A SIMPLIFIED STUDY OF EMBEDDED SYSTEM: REFERENCE ARCHITECTURE DESIGN FOR DIFFERENT DOMAINS OF EMBEDDED SYSTEMS

AKULA RAJITHA
M.Tech, Department of ECE,
CMR Technical Campus
Telangana.

Abstract:
In this paper, we have evolved generic software architecture for a domain specific distributed embedded system. The system under consideration belongs to the Command, Control and Communication systems domain. The systems in such domain have very long. A number of techniques and software tools for embedded system design have been recently proposed. However, the current practice in the designer community is heavily based on manual techniques and on past experience rather than on a rigorous approach to design. To advance the state of the art it is important to address a number of relevant design problems and solve them to demonstrate the power of the new approaches. We analyze the results obtained with our approach and compare them with the existing design underlining the advantages offered by a systematic approach to embedded system design in terms of performance and design time. The content of this paper addresses the issues regarding the reference architecture design for different domains in the context of a system of systems that is specific to today's embedded systems. The reference architecture contains core services of the domains included in abstract features package. The appropriate architectural style is provided by a knowledge base through service taxonomy. Services, commonality, variability management and rules for product derivation and configuration are the main issues considered in the architectural design process.

Key words: *embedded systems, Domain- Software Architecture, cross domain product line, Reference Architecture.*

1.0 Introduction:

Embedded systems are usually evaluated according to a large variety of criteria such as performance, cost, flexibility, power and energy consumption, size and weight. As these kinds of non-functional objectives are very often conflicting, there is no single optimal design but a variety of choices that represent different design trade-offs. As a result, a designer is not only interested in one implementation choice but in a well chosen set that best explores these trade-offs. In addition, embedded systems are often complex in that they consist of heterogeneous subcomponents such as dedicated processing units, application-specific instruction set processors, general purpose computing units, memory structures and communication means like buses or networks. Therefore, the designer is faced with a huge design space. Embedded systems are resource constrained because of tight cost bounds. Therefore, there is resource sharing on almost all levels of abstraction and resource types that makes it difficult for a designer to assess the quality of a design and the final effect of design choices. This combination of a huge design space on the one hand and the complexity in interactions on the other hand makes automatic or semi-automatic (interactive) methods for exploring different designs important. As a consequence, embedded systems are usually domain-specific and try to use the characteristics of the particular application domain in order to arrive at competitive implementations. In order to achieve acceptable design times though, there is a

need for automatic or semi-automatic (interactive) exploration methods that take into account the application domain, the level of abstraction on which the exploration takes place and that can cope with conflicting criteria. Following the usual hierarchical approach to embedded system design, there are several layers of abstraction on which design choices must be taken. Above the technology layer one may define the abstraction levels 'logic design and high level synthesis', 'programmable architecture', 'software compilation', 'task level' and 'distributed operation'. These terms are explained in more detail in Section 2. Design space exploration takes place on all of these layers and is a generic tool within the whole design trajectory of embedded systems. Embedded systems are found in a variety of common electronic devices such as consumer electronics ex. Cell phones, pagers, digital cameras, VCD players, portable Video games, calculators, etc.,

We argue with our experiences in the software architectures design and analysis for embedded systems domains and other researchers' recent studies that will be revealed during the paper.

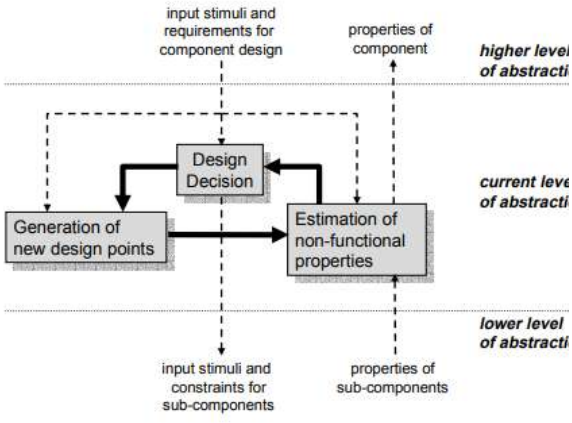

Figure 1: Embedding of exploration in a hierarchical design trajectory for Embedded Systems.

The processing units of the embedded system:

1. Processor in an Embedded System A processor is an important unit in the embedded system hardware. A microcontroller is an integrated chip that has the processor, memory and several other hardware units in it; these form the microcomputer part of the embedded system. An embedded processor is a processor with special features that allow it to be embedded into a system. A digital signal processor (DSP) is a processor meant for applications that process digital signals.
2. Commonly used microprocessors, microcontrollers and DSPs in the small-, medium-and large scale embedded systems.
3. A recently introduced technology that additionally incorporates the application-specific system processors (ASSPs) in the embedded systems.
4. Multiple processors in a system.

Embedded systems are a combination of hardware and software as well as other components that we bring together into products such as cell phones, music player, a network router, or an aircraft guidance system. They are a system within another system as we see in Figure 2

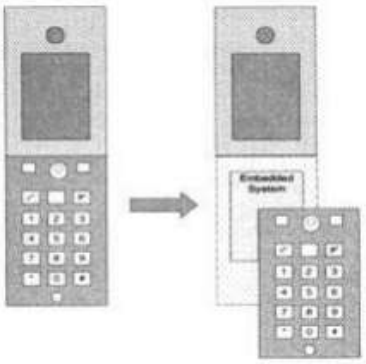

Figure 2: Simple embedded system

2.0 Literature Review:

Abdoulaye Gamatie, (2011), Modern embedded systems integrate more and more complex functionalities. At the same time, the semiconductor technology advances enable to increase the amount of

hardware resources on a chip for the execution. Massively parallel embedded systems specifically deal with the optimized usage of such hardware resources to efficiently execute their functionalities. The design of these systems mainly relies on the following challenging issues: first, how to deal with the parallelism in order to increase the performance; second, how to abstract their implementation details in order to manage their complexity; third, how to refine these abstract representations in order to produce efficient implementations. This article presents the GASPARD design framework for massively parallel embedded systems as a solution to the preceding issues. GASPARD uses the repetitive Model of Computation (MoC), which offers a powerful expression of the regular parallelism available in both system functionality and architecture. Embedded systems are designed at a high abstraction level with the MARTE (Modeling and Analysis of Real-time and Embedded systems) standard profile, in which our repetitive MoC is described by the so-called Repetitive Structure Modeling (RSM) package. Based on the Model-Driven Engineering (MDE) paradigm, MARTE models are refined towards lower abstraction levels, which make possible the design space exploration. By combining all these capabilities, GASPARD allows the designers to automatically generate code for formal verification, simulation and hardware synthesis from high-level specifications of high-performance embedded systems. Its effectiveness is demonstrated with the design of an embedded system for a multimedia application.

Karen Zita Haigh, (2015), We describe our application's need for Machine Learning on a General Purpose Processor of an embedded device. Existing ML toolkits tend to be slow and consume memory, making them incompatible with real-time systems, limited hardware resources, or the rapid timing requirements of most embedded systems. We present our ML application, and the suite of optimizations we performed to create a system that can operate effectively on an embeddded platform. We perform an ablation study to analyze the impact of each optimization, and demonstrate over 20x improvement in runtimes over the original implementation, over a suite of 19 benchmark datasets. We present our results on two embedded systems.

Salome Maro, (2017), Currently, development efforts in embedded systems development lead to a large number of interconnected artifacts. Traceability enables understanding and managing these artifacts as they evolve. However, establishing traceability is not a trivial task, it requires the development organization to plan how traceability will fit into its processes and provide tools to support traceability establishment. In practice, guidelines for how traceability should be established are lacking. Therefore, companies struggle with establishing traceability and making the most of traceability once it is established.

3.0 Methodology:

3.1 Design Description:

Our approach is applied to design architecture for various application domains of embedded systems (e.g. automotive, avionics, consumer electronics, control, etc.). Embedded systems applications are doing control. Control activities could be measuring physical variables (sensing), storing data, processing sensors signals and data, influencing physical variables (actuating), monitoring, supervising, enabling manual

and automatic operation, etc. We consider an embedded system domain a collection of cooperating services that deliver required functionality. These services may be executed in a networked environment and may be recomposed dynamically. We propose an approach that extends to three levels the architecture development method for embedded systems application domains. In Figure 2 the reference architecture includes core services and it focuses on commonality. Domain architecture includes domain specific services and it requires variability management concerns. Finally, the last level is dedicated to the set of products architectures. On this level rules for product derivation and configuration should be included.

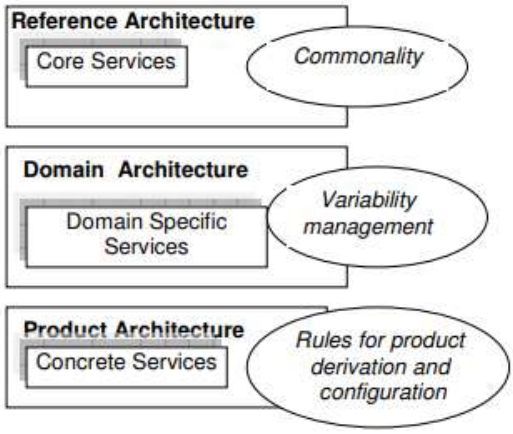

Figure 3. Cross domain architecture design.

A feature model is a prerequisite of this approach. We propose a feature model as described in Figure 3. This model is essential for both variability management and product derivation, because it describes the requirements in terms of commonality and variability, as well as defines the product line dependencies. Feature types may be mandatory, optional, optional or optional alternative. Dependencies specified by the model are called composition rules between domain features. The requires rule expresses the presence implication of two features and the mutually exclusive rule captures the mutual exclusion constraint on feature combinations. Reference architecture defines quality attributes, architectural styles and patterns and abstract architectural models (Figure 4). Quality attributes clarify their meaning and importance for core service components. The interest of the quality attributes for the reference architecture is how the quality attribute interacts with and constrains the achievement of other quality attributes (i.e., trade-offs) and what the user's view of quality (i.e., quality in use) is. Embedded systems services have to meet many quality attributes, such as modifiability and integrability. Modifiability of a service can be divided into the ability to support new features, simplify the functionality of an existing system, adapt to new operating environments, or restructure system services.

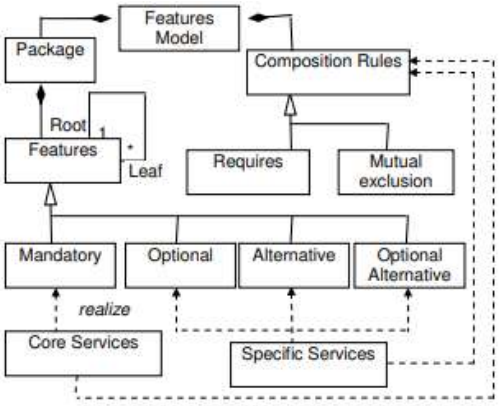

Figure 4. Reference architecture realization.

The styles and patterns are the starting point for architecture development. Architectural styles and patterns are utilized to achieve qualities. The style is determined by a set of component types, the topological layout of the components, a set of semantic constraints and a set of connectors. A style defines a class of architectures and is an abstraction for a set

of architectures that meet it. A small taxonomy identifies five style categories: independent components, data-flow cantered, data-cantered, virtual machine, and call-and return style. An architectural pattern is a documented description of a style or a set of styles that expresses a fundamental structural organization schema applied to high-level system subdivision, distribution, interaction, and adaptation. Design patterns, on the other hand, are on a lower level. They refine single components and their relationships in a particular context, and idioms describe how to implement particular aspects of components using the given language.

In this way the reference architecture creates the framework from which the architecture for new embedded systems is developed. It provides generic architectural services and imposes an architectural style for constraining specific domain services in such a way that the final product is understandable, maintainable, extensible, and can be built cost-effectively. Potential reusability is highest on the reference architecture level. Core services and the architectural style of the reference architecture are completely reused on the domain architecture level.

Reference architecture is build based on a service taxonomy. We adopted the idea from WISA of an existing knowledge on software engineering that is integrated and adapted to service engineering for embedded systems. The standards related to each embedded system domain, applicable architectural styles and patterns and existing concepts services and components are the driving forces of embedded systems development. A service taxonomy defines the main categories called domains. Typical features that have been abstracted from requirements characterize services. The purpose of the service taxonomy is to guide the developers on a certain domain and getting assistance in identifying the required supporting services and features of services.

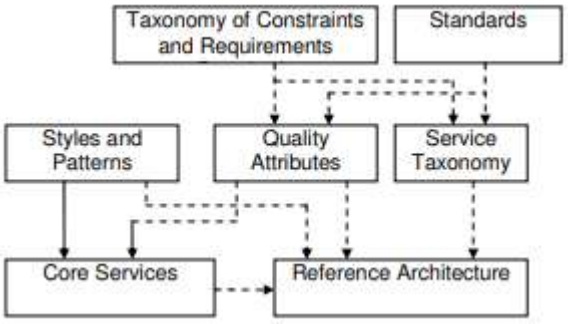

Figure 5: Reference architecture realization.

Domain architecture describes readymade building blocks that assist application/products developers in using specific domains services. When the reference architecture has been defined, the existing components and services are considered as building blocks in the architecture of set of products. The domains services provides variable assets repository. Variability appears in functional and non functional requirements (including quality attributes). A structured domain architecture repository may be provided. A schema for this repository has to be defined in a form of relationships between services. In this way we are mapping domain specific services to core abstract services. Specialization relation is a solution to be used for variability management. Modeling run-time quality attributes variability requires tool support. Approaches are under development and refining. Monitoring mechanisms, measuring techniques and decision models for making tradeoffs should be better defined and validated.

4.0 Analysis with example:

Description:

In this section we describe a simple multiple domains example to illustrate our

vision for product line architecture development. The cross-domain reference architecture considers measurement control, data acquisition control and data management domains. Our example is abstracted from our experiences with the architecture design of a scientific onboard silicon X-ray array (SIXA) spectrometer control software. SIXA is a multi-element X-ray photon counting spectrometer. It consists of 19 discrete hexagonally-arranged circular elements and specific domain hardware architecture. The SIXA measurement activity consists of observations of time-resolved X-ray spectra for a variety of astronomical objects. Figure 5 introduces the context view of SIXA considering it a measurement controller. External elements that interface with our measurement controller are a command interface and physical devices (detectors) representing sensors and actuators. The system is programmed and operates using a set of commands sent from a command interface.

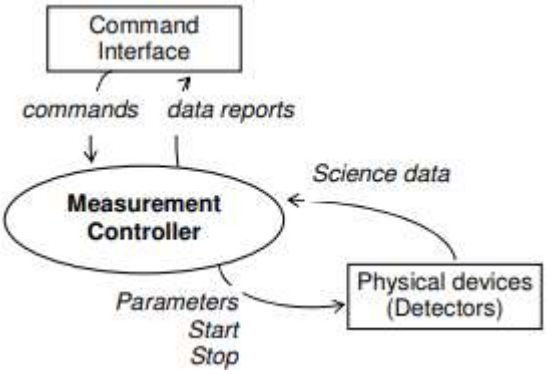

Figure 6. Context view.

The role of the spectrometer controller is to control the following measurement modes:

• Energy Spectrum (EGY), which consists of three energyspectrum observing modes: Energy-Spectrum Mode (ESM), Window Counting Mode (WCM) and TimeInterval Mode (TIM).

• SEC, which consists of three single event characterization observing modes: SEC1, SEC2 and SEC3.

Each measurement mode could be controlled individually. A coordinated control of the analog electronics is required when both measurement modes are on.

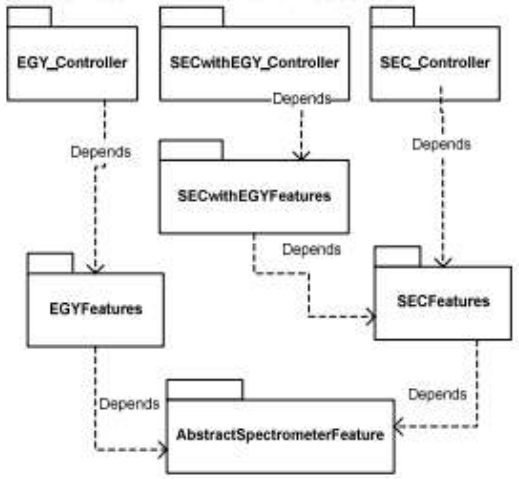

Figure 7. Mapping features into packages.

The analysis of requirements for domain engineering and scoping the product line has a result in a features model. The features model has been structured in packages (see. Figure 6). The with-reuse aspect of reusability is described by the abstract features of the PLA. The abstract features encapsulated into three main abstract domains Measurement Controller, Data Management and Data Acquisition, are completely reused in all product members. The Abstract Spectrometer Features package has the highest degree of reusability but also the highest degree of dependability. The abstract features depend on the commonality between EGY and SEC features. A change in the problem domain of one of the three products is mostly reflected in the degree of reusability of the abstract domain features.

The sets of products that could be derived from the domain specific services during application engineering are:

• P1 – EGY Controller includes specific services of a standalone control of EGY measurement mode;
• P2 – SEC Controller includes specific services of a standalone control of SEC measurement mode;
• P3 – SEC with EGY Controller includes specific services of coordinated control.

Cross-domain architecture of spectrometer controllers:

The spectrometer controller cross domain architectural approach is described in Figure 7.

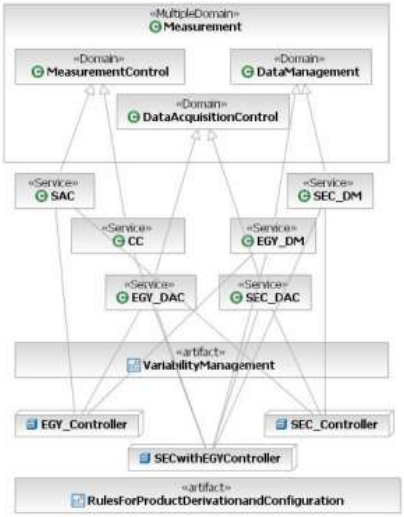

Figure 8. Spectrometer controller cross domain architectural approach.

Reference architecture. Looking top-down, the Reference Architecture encapsulated in the Measurement <<MultipleDomain>> is composed of three core abstract <<Domain>>s: Measurement Control, Data Acquisition Control and Data Management. In each core <<Domain>> abstract features are collected. The Measurement Control is responsible for services of starting and stopping the operating mode for data acquisition according to the commands received from the command interface and according to the events generated in other parts of the software.

Domain architecture: Domain architecture consists of domain specific services and variability management services. Each of the three core services is specialized in domain specific services. For example, Measurement Control is specialized in Stand Alone Control (SAC) and Coordinated Control (CC), Data Acquisition Control (DAC) is specialized in EGY_ Data Acquisition Control (EGY_DAC) and SEC_ Data Acquisition Control (SEC_DAC), Data Management (DM) is specialized in EGY_ Data Management (EGY_DM) and SEC_ Data Management (SEC_DM). Domain architecture includes services associated to variability management.

Product architecture: Product architecture for the sets of products includes rules for product derivation and configuration.

Table 1: Domain specific services and derived products.

Product		**P1**	**P2**	**P3**
Domains	**Specific Service**	x	x	
Measurement Control	SAC			
	CC			
Data Acquisition Control	EGY_DAC	x		x
	SEC_DAC		x	x
Data	EGY_DM	x		x
Management	SEC_DM		x	x

In Table 1 the set of products are horizontally distributed and the domain services are dispersed vertically. Each cell tij of the table is marked if product Pj uses component Ci . For example, two products, P1 and P2, includes a SAC service of the measurement control domain. The architecture have been documented around multiple views describing conceptual and concrete levels, for each view a static and dynamic perspective being offered. Architecture documentation addresses specific concerns

for measurement control, data acquisition control and data management. The views were illustrated with diagrams expressed in UML-RT, a real-time extension of UML. The conceptual level considered a functional decomposition of the architecture into domains. The relationships between architectural elements are based on pass control and pass data or uses. The concrete level has considered a more detailed functional description, where the main architectural elements are packages, capsules, ports, protocols.

Conclusions:

We have proposed an approach to design a cross domain reference architecture for embedded systems. The approach has been exemplified by a simple example. The problem dimension for the development of an embedded system reference architecture increases due to the larger number of requirements and constraints specified by a group of experts. Building the features model may require a tool in order to manage the analysis and structuring the abstract features in domains. The reference architecture contains core services of the domains included in abstract features package. The appropriate architectural style is provided by a knowledge base through a service taxonomy. A domain architecture repository is a solution for variability management of specific services. The decision support tool is proposed for product derivation. The role of this tool is to bound variability's in order to get a right configuration for product architecture. In our example, we used a tabular form for the decision model. When the problem complexity increases, a more elaborated tool is required and is a subject for our future research. However, our incremental design and analysis approach is based on a systematic service oriented approach, which is more practical, easier to follow and benefits of advantages provided by service engineering.

References:

1. Abdoulaye Gamatie, (2011), "A Model-Driven Design Framework for Massively Parallel Embedded Systems", Vol No: 10, Issue No: 4.

2. Bakshi, A., Prasanna, V. K., Ledeczi, A. (2001), "Milan: A model based integrated simulation framework for design of embedded systems", Vol No: 36, Issue No: 8, PP: 82–93.

3. Bastoul, C. (2004), "Code generation in the polyhedral model is easier than you think", In Proceedings of the 13th IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04), PP: 7–16.

4. Calvez, J.-P. (1993), "Embedded Real-Time Systems. A Specification and Design Methodology".

5. Domer, R., Gerstlauer, A., (2008), "System-on-Chip environment: A SpecC-based framework for heterogeneous MPSoC design", EURASIP J. Embed. Syst, PP: 1–13.

6. HA, S. (2007), "Model-Based programming environment of embedded software for MPSoC", In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'07). IEEE Computer Society, Los Alamitos, CA, PP: 330–335.

7. Karen Zita Haigh, (2015), "Machine Learning for Embedded Systems: A Case Study".

8. Salome Maro, (2017), "Addressing Traceability Challenges in the Development of Embedded Systems", ISSN 1652-876X.