

Components and Contracts in Software Development for Embedded Systems

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

DistriNet, Department of Computer Science, KULeuven
Celestijnenlaan 200A, B-3001 Heverlee
{Yolande.Berbers, Peter.Rigole, Yves.Vandewoude,
Stefan.VanBaelen}@cs.kuleuven.ac.be

Abstract. This paper presents CoCONES (Components and Contracts for Embedded Software), 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.

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. When developing embedded software, besides all kinds of software quality aspects, one has to consider non-functional and resource constraints. 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; according to [xxx] the complexity of embedded-system applications is increasing with 140% a year. It is no longer feasible to build such systems from scratch. Reuse of existing software is becoming vital, especially in the light of today's tight time-to-market demands in industry. Reuse should ensure that one can use validated software, only then will reuse result 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.

In the past few years, we have developed a methodology CoCONES (Components and Contracts for Embedded Software) for developing software for embedded systems, using a component oriented approach. Our approach uses contracts to model the non-functional constraints. The CoCONES methodology is backed by a tool chain that spans both the design-time and the runtime phase. CCOM (Component and Contract-Oriented Modeling) is a software design tool, enabling the developer to specify components and their interactions, including contracts for the non-functional constraints. DRACO (Dxxxx) is a middleware layer that at runtime will support our methodology. It allows components to be created and destroyed, organizes the communication between components, and monitors the contracts defined at design time.

We have applied our methodology using our tools to various smaller examples and to a full fledged case study. In this case study ..

This chapter gives a comprehensive overview of our methodology, the supporting tool chain and the case study. Elements of this work have been published in conferences and workshop: [xxxxxxx]. The presented work was started during the SEESCOA project (Software Engineering for Embedded Systems, using a Component Oriented Approach), and is continued in the CoDAMoS project (xxx). Both mentioned projects are funded by the Belgian IWT¹. This chapter is organized as follows: Section 2 gives an overview of our methodology. Section 3 and 4 respectively described the design-time tool and the runtime tool that together support our methodology. Section 5 presents our full fledged case. We compare our work with related work in section6, and conclude in section 7.

2 Core concepts of the proposed methodology

This section describes the Components and Contracts for Embedded Software methodology. Before giving details about CoCONES , we list the main strengths and characteristics of CoCONES :

1. CoCONES components are loosely coupled to facilitate reuse
2. CoCONES components communicate through ports
3. Connectors are used to connect communicating ports
4. CoCONES defines constructs for composing applications out of components
 - (a) Some constructs describe design-time compositions (blueprints)
 - (b) Other constructs describe run-time compositions (instances)
5. Contracts are used to specify and verify non-functional constraints
 - (a) Contracts can be used to specify and verify compositions at design-time

¹ The Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

- (b) Contracts can be used to verify the correct execution of compositions at run-time
 - (c) Currently, CoCONES supports contracts for timing and for bandwidth requirements
6. CoCONES is a methodology, supported by a CASE tool and by a runtime environment. These tools are described in sections 3 and 4

2.1 CoCONES components

The most common definition of a component was given by Szyperski in [1]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

In CoCONES, a distinction is made between components and component blueprints. The latter are reusable static entities that only exist at design time. They contain a complete description of the type of a component and its implementation (the code). It has a unique identifier, a version number and can be stored in a blueprint catalogue. In contrast, the term *component* is reserved for a runtime component instance containing a certain runtime state.

A CoCONES component complies with the definition of Szyperski: it is a reusable documented software entity, offering a coherent behaviour and is used as a building block in applications. In addition, all inter-component communication is explicit and takes place by sending asynchronous messages to the component through its external interfaces. In general, interfaces are an abstraction of the behaviour of a component and consist of a (subset of) interactions of that component, together with a set of constraints on when these interactions may occur. In CoCONES, a component interface consists of a group of messages that may be sent to or sent out from the component. These interfaces are formally specified using the port construct.

2.2 CoCONES ports

A CoCONES port represents a bidirectional communication access point of a component, consisting of an interface for incoming messages and an interface for outgoing messages. As with components, the distinction is made between port blueprints and ports. In CoCONES, a port is specified on 3 levels:

- Syntactic Level:** syntactic description of messages that can be sent and received.
- Semantic Level:** pre- en post-conditions associated to the messages.
- Synchronization Level:** description of the sequence in which the messages have to occur.

KANTLIJN OPMERKING YVES:
Eventueel kunnen we de QoS hier ook vermelden.

At the moment of writing, only the syntactic and the synchronization levels have been formally worked out in detail. Two ports can only be interconnected if their associated interfaces match on all levels. The number of connections that

can be made with a port is specified using the MNOI (Maximum Number of Instances) property of a port. A major advantage of this restriction is that with this additional knowledge about the usage of the component, the developer can make more accurate QoS statements about the services the component delivers. Evidently, these restrictions are enforced at runtime by our execution environment (see Sec. 4). CoCONES supports 3 types of ports with respect to this MNOI:

Single Port: A single port allows for one-on-one communications. This port is represented by a rectangle in our CCOM design tool (Fig. 1(a)).

Multiport: One multiport of dimension n is conceptually identical to n single ports as it allows for n connectors to be attached simultaneously. Although messages can be sent to the entire multiport as such (in this case it behaves as a multicast port), the intended behaviour of a multiport is to send messages to a specific index. Conceptually, a multiport is analogous to a call center: a connection is granted to a multiport unless it is already involved in its specified maximum of connections. Once connected, conversation is one-to-one. As depicted in Fig. 1(b), the symbol of a multiport depends on its dimension.

Multicast Port: A multicast port of dimension n is a single port that can have n connectors attached to it. Messages sent to a multicast port are always sent to all connectors attached to it. It is therefore not possible to differentiate between different receivers. Also, a multicast port can never receive messages. The graphical notation of a multicast port is a trapezium (Fig. 1(c)).

The dimension for both multiports and multicast ports may be ∞ .

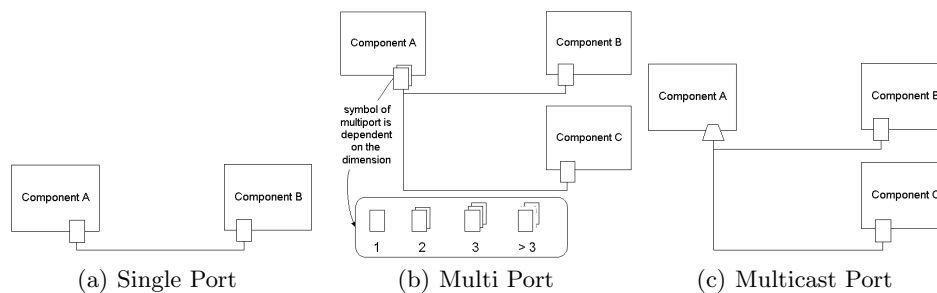


Fig. 1. Different ports in CoCONES

2.3 CoCONES connectors

Ports are connected using the *connector* construct. Compatibility of the port interfaces is checked both at design time and at runtime by the CoCONES tool

chain. As such, connectors act as a kind of tunnel during message transmission. Connectors provide a layer of abstraction of component location since they can cross node boundaries when different components are spread over various nodes in a distributed system: components are unaware whether they are communicating with local or with remote components. At runtime, the underlying middleware system (see Sec. 4) takes care of this transparency.

2.4 CoCONES contracts

Contracts are used in CoCONES to specify non-functional constraints. They allow a designer to impose constraints on the behaviour of components and on the interactions between them. Contracts can be attached by the designer when an application is constructed by composing components. They can be attached to all previously described constructs of the CoCONES architecture.

A CoCONES contract is used both for annotation and for verification. It is an important aspect for a designer for documenting applications. Furthermore, contracts are used to verify program correctness. 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 (see Sec. ??). Although the CoCONES contracts are a general construct, only timing contracts and bandwidth contracts have been worked out at the time of writing. Work is underway in order to support memory contracts as well.

A CoCONES timing contract specifies and imposes the timing constraints to which communicating components have to adhere. Timing contracts can be attached both to connectors and to ports (to specify constraints concerning multiple connections – e.g. 500ms after the arrival of message m on port $p1$, a response must be broadcast on port $p2$). Two types of timing contracts are currently supported: deadline timing contracts and periodicity timing contracts. A deadline timing contract imposes a constraint on the occurrence time of a particular event, given the occurrence time of an event that happened earlier. Possible events include the sending of a message, the receipt of a message, and the termination of the processing of a received message. A periodicity timing contract imposes a constraint on the periodic occurrence of a particular event.

CoCONES bandwidth contracts ... **PETER TODO**

2.5 CoCONES compositions

Applications are constructed by creating component compositions. In this process, guided by the CCOM design tool (see Sec. ??), existing component blueprints can be loaded from a component repository and visually be connected to each other. Additional components can be created and key properties (such as the number of ports and their interfaces) can be created by the tool. The CCOM tool generates the necessary skeleton code that can be filled in by the developer. This process is bidirectional, in that the properties of an existing code can be retrieved from its source code. During the design of the application, the CCOM

KANTLIJN OPMERKING YVES:
Als we hier al schrijven dat contract verification gebeurt, valt er in feite niet bijster veel meer te vertellen tijdens *composing compositions*. Eventueel uitstellen tot daar en hier enkel zeggen wat contracten zijn?

KANTLIJN OPMERKING YVES:

Wat kunnen we hier nog schrijven???

tool will check the compatibility of the connected ports, and, where possible, the feasibility of contracts.

2.6 Supporting tools

The entire methodology is supported by a toolchain. The CCOM (Component and Contract-Oriented Modeling) composition tool is a CASE design tool supporting the design and implementation of components and the construction of compositions. CCOM is capable of generating skeleton code that assists the developer in the implementation process. The code is then converted to standard Java with a preprocessor, and compiled. At runtime, DRACO (DistriNet Reliable And Adaptive Components) is the middlewaresystem responsible for the correct execution of CoCONES compositions. We discuss CCOM and its code generation in Sec. 3. DRACO is discussed in Sec. 4.

3 CCOM Case tool

The CCOM tool offers a designer the possibility to develop applications by means of the CoCONES concepts described in the previous section. The CCOM tool supports the creation and development of:

1. Component Blueprints: component blueprints can be created, specified and stored into a repository for later use.
2. 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:

1. Blueprint models: component blueprints, which are used in a composition, first need to be loaded into a blueprint model.
2. 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.
3. Scenario models: non-functional constraints are represented by contracts, which are attached to component instances, port instances and connectors.

3.1 Developing Component Blueprints

The development of a component blueprint occurs in two steps. The first step consists of the specification of the component blueprint, which includes the specification of the port blueprints attached to the component. The second step of a component blueprint development consists in 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 the component repository. Figure 2 shows a screenshot of the tool

during the development of a blueprint model. In our tool, blueprints are represented with dashed lines. Large rectangles are components, small ones are the ports attached to a component. Left in the figure is the repository of component blueprints, ordered hierarchically. Right in the figure is a blueprint model that groups several component blueprint needed in a car regulator or cruise control application.

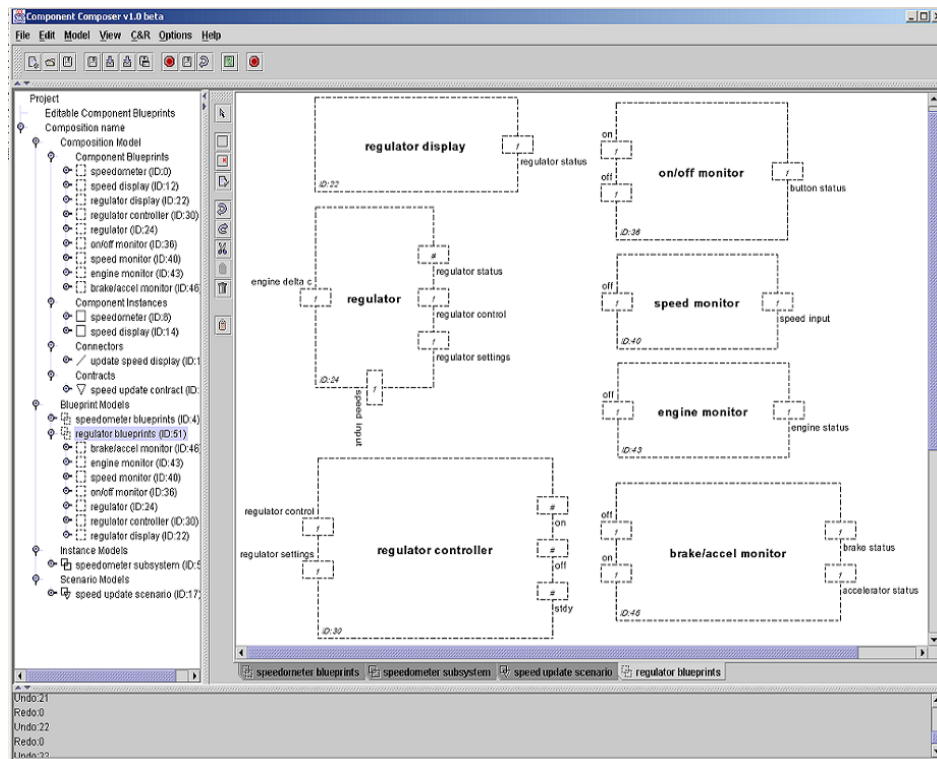


Fig. 2. A blueprint model in the CCOM tool

The car regulator system is used to align the speed of a car to a target speed requested by the driver using a cruise control. The regulator makes use of a speedometer to read the vehicle speed. At a frequency of 2 Hertz, that is every 500 ms, the regulator should calculate the new speed of the car, and pass this on to the engine. The regulator should be stopped a.o. when the brakes are hit, the driver accelerates, the drivers turns down the cruise control, the speed drops below a certain limit. Different components in our application take case monitor these happenings and possibly turn off the regulator. 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).

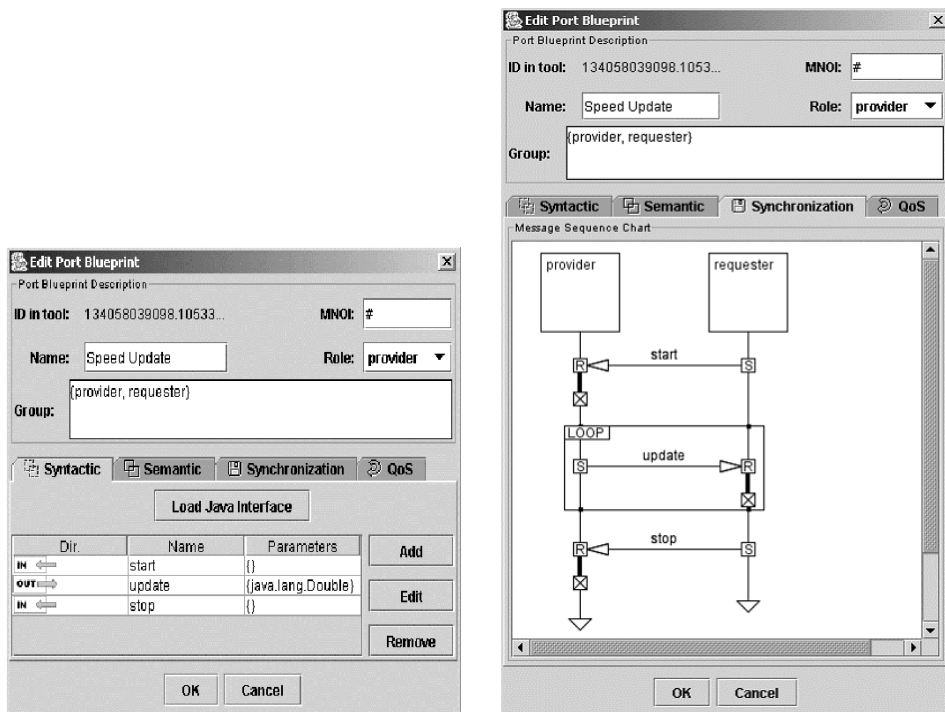
Figure 3 shows two more screen shots where the interface of a port blueprint is being specified. The port in this figure is the `speed_update` port, which is part of both the speedometer component and the `speed_display` component. The speedometer component calculates the current speed of the car at a frequency of 2 Hertz. It can output its speed calculation to an unlimited number of other components. Amongst other, the speed is send to the `speed_display` component, which displays the speed on the dashboard of the car. The left part of Fig. 3 shows how the syntactic elements of the port blueprint can be filled in: every message can be described including its name, parameters and direction. The right part of the figure shows how the synchronization level is specified. Here extended MSC's (Message Sequence Charts) are being used. From the MSC it becomes clear that the `speed_update` port of the Speedometer component first receives the start message and that it sends update messages in a loop. The interaction stops when the port receives a stop message. For each message interaction three hook types can be distinguished:

1. Send hooks (boxes with 'S') representing the sending of messages.
2. Receive hooks (boxes with 'R') representing the reception of messages.
3. End-of-activation hooks (boxes with 'X') representing the termination of the processing of a received messages.

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 CoCONES keywords and constructs. The synchronization between a component blueprint specification and its implementation occurs automatically by the CCOM tool.

3.2 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



(a) Specification of the Syntactic Interface (b) Specification of the Synchronization Interface

Fig. 3. Specification of the interfaces of a port blueprint

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 elaborate here on 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 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 example, a deadline contract could specify that the maximum duration between the send hook and the eoa hook may not exceed 500 milliseconds. A deadline contract has thus 3 parameters: a hook that starts the contract, a hook that ends the contract, and the maximum allowed time difference between the occurrences of these hooks. The second type of timing contract that CCOM supports is the periodicity timing contract. Figure 4 shows how such a contract can be specified in our tool. The window in the tool shows the MSC, with the hooks in each message, and the message names. The four necessary parameters of the periodicity contract can be filled in at the bottom: in this example the contract starts at the sending of the start message, the periodic event is the reception of the update message, the contract ends with the sending of the stop message, and the period is 500 ms. HIER MOET IETS KOMEN OVER DE CODE DIE CCOM GENEREERT, OOK IN VERBAND MET CONTRACTEN.

3.3 From design to execution: Code generation

To facilitate the development of components in CoCONES, the skeleton code of the component and its ports is generated by the CCOM tool. CCOM also ensures synchronization between a component blueprint specification and its implementation. For the further implementation of the component, a custom language is used. This language is a superset of Java that supports relevant component-based constructs. The code is automatically preprocessed, compiled and packaged into a binary that is ready for deployment in the runtime environment. As such, the design is directly used as input for the implementation.

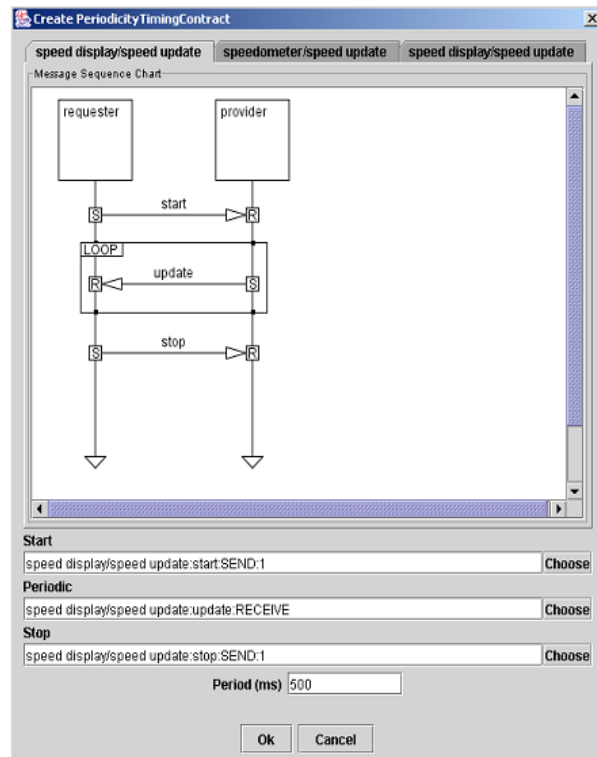


Fig. 4. The specification of a periodicity timing contract in the CCOM tool

We shortly illustrate the language with a small consisting of two components: a `NumberGenerator` and a `NumberDisplay`. The first component will generate a random number on request which it will broadcast on its `Out` port. The second component prints out all values it receives through its `Input` port. The implementation of these two components is shown in figure 5.

<pre> package components.numberGenerator; import java.util.Random; component NumberGenerator { Random \$rnd = new Random(); multicastport Out UNLIMITED; portgroup Control 1 { message SendNumber { message x = Number; x::value = new Integer(\$rnd.nextInt(100)); Out..x; } } } </pre>	<pre> package components.numberdisplay; component NumberDisplay{ portgroup Input 1 { message Number { System.out.println("Received_number:_" + \$\$inMessage::value); } } } </pre>
(a) NumberGenerator	(b) NumberDisplay

Fig. 5. Two simple components in CoCONES notation

KANTLIJN OPMERKING YVES:

Eventueel kan voor dit voorbeeld (een stuk van) de car regulator genomen worden ipv weer een ander vb? Dat moet dan nog wel geschreven worden.

The implementation of a component starts with the `component` keyword. It consists of zero or more attributes (e.g. `$rnd` in 5(a)) and methods (not shown for this trivial component) and the description of its ports. The declaration of a multicast port is straightforward since it can not accept messages. It suffices to specify its existence using the `multicastport` keyword. Two parameters are required: the name of the port and the maximum number of simultaneous connections that are allowed. A multiport has a similar declaration, but since it *can* accept messages, these messages must be declared as well using the `message` keyword. The definition of a message includes the code to be executed when the message is received on the port.

New messages can be created using the statement `message varName = messageName`. After its creation, this message can be sent out through any connected port. Message sending is asynchronous and as such, the sending of a message is always successful. If the component on the other side of the connector does not accept `messageName`, the system will return a `CannotDeliverMessage` message.

Finally, two additional operators were added: fields of a message are accessed with the `::` operator whereas the `..` operator is used on a port to send out a given message. Inside the implementation body for a message, the implicit parameter `$$inMessage` refers to the received message.

4 DRACO runtime system

Next to the CCOM tool, the toolchain supporting the CoCONES methodology also contains a runtime environment capable of executing CoCONES composi-

tions: the DRACO component system. DRACO is a middleware system that offers the underlying infrastructure that offers an execution environment for the component instances. The runtime system is highly modularized. As such, it can be configured and targeted to specific applications, while guaranteeing a minimal memory footprint.

The architecture of DRACO is depicted in Fig. 6. It consists of a core system which provides the minimal functionality to execute CoCONES applications. The most important tasks of the core system are as follows:

1. Management of component instances, connectors and contracts.
2. Support for introspection and naming.
3. Abstraction of the underlying hardware and OS.
4. Routing and scheduling messages sent between components.

The core system consists of 5 units and its footprint is less than 65kB, allowing it to be installed on embedded devices with stringent resource constraints. At startup, the core is dynamically assembled using the builder pattern [2]. Since the builder reads an XML file describing which implementation to use for each of the core units, modifying or replacing one core unit has no impact whatsoever on the rest of the system. The ability to easily customize its core makes DRACO an excellent platform for various types of research (e.g. replacing the scheduler would allow us to investigate the influence of the scheduling algorithm on the execution of a component based application, ...). Furthermore, it allows for further customization depending on the target platform. Once instantiated, however, the core is considered to be *fixed*. In order to keep the complexity (and size) of DRACO sufficiently low, no attempt was made to allow for unanticipated modifications of the DRACO core at runtime. 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.

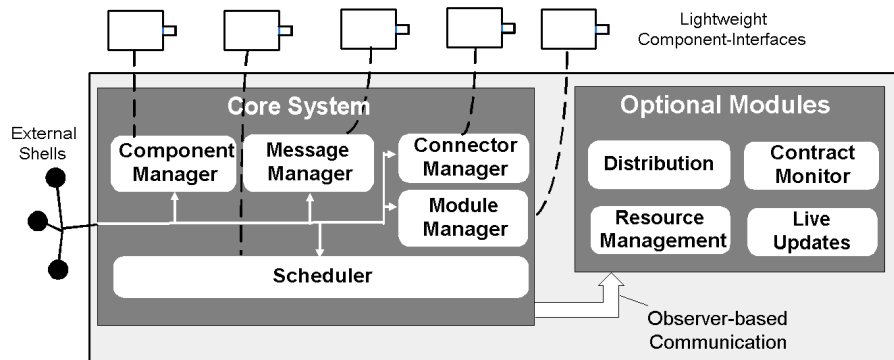


Fig. 6. Overview of the DRACO architecture

As shown on top of Fig. 6, each of the core modules exports a lightweight component-interface. These appear as components in the runtime system, and can be used by application components to query or configure the underlying middleware environment.

Interaction between DRACO and the user is handled by an external *shell*, which is provided to DRACO at startup and resides in a different binary. By separating the user interaction from DRACO, it is possible to use different interaction shells depending on the situation. An interactive shell with scripting capabilities is available for use during development on a high performance desktop machine, while a thin layer with minimal functionality can be used when resource consumption is an issue.

Although the functionality of the core system is relatively limited, DRACO offers an infrastructure which allows the addition of functionality that may not always be required: optional modules. These modules can be loaded and unloaded at runtime by the module manager.

The following optional modules have been worked out:

The distribution module (DM): this module adds distribution functionality to the core platform in a complete transparent way. It introduces the notion of proxy components, similar to the proxy pattern defined in [2]. These proxy components are lightweight components that represent remote components. As such, they offer the same ports and exactly the same semantic information. The DM is responsible for (1) setting up and tearing down connections² between remote DRACO systems, and (2) responsible for managing proxy components and generating them based on real components. No stubs or other design-time entities are required for making components communicate in a distributed way, which incorporates a considerable advantage

² In DRACO, a connection is an abstract concept that can be implemented by any kind of physical wired or wireless connection.

over traditional approaches that need additional constructs (e.g. the subs and skeletons used by Java RMI).

The contract monitor: this module checks whether contracts are violated at runtime. Depending on the type of the contract, the monitoring differs. For the timing contracts, messages are intercepted and time stamped by an eventgathering unit. These time stamps are used by the eventprocessing unit to verify the timing contract. The unit responsible for the verification of the contract, can be moved to another node to minimize intrusion on the target platform. Currently, contract violations are reported offline: a developer can analyze the occurred violations after an application's execution. In the future, a contract violation will be reported to the application, which then must take corrective measures.

The resource manager: this extension module is responsible for the negotiation of contracts with applications when these are started. The resource manager knows what contracts are currently active, and can then accept new contracts in function of available resources.

The Live update module (LUM): allows components to be replaced at runtime, even while they are part of a running application. The LUM achieves this by (1) putting the component to be replaced in an inactive state by temporarily holding back the messages, (2) instantiating the new version of the component, (3) possibly transferring the internal state from the old components to the new components, using routines provided by the developers of the components, (4) rewiring the connectors of the old version to the new component, (5) activating the new component by deblocking the messages that were held back in step 1 (6) removing the old component.

Since the exact tasks and thus requirements of extension modules are unknown in advance, they can make use of reflection mechanisms and may subscribe to one of the many events triggered by the core system.

In addition, they can interfere with the message flow and interact with the delivery of messages. In DRACO, messages are sent asynchronously between components. The path followed by a message traveling from component A to component B consists of 3 major parts (see Fig. 7(a)): the sending message chain, the scheduler and the receiving message chain. Each extension module can add message handlers to these message chains to implement the features they want.

The sending message chain comprises the journey of a message from the moment it is sent through the port of the originating component until it is scheduled for execution by the scheduler. Its detailed implementation in DRACO is shown in figure 7(b).

In the first step, the component contacts the port through which the message will be sent. Since ports are implemented as inner classes in DRACO, this is achieved with a local call (arrow 1). The port will pass on the message to the message manager (arrow 2) which will retrieve the attached connector from the connector manager (arrow 3). Each connector is associated with 4 message handlers (the first handler of both the sending and receiving chain of each direc-

tion: component *A* to *B* and vice versa). The message manager retrieves the two handlers associated with the current message direction. The receiving message handler is used for the delivery of the message after it has been scheduled for execution by the scheduler (see further). It is therefore simply passed on with the message to the sending message handler (arrow 4). This sending message handler is the first (and in the most basic scenario also the last) handler in a chain of message handlers. Each handler in the chain has the ability to intercept and modify the message, and will then forward it to the next handler in the chain (arrow(s) 5). The last handler is responsible for the delivery to the scheduler (arrow 6).

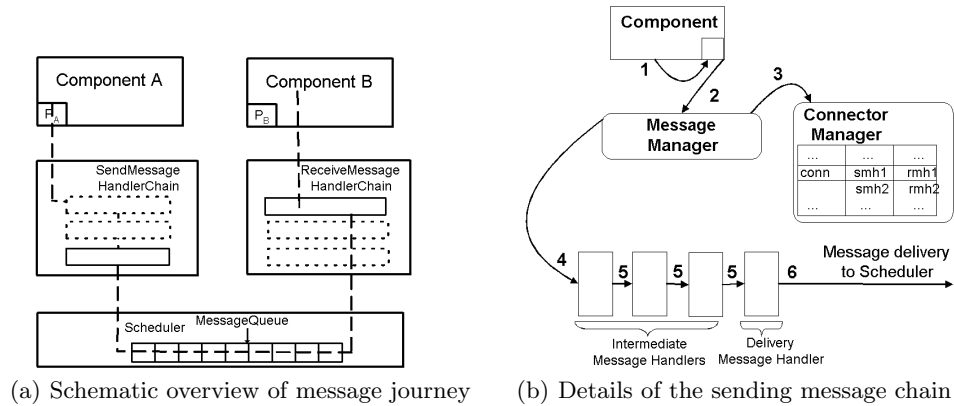


Fig. 7. Message Delivery in DRACO

After receiving both the message and its associated receiving message handler, the scheduler queues the message until it is ready for execution. The exact queuing mechanism depends on the scheduler that is used, but it is the responsibility of the scheduler to preserve the order of messages over a given connector.

When the scheduler has selected a message for delivery, it allocates a thread for the execution of this message, and passes on the message to its receiving message handler. The principle behind the receiving sequence is identical to the sending sequence: there is a chain of message handlers that process the message (e.g. the timing monitor can read out the time stamp added to the message by his peer in the sending message chain) and subsequently pass it on to the next handler in the chain. The last handler delivers the message to the port at the end of the connector. This port will then dispatch the message to the actual method associated to the message. After message execution, control is returned to the scheduler.

5 Extensive case study

Several embedded applications have been developed using the CoCONES design methodology and the supporting tool chain. One of these is the car regulator introduced in section 3. This application let us experiment with our timing constraints. However, as our implementation was not run on a car, and not on embedded hardware, it was in our eyes still a toy application.

We then developed a full fledged application, with the specific intention to test the reusability of our components and our designs. This led to a camera surveillance system of which several variations were designed and implemented. The surveillance system can be used for security related purposes such as physical intrusion detection and registration of activity in home and office buildings. A PC/104 embedded computer (holding the operating system, a Java virtual machine, our componentsystem and the test case code) was connected to a DFW-VL500 firwire digital camera. It was linked over a TCP/IP network with a desktop PC serving as a storage and control station.

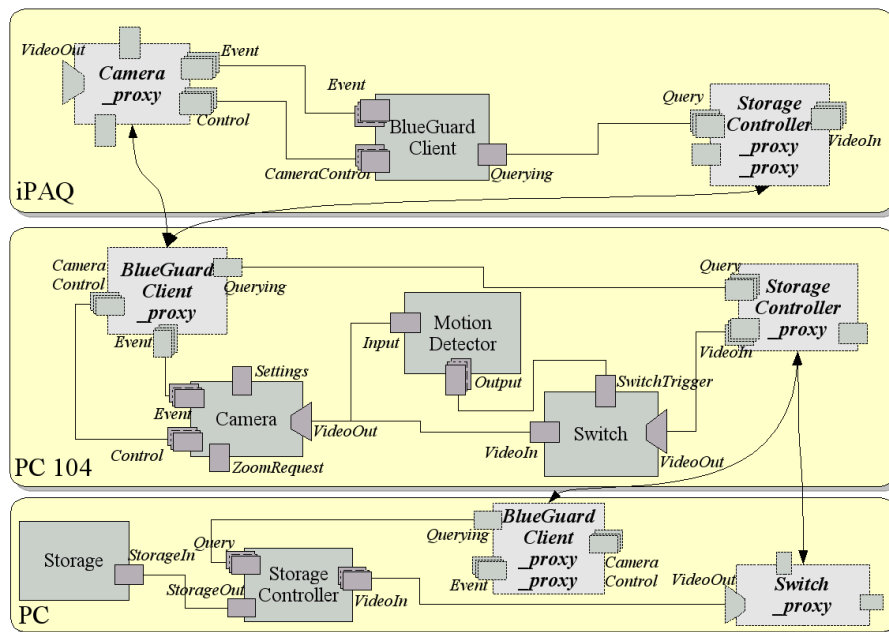


Fig. 8. Overview of the Camera Surveillance Case

Figure 8 gives an overview of the component compositions in the surveillance case. Device boundaries are indicated by surrounding boxes. The central Camera component on the embedded device (PC 104) continuously grabs images from the camera at a predefined rate and multicasts them towards the Motion De-

tector and the Switch component. The motion detector analyzes the images and produces an alarm-start output message when motion is detected. The switch, receiving the message from the motion detector, forwards the video stream towards its output port until the alarm-stop message is received, meaning that motion has ceased. The suspicious images are sent to the Storage Controller component, which is located on the desktop PC. Proxy components are introduced for handling the remote communication transparently. The StorageController proxy and the Switch proxy allow for remote communication between Storage and keywordSwitch. The Storage component, which encapsulates database access, eventually stores the images. The core application, as just described, was developed in CCOM and runs on top of DRACO, running a distribution module. In order to demonstrate reuse, several variations and extensions of the core application were developed. In one of the variations the motion detector component and the switch component were replaced by a component that passes one image out of 20 to the storage controller. This was a straight forward change, and all the other components could be reused without any change.

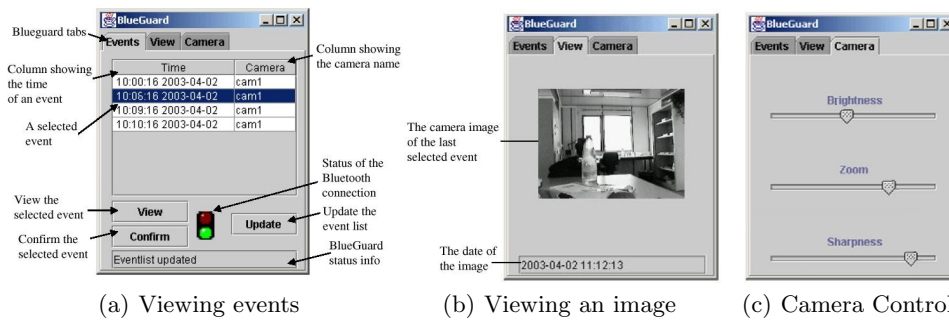


Fig. 9. BlueGuard client

We then wanted to take reuse one step further and ported our component system on an iPAQ. We then designed BlueGuard which provides security guards with more information about the safety inside the building. This extension allows the guards to query the recorded images using a handheld device when they are in the neighborhood of an observation station. For this extension a BlueGuard-Client component was created that is available on the handheld device.

Furthermore, the distribution module was extended with Bluetooth [?] connection support to enable short-range wireless access to our observation stations. The `BlueGuardClient` component, instantiated on the iPAQ handheld devices (see figure 8), provides the user interface used by the security guards. This component can be connected to a `StorageController` component for querying purposes and to several `Camera` components for adjusting camera settings such as focus and zoom. As depicted on the example setup (figure 8), the `BlueGuardClient` may be connected to a `StorageController` component through the embedded PC 104 module using proxy components for routing their messages. Figure 9(a) shows the tab view of all recent events and the Bluetooth connection status. The image associated with each event can be requested and is shown in the view

tab (figure 9(b)). The third tab (figure 9(c)) allows for changing the brightness, sharpness and zoom parameters of the camera. An in-depth description of this BlueGuard extension can be found in [?]. The core camera surveillance system, its variations and its extension, have proven the soundness of our methodology and our approach.

6 Relation to the state-of-the-art

TODO (Nog te refereren) :

- Beugnard et al. Screef dat component-interfaces op 4 niveaus worden beschreven: basic, behavioural, synchronization en qos. Lijkt me zeker related.

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 [?], 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, 4] (a model for field-devices with the emphasis on formal execution models using petrinets), Port-Based Objects [5, 6] (used in the Chimera RT operating system), VEST (A toolset for constructing and analyzing component based embedded systems) [7] and DESS (a generic component architecture and notation for embedded software development) [8, 9].

The Fractal component model [10] in particular is very interesting because it is in several aspects based on the same principles as CoCONES 's component model. Support for extension and adaptation is its prime concern and it is aiming at a broad range of host devices from embedded systems to application servers. It has, however, a less strictly defined component definition than CoCONES 's components and provides a language independent interface definition for its components. This interface definition can be used to connect components written in different languages. In addition, there is also support for composite components and it allows (sub)components to be shared between components. In the Fractal model, connections between two or several components are called bindings. There are two types of bindings, primitive bindings and composite bindings. Primitive bindings are language-level bindings (synchronous or asynchronous) whereas composite bindings are a composition of primitive bindings and components. Using this definition of inter-component bindings, flexible distributed applications can be built. Each component has a component controller that can control all internal behaviour of the component such as affecting operation invocations, influencing the behaviour of internal components, creating new components, etc.. The Java-based Fractal framework that supports the Fractal

component model consists of a core and several extensions, called increments. Like in DRACO, these increments can add new functionality to the core. The core offers a basic API for performing actions such as creating components, adding bindings between components and managing the content of components. Some of the increments under development allow for component bootstrapping, component distribution (distributed bindings), mobility and protection (resource management and distribution).

7 Conclusion

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 [11–13].

References

1. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley (2002)
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley (1994)
3. Winter, M., Genssler, T., Christoph, A., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Müller, P., Stich, C., Schönhage, B.: Components for embedded software - the pecos approach. In: *Proceedings of the Second International Workshop on Composition Languages*, Malaga, Spain (2002)
4. Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Müller, P., Zeidler, C., Genssler, T., van den Born, R.: A component model for field devices. In: *Proceedings of the IFIP/ACM working conference on Component Deployment*, Berlin (2002)
5. Stewart, D.B., Khosla, P.K.: The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering* **6** (1996) 249–277
6. Stewart, D.B., Volpe, R.A., Khosla, P.K.: Design of dynamically reconfigurable real-time software using port-based objects. *Software Engineering* **23** (1997) 759–776
7. Stankovic, J.A.: VEST — A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science* **2211** (2001) 390–??
8. : Definition of components and notation for components. Technical report (2001) <http://www.dess-itea.org>.
9. : Timing, memory and other resource constraints. Technical report (2001) <http://www.dess-itea.org>.
10. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: *Proc. of the Seventh International Workshop on Component-Oriented Programming*, Malaga, Spain (2002)

11. Urting, D., Holvoet, T., Berbers, Y.: Embedded software development: Components and contracts. In Gonzalez, T., ed.: Proc. of the IASTED Conference on Parallel and Distributed Computing and Systems. (2001) 685–690
12. Urting, D., Baelen, S.V., Holvoet, T., Rigole, P., Vandewoude, Y., Berbers, Y.: A tool for component based design of embedded software. In Noble, J., Potter, J., eds.: Proceedings of 40th International Conference on Technology of Object-Oriented Languages and Systems (Tools Pacific 2002). Volume 10., Sydney, Australia, Australian Computer Society Inc. (2002) 159–168
13. Rigole, P., Berbers, Y., Holvoet, T.: Design and run-time bandwidth contracts for pervasive computing middleware. In Urrahy, C., Sztajnberg, A., Cerqueira, R., eds.: Proceedings of the first International Workshop on Middleware for Pervasive and Ad Hoc Computing (MPAC)., Rio De Janeiro, Brazil (2003) 5–12