

# Draco : An adaptive runtime environment for components

*Yves Vandewoude*

*Peter Rigole*

*David Urting*

*Yolande Berbers*

*Report CW 372, 2003*



Katholieke Universiteit Leuven  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Draco : An adaptive runtime environment for components

*Yves Vandewoude*

*Peter Rigole*

*David Urting*

*Yolande Berbers*

*Report CW 372, 2003*

Department of Computer Science, K.U.Leuven

## **Abstract**

This paper introduces the architecture of Draco (DistriNet Reliable and Adaptive COmponents), a modular and extensible component runtime system for embedded devices. It adheres to the SEESCOA component methodology which explicitly models component interaction using port and connector concepts and allows for dynamic adaptations of component oriented applications by rewiring components at runtime. The SEESCOA component language and the constructs it introduces are presented using two example components. In addition, a mapping from the seescoa component language to Java is worked out. In order to preserve its small footprint, the draco runtime environment can be extended with extension modules to implement a variety of features (e.g. distribution, runtime contract monitoring or dynamic updating). These extension modules can influence the message delivery process using message handlers. To illustrate the power of this technique, two extension modules are worked out in detail.

**Keywords :** Middleware platform, Components.

# 1 Introduction

For a long time, increasing reuse and modularity has been the focus of enormous efforts in both academic and industrial research. The reasons for this are clear: reuse decreases development time and thus reduces time to market, increasing modularity isolates changes to one location and therefore reduces the complexity of software maintenance and modification. The result is that many modern object oriented languages as Java or C# successfully stimulate reuse through extensive open APIs. Since its introduction in the mid 1990's, component-oriented development takes this paradigm one step further by exploiting extreme low coupling between different components and integrating other features as concurrency, persistence and distribution. As such it can be argued that the component-oriented methodology is the first that truly realizes the idea of mass-producible software as described by McIlroy in 1969 [1].

Adoption for the development of embedded systems is significantly slower. Mainly due to their scarce resources and real-time behaviour, they have a history of low level design in which reuse and maintainability are often sacrificed for speed and efficiency. Recently however development for embedded systems is shifting towards high level languages and systems like Java. One important reason is that many problems that arise when using Java with embedded systems have been resolved or at least alleviated in the last couple of years (e.g. improved garbage collection algorithms, virtual machines with low memory footprint, ...). Perhaps the intention of Nokia to release 100 million Java-enabled units by the end of 2003 ([2]) illustrates best that in the near future embedded applications do not necessarily have to be low-level applications.

Components also, are a hot topic, since the majority of their advantages are also applicable to embedded systems: next to their reduced time-to-market and development costs, components fit extremely well in the context of large-scale distributed systems and are excellently suited for mass deployment.

In the context of SEESCOA<sup>1</sup>, a cooperation project between the KULeuven and three other Belgian universities, a component methodology was developed for embedded systems. One of the merits of SEESCOA is establishing a formal definition of a component and introducing an approach in which different components are dynamically connected to form the application ([3]). In addition, a tool to support component-oriented design (the CCOM Composer tool: [4]) and a component runtime were developed.

The current SEESCOA executing environment ([5]) is able to execute an application constructed using SEESCOA components and it has been deployed successfully on an embedded system in several case studies (e.g. see [6] for a distributed camera surveillance system).

However, from our experience implementing larger cases using the SEESCOA methodology, the current runtime environment proved to be too hard to extend. In this paper, we introduce DRACO, a much improved version of the current SEESCOA runtime. The DRACO component runtime is a lightweight, robust and extendible component system that adheres to the SEESCOA component philosophy. Components written for SEESCOA and that follow the SEESCOA guidelines and specifications are able to run on DRACO without large modifications.

The remainder of this paper is organized as follows. In section 2, we first introduce some relevant concepts of the SEESCOA methodology. Section 3 presents the DRACO architecture. This architecture has its impact on component development. In section 4 we discuss how a component can be written for

---

<sup>1</sup>SEESCOA stands for **S**oftware **E**ngineering for **E**mbded **S**ystems using a **C**omponent-**O**riented **A**pproach. The project is funded by the Belgian IWT.

DRACO. In section 5, we illustrate how the DRACO runtime can be extended using extension modules. Two examples are worked out in detail:

- A module for dynamically updating a running application.
- A module that can be used to monitor timing contracts.

Related work on components, dynamic updating and timing is discussed in section 6. We discuss future work in section 7 and conclude in section 8.

## 2 SEESCOA Component Methodology

This SEESCOA methodology is built around the following concepts: component, specification, port, connector and contract. The development of a component application in the SEESCOA methodology is supported by the CCOM Composer tool ([4]). This tool allows an application developer to compose an application from existing components and statically checks the feasibility of that composition. However, neither the CCOM Composer tool nor the SEESCOA methodology are the focus of this paper, and therefore we will limit ourselves to a short description of the most important concepts of SEESCOA that are relevant for the DRACO runtime environment.

### 2.1 Component blueprint and component

**Definition 1 (Component Blueprint)** *A component blueprint is a reusable entity and contains the type description and implementation of a component. It is a static construct that has no run-time meaning. Component blueprints have an identifier, a version and can be stored in a catalogue.*

**Definition 2 (Component)** *A component is a reusable documented entity that is used as a building block in applications. It is an instantiation of a component blueprint and as such it has a run-time existence and state. It performs a specific function, and to carry out this task it operates and interacts with other components within the component system. Components are composed by means of their interfaces: they can provide an interface to and require an interface from other components.*

To perform their services, components often need to interact. A component has three main activities: receiving messages, performing computations on behalf of these messages and sending out messages.

### 2.2 Ports

A port is used to allow for inter-component communication.

**Definition 3 (Port)** *A port is a communication gateway for components: a component uses ports to communicate with other components. Every component has zero, one or more ports associated with it.*

There is a strong dependency between a port and an interface: a port is a representation of an interface of a component. Therefore ports can only be connected if their associated interfaces<sup>2</sup> match.

---

<sup>2</sup>The interface of ports in SEESCOA is specified on 4 levels: syntactic, semantic, synchronization and quality of service. For more information see [3].

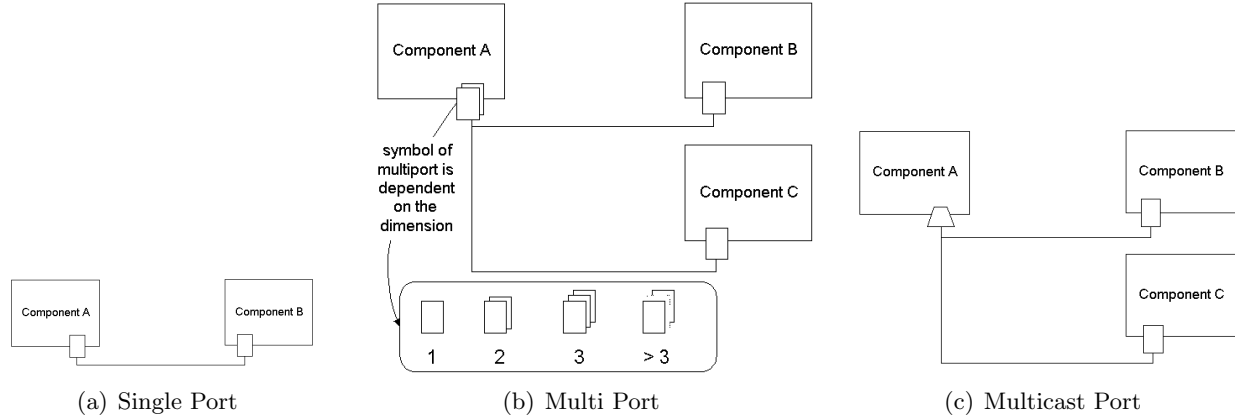


Figure 1: Different ports in SEESCOA

In SEESCOA, the port concept specifies which interfaces a component implements and how many connections are allowed simultaneously. 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. Hence, it is required that DRACO enforces both of these restrictions. In SEESCOA, three types of ports exist:

**Single Port:** A single port allows for one-on-one communications. In the CCOM Composer tool, this port is represented by a rectangle (figure 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 figure 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 (figure 1(c)).

The dimension for both multiports and multicast ports may be  $\infty$ .

### 2.3 Connectors and Contracts

**Definition 4 (Connector)** *A connector represent a functional connection between components. This means that components will send messages to each other using this connector as a kind of tunnel.*

When connecting two ports by means of a connector, the CCOM Composer tool checks the compatibility of both ports at design time. However, since foolproof checks are not always possible, it remains important that port compatibility can be checked at runtime.

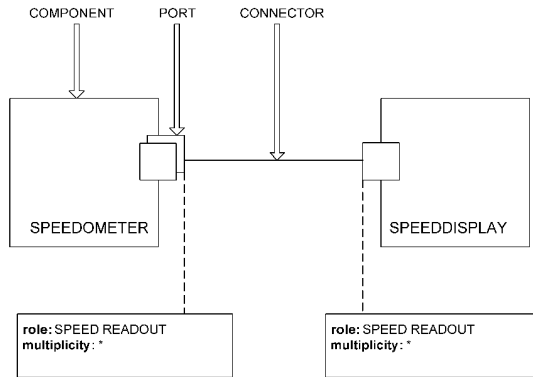


Figure 2: Component, Port and Connector

**Definition 5 (Contract)** *A contract imposes a non-functional constraint (for instance a timing or memory constraint) on a component or on a group of interacting components. Contracts can be associated with components, ports and connectors.*

We have decided not to model contracts explicitly in the DRACO runtime. However, if contract checking is desired, an optional monitor can be added (see section 5.2).

### Composing components

Two components can only be composed with each other, if their interfaces match. This principle is illustrated in figure 2. A basic static check of interface (port) compatibility is performed by the CCOM Composer tool at system design time.

## 3 The DRACO Runtime

During the design and implementation of DRACO, following requirements had to be met:

**Robust:** DRACO is intended to be used in systems with high reliability demands.

**Small:** Since DRACO will be deployed on systems with limited resources, a small footprint is required. The current DRACO implementation is contained in a `jar` with a size  $< 65$  kB.

**Extendible:** To keep the base system very small and lightweight, DRACO must be extendible in order to offer more advanced features when necessary. Extensions that will be implemented for DRACO include Distribution, Live Updates, Contract Monitoring and Resource Management.

**Performant:** The implementation will be as efficient as possible and will try to eliminate all unnecessary overhead. DRACO should be at least as performant as the current SEESCOA runtime.

### 3.1 Architecture overview

The DRACO architecture is shown in figure 3. Its core system consists of 6 modules. Each of these modules implements a strict interface by inheriting from fixed base classes. At startup, the core is dynamically constructed using the builder pattern ([7]). Since the builder reads an XML file describing which implementation to use for each of the core modules, modifying or replacing one core module 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, ...). 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.

**Component Manager:** The component manager is responsible for creating components and ports. It maintains references to all component instances in the component system<sup>3</sup> and will allow for queries to retrieve components that implement a specific component blueprint.

**Scheduler:** The scheduler is responsible for scheduling the messages for delivery. The scheduler must make sure that the order of messages are preserved for each port (i.e. if a message *a* is sent before message *b*, *a* will be scheduled for execution before *b*). In a distributed context, there is always exactly one scheduler available for each component system. The scheduler is also the only entity in the component architecture that deals with threads. It is the responsibility of the scheduler to make sure that no more than one execution thread accesses a component at any moment in time.

**Message Manager:** The message manager is the module responsible for the delivery of the messages. It will interact with the connector manager to retrieve the connector associated with the port from which the message originates.

**Connector Manager:** The connector manager is responsible for creating and maintaining the connectors between component ports. It also maintains the message handlers that are associated with these connectors and that guide the message delivery.

**DRACO control interface (DCI):** The DCI is responsible for interfacing the DRACO system with the outside world (e.g. to load new components, to start or stop applications, ...). It provides a clean and complete API that can be used to interact with the component system itself. This API is then used by a *shell* that takes care of the actual interaction with the user. By separating the user interaction from DRACO, it is possible to use different interaction shells depending on the situation. A graphical shell can be used on a high performance desktop, while a thin layer with minimal functionality can be used when resources consumption is an issue. The shell is a separate `jar` which is provided to DRACO at startup.

**Module Manager:** This core module is responsible for extending the DRACO component runtime with extra functionality that may not always be required: **extension modules**. These extension modules can be loaded and unloaded at runtime. A few examples of extension modules that

---

<sup>3</sup>In a distributed context, each node will run the entire DRACO system consisting of the six core modules. Therefore, the component manager is only aware of components in his own node.

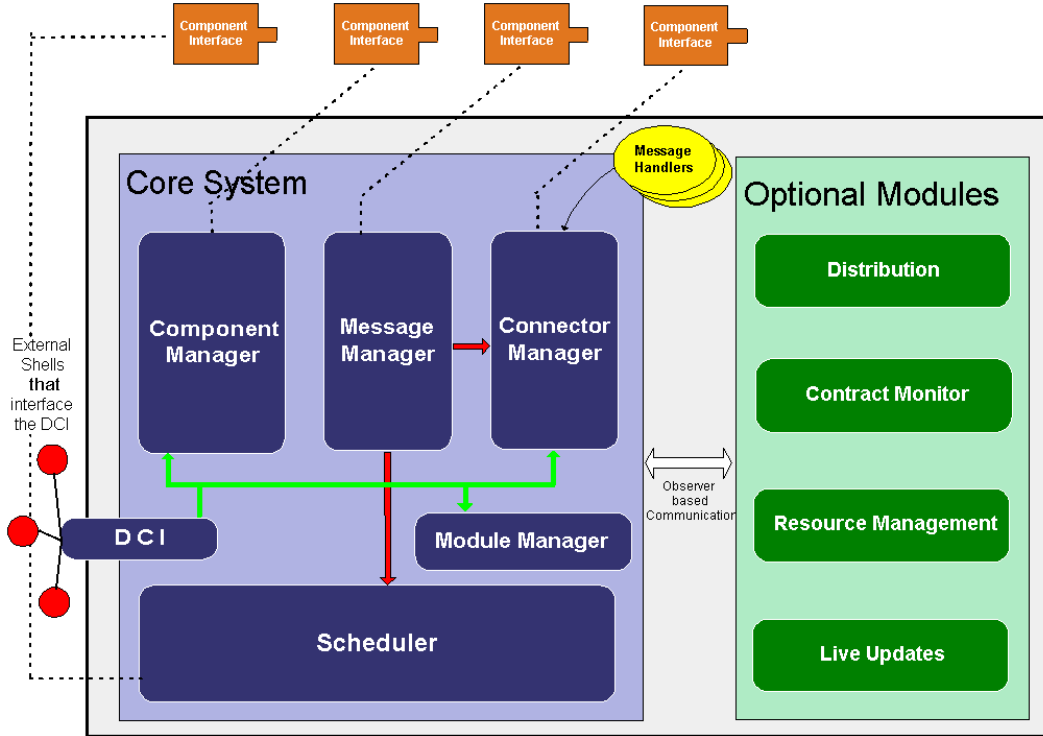


Figure 3: Overview of the DRACO architecture

can be inserted in DRACO are a Distribution module, a Dynamic Updating module, a Contract Monitoring module and a Visualization module. Since the exact tasks and thus requirements of extension modules are unknown in advance, they can influence the message flow at various points (see section 3.2). Also, extension modules can register themselves with the DRACO core modules to receive notification of a number of events.

To allow for application components to interact with the DRACO runtime environment, each of the core modules provides a (limited) component interface. These interfaces are multiports to which component ports can be connected and information can be retrieved. To separate the internal workings of DRACO from application development, this interface is deliberately kept limited. It is used to provide a variety of services to components (e.g. timer functionality).

### 3.2 Journey of a message

In DRACO, messages are sent asynchronously between components. In order to send a message from one component's port to another, several steps are required: the destination port must be determined, the message needs to be scheduled for delivery and finally it must be executed. In some cases (e.g. in a distributed context when messages pass system boundaries) additional steps are required. This path which is followed by a message is called the **message chain**.

It is clear that this chain should be kept short and that the steps constituting the chain should be implemented in an efficient way. However, extension modules should be able to subscribe themselves to

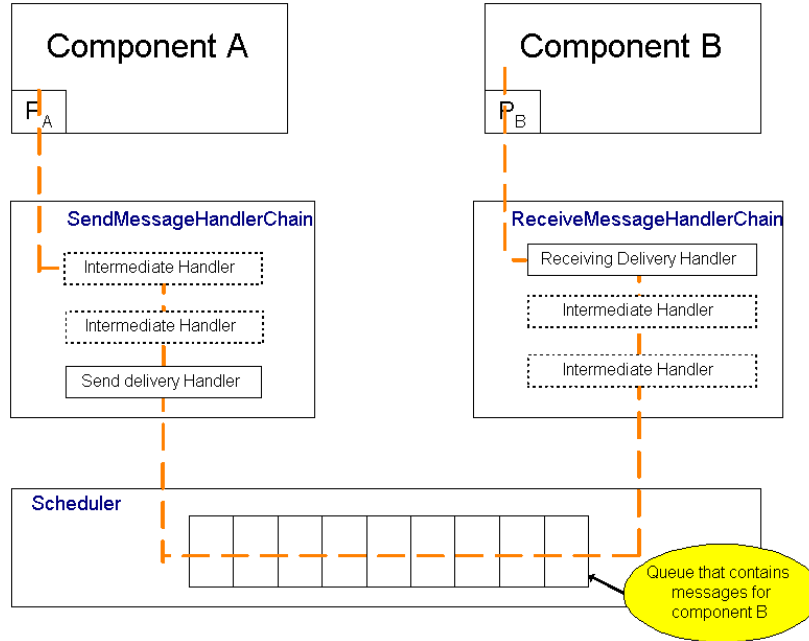


Figure 4: The journey of a message

various events of this message chain (e.g. the delivery of a message). In addition, they should also be able to control this message chain and interact with the delivery of messages in order to implement a variety of features as live updates, distribution, resource management, contract monitoring and others. For example, in order to replace a component at runtime, the dynamic update module will need to freeze the incoming flow of messages to a component in order to shut it down for replacement (see section 5.1). In DRACO, this is realized with message handlers.

The path followed by a message traveling from component *A* to component *B* consists of 3 major parts: the sending message chain, the scheduler and the receiving message chain. A schematic overview is shown in figure 4.

### Sending message chain

The first part of the journey begins when a component sends a message through one of its ports and ends when the message is passed on to the scheduler for delivery. As can be seen on figure 4, the component passes on the message to its port  $P_A$ . The message is then handed over to the message handler associated with the connector attached to the port. In the most simple case, this message handler immediately passes on the message to the scheduler for delivery. However, in more complex scenarios, additional message handlers can be introduced by core or extension modules in order to intercept the sending of messages. For instance, there could be a sending message handler for the Contract Management module to time stamp the messages exchanged between components (see section 5.2). Message handlers are linked to form a queue, such that messages will be passed on from the first to the last handler. The last handler is responsible for delivering the message to the scheduler.

Using the six core modules, this message chain is implemented in 6 steps which are shown in figure

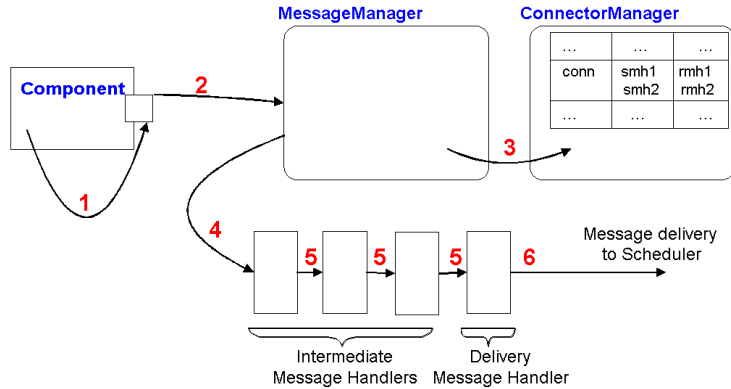


Figure 5: Details of the sending message chain

5:

**Component** → **Port**: When a component needs to send a message, it does so by contacting the port through which the message will be sent. Since ports are implemented as inner classes (see section 4) this is achieved using a local call (see arrow 1).

**Port** → **Message Manager (core)**: The port delivers the message to the message manager (arrow 2), which will retrieve the connector that is associated with the given port from the connector manager (arrow 3). For every connector, the connector manager manages 2 message handlers for each direction (from component *A* to component *B* and vice versa): a sending message handler and a receiving message handler. The receiving message handler will be used for the delivery of the message after it has been scheduled for execution by the scheduler. The sending message handler that is being retrieved is the first handler of the sending chain shown in figure 4. As such, the message manager will then pass on the message (with the receiving message handler) to this sending message handler (arrow 4).

**Message Manager** → **[Send Message Handler]\***: All the message handlers can inspect or modify the messages, and will then pass it on to the next handler in the chain (arrow 5). The last message handler will eventually hand over the message to the scheduler (arrow 6).

## Receiving message chain

As mentioned in the previous section, the scheduler receives two objects: the message to be scheduled and the receiving message handler associated with the current connector. The scheduler will queue the message until it is ready for execution. The number of queues used by the scheduler depends on the number of threads that were specified during the startup of DRACO. In a multi-queued system, it is the responsibility of the scheduler to ensure that the order of messages over a specific connector is preserved. Therefore, messages that are being send over the same connector will always be placed in the same queue.

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 the receiving message handler that is associated with the

<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

Figure 6: Two simple components in SEESCOA notation

message. The principle behind the receiving sequence is identical to the sending sequence: there is a chain of receiving 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.

When the message execution has been completed, control is returned to the scheduler. From this moment the thread that performed the message processing can be reallocated to another message or be suspended, depending on the actual implementation of the scheduler.

## 4 Implementing Components

As shown in the previous sections, the internal structure of DRACO is built for extendibility, and the concept of message handlers may not be the most intuitive to work with. Fortunately, when developing components, no knowledge of this structure is required whatsoever. In this section we illustrate the development of a component. The example we develop here is intentionally kept very simple to focus on specific issues concerning component development for DRACO.

We will work out a small application consisting of two components: a `NumberGenerator` and a `NumberDisplay`. The first component will generate a random number on request and will broadcast it to all interested parties. It has two ports: a `MulticastPort` *Out* through which it will output its generated number and a `PortGroup` *Control* through which he will receive all requests. The second component prints out all values it receives through its *Input* port.

### 4.1 SEESCOA syntax for components

The implementation of these two components is shown in figure 6. The syntax consists of standard Java code, enhanced with a extra keywords. The `component` keyword indicates the start of a component implementation. It consists of a number of variables (e.g. `$rnd` in 6(a)), a number of 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

<pre> package components.numberGenerator;  import draco.core.*; import java.util.Random;  public class NumberGenerator extends Component {      public NumberGenerator(String name)     {         super(name);         register(\$out);         register(\$control);     }      protected MulticastPort \$out =         new MulticastPort("Out", CompositePort.UNLIMITED, this);      protected PortGroup \$control =         new PortGroup("Control", 1, Control.class, this);      public class Control extends SinglePort {          public void SendNumber(Message inmessage)         {             Message x = new Message("GetNumber");             x.putField("value", new Integer(\$rnd.nextInt(100)));             out.sendMessage(x);         }     } } </pre>	<pre> package components.numberdisplay;  import draco.core.*;  public class NumberDisplay extends Component {      public NumberDisplay(String name)     {         super(name);         register(\$input);     }      protected PortGroup \$input =         new PortGroup("input", 1, Input.class, this);      public class Input extends SinglePort {          public void Number(Message inmessage)         {             System.out.println("Received_number:-" +                 (inmessage.getField("value" )));         }     } } </pre>
(a) NumberGenerator	(b) NumberDisplay

Figure 7: The generated Java code

`multicastport` keyword. Two parameters are required: the name of the port and the maximum number of simultaneous connections that are allowed. A port group has a similar declaration, but since it *can* accept messages, these messages must be declared as well using the `message` keyword. Inside the definition of a message, the code to be executed when this message is received on the port must be implemented.

New messages can be created using the statement `message varName = messageName`. After its creation, this message can be sent out through any connected port. If the component on the other side of the connector does not accept `messageName`, the system responds at runtime with a `CannotDeliverMessage`.

In addition, two operators have been added: (`::` and `..`). The operator `::` is used to access fields of a particular message. `messageName::fieldName = "Hello"` will assign the string `Hello` to a field called `fieldName` in the message `messageName`. Retrieval is similar: `String someString = (String) messageName::fieldName` will assign the content of `fieldName` to a newly created `String` variable. 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.

## 4.2 Mapping to Java

As shown in figure 6, the SEESCOA syntax offers a short and powerful notation to express component interactions. Using a preprocessor, this syntax is converted into standard Java. The result after conversion is shown in figure 7. As can be seen in these code snippets, the translation is straightforward. In Java, each port is implemented as an inner class. This has a number of important advantages:

1. Messages are defined for each port. Therefore, DRACO can enforce that messages can only be

sent to ports that accept these messages.

2. The use of inner classes allows for the implementation of messages to use all the variables or internal structures of the component.
3. All ports of a component and all their messages are defined in the same place, giving a good overview of the functionality of the component.

The preprocessor is also responsible for adding the constructor of the component and registering the different ports with DRACO (see figure 7).

## 5 Extending DRACO

To illustrate the power of message handlers, two extensions will be worked out in more detail. In section 5.1, we will describe a *dynamic update* module that will make use of message handlers to freeze a component before it is replaced. In section 5.2, a timing monitor is presented that uses the handlers to insert time probes.

### 5.1 Dynamic Updating of Applications

To safely replace a component at runtime, following steps are typically required:

1. The component is put in an inactive<sup>4</sup> state.
2. The new component blueprint version is instantiated.
3. The internal state of the old version is transferred to the new version.
4. The connectors are rewired to the new component version.
5. The new component is activated.
6. The old component is removed.

Suppose we have the initial situation depicted in figure 4 and we wish to replace component  $B$  with a new version. The user will first instruct the system to load the DU (Dynamic Update) module. This module will then interact with the receiving message chains for all the ports of component  $B$  (we assume only a single port on component  $B$  for brevity). In order to freeze component  $B$ , it will set itself as the receiving delivery message handler for component  $B$  (the DU Module will store the current delivery handler to restore the situation when the update has been completed) and it will send a message **Freeze** to the component. The situation after loading the Dynamic Updating module is shown on figure 8. When the update is initiated, a number of messages may be present in the scheduler. In figure 8, the green messages are intended for one of the ports of  $B$ . Blue messages are intended for other components and will not be seen by the DU Module as they have different message handlers. This reduces unnecessary overhead. The red message is the **Freeze** message sent by the DU Module itself.

---

<sup>4</sup>More accurately, it must achieve quiescence (see [8]). Since the focus of this paper is the use of message handlers in DRACO, an in depth explanation of dynamic update techniques is beyond the scope of this paper.

Simplified Schematic overview of message-delivery situation after the dynamic updating module is loaded and initiates an update for component B

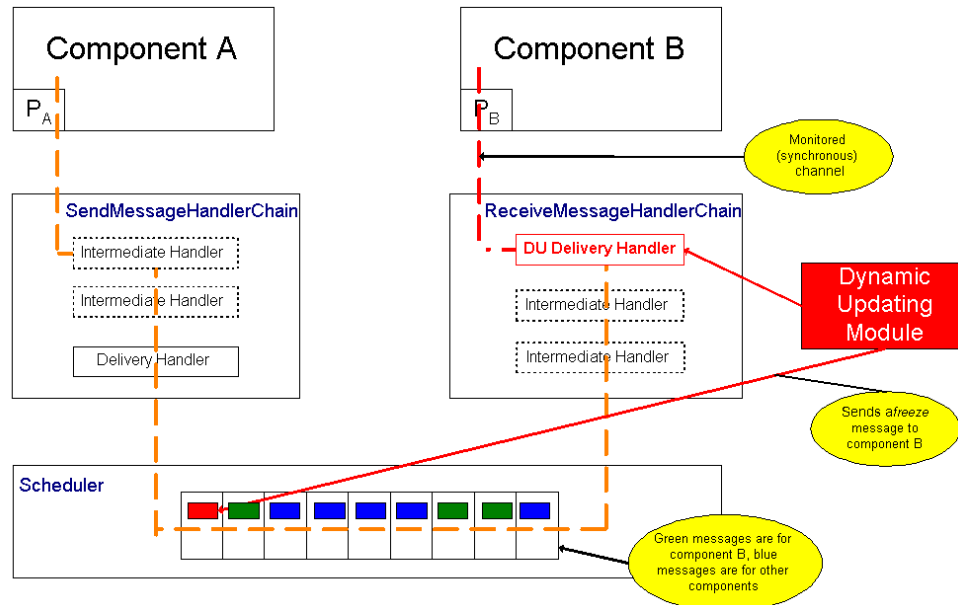


Figure 8: Situation after loading the DU Module for component *B*

The green (and red) messages are intercepted by the DU delivery handler as they pass through the message handler chain. As they are intercepted, the following actions are taken:

1. If the message is not the **Freeze** message that was sent by the DU Module itself, it is just passed on to the component for execution. Afterwards, the control is returned to the scheduler. In this case, the dynamic updating module performs exactly the same functionality as the original default delivery handler.
2. If the message *is* the **Freeze** message that was sent by the dynamic updating module at load time, it will pass on this message to the component for execution. Afterwards, however, it will not return control to the scheduler. Instead, it will first perform an update as described in the following section.

### The update process itself

The steps discussed earlier are then executed:

1. The component is already in inactive state after the **Freeze** message has been executed. Note that if the component does not support the **Freeze** message, a **CanNotDeliverMessage** response is returned to the DU Module. Regardless of whether the component supports the message however, it is indeed in inactive<sup>5</sup> state, since the DU Delivery Handler is blocking all future requests to this component.

<sup>5</sup>If the component is involved in transactions, achieving a quiescent state is more complex. In this case, techniques as those described in [9] can be used.

2. The DU Module will then create an instance of the new component version, by contacting the Component Manager. At the same time, the ports of this component are created as well. A reference to the Component Manager is received through the Module Manager. This creation is done through a direct synchronous call. A reference to the new component version is given to the DU module.
3. The old version of component  $B$  is passed on to the newly created component. At runtime, it is assumed that the new component version contains the necessary code to import the internal state of the previous component version (for information on how this can be achieved we refer interested readers to [10]).
4. The DU Module contacts the Connector Manager and updates the connectors for each of these ports of Component  $B$ .
5. Finally, after a possible but not required transition period (which may prove useful if rollback functionality would be required), in which the DU Module can continue to inspect messages that are sent to component  $B$ , the DU Module requests that the component manager removes the previous component version and its ports from the system to free resources. It then concludes by removing itself from the system by reinstating the DeliveryHandler it replaced.

## 5.2 Monitoring Timing Constraints

In this section, a second example that illustrates the power of message handlers is worked out: an extension module to monitor timing contracts.

The contract concept enables the specification and verification of non-functional properties. Contracts can be attached to components, their ports or the connectors between them. Key idea is that a designer can select from a set of predefined contract templates and attach these to a component model in order to impose a particular non-functional constraint. Once a contract has been added to a model, it can be verified by means of a contract verification algorithm that is part of an underlying monitoring system. This monitoring system will be implemented in DRACO as an optional module, namely the Contract Monitor module.

One particular contract type is the timing contract; it lets the designer specify timing constraints imposed on the interactions among components. Timing contracts are attached to connectors since the connector is the construct that represents such an interaction.

### Timing Contract Concepts

Timing contracts are based on two important concepts, namely *hooks* and *hook occurrences*. Both refer to a particular point of interest on the extended MSC<sup>6</sup> associated to a port or connector. A hook refers to an action occurring in an MSC and exists in three types: send hooks, receive hooks and eoa (end-of-activation) hooks. A send hook corresponds to the sending of a message, a receive hook to the receiving of a message and an end-of-activation hook refers to the end of processing of a message. Since an extended MSC can contain loop blocks, alternative blocks and optional blocks, one also needs a

---

<sup>6</sup>An MSC (Message Sequence Chart) represents the interactions among communicating entities. In SEESCOA *extended* MSC's are used to describe the communication protocol of ports and connectors. An extended MSC also provides support for modeling loops and alternative/optional branches.

way to specify the occurrence number of a particular hook. In this paper we will introduce a simplified notation for both concepts: `[ComponentName].[PortName].[MessageName].[HookType]` represents a hook, whereas `[ComponentName].[PortName].[MessageName].[HookType].[OccurrenceNumber]` represents the *OccurrenceNumber*<sup>th</sup> occurrence of the hook. *OccurrenceNumber* is also allowed to be ALL, which refers to all occurrences of that particular hook. More information on this notation and its semantics can be found in [11].

### Timing Contracts: DeadlineTC en PeriodicityTC

Two subtypes of a timing contract have been worked out: a deadline timing contract and a periodicity timing contract. A deadline contract imposes a deadline on the occurrence of two hooks, while a periodicity contract states that a particular hook has to occur periodically (each occurrence in its associated period). Both timing contract types have their corresponding runtime verification algorithms.

### Event Gathering and Contract Monitoring

In order to monitor contracts, and more in particular timing contracts, a distinction has been made between the gathering of events and the monitoring of contracts pertaining to them. In the case of timing contracts, one is particularly interested in hook occurrence events. Since hook occurrences correspond to message interactions, the monitoring system needs to intercept the messages exchanged among the components. Once intercepted, every event needs to be timestamped and then this information is sent to the contract monitoring subsystem.

**Event gathering:** Intercepting hook occurrence events is done through the insertion of specific message handlers (time probes) in the send and receive message chains associated to connected ports. Each time a particular message is delivered into the chain, the time probe will intercept the message. It then extracts information that uniquely identifies the event: the component and port that sent the message, the message name and the status of the message (send, receive or eoa). Finally, the time probe samples the system time and attaches it to the event information. The probe then puts this information in a non-blocking data structure that is read out by the contract monitoring subsystem.

It is important to note that the use of time probes also influences the timing behaviour of the application. Therefore, this probe needs to perform its computations in constant time and in a non-blocking fashion.

**Contract monitoring:** The contract monitoring system is responsible for collecting events coming from the various probes. These events are then sent to the contract verification algorithms that are registered within the monitoring system. For each type of contract there is a corresponding contract verification algorithm. Currently, two such algorithms have been developed, one for each type of timing contract.

Contract violations are logged in a file, which can be analyzed after the execution of an application. In a future release of the Contract Monitor module, we intend to include online violation feedback functionality. This will enable a DRACO application to reflect and adapt to non-functional constraint violations of its constituting components.

## Contract Monitor Module

The Contract Monitor module responsible for monitoring a DRACO application. It contains the necessary functionality for event gathering and contract monitoring, and has to be loaded before the startup of a DRACO application. Initially, the Monitoring module reads in a *probe file* and a *contract file*.

The probe file contains [ComponentName]. [PortName]. [MessageName]. [HookType]. [OccurrenceNumber] entries indicating which events are to be monitored. The contract file contains information about deadline and periodicity contracts that have to be monitored, including references to the involved probes.

The Monitor module uses several reflection and interception points, that are built into the DRACO component system, in order to perform its activities. It registers itself with the Connector Manager since it has to be notified when a connector is created to which a contract has been attached. If such a connector has been created, the Monitoring module will insert the necessary time probes in the message chains associated to the connected ports. Additionally, the Monitoring module notifies its monitoring subsystem that a new contract needs to be monitored.

## 6 Related Work

Other interesting approaches on the use of reusable components in embedded systems can be found in [12–14]. Although they have some additional application-domain specific features, each of these systems is based on the wiring of independent components. In PECOS, focus is on the use of components in heavily resource constrained devices. Components communicate through ports that are implemented as shared variables. The PECOS component model ([12]) is strongly formalized (including the behaviour of a component), and is intended for use in hard real-time systems. Both in DESS ([13]) as in Koala (developed by the EPSRC and Philips to enable the reuse of components in consumer electronics: [14]) components have provided and required interfaces. DESS interfaces are denoted by means of the lollipop notation without a specific port concept whereas koala components specify their interfaces using an Interface Definition Language. Both DESS and Koala allow components to be decomposed into connected subcomponents thus supporting the construction of hierarchical components.

With *Conus*, Kramer and Magee pioneered in the field of component-oriented evolution. In [8] they introduced the concept of quiescent nodes for safe removal of a component from the running system. Using provisions available in the Chorus Operating system, Hauptmann and Wasel achieve deterministic timing behaviour during updates ([15]). Just like the SEESCOA component system, it uses the concept of *ports* to reroute messages between objects. A very flexible approach can be found in meta-architectures (e.g. [16,17]). Through reification of object-oriented concepts (class, method-call, ...), a meta-model is built on top of the application. By changing this meta-model, structural changes of the application are possible. Other systems focus specifically on dynamic Java. In [18] the default Java class loader is extended so that class definitions can be replaced and objects or dependent classes can be updated. More theoretical work on dynamic updating is done by Gupta ([19]) and Hicks ([20]). A complete survey of this domain is beyond the scope of this paper (we refer to [19–22] for more complete overviews).

A lot of research has already been done on the specification and *static* verification of timing constraints (examples are Real-Time Logic [23] and timed MSC's [24]). Less efforts have been done till now concerning the *dynamic* verification of constraints. One particularly interesting approach is the

one used by [25] and [26], in which RTL formulas are monitored at runtime by a generic verification algorithm. RTL is a powerful formalism, but not all RTL formulas can be monitored efficiently at runtime. Our approach is not based on a formalism like RTL; instead a template-based approach is used in combination with efficient contract-specific verification algorithms.

## 7 Future work

Although the DRACO runtime system is fully implemented, the preprocessor is not. Implementation of this preprocessor will receive high priority in the near future, since development of components is dramatically simplified using the SEESCOA constructs. Having a fully functional preprocessor at our disposal also allows us to port more complex existing component oriented software (e.g. the camera surveillance system [6]) to DRACO with minimal effort. These more complex cases will then allow us to further evaluate the performance of DRACO in real life situations.

Next to the continuing work on the runtime environment itself, DRACO will particularly be used as a base platform for further research on component oriented development for embedded systems. Issues that will be investigated are state transfer during a live update of a component, performance and resource monitoring and component mobility. As such, many extension modules (among which the two proposed modules, but also a distribution module for instance) will be implemented. Additional complications may need to be resolved when different extension modules are being used at the same time (e.g. updating a component in the distributed case, ...).

## 8 Conclusion

In this paper we have introduced DRACO, a modular component runtime environment intended for embedded devices which adheres to the SEESCOA component methodology. First, a detailed overview of the DRACO architecture was given and the 6 core modules were discussed in detail. In order to preserve its small footprint, DRACO can be extended with external modules that implement additional features as distribution, runtime contract monitoring or dynamic updating. These extension modules can influence with the message delivery process using message handlers. The power of this technique has been illustrated with two example modules that have been worked out in detail. In addition, a trivial component was developed to illustrate the SEESCOA component constructs and a mapping from the SEESCOA language to Java was worked out.

## References

- [1] M.D. McIlroy. Mass produced software components. In *Software Engineering*, pages 138–150, January 1969.
- [2] Jari Suutarinen. Java in mobile world. Sun Developers Connection - J2ME & Wireless Java, February 2002.

- [3] David Urting, Stefan Van Baelen, Tom Holvoet, and Yolande Berbers. Embedded software development: Components and contracts. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 685–690, 2001.
- [4] David Urting, Stefan Van Baelen, Tom Holvoet, Peter Rigole, Yves Vandewoude, and Yolande Berbers. A tool for component based design of embedded software. In *Proceedings of Tools Pacific 2002*, February 2002.
- [5] Werner Van Belle. Refinement of the component architecture. SEESCOA Deliverable, October 2001.
- [6] Peter Rigole, Yolande Berbers, and Tom Holvoet. Bluetooth enabled interaction in a distributed camera surveillance system. In *Submitted to: Wireless Personal Area Networks Minitrack at HICSS 2004*.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
- [8] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
- [9] Nico Janssens, Sam Michiels, Tom Mahieu, and Pierre Verbaeten. In *Second International Workshop on Unanticipated Software Evolution*, pages 9–16, Warshau, Poland, 2003.
- [10] Yves Vandewoude and Yolande Berbers. Meta model driven state transfer in component oriented systems. In G editor, *Proceedings of The Second International Workshop On Unanticipated Software Evolution*, pages 3–8, April 2003.
- [11] David Urting and Yolande Berbers. Runtime verification of timing constraints. Technical Report CW345, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, July 2002.
- [12] O. Nierstrasz, G. Arévalo, and G. Ducasse. A component model for field devices. In *"IFIP/ACM working conference on Component Deployment"*, Berlin, Germany, 2002.
- [13] Definition of components and notation for components. ITEA-DESS public deliverable D1.4.4, 2001. <http://www.dess-itea.org>.
- [14] R. van Ommering, F. van der Linden, and J. Kramer. The koala component model for consumer electronics software. *IEEE Computer*, 2000.
- [15] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3th International Conference of Configurable Distributed Systems*. IEEE Computer Society Press, 1996.
- [16] Jim Dowling and Vinny Cahill. Dynamic software evolution and the k-component model. In *Workshop on Software Evolution, OOPSLA*, 2001.
- [17] Barry Redmond and Vinny Cahill. Iguana/j: Towards a dynamic and efficient reflective architecture for java. In *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, Cannes, France, June 2000.

- [18] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, June 2000.
- [19] Deepak Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, IIT, Kanpur, November 1994.
- [20] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, June 2001.
- [21] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for a dynamic updating. *IEEE Software*, 10(2):53–65, 1993.
- [22] Yves Vandewoude and Yolande Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proceedings of The International Conference on Software Engineering Research and Practice*, Las Vegas, USA, June 2002.
- [23] F. Jahanian, A.K. Mok, and D.A Stuart. Formal specification of real-time systems. Technical Report UTCS-TR-88-25, Department of Computer Sciences, the University of Texas at Austin, USA, 1988.
- [24] H. Ben-Abdallah and S. Leue. Analyzing timing constraints in message sequence chart specifications. Technical Report 97-04, Department of Electrical and Computer engineering, University of Waterloo, Canada, 1997.
- [25] Sitaram C.V. Raju, Rangunathan Rajkumar, and Farnam Jahanian. Monitoring timing constraints in distributed real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 57–67, 1992.
- [26] Aloysius K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 252–262, 1997.