized in 30-calendar-day. The Sprint Planning practice *(ii)* consists of two meetings.

In the first meeting, the product backlog which contains a list of features, use cases, enhancements, and defects of the system is refined and re-prioritized by the product owner, stakeholders and goals for the next iteration are chosen. In the second meeting, the Scrum team figures out how to achieve the requests and creates the sprint backlog that contains detailed tasks to be accomplished in the current iteration. In the Sprint Review practice *(iii)*, the Scrum team presents the results obtained at the end of each iteration by showing the working software for the product owner, customers and other stakeholders. In the Daily Scrum practice *(iv)*, daily meetings are held at the same place and time with special questions to be answered by the Scrum team.

Scrum employs the empirical process control model, i.e. the practices aim to inspects the condition of activities and empirically determines what to do next in order to produce the expected outcomes *(product)*. The productivity and quality strongly depend on both skills and motivation of the people involved in the process. Section 4 shows how the Scrum practices were adapted into the proposed methodology.

## 3.3 Patterns for Agile Software Development

The agile patterns described by [5] can be combined with XP and Scrum agile methods with the purpose of structuring the software development process of the organizations. These patterns are split into four different pattern languages as follows: The *project management pattern language* provides a set of patterns that help the organization manage the product development, clarify the product requirements, coordinate project's activities, generate system's build, and keep the team focus on the project's primary goals.

The *piecemeal growth pattern language* provides a set of patterns that help the organization define the high-level management and amount of team members per project, ensure and maintain customer satisfaction, communicate the system requirements, and ensure a common vision for all people involved in the product development team. The *organizational style pattern language* provides a set of patterns that help the organization eliminate project's overhead and latency, ensure that the organization structure is compatible with the product architecture, organize work to develop products by geographically distributed teams, and ensure that the market needs will be met.

The *people and code pattern language* provides a set of patterns that help the organization define and keep the architecture style of the product, ensure that the architect is materially involved in implementation, and assign feature development to people in nontrivial projects. The *software configuration management pattern language* is not part of the organizational patterns but they were integrated into the proposed methodology. These patterns were defined by [3] and they offer patterns that help the development team define mechanisms for managing different versions of the work products, develop code in parallel with other developers and join up with the current state of development line, and identify what versions of code make up a particular component. The next section describes the proposed methodology and its main components.

# 4. PROPOSED DEVELOPMENT METHODOLOGY

The proposed methodology aims to define roles and responsibilities and provide processes, lifecycle, practices and tools to be applied in embedded real-time system projects. It contains three different processes groups that should be used during the system development: *system platform, product development and management*.

The *system platform* processes group aims to instantiate the platform for a given product. It means that the system designer must choose the system components that will be part of the architecture and API platforms from a platform library. After that, the system designer has still the possibility to customize the architecture and API platforms in order to meet the application constraints. The customization process is carried out by programming the designer-configurable processors and runtime-reconfigurable logic integrated into the platform. The customization process is carried out by successive refinements in an iterative and incremental way into the proposed methodology.

The *product development* processes group offers practices to develop the application's components and integrate them into the platform. The functionalities which make up the product are partitioned into either hardware or software elements of the platform. Our partitioning algorithms used to carry out this task takes into account the energy consumption, execution time, and memory size of the application's components. In addition, the partitioning technique is also applied in an iterative and incremental way. The mechanical design is also part of this processes group, but it is out of the scope of this paper.

The product scope, time, quality, and costs parameters are monitored and controlled by the *product management* processes group. These parameters also influence the *system platform* and *product development* processes groups. When the project starts with an infeasible project plan which needs corrective actions to be carried out then this processes group aims to get the project back on the track and ensure that the project's parameters are met. The *product management* processes group consists of the practices promoted by the Scrum agile method as well as the agile patterns described in [5, 15]. The next subsections are concerned with describing the processes groups, roles and responsibilities, and the processes lifecycle of the proposed methodology.

## 4.1 System Platform Processes Group

The *system platform* processes group is composed of the following processes: *product requirements, system platform, product line, and system optimization*. The *product requirements process* aims to obtain the system's requirements (func-

tional and non-functional) that are relevant to determine the system platform in which the product will be built. The *platform instance process* helps the development team define the system platform by making use of a set of design tools and benchmarks.

After defining the system platform, the *product line process* helps the development team setup the repository in which the system platform components will be available to the product development. This process also allows the development team to implement and integrate system's functionalities into the system and release new product versions into the market. After implementing and integrating the system's functionalities into the product development line, the *system optimization process* provides activities to ensure that system's variables such as execution time, energy consumption, program size and data memory size satisfy the application constraints.

## 4.2 Product Development Processes Group

The *product development* processes group is composed of the following processes: *functionality implementation, task integration, system refactoring, and system optimization.* The *functionality implementation process* ensures that test cases are created for every product's functionality. This process helps increase the product quality and reduce the creation of complex functions. Moreover, it also helps create a comprehensive test suite for testing and validating that the API Platform layer will function properly for the software applications.

The *task integration process* provides means to integrate new implemented functionalities into the development line of the product without forcing the other team members to work around it. The *system refactoring process* helps the development team identifies opportunity to improve the code and changing it without altering its external behavior. After refactoring the code, the *system optimization process* allows the development team to optimize small part of the code by making use of profiler tools that monitor the program and tells where, for instance, it is consuming time, energy, and memory space [12]. This process guarantees that software metrics meets the system constraints.

## 4.3 Product Management Processes Group

The *product management* processes group is composed of the following processes: *product requirements, project management, bug tracking, sprint requirements, product line, and implementation priority.* The *product requirements process* (that also belongs to the system platform processes group) aims to obtain the system's requirements (functional and non-functional) that must be part of the product. The *project management process* allows the development team to implement the system's requirements by managing the product and sprint backlog, coordinating activities, generating system's build, and tracking the product's bug.

The *bug tracking process* allows the product leader to manage the lifecycle of the project's issues (bug, task, and enhancement) and provide the needed information about the product quality through the release notes for the end user. The *sprint requirements process* allows the development team to analyze, evaluate, and estimate the system's functionalities before starting a new project's sprint. This information is included into the sprint backlog which will help the development team partition the system functionalities into either hardware or software before starting the sprint.

The *product line process* guarantees that the system functionalities implemented during the sprint will be integrated into the product development line. This process also helps the development team to release new product versions into the market. The *implementation priority process* helps the product leader manage any kind of interruptions that may impact the project's goals. This process guarantees that the project's tasks are 100 percent completed after initiated.

## 4.4 Roles and Responsibilities

The proposed methodology involves four different roles and the responsibility of each role is described as follows:

**Platform Owner:** Platform owner is the person who is officially responsible for the products that derive from a given platform. This person is responsible for defining quality, schedule and costs targets of the product. He/she must also create and prioritize the product backlog, choose the goals for the sprints, and review the product with the stakeholders.

**Product Leader:** Product leader is responsible for the implementation, integration and test of the product ensuring that quality, schedule, and cost targets defined by the platform owner are met. He/she is also responsible for mediating between management and development team as well as listening to progress and removes block points.

**Feature Leader:** Feature leader is responsible for managing, controlling and coordinating subsystem projects, preintegration projects, external suppliers that contribute to a defined set of features. The feature leader also tracks the progress and status of the feature development (deliverables, integration and test status, defects, and change requests) and reports the status to the product leader.

**Development Team:** The development team which may consist of programmers, architects, and testers are responsible for working on the product development. They have the authority to make any decisions, do whatever is necessary to do (according to the project's guidelines), and ask for any block points to be removed.

If the product to be developed is small, i.e. it is composed of few components (less than 50 KLOC) and does not require other development teams to implement the product's functionalities then one product leader and the development team are enough for the product development. On the other hand, if the product is composed by several components (more than 50 KLOC) and requires other development teams to implement the product's functionalities then the Feature Leader role must be involved in the processes. In this context, one product leader requires feature leaders to manage, control and coordinate components' projects. Therefore, for medium and larger projects, one product leader and several feature leaders and development teams may be involved in the processes.

## 4.5 Processes Lifecycle

The proposed agile methodology consists of five phases: *Exploration, Planning, Development, Release, and Maintenance.* In the *Exploration phase*, the customers provide requirements for the first product release. These requirements are included into the product backlog by the platform owner. After that, the platform owner and product leader estimate the requirements with no item larger than 3 person-days of

effort. In this phase, the development team identifies the platform and application constraints and estimates the system's metrics based on the product backlog items. With this information at hand, the development team is able to define the system platform that will be used to develop the product in the next phases.

In the *Planning phase*, the platform owner and customers identify more requirements and prioritize the product backlog. After that, the development team spends one day to estimate the sprint backlog items and decompose them into tasks. The tasks that make up the sprint backlog must take from 1 to 16 hours to be completed. Explanatory design and prototypes may also be developed at this phase in order to help clarify the system's requirements.

In the *development phase*, the team members implement new functionalities and enhance the system based on the items of the sprint backlog. The daily meetings are held at the same time and place with the purpose of monitoring and adapting the activities to produce the desired outcomes. At the end of the each iteration, unit and functional tests are executed in a continuous integration build. System optimization also takes place during this phase. The last sprint provides the product to be deployed in the operational environment.

In the *Release phase*, the product is installed and put into practical use. During this phase, it usually involves the identification of errors and enhancement in the system services. Therefore, the platform owner and customers decide if these changes will be included in the current or subsequent release. This phase aims to deliver the release product and needed documentation to the customer. The *Maintenance phase* may also require more sprints in order to implement new features, enhancement and bug fixes raised in the release phase.

The next subsections describe only a subset of processes of the proposed methodology that focuses on achieving the aims of the embedded control systems.

# 5. APPLYING THE PROPOSED METHODOLOGY

This section is concerned with describing the application of the proposed methodology in the development of the digital soft-starter and induction motor simulator equipments. We chose these equipments as case studies because they impose several challenges to develop the ESW that is under stringent hardware requirements (e.g., real-time and code size) and also require a close interaction among the engineers in order to develop the products.

Both projects were split into 2 different sprints and developed by four embedded system engineers (each project had two engineers), one product leader, and one platform owner. The next subsections describe the characteristics and architecture as well as the processes of the proposed methodology and the build infrastructure.

## 5.1 System's Characteristics and Architecture

Generally speaking, the digital soft-starter is an equipment that makes use of an efficient method for starting motors. Efficient here means low energy consumption levels obtained by the automatic (adaptive) adjust of its working parameters. A detailed description on the functioning of the soft-starter may be found at [19]. A widely adopted method used to control induction motors which was implemented in our digital soft-starter equipment is the so called PWM (Pulse Width Modulation). Summarizing its functionality the output signal of the inverter is controlled by the variation of the pulse width of another signal in every voltage cycle [1].

Therefore, the main system's requirements that were implemented to the digital soft-starter include: *(i)* the system should be able to automatically control the start of the induction motor, *(ii)* the system should read the voltage signal provided by the sensor through an analog-to-digital converter, *(iii)* the PWM signal generated by the microcontroller should meet all timing requirements of the application, and *(iv)* the user interface of the equipment should have a keyboard and a graphical display.

In order to validate the digital soft-starter equipment and also the proposed methodology, we have constructed an induction motor simulator. An important characteristic that distinguishes the induction motors is that they are machines with a *single excitement*. Although such machines are equipped with a *field winding* and with *armature winding*, in normal conditions of use the source of energy is connected to a single *field winding*. It is important to point out that the induction motor needs a device departure starter, so that it can be used. Such devices act basically in the sense of creating an unbalance in the field of the *estator*.

In general, the main characteristics implemented for our induction motor simulator include: *(i)* the system should simulate the behavior of the motor through a mathematical model; *(ii)* the system should reproduce the signal supplied PWM (for the digital soft-starter) through I/O microcontroller ports; *(iii)* the system should calculate and show in the display the voltage and current values based on the PWM signals supplied by the soft-starter; and *(iv)* a man-machine interface (display and keyboard) should be present in the final solution. Figure 1 presents an overview of the connection between the induction motor and digital soft-starter equipments.

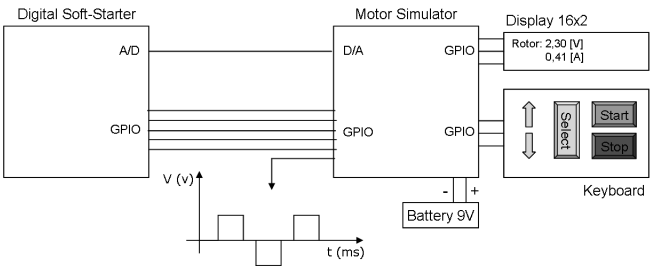


**Figure 1: Induction Motor Simulator Overview.**

As can be seen in this figure, the soft-starter sends the PWM signals through the microcontroller's general purpose I/O (GPIO) to the induction motor simulator. The GPIO provides flexible software-controlled digital signal between both platforms. The mathematical model that represents the induction motor is implemented in another platform that has the same configuration (microprocessor and peripherals) as the digital soft-starter.

In order to check this mathematical algorithm of the induction motor in our embedded software, we could use the Matlab to simulate the behavior and generate the test data for comparison. As described in Subsection 5.4, this tech-

nique allows us to easily implement and test changes and carry out integration and unit tests of the algorithm simultaneously. These test data files are of great benefit to exercise the code paths and the effort to set up them can be paid back many times over.

Our architecture platform for both projects contains a RS-232 serial converter, a microcontroller AT89S8253 which has an 8051-like architecture with code and data memory integrated on a chip, a real-time clock PCF8583, four channels A/D converter and one channel D/A converter. Additionally the platform has 12 KB of flash memory and 32 KB of RAM. The communication between the converters and the embedded software is carried out by the communication protocol $I^2C$ [16].

The API platform is composed of a set of helpful components which include: the Services subsystem (*System Log, PWM Generator and Unity Converter* components) and the Platform Drivers (*RS-232 Serial, Timer, LCD 16X2, Keyboard and A/D Converter*). This API can be seen as an abstraction layer that hides several details of the platform's resources [20]. Therefore, we chose this platform because it has already a set of interconnected HW/SW components that together implements a set of functionalities and decreases significantly the development speed and costs.

## 5.2 Process for Managing the Product Requirements

This process helped us identify the market needs for the digital soft-starter product line and manage the product requirements. At the beginning of the project, we arranged a brainstorming meeting in order to capture high-level requirements of the product. After that, we created an initial product backlog with the purpose of capturing more requirements and creating a first product prototype.

The first project iteration allowed us to answer questions such as whether the technology needed for the system exists, how difficult it would be, check the platform performance from different vendors, and implement a couple of system's functionalities. In further iterations, we implemented more system's requirements by focusing on the items with highest business values (the business values range from 1 lowest to 5 highest). After that, the final configuration of our development platform would dispense unneeded components and combine everything on one board for economical production costs.

As a requirement management strategy, we put much emphasis on delivering the system's functionalities *(i)*, *(ii)*, and *(iii)* in the beginning of the sprints for the digital soft-starter project and system's functionalities *(i)* and *(iii)* for the motor simulator project. Delivering these functionalities with highest business value, helped our customer and platform owner get feedback on functionality earlier and allow them to spot any misunderstanding more quickly. At the end of each iteration, the product leader and customer verified if the product was still feasible or not. If the project would not be feasible then it could be canceled just after the end of the iteration *(risk management)*.

## 5.3 Process for Managing the Project

This process helped us refine and prioritize the product backlog that contains the system's functionalities. In the sprint planning, the product leader and our customer chose the goals of the next sprint based on the highest business value and risks of the product backlog items. After that, we had a meeting to consider how to achieve the sprint's goals and to create the sprint backlog. The sprint backlog should contain only tasks in the 4-16 hour range in order to make the management activities easier *(risk reduction)*.

During the system development, the sprint backlog was updated regularly as the activities were being accomplished. The product leader held two meetings per week with the team members in order to monitor and control the complexity of the tasks. These daily meetings provided a great feedback to the product leader and created the habit of sharing the knowledge. After starting the sprint, we implemented first the functional requirements and then focused on the non-functional requirements of the system. This approach helped us obtain better optimization results because we optimized the global system instead of only parts of the system which sometimes might not lead to the global optimization [6].

During this phase, system's builds were also generated on a weekly basis which helped our customer clarify the requirements and asses the risks earlier in the development process. At the end of the sprint, the product leader and development team showed the results of the work to the customers. This meeting aimed to present the product increment, technology and business situation. These artifacts helped the product leader and customers decide the goals of the next sprint. In addition, after each sprint review there was a retrospective meeting which had the purpose of collecting the best practices used in the sprint and identifying what could be improved for the next sprint.

## 5.4 Process for Implementing New System's Functionalities

This process supported us for implementing the system's functionalities of the digital soft-starter and induction motor simulator prototypes in a systematic way. According to the business value of the systems functionalities defined in the Section 5.2, we started implementing the requirements responsible for generating and handling the PWM signals. In order to implement these systems functionalities, we strove to write first the unit test for each stage of computation. However, this kind of activity required certain level of experience from the embedded system engineers. Nonetheless, we had to successfully compile the unit test before really writing the functionality's code.

In order to test each computation stage of the systems functionalities, we had to run the embedded software on the PC and target platform. We used this approach throughout the development cycle in order to avoid debugging hardware and software simultaneously. By running the embedded software on the PC platform, we could exercise all code paths and gain confidence in our code before running it on the target platform. Another way to gain confidence in the code is to use the JTAG debug capability.

After that, we could run the embedded software on the target platform to verify the application's timeliness. For both projects, we created data files on the PC that had all possible parameters combinations that make sense for the system's functions inputs. In this way, we could provide these data to our unit tests to exercise the code's paths of the functions. Figure 2 depicts an example of unit test applied to PWM signal generator of the soft-starter device by using the embUnit framework test tool [18].

```
1 static void setUp(void) {
2     fillData("signal01.txt", signalData01);
3 }
4 static void testGenerateSignalNormal(void) {
5     Q1 = 0; Q2 = 1; Q3 = 0; Q4 = 1
      .
6     .
7     for (i = 0; i < SIGNAL_NUMBER; i++) {
8         generateSignal();
9         counterQ1 += TICK;
10        counterQ3 += q3Enable;
11        ASSERT_EQUAL_INT(signalData01[4 * i], Q1);
12        ASSERT_EQUAL_INT(signalData01[4 * i] + 1, Q2);
13        ASSERT_EQUAL_INT(signalData01[4 * i] + 2, Q3);
14        ASSERT_EQUAL_INT(signalData01[4 * i] + 3, Q4);
15    }
16 }
```

**Figure 2: Unit test for the *generateSignal* function**

This *generateSignal* function uses the counter values to generate the control signals in pins *Q1*, *Q2*, *Q3* and *Q4* of the target platform. In order to verify the timeliness and correctness, we had to: *(i)* fill in the integer array *signalData01* with all possible combinations that make sense to the control pins, *(ii)* initialize the control pins *Q1* and *Q3* with the angle $\alpha$ to be triggered (the other pins *Q2* and *Q4* are complementary), *(iii)* call *generateSignal* function to simulate the PWM signal generation, and finally *(iv)* compare the actual value of *Q1*, *Q2*, *Q3* and *Q4* with the value expected into array *signalData01*.

To evaluate the correctness of this function, it was necessary to create the *setUp* function (line 5) with the purpose of providing the *signalData01 array* with all possible values to *Q1*, *Q2*, *Q3* and *Q4* control pins. Moreover, we had to declare the *counterQ1*, *counterQ3* and *q3Enable* as global variables and initialize them with defined value in the beginning of test case execution. It is important to mention that we exercised the "happy" path, error conditions, and corner cases. To evaluate the timeliness, we had to perform this test case in the target platform. Therefore, we obtained a worst case execution time (WCET) of approximately 107.415 $\mu$s.

For those software components that touch the hardware, we just used the #if and #else statements while running the embedded software on the PC platform. Later on, we developed and run application tests on the target platform in order to stress the hardware dependent code. To implement all systems' functionalities, we followed the product's coding standard defined at the beginning of the project with the purpose of keeping consistence throughout the system's code. An important point to take into account is that if there is the need for splitting a given system's functionality into different functions then the unit tests should be created for each system's function.

## 5.5 Process for Refactoring the Code

After implementing the system's functionalities, we identified in further sprints opportunities to improve an existing code. For instance, we identified during the digital soft-starter and induction motor simulator projects that the functionalities could be split into different modules. Moreover, we also identified duplicated code in both projects. Therefore, the application of this process led to elimination of duplicated code, reduction of the amount of system's functions, and improve the system performance. However, before improving the code for those tasks in common, we first created branches in the system repository for not breaking an existing working code.

After that, we verified if there was some need for updating the unit test of the functionality. If there was no need to update the unit tests then we could start improving the code without altering its external behavior. After refactoring the code, we run the unit test in order to verify if the changes were working correctly. If there was no compilation problem and the unit test would not fail then we could integrate our changes into the product development line. After integrating the code, the regression tests could also be run in order to check if there was no compilation and semantic problems. If there was no problem then the refactoring was completed.

## 5.6 Process for Choosing the Sprint Requirements

This process helped us choose the system's functionalities for the sprint based on the following criteria: *(i)* splitting system's functionalities depending on its estimates and on the load of each iteration and *(ii)* splitting by across data boundaries, for example, selecting a subset of operations (e.g. fill in and sort array) supported for a given functionality [4]. Coding stubs and mock objects were used in order to compensate for the absence of component's functionality.

Because system's components (e.g., LCD or keyboard driver) have little coupling to the rest of the system, it is easier to build mock objects to cover those inputs and outputs of the components. In addition, when seen from the whole, having such mock objects timely available to other subsystems might be essential for allowing the rest of the system to grow optimally. Therefore, in order to enable incremental integration to happen in embedded real-time projects, the notion that subsystems should expect a level of rework between iterations should be introduced.

## 5.7 Process for Managing the Product Line

This process allowed us to set up the system repository by populating it with the system platform components which consists of the API platform. Collection of components that comprises the architecture platform were also chosen based on the application constraint's. The repository contained only the system components that were needed to derive new product lines. In other words, the development team derived the product line (or the platform instance) from the platform just by choosing a set of components from the platform library or by setting parameters of the library's reconfigurable components.

After that, they created in the system repository the development line in which the product would be built. This product development line allowed the team members to develop and optimize the product components. In order to implement new system's tasks (which may include new requirements, enhancements and bug fixes), each team member should create a branch in the product development line. This branch would help the team member implement the task without forcing the other team members to work around

it. After implementing and optimizing all system's components that make up the product, the development team could create a release branch of the product with the purpose of not interfering with the current system development.

## 5.8 Process for Tracking the Product Bugs

This process for tracking the product's bugs provides activities to manage the lifecycle of the project's issues (e.g., bug, task, and enhancement). Therefore, this process allowed us to identify a new issue during the product development by using a log system as depicted in Figure 3. This log system is an extension of the idea put forth by [14] which aims to eliminate the overhead caused by the *printf* call. Therefore, this log system uses a circular buffer in RAM to hold brief fixed-length text messages. In order for the development team to carry out the system diagnostic, they wrote the log statement at the start and finish of an Interrupt Service Routine (ISR) and then subtract the timer to see how long the ISR took to execute.

```
Timer   Component   Function:Filename(line)
12320   c_LCD   ->  LCD_Driver_InitModule: lcd_class_driver.c(85)
11789   c_LCD   ->  LCD_WriteData: lcd_class_driver.c(90)
13452   c_LCD   ->  LCD_InterfaceDescriptor: lcd_class_interface.c(102)
11216   c_LCD   ->  LCD_InterfaceContext_Create: lcd_class_interface.c(18)
12834   c_LCD   ->  LCD_initialize: lcd_class_interface.c(80)
```

**Figure 3: Log Output Sample.**

After identifying bugs in the code through either the log system or the test strategy described in the Section 5.4, the issue could be open in the bug tracking tool by any team member. He/She should only provide the needed information in order to reproduce the bug such as: *summary, platform, product, software version, functional cluster, component, log, and issue description*. After that, the issue could be automatically assigned to the person responsible for the functional cluster in which the bug was open. If the person responsible for fixing the issue was not able to reproduce the bug then he/she could talk directly with the person who opened the bug in order to get more information on how to reproduce it.

## 5.9 Build Infrastructure and Tools

Figure 4 shows our proposed build infrastructure that aims to support the processes of our proposed methodology. This infrastructure allows the team members to integrate new tasks into the system and hence manage the product development line as described in Subsection 5.7. The *CVS* repository has the purpose of controlling the system's code version. In our projects, this repository is hosted in the sourceforge website.

Our proposed build infrastructure for continuous test and integration works as follows: *(i)* firstly we must check out the code that is in the repository to a local workspace. Therefore, it allows us to implement new system' functionalities, fix bugs and improve the system' code. Moreover, we are also able to generate new product' builds in the local repository. *(ii)* after implementing and testing the code by following the activities described in the Subsections 5.4 and 5.5, we can make the code available in the repository.

*(iii)* After that, the *Cruise Control* tool looks for code modifications in the repository. If the file date/time changes

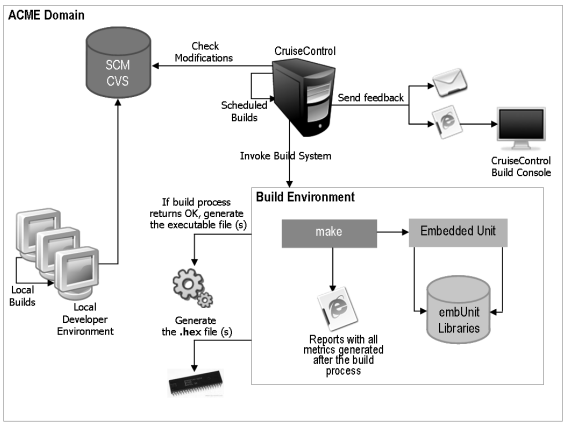


**Figure 4: Continuous test and integration process.**

then the *Cruise Control* starts the build process in an automated way. If there is a compliation error then *Cruise Control* tool sends an e-mail to the person responsible for breaking the code. Otherwise, it generates the .hex file that will be loaded into the flash memory of the embedded device and it runs other tools (e.g., *CCCC* and *EmbUnit*) in order to capture the metrics and test the code. The next subsection describes the experimental results of our proposed methodology applied to the digital soft-starter and induction motor.

# 6. EXPERIMENTAL RESULTS

In this section we present the results of our proposed methodology. As described in Section 5.2, we created a list of new features and requirements in order to gather all our needs (product backlog). Based on the business value, we chose from the product backlog a set of features and requirements to be implemented in the project's sprints. Therefore, we put much emphasis on delivering the systems functionalities with highest business value in the beginning of the sprints. Table 1 shows the measured effort, business value, and development speed for each sprint of both projects.

**Table 1: Measured Effort (in hours)**

| | Sprint 1 | Sprint 2 |
|---|---|---|
| **Digital Soft-Starter** | – | – |
| Measured Effort | 70 | 84 |
| Business value | 11 | 16 |
| Sprint Velocity | 0.1571 | 0.1904 |
| **Motor Simulator** | – | – |
| Measured Effort | 125 | 219 |
| Business value | 13 | 21 |
| Sprint Velocity | 0.104 | 0.0958 |

As can be seen in Table 1, the sprint velocity of the soft-starter increased as the system was being developed. This situation took place due to the fact that we were still learning the involved technology, development environment, and the application domain. On the other hand, the sprint velocity of the motor simulator decreased due to the fact that a team member was moved from the soft-starter to another project. Therefore, the tasks that were allocated to

him had to be transferred to an engineer of the induction motor simulator.

As mentioned in Section 5.4, we had hardware independent code and hardware specific code for the digital soft-starter and induction motor simulator projects. For hardware independent code, we applied our proposed test techniques which checks not only the logic but also the timing. But for hardware specific code, we had to skip over it when running on the PC desktop. For this kind of embedded software, our approach was to develop and run the test applications that aim to exercise all code paths of the function in the target platform manually. Therefore, we had approximately one test line for each two code lines using these proposed test techniques to ensure the timeliness and correctness of the prototypes.

Table 2 shows the relationship between the test and code lines of both projects. The final size of the digital soft-starter and motor simulator embedded software was approximately 1615 and 854 lines of code respectively. We also used the CCCC tool to count the effective source lines of code (ESLOC) of our embedded software. This tool counted 589 and 385 lines of code and 220 and 101 lines of test for the digital soft-starter and motor simulator respectively. This tool did not include blank lines, comment lines, and lines with a single brace "}". Source files with long preambles are mainly the cause for the high percentage of non-code lines.

**Table 2: Total LOC (Application and Test)**

| Project | Application | Test |
|---|---|---|
| Digital Soft-Starter | 1615 | 854 |
| Motor Simulator | 957 | 243 |

The digital soft-starter and induction motor simulator embedded software had to run in a constrained environment. Our development platform had just 12KBytes of flash memory. Therefore, we used the Big Visible Chart (BVC) proposed by [2] with the purpose of tracking the memory usage and power consumption metrics. Both charts were regularly updated and kept visible in order to look for trends. Table 3 and 4 show the memory usage and power dissipation values for both projects. The current consumption was measured by connecting a multimeter in series with the energy source. The final power dissipation was then obtained by multiplying the current consumption by the supplied voltage. This power is dissipated in the whole system by the digital and analog components.

**Table 3: Memory Usage (Bytes)**

| Project | RAM | Flash |
|---|---|---|
| Digital Soft-Starter | 3631 | 3252 |
| Motor Simulator | 600 | 6398 |

The test techniques described in the Section 5.4 were the perfect vehicle for software design and modularity of the digital soft-stater and motor simulator embedded software. Table 5 shows the average cyclomatic complexity of the systems. Cyclomatic complexity (v(G)) is a measure of the complexity of a module's decision structure that indicates the number of linearly independent paths [9]. Data

**Table 4: Power Dissipation (mW)**

| Project | Power |
|---|---|
| Digital Soft-Starter | 855 |
| Motor Simulator | 774 |

on source code size, number of functions and cyclomatic complexity were obtained using CCCC tool which analyzes C/C++ files and generates a report on HTML format [17].

The final solution of the digital soft-starter and motor simulator prototypes has approximately 35 and 31 functions in C code respectively. Therefore, all sprints were analyzed and the result was an average cyclomatic complexity of 1.43 and 1.61 at the end of second sprint for the digital soft-starter and the induction motor simulator. These low cyclomatic complexity levels make the white-box testing easier due to the fact that they decrease substantially the number of paths that should be tested to reasonably guard against errors. For more detail on this metric, refer to [9].

**Table 5: Cyclomatic Complexity**

| Project | v(G) |
|---|---|
| Digital Soft-Starter | 1.43 |
| Motor Simulator | 1.61 |

The next section concludes this work and provides goals for future research.

# 7. CONCLUSIONS

This paper described an agile development methodology and its application in the development of the digital soft-starter and induction motor simulator projects. In order to create the methodology, we chose two agile methods XP and Scrum as well as organizational patterns named in this paper as agile patterns. When XP, Scrum and agile patterns are combined they cover many areas of the system development life-cycle. However, the combination of Scrum, XP and agile patterns does not mean that they can directly be used to develop embedded control software.

Therefore, slightly changes were needed to addreess the challenges involved to develop such kind of software that include: *(i)* adopt processes and tools to optimize the product's design rather than take paths that lead to designs that have no chance of satisfying the constraints, *(ii)* support software and hardware development through a comprehensive flow from specification to implementation, *(iii)* instantiate the system platform based on the application constraints rather than overdesign a platform instance for a given product, and *(iv)* use system platform to conduct various design space exploration analyses for performance.

To illustrate the use of the processes and tools of the proposed methodology, we described how it was applied to develop the hardware-bound embedded software of both control systems. In these case studies, the development platform reduced substantially development time and costs of the product. In addition, we also applied a set of test techniques in order to guarantee the timeliness and correctness of the embedded control software.

These test techniques led to better software design and

modularity. Therefore, we obtained 1.43 and 1.61 average cyclomatic complexity levels for the digital soft-starter and motor simulator equipments respectively. For further steps, we are researching models that can carry enough information about the ultimate physical implementation and achieve better results in terms of functional correctness. Moreover, we are performing more experimental studies where the methodology will be observed while applied.

## 8. REFERENCES


[1] A. Ahmed. *Power Electronics*. Prentice Hall, Inc., 2000.

[2] K. Beck and C. Andres. *Extreme Programming Explained - Embrace Change*. Second Edition, Addison-Wesley, 1999.

[3] S. Berczuk and B. Appleton. *Software Configuration Management Patterns*. First Edition, Addison-Wesley, 2002.

[4] M. Cohn. *Agile Estimating and Planning*. Robert Martin Series, Prentice Hall, 2005.

[5] J. O. Coplien and D. Schmidt. *Organizational Patterns of Agile Software Development*. First Edition, Prentice Hall, 2004.

[6] P. Cybernetica. *Suboptimization Problem*. Available at http://pespmc1.vub.ac.be/SUBOPTIM.html. Last visit on 26th December, 2007.

[7] D. Gajski, F. Vahid, and S. Narayan. A system-design methodology: Executable-specification refinement. *European Conference on Design Automation, Paris, France*, 1994.

[8] B. Greene. Agile methods applied to embedded software development. *Proceeding of the Agile Development Conference (ADC'04).*, 2004.

[9] S. E. Institute. *Cyclomatic Complexity*. Published at the Carnegie Mellon University, 2007.

[10] P. Koopman. Embedded system design issues (the rest of the story). *Proceedings of the International Conference on Computer Design (ICCD96)*, pages 310–317, 1996.

[11] P. Manhart and K. Schneider. Breaking the ice for agile development of embedded software: An industry experience report. *Proceedings of the 26th International Conference on Software Engineering (ICSE04)*, page 3647, 2004.

[12] M. J. Oliveira, S. Neto, P. Maciel, R. Lima, A. Ribeiro, R. Barreto, E. Tavares, and F. Braga. Analyzing software performance and energy consumption of embedded systems by probabilistic modeling: An approach based on coloured petri nets. *ICATPN 2006, LNCS 4024, pp. 261281, 2006.*, page 261281, 2006.

[13] J. Ronkainen and P. Abrahamsson. Software development under stringent hardware constraints: Do agile methods have a chance? *eXtreme Programming Conference*, 2003.

[14] N. V. Schooenderwoert and R. Moriscato. Taming the embedded tiger - agile test techniques for embedded software. *Proceeding of the Agile Development Conference (ADC'04).*, 2004.

[15] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. First Edition, Series in Agile Software Development, Prentice Hall, 2002.

[16] P. Semiconductors. *The I2C-bus and how to use it*. Available at http://www.mcc-us.com/i2chowto.htm. Last visit on 22th October, 2007.

[17] SourceForge. *C and C++ Code Counter*. Available at http://sourceforge.net/projects/cccc. Last visit on 18th October, 2007.

[18] SourceForge. *embUnit: Unit Test Framework for Embedded C Systems*. Available at http://embunit.sourceforge.net/. Last visit on 18th October, 2007.

[19] V. D. Toro. *Basic Electric Machines*. Prentice Hall, Inc., 1990.

[20] A. S. Vicentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.