Using Aspects and Component Concepts to Improve Reuse of Software for Embedded Systems Product Lines

Edison Pignaton de Freitas¹, ², Marco Aurélio Wehrmeister², ⁴, Carlos Eduardo Pereira³, ³ and Tony Larsson¹

¹ School of Information Science, Computer and Electrical Engineering, Halmstad University, Sweden
² Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil
³ Dep. Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Brazil
⁴ Heinz Nixdorf Institute – University of Paderborn, Germany

{edison.pignaton, tony.larsson}@hh.se, maweehrmeister@inf.ufrgs.br, cpereira@ece.ufrgs.br

Abstract

Embedded systems have several characteristics, such as application specific needs, real-time constraints and intrinsic embedded concerns (i.e. energy consumption), which hinder the reuse of previously developed components as well as their adaptation to provide variability in a software product line. As more applications require embedded system solutions, it is clear that if each time a new application is developed, it is not suitable to develop its supporting embedded system components from scratch. Resulting in an inability of the industry to follow the needs imposed by the market. To cope with this problem, the reuse of components within software product lines is a key issue. It seems to be easy task, but in fact it is not, due to increasing amount functionalities and crosscutting concerns present in those applications. In this paper we present an approach to handle components at a higher abstraction level, whose handling of crosscutting concerns is weaved by system level aspects, in order to address such complexity and also to make it easier the handling of variations in a software product line.

1. Introduction

Nowadays, the complexity of system development requires the use of more abstract representations that provide an easier way to improve reuse [1] in order to reduce the time to market. This is particularly perceived in the development of embedded systems, as they have several characteristics that impose difficulties to reuse parts of previous projects [2] [3]. The use of product line concepts bring a nice way to think about how software components are envisioned from their early conception to their eventual reuse in further projects. It is an emerging trend in this area, as developers have realized that components cannot be developed in an “opportunistic” reuse way, in which an implemented component is stored in a library with the hope that it can be reused in the future. With these ideas in mind, traditional development approaches do not provide the necessary tools and abstractions required in the development of large systems in which the result of previous work is intend to be reused.

Traditional programming languages are good for description of data and functional transformations or mappings of data. The composition of individual mappings into larger composed systems can be described but is harder to get an overview of. We argue that many design difficulties arises due to the fact that traditional software development is in its nature close ranged and not suited for the description of “programming in the large”, system oriented design problems. Structured, object oriented and functional programming styles helps but description of entire software systems in a textual language (mainly made for algorithm and data description) also has its limitations.

The use of languages that raise the abstraction level, as those based on UML (a de facto industrial standard for graphical notations), helps developers in their task to design embedded systems, as discussed in [4]. However, the increasing complexity of nowadays embedded systems leads to very large and tangled designs, with diagram representations that are sometimes hardly readable due to the large amount of elements and their interconnections. This difficulty indicates a need for a higher-level approach in which the system can be seen as a composition of more coarse grained elements in a way that the whole view of the system is not lost.

Furthermore, features such as triggering conditions, timing, and concurrency often crosscut the embedded system design documents over several different parts and in different ways. It is extremely hard to decouple system elements and, consequently, reuse them. Another facet of this concern is that using the current paradigms, it is not trivial to get an overview of how these crosscutting
concerns affect the whole system. In general, only local views of what happens in parts of the system are provided, but not a composed overview of the entire system. This overview of the system as a whole, or major modules, is very useful if internal details are disregarded for a while.

Many of the above mentioned problems and others more linked with later phases, from design to implementation and deployment, come from the influence that component-based development got from the object-oriented approach. A good discussion about this influence is presented in [5].

Our proposal is to use the concept of components [1] in a simple way to describe meaningful and specialized functional elements of an embedded system that can be interconnected and composed with other components in order to provide the total system functionality. Moreover, these components must be decoupled in order to really allow changes within them without modifying other parts of the system, as well as to make them really reusable. These modifications allow the creation of variations in a product line without much extra effort. An inspiring idea comes from Operating Systems (OS), which use the concept of services to partition the handling of concerns into different specialized elements. However, the use of OS services and their effect are not easy to overview since the cooperation between application parts and OS services are intertwined and spread over the code.

In order to handle crosscutting concerns, which are generally intertwined in system functionalities, we propose to borrow the concept of aspects from [6], but more towards the early-aspects approach based on ideas presented in [7] and [8]. Thus, aspects affect higher abstraction level entities (in this case components) in order to separate the handling of functional and non-functional concerns.

The remaining of this paper is structured as follows. In Section 2 the concept of components is briefly described. Section 3 describes how aspects can be used together with components, allowing an effective reuse, based on the product line concept. Section 4 is dedicated to the analysis of related work while in Section 5 the conclusion and future work directions are presented.

2. Component Concept

The value of modular thinking and components are well established concepts that have solid backgrounds as presented in [9] and still nowadays is a very useful and promising design direction [10].

To bring forward the ideas about how aspects can support components we aim to keep the component discussion simple. Thus the used concept of components follows the general principles presented in references already mentioned, such as in [1] and [10]. Other reference studies in the area are considered, such as [11], which provide consensus statements about component based software engineering. On other words, the intention of this section is to present concepts of the mentioned component models that are used in the scope of the proposed work, instead of presenting new components concepts.

Therefore, the basic idea is to use components as the primary building blocks of a system, expressing well defined functionalities, and connecting them to other components to achieve the overall systems desired functionality. Figure 1 graphically represents two connected components.

![Figure 1: Two Connected Components](image)

The main elements used by components can be identified in the representation shown in Figure 1. A component has a certain number of ports that represents its interface to communicate with other components. Those ports are labelled in order to provide their identification, and also act as incoming or outgoing paths for data, which are represented by “IN” or “OUT” ports. A port can also be both “IN” and “OUT”, depending on the specification of the component.

The connection of components, provided by ports, is represented through a link that connects compatible components’ ports. This link is oriented by the notion of connection, which shows the direction of the data flow through the established link, as shown in Figure 1.

A component is an entity that provides a certain kind of service(s) that can be used decoupled in the system, through mappings between input and output ports. Alternatively a component can be described as a composition and interconnection of more basic and simple component instances. Components can thus be composed hierarchically. Figure 2 illustrates this hierarchical composition. The input and output ports of the component instances (A and B) within the composed component (C) can be connected via internal ports acting as places where data actually can be exchanged.

Components can have several ports, each one expressing a certain kind of data that is expected to come in
or go out of them. A simple illustrative example is an arithmetic component, which has two IN ports to receive the operands and two OUT ports: one for the result of the operation and another to inform an occurrence of a possible overflow.

The internal representation of a component may not interest the developer, thus the component can be used as a black-box. It does not matter if the component is implemented using object-oriented or structure-oriented paradigms. It is only important that both “IN” and “OUT” ports, and the “links” are clearly identified.

![Figure 2: Hierarchical Components Composition](image)

Some extensions can be made in these basic concepts of components. To mention one, port-wrappers can include some additional features to component ports. A useful feature is a buffering-wrapper that modifies a port in order to transform it in a buffered-port.

In other to be reusable, a clear component description has to be provided and stored in the component library. The description must provide all key information, such as: available ports and their descriptions; component functionality; component properties (i.e.: size in KBytes); and extended features if any. Additional optional information can also be included in order to improve the reusability, such as performance characteristics.

3. Aspects Weaving Components

Components may be triggered by events that can be periodic or not, and must also complete the execution of their behaviour within deadlines. In order to provide a decoupling of the component functionality from timing concerns, from the early stages of the system development, it is proposed the use of early-aspects to handle the desired time behaviour of the component.

As stated before, the component should represent a well-defined functionality. What we claim is that requirements such as timing, data encryption and other crosscutting concerns must not be programmed inside a component description. They can be enclosed in separate elements, i.e. aspects, which add their features to the affected component. Like this, different variations in a product line can be easier delivered. Non-functional features, such as timing, can be added in a modularized way, without the need of changes in the base components. Such modularization allows variations in the implementation of crosscutting concerns handling without changing components functional concerns. It means that, the functional and non-functional requirements are handled in separate and modularized way from the domain engineering phase, in order to provide easier composition that facilitates the family variation in a software product line.

The goal is that when a component is being developed, it is intended to be reused. This reuse depends on many conditions, especially when real-time constraints must be handled. The introduction of these concerns handling in the component makes it coupled with the context in which it is being developed, hindering the reuse. In order to enable components’ reusability, late binding of the crosscutting concerns must be provided. In this paper, an aspect-oriented approach to address non-functional requirements handling at early development stages is proposed.

As mentioned in our motivation, the intention is to provide reasonable reusability of components for future application project and support to family variations in product lines. A component can be reused in several different ways in the same context. For instance, if a component is been reused in a context where the handling of timing concerns is performed in different way (compared with the handling described in the component), a lot of work will be necessary in order to adapt to this new context.

Using aspects, a generic and decoupled component concept can be used, that does not care for the non-functional constraints that can affect it. This generic component is placed in a specific part of the system, taking into account the conditions over which it runs. After choosing components from a repository, the handling of the mentioned non-functional requirements, which is encapsulated within aspects, can be inserted into components through a tool such as GenERTiCA code generation and aspects weaving tool. The initial version of GenERTiCA, which was presented in [19], uses object- and aspect-oriented models specified in UML as input, generating code from them as well as applying aspects adaptations in the functional code. Currently, GenERTiCA is been adapted by means of adding concepts of components as input to the tool. Therefore it will be possible to generate code and also configuration files from the input component model of the application. Moreover, it
will also support the aspect weaving approach proposed by this work. In this paper, examples of two situations are presented in order to illustrate the proposed idea. The first one is well known within the aspect-oriented community, namely the support for logging or observation of internal probe points. This example was chosen for two reasons: (i) to explore component’s hierarchy concept (explained in Section 2); (ii) to provide an easy example to explain the proposed approach, before presenting a more representative aspect. The second example presents the handling of a conceptually important concern in the design of several embedded systems, which is related to the handling of timing conditions. It is important to highlight that other non-functional requirements have strong impact in the development of embedded applications. Such requirements are being considered in extensions of this work.

3.1. Logging Aspect

The hierarchical composition of components can hide information that in certain situations are important to reach, e.g. for debugging proposes. In order to allow this kind of information inspection, a logging aspect is provided. Thus the state of input and output ports of components that composes a major component can be assessed.

Going back to Figure 2, in order to verify if the delays are being respected and also if the output port of the major component really provides correct data, the logging aspect assesses the transitions of the internal ports. This allows the verification of the internal functionality of a component. As can be seen in the picture, the output of the major component is the result of a composition of the partial results from inner components. A great problem that can occur is the incorrect overall output resulted from unsynchronized data at output ports of inner components caused by the incorrect tuning of temporal parameters.

This situation can be predicted using static analysis of the schedule for the interaction of inner components, followed by the tuning of these components by choosing the correct timing parameters. However, as stated before, nowadays the emerging systems are composed of several different parts (possibly reused from previous projects) going together in the final system. Such situation leads to a very error-prone assemble/composition task, requiring too much work that makes the reuse sometimes more expensive than the new development of these parts.

In order to avoid this, proposals like the one in this paper are presented. However, even with a good overall knowledge of the system and with a hierarchical division in different levels of abstraction, the inspection of the inner parts must be available. Through the use of logging aspects like the one present in this section, the developer can choose by sampling different data in different parts of the system and in different levels of abstraction. It is necessary just to apply the aspect over the interested part, avoiding changes in the internal behaviour of components in order to include logging mechanisms. Modifications in the internal behaviour to include such actions do not scale. Thus undesired additional work is required as new components are aggregated in the system, if they require the same type of handling. Another drawback is that if another property, such as accuracy, must be observed instead of delay, all the work must be redone in each of affected components. Encapsulating the handling of this concern into an aspect, it is enough to modify it once and the changes will be spread over the affected components without unnecessary work.

The logging aspect presented in this work has no intention to provide, by itself, a great novelty as a logging mechanism. The aim is to use it in the context of information assessment and inspection to support the requirements compliance analysis of components composition.

In order to perform this kind of inspection explained above, the logging aspect can check the correctness of a value in a connection before and after the respective attribution. Thus it is required the specification of just a joinpoint, where adaptations are inserted before and after its occurrence in the affect behaviour. Figure 3 presents an example in which the connection over the link between A and B is specified as a joinpoint (named as JPCon) that can be referenced by the logging aspect.

Figure 3: JoinPoint – Connection between Components A and B

The logging aspect is presented in Figure 4. JPCon is used by the logging aspect to make the inspection of the value that is passed from component “A” to component “B” according to certain timing parameters that are specified inside the behavioural adaptation Inspect. It verifies if the value in the connection is equal to zero before the expected time when component “A” will deliver the data to component “B”, and if the value delivered is correct according to the test conditions. The pointcut DDelivery
relates the behavioural adaptation Inspect to the joinpoint JPCon. It is defined as AROUND because Inspect encloses the joinpoint represented by the connection. Another possibility could be defining the joinpoint as a transition of the value in the connection from zero to something different of zero, relating it to a behavioural adaptation that could inspect that value after the transition. However, it would assume that the component is at least working properly, providing a value equal to zero until the new value is provided. As we do not want to take this as an assumption, we suggest the use of the initial proposal because it monitors both zero and non-zero values after the computation performed by the component “A”.

Figure 4: Logging Aspect

3.2. Timing Aspects

The basic idea is to weave timing aspects of behaviour into a component in order to adapt it to the target context. For instance, components can be triggered periodically, aperiodically or sporadically. A measurement device is an example of situation in which a component can be used in both periodically and aperiodically ways: (i) it performs periodical sampling of the rotation of a rotor in an unmanned helicopter in order to control speed or altitude [16]; or (ii) it performs a more demand-driven aperiodic sampling of an environment variable, such as temperature or humidity, as the helicopter comes to a certain area of interest. The sampler component can be functionally the same in both cases; it just takes data from sensor devices as inputs, computes the result of the sampled sensor values and provides digital data as output. However, as explained before, one task must be executed periodically and the other one just under a request, i.e. aperiodically. This sampler device could also be employed as a sporadic task, in which it can be called any time, as in the aperiodic data gathering example, but with an assumed minimum time interval between two activations.

Concretely speaking, the issue in addressing the use of the component in a periodic, aperiodic or sporadic ways brings several practical problems that must be addressed:

1. management and setup of timing parameters;
2. periodic execution mechanism, such as schedulers and loop controls; and
3. feasibility and priority list management to allocate and schedule the execution on processing units.

If the component was designed to be periodic activated, it may be hard to reuse it in an aperiodic operation, due to support of periodicity which is intertwined in the component, such as frequency control of execution. The same reasoning is valid to aperiodicity support, which hinders the reuse of the component in contexts with periodicity requirements. Of course there are alternatives that can be thought to overcome the last situation, but they are not systematically efficient, as one could propose to reuse the aperiodic sampler within a periodic task, by using another component that generates periodical triggers. It is possible, but too inefficient considering that those components would be completely coupled to each other, and also the insertion of an extra unnecessary component into the system. This situation is undesired because we are dealing with complex and constrained systems that have already several kinds of different needs handled by different elements.

In order to address these needs, this paper proposes an extension to the work presented in [12] in order to cover the concept of components. Aspects defined in the DERAF library are used to weave adaptations within components in an abstract level. DERAF is an extensible high-level aspect framework based on the aspect oriented conceptual model proposed in [14], which contains aspects to handle different embedded systems concerns, such as timing, memory footprint and energy consumption. As an example, consider the Timing Package (Figure 5) from DERAF, adapted for the component usage. For more details about DERAF interested readers are referred to [12] and [15].

TimingAttributes: adds timing attributes to components (e.g. deadline, priority, WCET, start/end time, and so on), and also the corresponding initialization of these attributes;

PeriodicTiming: deals with periodic activation of components adding the corresponding period value to the scheduler that triggers the component;

SchedulingSupport: inserts a scheduling mechanism to control the execution of individual and concurrent components;

TimeBoundedActivity: temporally limits the execution of an activity, that is, adds the mechanism to restrict the maximum execution time for an activity. The time counting begins immediately before the start of the activity and must provide an exception handling mechanism in case the maximum allowed execution time is exceeded.

1 Sporadically, means that there is at least a minimum known interval between the occurrence of aperiodic events.
In the current DERAF version those concepts are more related to the design in a finer grain (weaving of objects). In the present work similar ideas are used but in a more abstract and high-level way, as we are dealing with the system on a coarse grained level. For example, a deadline or WCET value described on this level for a component can split into different values for objects in a representation provided in [12].

A serious concern when dealing with embedded systems is the rational use of shared processing resources, especially in cases where real-time requirements must be met. This problem involves the scheduling of the components allocated to distributed processing resources. System requirements include lots of complex information to be handled by developers in order to develop new components or to assemble components from a library. This problem was already discussed in related works, as presented in [17]. Thus, in order to free developers from the problem of inserting the mechanisms to provide the handling of schedulability concerns, an aspect that handles this feature, the SchedulingSupport aspect, is proposed. The details on such mechanism are provided in [12][15], but basically its task can be summarized as the insertion of elements that perform the scheduling mechanism, e.g. the scheduler itself, the timer and execution frequency control mechanism. Additionally to that, it also performs the components insertion in the feasibility list of processing resources.

In the original concepts presented in the DERAF, the inserted elements are objects that interact with processing units and the objects that compose the system. However, the proposal to make the SchedulingSupport aspect compliant with the coarser grain idea is to insert an additional component that handles the complexity related to the scheduling tasks in the system. This component is responsible for the scheduling mechanism, overwriting it to support the addition of components in the processing feasibility list for execution in the processing unit. As in the original work, the developer is free to choose the scheduling policy that will be used, which is related to system requirements.

3.2.1. Practical Example

In order to illustrate the concepts explained above a practical example describing the usage of the timing aspects is presented.

One of the OUT ports of a component is wrapped by a buffer-wraper. Additionally, this port is supposed to deliver data of when the buffer is full. Suppose that this component is reused in a project that needs a component with the same functionality and the same characteristic of its OUT ports. However, there is a timing requirement telling that after the expiration of “T” milliseconds the buffer must deliver its data even if it is not full or not. How to adapt this component to fulfil this requirement without changing its internal mechanisms? An aspect that weaves this behaviour can be a solution to this problem.

Figure 6 shows the above exemplified component. As can be seen the OUT Port 2 is wrapped by the buffering-wraper. This wrapped port is identified as a JoinPoint, designated by “JPWraPort”. The OUT port keeps itself its identity, thus can also be addressed and identified using a JoinPoint, e.g. the JPOUTPort joinpoint.

By using the TimingAttributes aspect, it is possible to relate a deadline to the wrapped-port. Thus, when this deadline expires, it delivers the data, transforming that port to a timed-buffered port. The mechanism to check the deadline expiration is inserted by the SchedulingSupport aspect, which just setups a timer to announce when the deadline is expired. If there is no timer in the target system, it can insert a timer component. Figure 7 shows the TimingAttribute and SchedulingSupport aspects using a notation presented in [12] that are explained in the following.

In the TimingAttributes aspect, the pointcut BuffPort relates the joinpoint JPOUTPort to the structural adaptation Deadline, which inserts the deadline attribute to the buffered-wrapped port.
pointcut OUTP in the SchedulingSupport aspect links the joinpoint JPOUTPort to the TriggerTimer behaviour adaptation, which triggers the count down until the deadline expiration. The modifier AFTER is applied in the OUTP pointcut to indicate that the deadline timer starts count down after the insertion of the first data in the empty buffer.

4. Related Work

SysML [13] is an effort to refine UML defining a general purpose modeling language for systems engineering. It aims at supporting the design of complex heterogeneous systems based on a mix of software and hardware architecture. It presents a linkage between requirements and test procedures. However, it does not address non-functional requirements directly. It just presents reference to these requirements in annotations along the project diagrams. In contrast, in this paper we present dedicated elements, i.e. aspects, to handle non-functional requirements in a system abstraction level, instead of only reference annotations along diagrams.

A set of tools named VEST (Virginia Embedded System Toolkit) [17] uses aspects to compose a new distributed embedded system based on a component library. Those aspects check the possibility of composing components with the information taken from system models. This work also has a library of aspects and supports a sort of model weaving, but it does not provide an overview of the system as a composition of high-level components what makes it harder to make changes and provide variations of a software product if compared with the ideas presented in our work. Moreover, when a change is decided, what is done is begging the proposed composition process again.

ACCORD [18] is an AO methodology to design component-based software for real-time systems. A library of software artifacts, a modeling environment and a configuration compiler composes the toolset supporting ACCORD, which uses UML to specify components selection. Then a “compilation file” is generated in order to compile the source code of aspects and components. In our approach, a similar modeling approach is provided, but with the difference that we focus more in the system level view instead of the finer grained design. Another difference is related to the transition from design to implementation, but it is out of the scope of this paper. Interested readers are referred to [19].

CoSMIC is an approach that works with a set of Domain Specific Modeling Languages (DSML) in order to provide a component-based implementation for embedded real-time systems. In our approach we propose a more general notation that as a future work will be standardized with UML notation, overcoming the lack of standardization problem. CoSMIC do handle crosscutting concerns, such as QoS control. However, the separation of functional and non-functional dimensions is not that clear if compared with a solution based on the use of aspects to handle non-functional requirements.

AspectCCM [5] is a proposal that addresses the problem of crosscutting concerns handling in CORBA Component Model (CCM) through the use of aspect oriented programming. The authors present the problem of crosscutting components by giving an illustrative example of a bank account controller system, in which they show how the occurrence of the tangling code can be prejudicial in the reuse of components. Further, the authors present an interesting way to address this problem by aspects as the means to eliminate the crosscutting dependencies among CCM components. The ideas presented are interesting but they are related to late phases of design to implementation. On the other hand, our work aims to address the same problem in early phases of development, i.e. from the transition of the requirements to the design, or at least the early stages of the design. Besides, our approach focuses on embedded software product lines, aiming at providing variability in a software family related to embedded systems concerns, while the mentioned work focused in general information systems.

5. Concluding Remarks and Future Work

This paper presents an initial phase of an ongoing work that proposes to take advantage of two paradigms, namely components and aspect orientation, in order to improve the reusability in the development of embedded software product lines. The proposal consists of a high-level component description that enables the developer to have an overview of the entire project, as well as the possibility to go deeper inside the components. However, the major contribution is bringing together the concepts of aspect to the conceptualization of the system. This allows the developer not only to separate the functional and non-functional dimensions of the system, but also insert mechanisms to control the quality and the correctness of the development with all problems that an intertwined development (mixing the handling of functional and non-functional requirements) has [21].

Related work in the area, such as [17] and [18], presents approaches that deal with important subjects, such as the tool support for implementation and language usage [13]. However, it is remarkable that much of the work done so far misses an approach that handles the requirements and shows an overview of the system under development at the same time. We believe that the proposed approach covers this gap with an easy to use and understandable concept, which brings together components and aspects in a high-level of abstraction.
The next step is to improve the graphical representation of the presented concepts by using UML and the methodological tool support from our previous work presented in [12], [15] and [19]. After that, the idea is to extend other aspects from DERAF, namely those specifically related to embedded and distributed concerns, such as energy consumption and concurrency control, the same way that was done with those related to the handling of timing requirements. Another important task is to provide data to assess how the use of these ideas improves the reusability in embedded systems development. In order to that, we will extend the assessment framework present in [22] and apply it to a case study in a similar way we did in [23]. Another goal is to use the presented approach to model a flexible and modular middleware to heterogeneous wireless sensor networks, in which components and aspects will play key roles in the middleware adaptation and customization. Such adaptation must consider the node type in which the middleware will be deployed, characterizing variations in a family of software product line.

6. Acknowledgment
Author Edison Pignaton de Freitas thanks the Brazilian Army for the grant given to him to follow the PhD program in Embedded Real-time Systems in Halmstad University in Sweden in cooperation with UFRGS in Brazil.

7. References