Components and contracts in software development for embedded systems

Berbers, Yolande
Rigole, Peter
Van Baelen, Stefan
Vandewoude, Yves

1st April 2004

“KULeuven” Department of Computer Science
Celestijnenlaan 200A, B-3001 LEUVEN
Yolande.Berbers@cs.kuleuven.ac.be

Abstract
This paper presents a methodology for the development of embedded software, supported by a tool chain. The methodology is based on the composition of reusable components with the addition of a contract principle for modeling non-functional constraints. Non-functional constraints are an important aspect of embedded systems, and need to be modeled explicitly. The tool chain contains CCOM, a tool used for the design phase of software development, coupled to DRACO, a middleware layer that supports the methodology at run-time.

Keywords: embedded software, components, contracts, middleware.

1 Introduction
Embedded systems are typically characterized by a specific functionality in a specific domain, where the software element is taking an increasingly important role. In embedded software, one has to consider non-functional and resource constraints when building a system (besides software quality aspects such as reusability). Embedded systems often have limited processing power, storage capacity and network bandwidth. A developer has to cope with these constraints and make sure that the software will be able to run on the constrained system. Often, embedded systems also have timing constraints on their computations.

Today, embedded software is becoming complex and it is no longer feasible to build every system from scratch. Therefore, it is important to reuse existing software as much as possible. This ensures that one can use validated software,
which in turn results in shorter development time. To enable reuse, we have
chosen for a component-based approach for building embedded systems.
Component software is quite common today in traditional applications. A large
software system often consists of multiple interacting components. These
components can be seen as large objects with a clear and well-defined task.
Different definitions of a component exist [1]; some see objects as components,
while others define components as large parts of coherent code, intended to be
reusable and highly documented. However, all definitions have one thing in
common: they focus on the functional aspect of a component.
For embedded software the non-functional constraints cannot be discarded.
Modeling explicitly these non-functional constraints enables one to safely reuse
components in a design, while being sure that the non-functional constraints will
be met. Our CCES (Components and Contracts for Embedded Software) approach
uses contracts to ensure this.
The paper discusses the core concepts of the proposed methodology in section 2.
Section 3 describes the supporting tool chain. Section 4 briefly relates our work
with the state-of-the-art and concludes this paper.
The presented work is the result of the SEESCOA project (Software Engineering
for Embedded Systems, using a Component Oriented Approach), funded by the
Belgian IWT1.

2 Core concepts of the proposed methodology
This section describes the Components and Contracts for Embedded Software
methodology. Before giving details about CCES, we list the main strengths and
characteristics of CCES:
• CCES components are loosely coupled to facilitate reuse
• CCES components communicate through ports
• Connectors are used to connect communicating ports
• CCES defines constructs for composing applications out of components
  o Some constructs describe design-time compositions (blueprints)
  o Other constructs describe run-time compositions (instances)
• Contracts are used to specify and verify non-functional constraints
  o Contracts can be used to specify and verify compositions at
design-time
  o Contracts can be used to verify the correct execution of
compositions at run-time
  o Currently, CCES supports contracts for timing and for bandwidth
requirements
• CCES is a methodology, supported by a CASE tool and by a runtime
environment. Both these tools will be described in the next section.

1 The Flemish government institution IWT (Institute for the Promotion of Innovation by
Science and Technology in Flanders).
2.1 CCES components
A CCES component is a reusable documented software entity, offering a coherent behavior, and is used as a building block in applications. A component is defined at design time, but also has a run-time existence and state. Components are composed by means of their interfaces: they can provide an interface to and require an interface from other components. Other components can access a component’s behavior by asynchronously sending messages to the component. Such communication can only take place if both components have a port, connected by the same connector.

2.2 CCES ports
There is a strong dependency between a port and an interface: a port is a representation of a bidirectional communication access point of a component, consisting of an interface for incoming messages and an interface for outgoing messages. Therefore ports can only be connected if their associated interfaces match.
To control non-functional constraints it is important to be able to control the number of connections that can be made from a port. This is described by a MNOI (Maximum Number of Instances) property of a port.

2.3 CCES connectors
A connector connects ports of components so that these components can communicate. Components send messages to each other using a connector as a kind of tunnel.
Only ports with compatible interfaces can be connected to each other. The compatibility of both the syntactic interface and the synchronization interface is checked both at design time and at runtime by the CCES tool chain.
Connectors can possibly cross node boundaries since components can be spread over various nodes in a distributed system. Components are unaware of the fact that they are communicating with local or with remote components. The underlying component system DRACO takes care of this transparency.

2.4 CCES compositions
An application is made up of a composition of components. For this purpose, CCES defines several constructs.
CCES distinguishes between blueprints and instances, both for components and ports. Blueprints are reusable static entities that only exist at design time. In contrast, instances represent instantiations of blueprints that have a runtime meaning; instances exist at runtime and have a particular state. A component blueprint contains the type description and implementation (the code) of a component. It has an identifier, a version and can be stored in a catalogue. A port blueprint represents the interfaces of the component blueprint. It describes the
messages that can be sent and received by the port blueprint, and it also prescribes
a particular protocol that needs to be adhered to when communicating with
instances of the port blueprint.
A component instance represents an instantiation of a component blueprint. One
application can hold several component instantiations of the same component
blueprint. A port instance is a gateway that has to be used by other component
instances to communicate with a component instance.
At runtime, instances are managed by the underlying component system DRACO.
Connectors are used to connect port instances.

2.5 CCES contracts
Contracts are used in CCES to specify non-functional constraints. They allow a
designer to impose constraints on the behavior of components and on the
interactions between them.
A CCES contract is used both for annotation and for verification. It is an important
aspect for a designer for documenting applications. Furthermore, it can be used for
verification. Some verifications can be done statically and are performed by
CCOM, our component composition tool. Other verifications are done dynamically
by a contract monitoring module in DRACO, which uses runtime contract
instances for verification. DRACO supports both offline verification, where logs
are made by the monitor for later analysis, and online verification, where the
monitoring system gives feedback of contract violations to the running application.
Currently CCES supports timing contracts, including deadline and periodicity
contracts, and bandwidth contracts. Work is underway in order to support memory
contracts as well.

3 Supporting tools
In order to support the CCES methodology two tools have been built: a CASE tool
enabling the development of CCES compositions, called the CCOM (Component
and Contract-Oriented Modeling) composition tool, and a runtime component
system responsible for the execution of CCES compositions, called DRACO
(DistriNet Reliable And Adaptive Components). This section gives an overview of
both tools.

3.1 CCOM Case tool
The CCOM tool offers a designer the possibility to develop applications by means
of the CCES concepts described in the previous section.
The CCOM tool supports the creation and development of:
• Component Blueprints: component blueprints can be created, specified and
  stored into a repository for later use,
• Compositions: compositions consisting of component instances,
  connectors and contracts can be created. A designer can make use of three
model types (or model views) in order to decompose the structure of a composition:

- Blueprint models: component blueprints, which are used in a composition, first need to be loaded into a blueprint model.
- Instance models: loaded component blueprints can be instantiated and added to instance models. These component instances can then be connected to each other by means of connectors.
- Scenario models: non-functional constraints are represented by contracts, which are attached to component instances, port instances and connectors.

### 3.1.1 Developing Component Blueprints

The development of a component blueprint occurs in two steps. The first step consists of the specification of the component blueprint.

The interface of a port blueprint can be specified in the tool on two levels:

- **Syntactic level:** the designer has to specify the messages that the port can receive or send, including a description of the types of the parameters associated with each message.

- **Synchronization level:** a designer also needs to specify the communication protocol of the port. This protocol specifies the send and receive sequence of exchanged messages. The protocol describing the interaction is represented by means of extended Message Sequence Charts (MSC).

The interface of a port blueprint is used to verify if it can be connected to other port blueprints: connecting ports is only possible if their interfaces match. The compatibility of ports can be verified both for the syntactic as for the synchronization level.

When developing a component blueprint the tool generates skeleton code. The syntax resembles the Java language syntax, but it has been extended with CCES keywords and constructs. The synchronization between a component blueprint specification and its implementation occurs automatically by the CCOM tool.

The second step of a component blueprint development consists of providing an appropriate implementation for the messages the component can receive. Once a component has been specified and implemented, it is transformed into an XML representation and stored in a component repository.

### 3.1.2 Developing compositions

A composition can be built by retrieving component blueprints from the repository and loading them into the composition. Next, instantiations of these component blueprints can be created and put into instance models. Connecting component instances is done by (1) instantiating the port instances that will communicate with each other and (2) creating a connector and attaching it to the created port instances.
The scenario model, used in the following step, enables one to impose non-functional constraints on parts of a composition by attaching contracts. A CCOM contract can be attached to one or more participants (component instances, port instances and/or connectors). The actual number and type of participants in a contract are dependent on the particular type of contract: a contract constraining the memory usage of a component is attached to a component instance, while contracts imposing timing constraints on the interaction between components are attached to the ports involved in the interaction.

To make it more concrete, we give more information about the timing constraints. In CCOM, timing constraints are specified by means of templates with properties that have to be filled in by the application designer. Using templates makes it easier to specify constraints, without the need to learn a particular formal specification notation.

In general, a CCOM timing contract specifies and imposes the timing constraints to which communicating components have to adhere. A timing contract is concerned with the communication between components. As such, it is straightforward to attach the timing contract to their ports, since these are the communication gateways between components. Furthermore, the communication between components is fully specified by the MSC of the involved ports. So this MSC plays a key role in the specification of a timing contract.

A hook is a point on an MSC that represents a particular communication action, e.g. we can distinguish a send hook, a receive hook and an end-of-activation (eoa) hook. A timing contract can be specified by means of these hooks. For instance, it is possible to specify that the maximum duration between the send hook and the eoa hook may not exceed 500 milliseconds.

### 3.2 DRACO runtime system

Next to the design-time tool support, we have also developed a runtime component system that enables the execution of CCES compositions: the DRACO component system. This runtime system is highly modularized in order to be configurable and targetable to specific applications (and thus guaranteeing a minimal memory footprint). The DRACO component system is the underlying infrastructure that offers an execution environment for the component instances. DRACO has been extended with the possibility to load optional functionality as optional modules. Through these optional modules one is able to add extra functionality to the component system.

The most important tasks of the core system are as follows:

- Management of component instances, connectors and contracts.
- Offering support for introspection and naming.
- Abstracting the underlying hardware and OS.
- Routing and scheduling messages.
The DRACO core system consists of 5 core modules. It has a size of approximately 62kB and is essential to execute CCES applications. Additional functionality can be added through optional modules. Optional modules can make use of the functionality of the core subsystem by means of reflection mechanisms and Publish-Subscribe patterns built into the core subsystem.

The 5 core modules are:

- **Component Manager:** is responsible for loading component blueprints, creating instances and removing them. It also keeps a repository of created component instances, with a basic directory mechanism mapping names onto component instances.
- **Connector Manager:** is a repository containing the connectors that exist between component instances in a composition. Each connector refers to the ports to which it is connected. Each port has a send message handler queue and a receive message handler queue associated to it.
- **Message Manager:** this module is responsible for delivering messages sent out by components. By means of the Connector Manager it retrieves the send message handler queue of the sending port and the receive message handler queue of the receiving port. The messages then traverse the send message handler queue of the sending port and arrive at the Scheduler.
- **Scheduler:** accepts messages coming from a send message handler queue and schedules them for delivery to the receive message handler queue.
- **Module Manager:** responsible for loading and unloading optional modules, which can be used to extend the functionality of the DRACO component system.

4. **Relation to the state-of-the-art and conclusion**

The state-of-the-art of component based development is too large to be presented here. We just contrast our work with alternative component-based frameworks specifically targeted to embedded systems, which have been developed in recent years.

In Koala [2], components are implemented in C and specify provides and requires interfaces that cannot be changed. Interfaces can be connected if the provided interface implements at least all methods for the required interface. The binding of these interfaces is made at the product level. All external information (including memory management) must be retrieved through require interfaces.

Other embedded component systems worth mentioning are PECOS [3] (a model for field-devices with the emphasis on formal execution models using petri-nets), Port-Based Objects [4, 5] (used in the Chimera RT operating system), VEST (A toolset for constructing and analyzing component based embedded systems) [6] and DESS (a generic component architecture and notation for embedded software development) [7, 8].

In conclusion we can say that our methodology is original in that it is supported by a tool chain, where both functional and non-functional constraints are checked both
at design-time and at run-time. These checks are generated by the tools, based on the design made by the developer.
More detailed description of this work can be found in [9, 10, 11].

References