STWW-Programma

SEESCOA:
Software Engineering for Embedded Systems using a Component-Oriented Approach

Working Definition of Components

Deliverable D1.4

Contents

Introduction

The very nature of real-time embedded applications makes certain characteristics of their implementation (such as timing and implementation architecture) critical. Usually, the software in these applications is responsible for the control of other equipment; so designing a correct solution requires a good expertise to glue non-standardized components together.

Most real-time embedded systems are, by nature, multitasking solutions to real-world problems. They typically deal with the interface and control of multiple external devices. The different parts of these systems usually run at different priorities and with different run-time characteristics. The notion of multiple tasks or threads being active in the system at the same time is common. Many of these real-time systems are deployed on a set of microprocessors in a distributed architecture. Designing a solution for this type of problem requires a new adapted view to components.

This document describes the definition of a component and component system, as it will be used in the SEESCOA project.[1] The definition is split in three parts:

- The first part describes what a component is and gives a good idea how to think about components.
- The second part describes the requirements and the responsibilities of a component system. The component system is the context in which a component has to operate.
- The third part looks at some additional characteristics and issues concerning components and the component system.

[1] Currently, some definitions are missing. For example, how synchronization should be described and other 'formal' issues are left out because this is partly the scope of the research proposal. Note also that it could be needed to adapt and change the SEESCOA component definition based on the experience we will gain during the project.

Components

A component is a *reusable documented* entity that is used as a building block for software systems. It is used to perform a *particular function* in a specific *application environment* within a specific *component system*. Components are composed (glued together) using their *interfaces*. These interfaces consist of *provided interfaces* and *required interfaces*. A provided interface describes how the functionality has to be accessed. A required interface describes what is needed to perform this functionality.

In this definition a distinction has to be made between a component blueprint and a component instance. A *component blueprint* is a description of a reusable software element; a *component instance* is an instantiation of this description. A component blueprint does not have a state, a component instance does. It also doesn't make sense to talk about the runtime properties of a component blueprint; only component instances have runtime properties. This distinction is important for a clear definition. The term component is more general; by using it we mean both aspects.

Component instances are not objects. And in consequence, component blueprints are not classes. First of all, components cannot inherit from other components. Objects do inherit form each other (they inherit the implementation). Component blueprints are also extensively documented, classes are not. Note that in some cases classes are documented with diagrams, semantics, call protocols, and so on, but this information is not explicitly described in the *object definition*. This information is often added in an informal way. When a component is implemented, it will probably use different objects to perform its functionality (of course in the case an OO language is chosen). Therefore some books talk about components as if they were big objects. This is true in some extent, but limiting the component definition to this would be wrong. A component instance should be thought of as having its own *code & data space* and also its own *thread of control*. This is necessary to have the ability to use different synchronization principles and make components reusable. A component worked out, thinking it has its own control flow will be more general, than a component which enforces certain calling strategies upon other components.[2] Or, a component written in the assumption its memory will be accessed by other components is more specific than a component which doesn't share its data via these kinds of techniques.

A component (the blueprint as well as the instance) is always used in a certain application environment and in addition it also offers an application environment to its users. So two views are distinguished:

Environment of a Component: Outside View

- **Provided environment**

[2] For example, database applications written as if they are standalone programs will receive a message and send an answer back to the caller some time later while the originating component can do other things. Written in the other way, the database component will block the sender which is not reusable without writing adapters and other middleware.

This is the application environment that the component offers. It can be described by means of a text, a figure, … For the component composer[3], this is important information because it determines if a component is usable in a given situation. A component that offers a 'sales environment' (for example: a component that calculates Value Added Taxes) will not be used in a situation where one needs a component to decode a video stream. Thus, every component offers a certain environment, which is usable or not for the component composer.

- **Required environment**

A component can also require to be deployed in a certain application environment. The VAT component (see above) could for example need a `database environment`, if it needs to make some data persistent. A component could also need specific hardware, so this hardware makes part of the needed environment of the component.

Unique Identification

Every component blueprint has to be named uniquely. This simplifies its reference (if, for example, the component blueprint is stored in a catalog). It should also be possible to have multiple versions of a component blueprint. Therefore the identification of a component blueprint consists of an **identification name** and a **version number**.[4] When component instances are instantiated from the same component blueprint, they also have to be distinguishable by means of a unique name[5]. This name should be used to get a reference to the component instance (for example, if one wants to send a message to the component). This of course requires a naming and referencing mechanism.

Naming and referencing of component instances are not the same. Often, the name of a component instance is a human-readable string. Knowing the name is not enough to send messages to the component: a reference has to be obtained. This reference is then used by the component system to route messages to the correct component. Examples of references: a pointer in C++, a Java reference or a Java RMI reference. A Java reference is a `smart` reference since it knows how many times an object is referred.

Boundary View

The boundary view describes which interfaces a component offers and eventually which interfaces it needs from other components. The difference with the outside view is the following: the boundary view specifies formally how to use a component. The outside view specifies only informally when to use that component. Or put differently: the outside view specifies the usability of a component, where the boundary view specifies how to use it.

Three views are distinguished: [6]

[3] The component composer is the person or tool that uses components and connects them to build applications.

[4] When using the term 'version' here, the 'implementation version' of the component is meant, not the version of the component's interface (like COM).

[5] Note that the unique identification of a component blueprint and a component instance are not exactly the same.

- **Provided Interface**
  This interface represents the services a component supports. The provided interface shows which operations can be performed on the component and which events the component can generate.

- **Required Interface**
  Via this interface the component describes which interfaces it needs to perform its services. Note that a component does not require a specific implementation of an interface: it only requires the interface. It is possible that a component does have an empty requires interface; this means that the component is self-contained.

- **Management Interface**
  Via this interface the component user is able to tune and to manage a component. This is useful for testing and debugging purposes. This interface is also used for creating, starting, cloning, stopping and destroying components.

An interface can be specified at four different levels. Only the first level is supported in most component systems today. Other levels are often specified informally, without any support of the component system.

1. Syntactical Level

On this level the supported services are specified in a syntactical manner.
For each operation of the interface it consists of the name of the operation, the parameters, the return value and the exceptions that can occur.

2. Semantic Level

A more detailed step is specifying the behavior of a component. The user knows what he has to do, but he also knows what he can expect. The specification of behavior is often done informally, but it can also be done in a formal way with pre-conditions and post-conditions. A formal description has the advantage that behavior can be checked automatically.

3. Synchronization Level

This level specifies the protocol that has to be followed when interacting with the component. This protocol can for example determine in which sequence operation calls should be made, if certain operations have to be performed in an atomic way, and so on. The specification should be done in a formal manner either by drawing a dynamic diagram (such as MSC) or drawing the control flow between components or other ways of specifying the synchronization behavior between components.

4. Quality of Service Level

[6] A component is of course free to offer more than one interface and it is free to require more than one interface. For example a component can require a certain interface from a database component while it requires another interface from a scheduling component.

This last level specifies non-functional properties of the component. This is of course strongly dependent on the context of the component. Non-functional aspects are for example the memory usage, the time complexity, the worst-case time complexity, … Other aspects have to do with the loss of precision, the loss of messages, and so on. QoS specification can also be formal or informal. One could for example supply estimation information, measurements, …
This level is particularly important for embedded systems, since it enables us to specify the resource needs of a component. This specification will only be an approximation, but it can help in the development of the application. Also, when specified formally, the component system can check automatically at runtime if the constraints are met.

Component System

The component system is the infrastructure (framework, architecture or kind of operating system), which makes component instances work together, which glues them and creates a homogenous environment for them. The component system can be seen as the middleware which connects different components and which makes them work together. To put it differently: the component system provides the streets while the components are the cars driving on it.

The component system

- Makes components to work. The component system can **create** and **destroy** component instances and is able to **start** and **stop** component instances.
- The component system can have **support for introspection**. When working with components we need the ability to find, name and rename components. These abilities should be provided by the component system. Furthermore, sometimes it is necessary that a client can query a component about its services. Mostly the client is bound to the component's interface at client construction time (e.g. when the client is compiled). When introspection is possible, the client is not bound at client construction time, but it can dynamically (at runtime) find the services of a component. This can be compared to the reflection mechanism in Java.
- **Abstracts the hardware** and **the operating system** such that all components can run in the same environment. For example, if we work in an embedded system with segmented memory or in a system with five flavors of memory access, the component system should tackle this and offer a more or less flat interface to it. This abstraction should be as lightweight and as performant as possible in embedded systems. The component system should be mapped upon the operating system and language as closely as possible. It is not said that all hardware dependent issues should or could be put into the component system. All general hardware aspects that have impact on all of the code (like the memory access example) should be put into the component system. Modular hardware access, such as devices, can be put into separate components.
- **Handles message passing** between components: If a component wants to make another component to do something, or whenever the state of another component has to be changed, a message is send to the component in question. Components can send messages to other components using a reference (which can be obtained by using the unique name of the component). The component system takes care of sending data (over a network for example), calling the right function on components and eventually other ways of passing messages between components. This includes changing the data format if necessary, as is done in CORBA. Nevertheless, the component system is not necessarily a distributed environment.

- The component system handles the **scheduling** between components. Because components are thought of as active entities it is necessary to map this view to a real operating environment. This is done by the component system, which ensures priorities of messages between components, which takes care of (hard) real time constraints and scheduling in general.
- The component system has some standard **glue components** to adapt interfaces between different components. For example, a certain component can return a callback with a specific name, whilst the receiver expects the message with another name. This can be done by certain glue components.

The component system can help in **debugging** by checking whether interfaces are used in the right way. The component system understands the synchronization interfaces provided by the components and can automatically check whether the right calling sequence is used. Another possibility is logging all sent messages.

Additional Possibilities

Optional Features

There are some additional properties which can be added to the component definition but which are not required.

Inside View

Every component has an inside view. But that does not mean that this inside view is shown to the outside world (= the component composer). The inside view of a component comprises all documents that explain the implementation or the design of the component. This ranges from analysis documents to the implementation code of the component. The inside view of a component can give more insight in the workings and the use of this component. But showing the internal workings of a component also has its disadvantages: one does not only use the interfaces of the component but also extra knowledge, this can have an impact on the replaceability of the component.

Verification and Validation Information

To *trust* the correct working of a component it is important to provide testing and validation information. This information can determine if a component can be used or not in a certain product. As information one could for example provide test cases, together with the results that were achieved when performing those test cases. Formal proofs of correctness could also be used as a validation of the component.

Source Code/Binary Form

A component can be delivered with its source code. Using the component thus also means compiling that component. When a component has its source code associated with it, it can be more easily adapted and understood. Components that are used within a company should ideally be provided with their source code. Components bought from third parties will not always be delivered with their source code.

A component can be deployed in a binary form. That means that the component composer cannot see the internals of the component (its source code). The reasons for this can be multiple: the source code is not available, is not sold, … Delivering a component in binary form also means that the component can only be used on platforms that do ‘understand’ this binary form. In the embedded systems world this is rather rare, knowing that there exists no single standard platform.

Implementation Issues

This section highlights some possibilities in implementing or constructing a component system. Furthermore, it also contains some implementation characteristics of components. Some issues are raised with their advantages and disadvantages.

Dynamic binding

In some component systems components can be removed, added or replaced at runtime. This feature can help to maintain a running system or to adapt a running system.

Active/Passive

It is possible to create passive components by only letting them respond to messages. An active component is a component that initiates communication with other components. A passive component can be looked at as a component without thread; an active component does have a thread. Whether all components are active or not is a question that is highly dependent upon the component system used. The component system determines the kind of scheduling and threading.

Single-Threading/Multi-Threading/Processes

A component can be single-threaded or multi-threaded. A single-threaded component has one thread of control, while a multi-threaded component has more than one thread of control. Even more, a component can be a complete process on its own.

State/Stateless

A component with state is a component for which the actions taken are dependent on the time the messages were send to it. A stateless component always acts the same to the same message. Passive components can be stateless but are not bound to be stateless, just as stateless components are not necessarily passive.

Persistent state

Sometimes it is necessary to store the state of a component when the component has to remain persistent. An example of such a component is an Address Book component: when the system shuts down, the component has to be made persistent otherwise the content of the Address Book is lost. Although, it is not necessary to make all components persistent, this feature is optional.

Migration/Remote execution

Sometimes it is necessary to execute a component on a remote system. If this is necessary the component system should take care of this and do the communication where needed. If we want to go one step further we may need the ability to migrate components from one component system to another whilst the code is executing. This feature can be used to balance the load in a system.

Cloning/Classing

Whether components are instantiated from a component blueprint (source and code on disk/memory) or components are instantiated from another component (cloning) is a matter of choice. Both ways have their pros and cons.

Object Oriented/Imperative/Logic/Functional Components Language

It does not matter whether components are written in an object oriented, imperative, functional, or logic fashion. This is a choice left to the implementer. The one and only thing that is fixed is the interfacing with the component system. This also means that the language a component is written in doesn't matter. As long as the interfacing with the component system is as expected nothing can go wrong. Of course, some languages are more suitable to be used in a component system than others. For example, Java is much more suitable than assembler with a macro language on top of it.

Glue Components

The glue components provided by the component system should be generic well-designed components with as little as possible overhead towards the global system and which can eventually be removed when compiling.

Interfacing towards the component system

The interfacing towards the component system should be as simple as possible. Sending a message to the component system will call the component system. Receiving a call from the component system is a bit trickier. Sometimes we may need our own dispatching function to dispatch an incoming call to the effective method, while in other languages suitable language constructs exist to reach the specific method. How messages are passed between components is another question: we can for example use serialization and deserialization to transfer object graphs, or we can simply use a string to send messages between components. Which way is used depends upon the component system.

Introspection/Reification & Absorption

Introspection is needed whenever we are faced with dynamic components that want to change their name at runtime, dynamic components that look for their communication partner in the system and components which change their behavior at runtime. The possibility for remote/runtime uploads depends heavily on this feature. However it may not be necessary to create a full reflective system in which we can absorb certain kinds of primitives and workings into the component system.

The component system as a component

It can be an advantage to do the interfacing to the component system as if it were a component itself. This has the advantage that whenever we

work in a distributed environment the component system can always be reached with the same identifier. Another advantage is the possibility of tuning the component system with a larger, more manageable component system-component, which will be removed in the production version of the system.

Summary

All presented properties will now be summarized in a has/gives table. The 'has' column of the table specifies if the component has a specific characteristic or not. The 'gives' column specifies if this characteristic should be made visible to outside world.[7]

	Characteristics	Has	Gives
	Component Characteristics		
1	Unique Identification	Mandatory	
2	Context		
	- Provided View	Mandatory	
	- Required View	Mandatory	
3	Boundary View		
	- Provided Interface	Mandatory	
	- Required Interface	Mandatory	
	- Management Interface	Mandatory	
4	Boundary View Levels		
	- Syntactic Level	Mandatory	Mandatory
	- Semantic Level	Mandatory	Mandatory
	- Synchronization Level	Mandatory	Mandatory
	- Quality of Service Level	Optional	Optional
5	Inside View	Mandatory	Optional
6	Verification and Validation Information	Mandatory	Optional
7	Binary Form	Mandatory	Optional
8	Source Code	Mandatory	Optional
	Implementation Characteristics		
1	Dynamic Binding	Optional	
2	Persistent State	Optional	
3	Migration/Remote Execution	Optional	
4	Supports Cloning	Optional	
6	Language		
	- Functional	Optional	
	- Imperative	Optional	
	- Logic	Optional	
	- Object Oriented	Optional	
7	Persistent	Optional	
8	Single-Threaded	Optional	
9	Multi-Threaded	Optional	
10	Contains State	Optional	
11	Is Active	Optional	

[7] The table also contains the required properties of components

Discussion

Components and embedded software development

In this section we will look at the opportunities offered by the SEESCOA component definition for embedded software development. The discussion is split in two parts: the first part looks at the implications for software development in general. The second part focuses on embedded software.

Opportunities for software development

A problem that often comes back, is the lack of component reuse over several projects or development teams in a company. To make reuse possible, components have firstly to be defined in a formal way to eliminate misinterpretations. Without a clear definition, we cannot talk about reuse – because we don't even know what is reused. That's why the definition is quite formal on some points. Some parts of the definition have still to be filled in, for example the languages that will be chosen to specify semantics and synchronization are still an open issue.

It is also important to notice that a clear definition does not imply the correct use of components. To use and reuse components a method is needed. This method should enable the discovery of reusable components. A general guiding rule is the high cohesion – low coupling rule. High cohesion means that when one develops a component he should only put functionality in it that is related. A component that does everything is not reusable. Low coupling is also needed between the different components. If low coupling is not maintained, the involved component cannot be reused without also deploying the other components to which it is coupled. This also breaks reuse. To summarize, not only a good definition is needed, but also a good method.

So, what are the consequences of the component definition for the software development?

First of all, in our definition, a component has to be named and versioned. The naming and versioning enables the unique referencing of components. The importance of this naming and versioning is quite present for companies, since several people will develop components and others will need to use them. It is clear that this should be done in a uniform way. The naming and versioning also enables the storage of a component in a catalog; this catalog can then be browsed when looking for a specific component.

Our definition also stresses the importance of specifying the usability of a component using an outside view. When this usability is described, a component composer will know if a component is relevant to him or not. What is meant here is that a component always plays a role in a system, and to fulfill its role, it probably needs other components. A component composer needs to know this information, otherwise reuse won't be possible.

When a component is finally selected, it is important to eliminate all misinterpretations that could occur when using that component. That's why much attention is put on the boundary view of a component. Only showing component operations is not enough to (re)use a component! The different introduced levels enable the specification of additional information concerning the use and behavior of a component. When used in conjunction with the component system, the provided information can help in the debugging of the software. The component system could for instance check if the semantics of an operation are obeyed or if the synchronization rules are adhered to.

Opportunities for embedded software development

Besides the advantages shown above, the definition also has additional advantages for embedded systems.

First of all, components can be annotated with resource information. This can be done at the QoS level of the boundary view. Since this resource information is strongly dependent on the context, it will often only be an approximation. However, if this information is specified formally, tools can be built to analyze the system and its correctness. Even when it is not done formally, the annotated components will help the component composer in the selection of a component.

Also, if the component system could check automatically if the constraints can be met it will be a great help for the system developer. The component system could also schedule the resources needed by the different components in a way that doesn't break the system. This automatic scheduling enables the construction of dynamic systems, where resource needs can change at run-time. When the component system supports dynamic resource scheduling, it also becomes possible to build systems that allow graceful degradation. The component system can inform a component of a resource shortage; this component could then take a decision to solve or mask this shortage. Of course, if resources have to be available at all time, a static analysis will still be needed.

The component system can also be used to debug the developed software: it can intercept the sent messages between the components. This enables logging of messages or events together with timing information, which can help the debug process. As stated previously, the component system can also check if the components adhere to what is specified in their outside view.

Another advantage of using a component system is the shielding from the hardware. This facilitates reuse of components, because they are not too dependent on hardware anymore. Though, if a specific piece of hardware needs to be used, it could be encapsulated in a component. When a component needs this hardware, it will communicate with the associated component. In fact, only the interface is important. In some cases, when the hardware changes, it is possible to retain the interface. In that way the dependency of other components on the hardware is not broken. Even if the interface has to change, glue components can be used to solve this problem.

References

- [Szyperski] Component Software, Addison-Wesley/ACM Press, 1997
- [Beugnard, Jezequel, Plouzeau, Watkins] Making Components Contract Aware, Computer (IEEE), 1999
- [Della Torra Cicalese, Rotenstreich] Behavioral Specification of Distributed Software Component Interfaces, Computer (IEEE), 1999
- [Francis D'Souza, Cameron Wills] Objects, Components, and Frameworks with UML: The Catalysis Approach, Addison-Wesley
- [Van Belle, Verelst] The mobile multi-agent system Cborg, http://progwww.vub.ac.be/poolresearch/Cborg/
- [Noble] Three features for Component Frameworks, WCOP '99
- [Terzis, Nixon] Component Trading: The basis for a Component-Oriented Development Framework, WCOP '99
- [Dong, Alencar, Cowan] Correct composition of Design Components, WCOP '99
- [Sametinger] Software Engineering with Reusable Components, Springer, 1997
- [Keller, Hoelzle] Binary Component adaptation, ECOOP '98
- [Mezini, Lieberherr] Adaptive Plug-and-Play Components for Evolutionary Software Development, Object-Oriented Programming Systems, Languages and Applications Conference, SIGPLAN Notices vol 33, nr 10, 1998
- [Cornwell] Reusable Component Engineering for Hard Real-Time Systems, PhD thesis, University of York, UK, 1998
- [Rastofer] A Component Model for Distributed Embedded Real-Time Systems, GCSE '99 Young Researchers Workshop
- [Nierstrasz, Tsichritzis] Object Oriented Software Composition, Prentice Hall, 1995
- [Szyperski] Components and Objects Together, Software Development Magazine, May 1999