



Information Technology for European Advancement

Task 1.4 - Guidelines for Component-based Development (D.1.4.3)

Version 02 - Public

Edited by Stefan Van Baelen

Software Development Process for Real-Time Embedded Software Systems (DESS)

ITEA COMPETENCES involved:

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and services**

April 2001

Table of Contents

Table of Contents	1
Purpose of this document	2
WP1.4 partners	2
Document history	2
1. Introduction	3
2. General component properties	3
Decomposability	3
Composability	3
Understandability	3
Continuity	4
Protection	4
3. Identification of components	4
Device interface control layer (DICL)	5
Application layer (AL)	5
Aggregation and application support layer (AASL)	5
4. Capturing properties of a component	6
5. Development of components	7
6. Composition and decomposition of components	7
<i>Retrieval of components</i>	8
<i>Design exploration and validation using components</i>	8
References	8

Purpose of this document

The purpose of this document is to describe a number of guidelines for component identification, component-based development (CBD), component composition and component decomposition. These guidelines are aimed at a component-based development process for real-time embedded software systems.

This document is a first deliverable of CBD guidelines within the DESS project. During the second stage of the DESS project, further research will refine the guidelines described in this document. The refined CBD guidelines will be described in deliverable D.1.4.3, that will become available at the end of the DESS project.

WP1.4 partners

The WP1.4 partners are listed below in alphabetical order:

1. Barco Display Systems (B)
2. Bull Italia (I)
3. DaimlerChrysler Research Information Technology (G)
4. France Télécom Cnet (F)
5. GMD-FIRST (G)
6. *K.U.Leuven (B) [Task Leader]*
7. Magdeburg University (G)
8. Paderborn University C-LAB (G)
9. Siemens C-LAB (G)
10. Thomson CSF (F)
11. Thomson Multimedia (F)
12. Unis (CZ)

Document history

March 2000	1.4.1 V00 Draft A	-- NEW DOCUMENT --
June 2000	1.4.1 V00 Draft B	Removed type errors
		inserted reference to COSIMA
October 2000	1.4.1 V01	included purpose and partners
April 2001	1.4.3 V01	included section on self-containment

1. Introduction

When generally looking at the component development process, three essentially different kinds of goals of that process can be distinguished.

- Firstly, the developed components can be sold by the producer as (more or less standardised) pieces of software which can be deployed in many different system environments determined by the purchaser after the component has been developed (COTS, commercial off-the-shelf components).
- Secondly, the supplier can develop the components according to a precise requirement specification provided by the purchaser before the development starts.
- Thirdly, components can be developed for “internal” use only just to make the own system more flexible.

Because most partners are not software component suppliers, only the third case will be considered in this document. Nevertheless, the second case (development of components according to a given specification) is also of great interest for the partners, because they may become big purchasers of components exactly tailored to their needs.

2. General component properties

To develop good components, modularity should be used as the basis for system construction. Criteria to assess the potential for modularity of a system are:

Decomposability

A component-oriented development approach must help the software architect to define a system that is made of simpler subsystems with a simple structure between them. The identified subsystems (components/objects/ modules/...) might be independent enough to allow the same decomposability property on each of them.

Composability

The component must be made so that they reinforce the reusability in different contexts. Each component might be freely combined with others just relying on their interface.

Understandability

The component structure of the system must favour the understandability of each of the created component. To understand each component, one might only need to look at the component itself and not at the system into which it is integrated.

Continuity

Continuity means that a local perturbation has only a local consequence. The continuity is a major concern of software maintenance because it is supposed to allow mastering the cost of change in software production. We believe this principle of continuity must be a very important goal when developing new components.

Protection

To ensure the safety of the whole system in to which components are gathered, each component must ensure that a failure or any other abnormal state is not propagated throughout the whole system.

Self-containment

The component concept in itself has a number of consequences. In order for a component to be reusable without limitations, it must be independent from any application. This means that every component must be treated as a small, self-contained project in its own right, with its own life-cycle. To be able to accomplish this, a component must have its own:

- Requirement set
- Documentation set
- Test cases
- History (configuration management), which uniquely identifies the combination of the component's software and the aspects mentioned above

Only this way, it will be possible to clearly track the life cycle of a component. By keeping an individual configuration history per component, an application can "decide" whether or not to upgrade to a new version of a component, when it becomes available.

Maybe the point distinguishing component versus object approach is that when we talk about a component we focus on domain technology with business goals. We usually apply the term *component* to mention a somehow autonomous piece of code with services that are of strategic importance for our business.

3. Identification of components

In order to describe the identification of component, a general component-oriented software architecture is used, that can be based on the COSIMA architecture of DaimlerChrysler [3]. Also the component-based approach of BARCO uses the same layered model. The architecture consists of three non-overlapping layers:

- The lowest layer is established by components encapsulating hardware devices and hardware functionality. This layer is called the Device Interface Control Layer (DICL). It can be seen as a device initialisation and low level protocol layer.
- The middle layer is built up of managers. These components use the functionality provided by the device interface control layer to provide specific high-level sup-

ITEA

port for applications. This layer is called the Aggregation & Application Support Layer (AASL). It can be seen as a high level protocol driver layer.

- The upper layer is established by application components. Each application component usually implements a specific service. This layer is called the Application Layer (AL). This layer can be seen as an application dependent part.

As such, each component belongs to exactly one of the three defined layers. The procedure of component identification depends on the layer the component should belong to.

A component-based development approach must be applicable to all layers in the architecture of such a system, i.e. basic run-time environment and operating system layer, object and component middleware layer, high level behavioural layers, etcetera.

Device interface control layer (DACL)

The identification of components belonging to the device interface control layer is quite simple. Each independent hardware device can be represented by exactly one component. So, when the decision to integrate a new hardware device with a new hardware interface into the system is made, a new component must be developed which provides to the rest of the system a complete compatible software interface to the new device. Thus, the problem of identification of new DAACL components can be reduced to the problem of identification of new hardware interfaces. Whether or not a particular device should be integrated into the system depends primarily on two questions:

- Does this device support desirable applications (cf. application layer) which could not be accomplished without this device or, at least, does this device significantly improve already existing applications?
- Is this device available (now or in the next future) at “reasonable” cost?

Application layer (AL)

The upper layer is established by application components. Each application component usually implements a certain service. The identification of potentially necessary application components is highly driven by currently available technological means (what is feasible) and customer’s desires (what could be interesting for the customer). Thus, there is currently a tight connection between the identification of DAACL components and the identification of AL components. When a system is changed in one of these two layers it should always be made sure that each hardware device is really needed by some application and that any application gets adequate hardware support.

Aggregation and application support layer (AASL)

These components use the functionality provided by the device interface control layer to provide specific high-level support for applications. The entirety of functional

interfaces provided by components belonging to this layer can be viewed as a kind of abstract operating system. Actually, the AASL layer is the heart of each system.

To identify the components of this layer is certainly the most challenging part of component identification. On the one hand, the functions provided by these components should really be more abstract and easier to use than the functions provided by the components of the underlying DICL. Ideally, an AASL component gathers device-dependent information from different hardware devices of the DICL, pre-processes it and offers it in a hardware-independent abstract shape to the AL components. On the other hand, the abstract functions offered to the AL must be complete enough such that no application needs direct access to the DICL. It is a good sign if such a higher-level function is used by more than one application.

This independence from the hardware can be a strength and a weakness. On the one hand, maximal flexibility is provided. The hardware and the components of the aggregation and application support layer can be exchanged independently. On the other hand, every abstraction means loss of information. The special features of the respective hardware cannot be used, the implementation cannot be as effective as it could be if the hardware would be accessed directly.

Thus, the process of identification of AASL components can roughly be divided into three steps:

- Identify the abstract functions needed by the current and the future (as far as possible) applications.
- Check whether the DICL components provide sufficient functionality to realise these functions.
- Divide the identified set of abstract functions into groups according to their semantic meaning. Each group of functions represents an AASL component which can be implemented using DICL functions.

4. Capturing properties of a component

Electronic and mechanical components are normally equipped with an expressive data sheet that describes all the necessary information for the use of that component. The data sheet usually contains different views on that component, e.g. physical dimensions, power consumption, operation modes, etc. and uses various description means, e.g. textual explanations, diagrams etc.

For software components similar approaches should be followed. Key to the success is a proper representation of meta-information about components. During design, important data should be captured and transformed in a description format, either as separate documents or by a reflection mechanism of a component itself. Emerging standards, especially in the context of XML, could then be used to represent such data.

5. Development of components

After the components and their functionality have been identified, a design specification is developed, e.g. using UML and an appropriate UML tool. Because a component is a unit of independent deployment, a specification of a particular component can be carried out rather independently from other components.

For the design specification, primarily interaction diagrams and class diagrams can be used. By interaction diagrams the most important scenarios are described helping to identify the necessary objects with their functions. The class structure of the component as well as the signatures of the classes are depicted by several class diagrams.

The implementation phase is loosely coupled with the design specification. The implementation of each particular component consists of two parts, the *worker* implementing the functional part of the component and the *control* implementing the extra-functional and non-functional part. The internal interface between these two parts must be implemented by both sides. The worker must completely be developed from scratch; substantial parts of the control can be supplied by a framework. Thus, the design specification described in the previous paragraph concentrates on the worker part of the component. As described so far the development process is language-independent.

Obviously the probability of reuse is proportional to the generality of a component, i.e. it is extremely improbable that a highly specialised component may ever be reused. On the other hand very general components tend to be highly inefficient. A solution to this problem is customisation, i.e. not specific components are designed and stored in a component library but more or less customisable ones, customisable via a set of parameters. This implies a move from implementation level to a specification level. Then it is possible to customise components at the specification level, independent of the implementation and to make late decisions concerning the way of realisation. Obviously specification level components make sense only with a sophisticated design methodology available. One approach to this has been developed at the University of Paderborn and is called PARADISE, which stands for design environment for PARAllel and DIStributed Embedded Real-Time Systems [2]. This environment is currently being extended to cover aspects of specification-oriented component design.

6. Composition and decomposition of components

The decomposition of a system specification into components and the composition of a system from components is driven by the layered component architecture.

Retrieval of components

If a component is intended to be used more than once and has been designed to be deployed in various environments by its supplier, then this component has to be marketed properly. For electronic and mechanical components in the classical engineering disciplines, there is a long tradition to organise these components in product catalogues, that for a long time have only been published and retrieved using paper catalogues, later on CD-ROMs and nowadays by means of internet based services in connection with database storage and sophisticated retrieval techniques.

A interesting solution to a world-wide market place for components is given by the Anteros approach [1]. Anteros services allow precise and efficient searches according to standard engineering classifications. Information and services can remain in legacy systems and user profiles ease the search and allow the configuration of the result presentation.

Design exploration and validation using components

As software components are intended to be building blocks in a system design there is a need for a design methodology that supports the synthesis of a system out of single parts. If a component has been identified for a potential use in a system, e.g. by one of the above mentioned retrieval mechanisms, then it must be validated whether it really fits into the overall system design, i.e. interfaces have to match, communication demands and processing power have to be sufficient, required services and hardware modules have to be available etc. Furthermore, any design freedom has to be resolved, e.g. if the component can be allocated on different hardware nodes or if the component itself can be configured towards optimal use on a given hardware. Obviously, there are still a number of open questions to be solved before this process is really understood and supported by appropriate tools. But it is likely that methodologies in disciplines where use of components is well established will at least help to solve those problems in the case of software building blocks.

References

- [1] E. Radeke. GENIAL -- Turning Engineering Knowledge into an Accessible Corporate Asset. In Proc. of the 2nd Symposium on Global Engineering Networking (GEN'97), Antwerp, Apr. 1997.
- [2] W. Hardt, A. Rettberg and B. Kleinjohann. The PARADISE design environment. In Proc. of 1st New Zealand Embedded System Conference, Auckland, New Zealand, July, 1999.
- [3] M. Stümpfle et al., COSIMA - A Component System Information and Management Architecture, IEEE Intelligent Vehicles Symposium, Dearborn, USA, 2000.