A component framework of a distributed systems family

Eila Niemelä

VTT Electronics, P. O. Box 1100, FIN-90571 Oulu, Finland

Email: Eila.Niemela@vtt.fi

Abstract

A component framework has a dedicated and focussed software architecture, components and their interaction mechanisms. In this paper, we present a component framework of a product family that has three tiers: the subsystem, integration and product-family tier. Each tier points out a view of a component architecture using architectural styles and patterns. The development and the effects of a component framework are presented briefly and they attempt to give an overview of how a component framework could be developed and when it can be started.

1 Introduction

Distributed systems are developed in concurrent engineering processes, each of which is concentrating on its own aspects, e.g., mechanics, electronics, and software (Rossak et al. 1997). Due to the complexity of the software of distributed systems, the software is normally decomposed into smaller, less complex parts, which are allocated to different people or subcontractors. Therefore, there is a need for a systematic way that supports the definition and implementation of software components, which are interoperable but can be produced independently. Independence of a software component does not only assist in allocating resources but also assists to integrate a system through carefully designed interfaces and guidelines how to use them.

Software architecture is an abstract and overall design description of a system integrating different issues that are separate but have a contrary influence on each other (Szyperski 1997). Component-based software architecture is a structure of the system including software components, the externally visible properties of those components and relationships among them (Bass et al. 1998). Furthermore, a component framework is a skeleton of a system with a focussed architecture and an integrated set of components that can be reused and customised (Johnson 1997; Brugali et al. 1997; Szyperski 1997).

The aim of this paper is to present what a kind of a component framework can support the development and evolution of a distributed control systems family. It also outlines what has to be done in order to developing a component framework successfully. The motivation of the work is that component-based software is essential for the development of distributed systems. The importance of component frameworks is also growing, because software products and components should be developed more cheaply and quickly to reduce time-to-market, as well as more easily customised according to the needs of diverse customers and end-users.

2 Characteristics of distributed systems

For achieving the understanding of the characteristics of distributed systems, we studied the problems that appear in the practical component-based software development of distributed control systems. Although the studied domain areas, machine, process and manufacturing control systems, had significant differences, we discovered the following similarities:

- The problem domains were quite stable, and therefore, the component framework could be used at the product-family level.
- Each systems family had a multitude of technical features and several variations that had to be defined and implemented by the component framework.
- Heterogeneous design and implementation techniques were used in all systems families.
- Application developers might be subcontractors that had to use the same component framework as the systems integrators.
- The organisational culture favoured the use of third-party components.
- The component framework had to support the roles and responsibilities of different stakeholders.
- The component framework could have to be used in the concurrent software development.
- The size of product variants varied, and therefore, extendibility and scalability were required.

Briefly, the component framework has to support simultaneously the application developers, the system integrators and the suppliers of distributed control systems.

3 Evolution of a product family

In order to resolve the problems caused by the evolution of a product family, we applied the adaptive design by focusing on the changes of the control systems families. Changes in the markets, used technology and the application domain are the main reasons for the evolution of the component framework. Therefore, the component framework has the product family, integration and subsystem tiers that have their own scope and techniques for achieving adaptability (Figure 1).

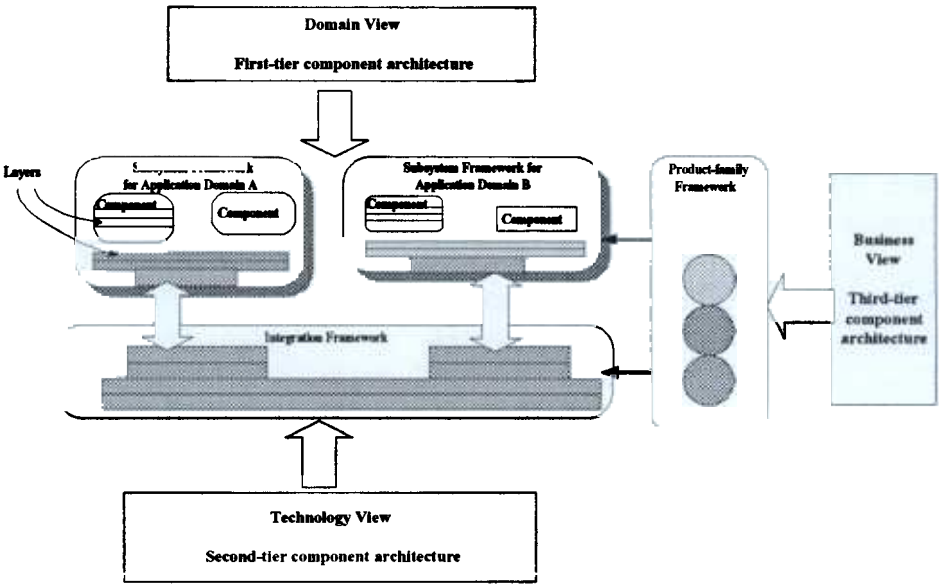

Figure 1. Tiers of the component architecture.

At first, the product-family tier clusters the product features to the semantic components that are implemented by the properties of the architectural components of the subsystem tier and the services of the integration tier and their interaction mechanisms. Fine-grained features are mapped to the properties of the building-blocks, primary components and their configuration parameters. The subsystems use standard interfaces with strictly defined policies, demanded by the integration tier that hides the changes in the used technology providing an application-specific layer for the subsystems integration. The architecture of the product-family tier balances the architectures of the subsystem and integration tier providing a systematic way to describe, manage and change product features. The ability to configure applications dynamically is the support software in the product-family tier, implemented as a part of the integration platfrom.

The support for systems' evolution can be scaled in each tier. The reconfiguration support for applications, integration platform and product features can be implemented in a way that is suitable for the needs and used technology. The larger the systems family is, the more comprehensive support is needed for product variations.

4 Development of a component framework

The development of the component framework embodies the following reuse assets: product features, product-family architecture, components, and mechanisms and policies for the use of components. Domain engineering produces the features model, scenarios and time-threads that give the overall understanding of the structural and behavioural properties of the systems and their execution constraints. We propose the QFD technique (Day 1993) for qualifying product features and COTS components. The used modelling and qualifying techniques are technology-independent due to heterogeneous design methods and tools used in the development of control systems.

The product-family architecture combines the layered, presentation-abstraction-control and client-server architectural styles. The layered architecture is used for:

- providing the portability of the framework,
- isolating the frequently changing parts from the more stable parts, and
- separating technology-dependent parts from application-dependent parts.

Layers are used in the subsystem and integration tiers. PAC agents have been applied for applications with the ability of the run-time configuration. The client-server architecture is obeyed between the applications and the integration tier that provides transparent communication, co-ordination, allocation and configuration services for the systems. The timing requirements define which communication service is selected and how the integration tier has to be allocated.

The amount of required adaptability depends on the used COTS, heterogeneity of communication media, operating systems, and environmental devices. Design patterns and standard interfaces for communication and configuration have been applied to achieve adaptability. An adapter, wrapper and filter are used for the adaptation of legacy software to the integration tier.

Due to the incremental integration and long installation phase, control systems need the configuration mechanisms that are integrated parts of the integration tier and are used for customising application and product features. In small systems families, the configuration support is a simple user-interface with the ability to change product features by data-manipulation. Reconfiguration support for fine-grained features is only needed in the complex systems that have a long installation and introduction phase.

5 Impacts to the development process

Our approach to the development of the component framework of a product family sets some preconditions and restrictions, but it also gives freedom to select the methods and tools applied within the tiers that are often used by different organisations. However, the systematic way to develop and manage reuse assets set the following preconditions:

- The definition and management of product features require a mature product family and development process.
- The whole organisation has to be committed to the systematic way to produce and use the reuse assets.
- The suppliers of COTS have the responsibility to describe all features of their components.

The purchasing of COTS has to be guided by required features including reuse requirements.

Stakeholders share the product-family architecture that has to be kept as stable as possible within each tier.

- Developers have to be familiar with the architectural styles and patterns.

The interfaces and policies of components have to be defined thoroughly.

The applications have to meet the specification of the interface layer.

The configuration of a system is a top-down activity that can be automated.

The above-mentioned conditions require that the stakeholders of the product-family -- marketing staff, application developers, integrators, and maintenance staff -- work together more closely than they do nowadays. Owing to COTS and distributed application development the component framework needs an organisational infrastructure that provides the services needed for sharing the knowledge of reuse assets.

6 Conclusion and future work

In order to get an understanding of the properties of a component framework, applied in a product family, we presented the characteristics of the distributed control systems domain and the need of adaptability at the product-family level. We also illustrated the used techniques: technology-independent methods, architecture styles and design patterns, by which the required adaptability in each tier can be achieved. However, the approach sets significant preconditions and restrictions that have to be committed by the whole organisation before starting the development of the component framework.

We have developed the integration platforms for machine and small process control systems and used the commercial ORB as a basic component for the manufacturing systems family. Our continuous work focuses on a generic integration tier that could be used for most embedded systems. This provides that the features of the services and components in the integration tier have to be defined and managed in a way that the tier could be configured for the needs of the target systems. Quality services will be new properties that are especially needed for Internet applications.

The other issue that needs further research s the enhancement of the feature modelling method. A formalised feature modelling method with variation support is the precondition that the semantic components can be defined, and thereafter, the configuration rules can be produced automatically. This presumes that the generative approach of the features modelling is integrated with the services of the integration tier and the components of the subsystem tier. The integration of the generative and component-based software development is more important for the continuously evolving domains, such as telecommunication systems and consumer electronic

products, than for the quite 'fixed' control systems families. They also need special tools for modelling and managing features. The existing design tools are appropriate for small systems, in which the reuse assets need not to be shared. The developers of large systems, the kind that most networked control and embedded systems will be in the future, need the supporting infrastructure with the ability to share and manage reuse assets over organisational boundaries.

The ROOM method that was used in the case study proved to have appropriate support for architecture modelling. However, its support for defining variation points is limited. Therefore, there is a need for further studies to develop an integrated development environment with the following properties:

The product-family tier should have to be supported by the tools to define and manage product features, create semantic components, and generate and validate the configuration rules of the target systems.

The subsystem and integration tiers need a tool for clustering component variants according to the defined architecture styles, mechanisms, and policies.

The evaluation of the architectural alternatives must be able to be validated as regards functional and quality requirements.

Reuse assets have their own quality requirements, such as openness, maintainability, and portability, which have also to be validated at the architecture level.

The subsystem tier that used the PAC architecture in the control domain needs alternative architectures for different problem domains. In order to get a better understanding of the suitability of the component framework, it needs to be applied for several domains. The application areas that have mature product families and development processes are promising domains for component frameworks.

References

Bass, L., Clements, P., Kazman, R. 1998. Software Architecture in Practice. SEI series in software engineering. Reading, Massachusetts: Addison-Wesley, 452 p. ISBN 0-201-19930-0.

Brugali, D., Menga, G., Aarsten, A. 1997. The Framework Life Span. Communications of the ACM, Vol. 40, No. 10, pp. 65-68.

Day, R. 1993. Quality Function Deployment. Linking a Company with Its Customers. Milwaukee, Wisconsin: ASQC Quality Press, 245 p. ISBN 0-8739-202.

Johnson, R. E. 1997. Frameworks =(Components+Patterns). Communications of the ACM Vol. 40, No. 10, pp. 39-42.

Rossak, W., Kirova, V., Jololian, L., Lawson, H., Zemel, T. 1997. A Generic Model for Software Architectures, IEEE Software, July/August 1997, pp. 84-92.

Szyperski, C. 1997. Component Software. Beyond Object-Oriented Programming. New York: Addison Wesley Longman Ltd, 411 p. ISBN 0-201-17888-5.