Abstract

While engineering of embedded & real-time systems has moved much into the focus of the research community, being strongly promoted by those prominent application areas as the automotive, aerospace & defense, or telecommunications industry, small embedded & real-time systems, as they can be found in somehow marginal application areas as the industrial automation, are still treated a bit stepmotherly. In particular, profound methodical support for the software development of such small devices is almost unavailable. With MeDUSA we especially target the domain of such small embedded & real-time systems, and explicitly address the very special technological, economical, and organizational constraints that have to be faced in such marginal application areas.

1. Introduction

The initial incentive for the development of MeDUSA (Method for UML-based Construction of Embedded & Real-Time Software) resulted from the evaluation of several pilot projects, being jointly executed at the ABB Corporate Research Centre and the ABB Automation Products GmbH, to gather experience regarding the applicability of the object-oriented Concurrent Object Modeling and Architectural Design Method (COMET) [3] for the development of small embedded & real-time systems in the industrial automation application area. While COMET was regarded to be a promising starting point, the evaluation results revealed some noticeable shortcomings [11], so that we started development of a method, namely MeDUSA, suitable to meet the very special characteristics in the respective domain.

Having first started as a mere evolution of COMET, MeDUSA has undergone several changes and may now be regarded as a distinct and self-contained method. Amongst other things, this expresses itself in the characteristic of the method to be indeed not object-oriented but class-based (compare [18]), in the fact that MeDUSA is not a simple design but an overall construction method (thus covering also a seamless transition from detailed design into a procedural implementation language), and that it is based on the current UML language standard [13], taking in particular advantage of the modeling capabilities in terms of composite structures, which have shown advantages in the domain of small embedded systems (compare [10]).

The paper is outlined as follows. First, the domain being targeted with MeDUSA is introduced. Based on this, challenges regarding the software development in this domain are accentuated, leading to a set of goals for MeDUSA. Then, important characteristics of MeDUSA to fulfill these goals are described, before the method’s workflow is introduced in detail. The paper is concluded with some comments on the state of its validation and on future work.

2. Scope

The domain covered by MeDUSA may be characterized, as denoted by its acronym, as software construction of small embedded & real-time systems. However, as this domain is rather broad - and even if we think that MeDUSA might be applicable to quite a few application areas within it - understanding the method and its characteristics can be best achieved by taking into consideration the application area, MeDUSA was initially developed for, namely software of small embedded & real-time systems in the industrial automation area, i.e. field device software - to be more concrete.

Field devices are rather small embedded & real-time systems. Ranging from drivetrains and positioners to measurement devices, field devices are used for process automation purposes across various industries such as food, chemicals, water and waste water, oil and gas, pharmaceutical, and oth-
ers. They thus come in a multiplicity of different variants. Measurement devices for instance range from simple low cost mass products like temperature measurement devices for unhazardous environments up to very upscaled marginal products as flow measurement devices to measure explosive gases.

Being imposed on them by the industrial contexts the systems are used in, measurement devices - or field devices in general - may be characterized by hard resource constraints in terms of memory consumption, power consumption, and computation time. Strong constraints regarding reliability are always natural for such devices as well, whereas some devices, being used in hazardous environments, are additionally liable to strong safety requirements. From a hardware viewpoint, measurement devices may thus be characterized as single or simple 16- or 32-bit multiprocessor systems, being equipped with physical memory of about 32-512 KByte ROM and 0.5 to 512 KByte RAM.

Due to those stringent resource constraints, the software running on those devices is mostly realized in the C implementation language, which pretty much reflects the current state that can be faced in the overall embedded & real-time community (compare surveys in [1] and [14]). Besides this, the software running on those systems may be characterized as having a rather low complexity. That is, it has a rather static run-time structure in such a sense that the software is initialized at device startup and does not dynamically reconfigure itself during device operations. Even in case of a distributed hardware architecture in terms of a multiprocessor system, the software may not be regarded to be intensely complex concerning the distribution aspect, as each peripheral microcontroller unit is normally connected to the main microcontroller via a separate serial communication interface, and all communication is initiated from the main microcontroller. Regarding its real-time properties, the software may be characterized to be real-time in terms of its main measurement task, that is related to the gathering of analogue data samples via the connected A/D converters, the calculation of measurement values (signal processing), and the output of computed measurement values on the communication buses and the local display, the device is equipped with. All other functionality, related to configuration, diagnosis, or maintenance, is usually not real-time critical.

From an economical and organizational perspective, strong constraints are as well observable. That is, an inherently existing cost pressure is characteristic for the development of such devices, as well as regulatory constraints, resulting from safety and reliability requirements. Further organizational constraints that result from the distributed development, such systems are normally realized in, are restricting their development. What is as well characteristic is that most of the software development is indeed not performed by software engineering professionals, but by electrical engineers, communication engineers, process engineers or physicians. While this should not be understood as a discrimination of those professions it points to the problem that software is developed mostly by domain experts, lacking profound software engineering skills.

3. Challenges & Goals

Model-based software engineering seems to be a reasonable means to deal with the increased complexity and the special technical, economical, and organizational constraints being faced in the respective application area. Interestingly, it does not have penetrated the domain to a large extend, as surveys on the state-of-the-practice in software engineering of embedded & real-time systems unveil ([16] [15]).

The reasons for this are manifold. First, while model-based software engineering does not lead to larger and less efficient code, most model-based software engineering approaches seem to facilitate the use of higher abstracting implementation technology as object-oriented programming languages or component-based middleware, which is somehow contrasting with the very restrictive technical constraints being faced in the context of small embedded & real-time systems. Often model-based methodical support is also not continuous in a sense that the transformation of a detailed design model into source code is left open or - leading to the same result - regarded to be obvious, what is mostly not the case if no object-oriented but a procedural implementation language is used.

It is the goal of MeDUSA to overcome those problems. In detail, the method was designed according to the following goals:

- **Methodological Completeness** - A continuous methodical approach, covering all constructive software engineering activities from the early elicitation of software requirements, via analysis and design, up to the late transition into the resulting implementation, has to be developed.

- **Constraint Adequateness** - The method has to adequately deal with the special technical, organizational, and economical constraints being faced in the application area of small embedded & real-time systems. That is, it has to avoid that concepts or technologies are used, which are contrasting the technical restrictions being faced, leading to a breach between detailed design and implementation. It further has to assure its practical applicability and easy adoptability. It thus has to be based on standard languages and standard tools wherever possible.
3.1. Characteristics

To meet these goals, MeDUSA was designed to be a methodologically complete construction method, covering all constructive activities of the software development lifecycle from the early requirements gathering up to the concluding implementation. To meet the special organizational constraints in the respective application domain, it was explicitly designed to be iterative, what seems to best match the requirements in terms of flexibility and customizability that result from the integration of software development into a larger system engineering context. Furthermore, MeDUSA was designed to be based on standards (the UML is used as notation), so that easy adoption and understanding is achieved, and market-available tools can be employed.

In contrast to its predecessor COMET, MeDUSA was designed to not facilitate object-oriented concepts like inheritance and polymorphism, as this would hinder a seamless transition of the detailed design into a procedural implementation, and may thus be characterized as being class-based (compare classification provided in [18]). This is achieved by what we denote as the instance-driven nature of the method. That is, all of the analysis as well as most of the design are performed by modeling on the instance level, i.e. in terms of objects rather than classes, and class design is postponed to the very late detailed design. This way, object-oriented concepts, which are expressed on the class level, are disregarded up to then and are thus omitable.

What has to be especially emphasized is that MeDUSA was designed to comprise a continuous real-time analysis, which is performed based on the early requirements, as well as on the results of the analysis and design activities, to gain understanding and awareness on how the software is able to meet the non-functional real-time requirements, which are imposed on it.

4. The MeDUSA Workflow

MeDUSA is defined in terms of a workflow [1] comprising five phases, corresponding to the development lifecycle phases being covered, as indicated in Figure 1.

All phases comprise iterations over a set of tasks, being initially executed in the respective phase, as well as iterative backflows to tasks that have been initially performed in earlier phases, as indicated in Figure 2 exemplarily for the Architectural Design Phase.

The tasks being primarily executed in a phase together with their dependencies are defined by a respective workflow pattern. They are explained in detail in the following sections.
4.1. Requirements Workflow Pattern

MeDUSA is a use-case driven method, so Use Case Modeling, i.e. the construction of a UML use case model, as well as Use Case Description Modeling, the development of UML-based or textual detailed descriptions for each identified use case, are the essential tasks being performed. As denoted by Figure 3, they are executed in parallel, although Use Case Modeling will quite naturally be started with a slight advance, as an initial set of use cases will have to be identified, before detailed descriptions for the use cases can be created. While use case modeling is first and foremost suited to capture functional constraints, in the context of embedded & real-time systems, especially timing and concurrency constraints have to be regarded.

4.2. Analysis Workflow Pattern

Based on the Requirements Model, which captures the functional and non-functional requirements of the software in terms of use cases, the Analysis Model, being build to gather profound understanding of the problem domain, is constructed in terms of collaborating (analysis) objects.

As described in [9], the explicit modeling of such constraints is supported by the MeDUSA Actor Taxonomy, as outlined by Figure 4, which is used to denote distinct actors as sources of all periodic and aperiodic events. Based on this, a Preliminary Real-Time Analysis may be performed, as motivated and described in [9] and [8].
showing the identified (analysis) objects and their relationships.

Further analysis objects (control and application-logic) are defined during the successive Inter-Object Collaboration Modeling. Here, object collaborations are developed, whose participants - already identified entity, interface, and trigger objects, as well as newly identified ones - collaboratively perform the scenarios subsumed by the use cases, captured in the Requirements Model, by means of UML communication or sequence diagrams.

Having identified the collaborations, and having defined how the system behavior manifests itself in the inter-object behavior of those collaborations, the conceptually last modeling step of the Analysis Workflow Pattern, namely Intra-Object Behavior Modeling, is concerned with modeling of internal object behavior for those objects, where this is not trivial. Analogously to the Requirements Workflow Pattern, a successive Intermediate Real-Time Analysis is performed based on the Analysis Model, again not to proof schedulability and performance of a later system design - which actually has not been developed yet, but to indicate initial performance problems and to identify hot spots that have to be regarded in particular in the following.

4.3. Architectural Design Workflow Pattern

Having gained decent and profound understanding of the problem domain, architectural design is the first step to compose a solution. All tasks needed to construct a Design Model, i.e. specifying the overall software architecture in terms of a decomposition of the system into fully encapsulated and self-contained subsystems, are correspondingly subsumed by the Architectural Design Workflow Pattern.

As indicated by Figure 7, architectural design is started by Identifying Subsystems. That is, the (analysis) objects are grouped together, resulting in a couple of subsystems. A number of potentially conflicting design principles like Locality in Changes, Functional Coupling [5], or Task Coupling [8] may be quoted to guide and support the identification. Additionally to grouping together objects to subsystems, the initial externally visible interfaces of the subsystems have to be defined, by deriving them from the message communication established between the aggregated objects and all external ones. A UML composite structure diagram denoting the internal decomposition of each subsystem (in terms of aggregated parts) as well as its external interfaces in terms of exposed required and provided interfaces, as well as a UML class diagram showing the signatures of the exposed interfaces, are developed to document the outcome of this task. Further, UML sequence and (protocol) state machine diagrams are developed to denote behavioral aspects of the subsystem and its externally visible interfaces.

Subsystem Consolidation is then performed to ensure that the initial subsystem decomposition is sustainable under design considerations. That is, it literally turns the initially partitioned analysis objects into design objects, by splitting respectively merging together objects, while indeed also the preliminary identified interfaces of the subsystems are consolidated. That is, they are as well restructured according to their later integration needs and additionally, a detailed class design is developed for the interfaces themselves, as well as for all data types being exchanged via the interfaces.

Subsequently, Structural System Architecture Modeling and Behavioral System Architecture Modeling are performed, being concerned with integrating the different subsystems, which have been individually defined in terms of their provided and required interfaces, to an overall software system. Here, a UML composite structure respectively component diagram is employed, denoting how the subsystems are interconnected via their required and provided interfaces to form an overall system, and UML sequence diagrams are further used to specify how system behavior manifests itself on the system level, i.e. in terms of collaborating subsystems.

To analyze feasibility of the software architecture in terms of performance and schedulability, a Conclusive Real-Time Analysis is performed. Unlike the previously executed Preliminary or Intermediate Real-Time Analysis, the analysis is now based on the actual task design, which manifests itself after the division and consolidation of the active tasks that the initial subsystem decomposition is sustainable under design considerations. That is, it literally turns the initially partitioned analysis objects into design objects, by splitting respectively merging together objects, while indeed also the preliminary identified interfaces of the subsystems are consolidated. That is, they are as well restructured according to their later integration needs and additionally, a detailed class design is developed for the interfaces themselves, as well as for all data types being exchanged via the interfaces.

Subsequently, Structural System Architecture Modeling and Behavioral System Architecture Modeling are performed, being concerned with integrating the different subsystems, which have been individually defined in terms of their provided and required interfaces, to an overall software system. Here, a UML composite structure respectively component diagram is employed, denoting how the subsystems are interconnected via their required and provided interfaces to form an overall system, and UML sequence diagrams are further used to specify how system behavior manifests itself on the system level, i.e. in terms of collaborating subsystems.

To analyze feasibility of the software architecture in terms of performance and schedulability, a Conclusive Real-Time Analysis is performed. Unlike the previously executed Preliminary or Intermediate Real-Time Analysis, the analysis is now based on the actual task design, which manifests itself after the division and consolidation of the active tasks that the initial subsystem decomposition is sustainable under design considerations. That is, it literally turns the initially partitioned analysis objects into design objects, by splitting respectively merging together objects, while indeed also the preliminary identified interfaces of the subsystems are consolidated. That is, they are as well restructured according to their later integration needs and additionally, a detailed class design is developed for the interfaces themselves, as well as for all data types being exchanged via the interfaces.
trigger objects, from which all concurrent system behavior originates.

4.4. Detailed Design Workflow Pattern

Having specified the software architecture in terms of fully encapsulated and self-contained subsystems, the detailed design of each subsystem has to be developed. As indicated by Figure 8, this is performed in terms of Class Design Modeling, where for each part (object), belonging to a subsystem’s internal decomposition as well as for each port, forming its externally visible interface by aggregating required and provided interfaces, a respective class is designed. While all preceding tasks have been performed on the level of the overall system, Class Design Modeling may be performed individually and in parallel for each subsystem, as all external interfaces have already been defined.

4.5. Implementation Workflow Pattern

Last, the detailed design has to be transferred into source code, where MeDUSA, as already mentioned, facilitates a seamless transition into a procedural implementation in the C-language. Conceptually, the transition is performed in two steps. First, the information already captured explicitly in the Design Model is transferred into source code equivalents. Second, all detail code needed to build up a decent and complete source code base for the overall system has to be added. While the first, Code Generation, is more or less automateable by a code generation tool, the latter step has to be performed manually. While conceptually one step, the second part (adding the code details) is actually performed in terms of two tasks, namely Implementing and Integrating. While the first is concerned with completing the code base of each subsystem, and may thus be executed in parallel for each subsystem, the latter is concerned with programming the glue code that is needed to integrate the individual subsystem specific code fragments, as well as the code that is concerned about non-subsystem specific aspects, like startup and initialization.

4.6. Conclusion & Outlook

Model-based software engineering seems to be a very promising approach for the development of small embedded & real-time systems. The intense use of models throughout all phases of the development does not only facilitate a concise and systematic procedure, it also offers an increased potential in terms of traceability and analyzeability, which seems to be an adequate means to deal with the stringent technical constraints, those systems are exposed to.

However, it obviously has not yet achieved to penetrate the domain of small embedded & real-time systems, as the special technical and organizational constraints inherent to the domain are not adequately addressed. Therefore, we developed a method that especially suits the needs of software development in the respective domain. Although the method was initially designed targeting the development of field devices, we think that it might be applicable to a broad range of small embedded & real-time systems.

The Second Edition of MeDUSA [8] is currently being evaluated in a pilot project, being executed at ABB Automation Products GmbH. Any lessons learned therein will - as with earlier revisions - be incorporated into the method, whose most recent definition can always be retrieved from the MeDUSA project website [7] in electronic form.

While basic tools for the MeDUSA-specific generation of C code [2][6] and for the specific methodical support [4] have already been developed based of our ViPER platform [17], we are currently spending further efforts on these topics, as a lack of profound tool support seems to be a major hindrance, most recent development methods are suffering from.
References


