# Enforceable Component-Based Realtime Contracts

–

# Supporting Realtime Properties from Software Development to Execution

Hermann Härtig[†] Steffen Zschaler[†] Martin Pohlack[†]
Ronald Aigner[†] Steffen Göbel[‡] Christoph Pohl[‡]
Simone Röttger[†]

[†]Technische Universität Dresden, Department of Computer Science
[‡]SAP AG, SAP Research CEC Dresden

**Abstract**

We present *enforceable component-based realtime contracts*, the first extension of component-based software engineering technology that comprehensively supports adaptive realtime systems from specification all the way to the running system.

To provide this support, we have extended component-based interface definition languages (IDLs) and component representations in repositories to express realtime requirements for components. The final software, which is assembled from the components, is then executed on a realtime operating system (RTOS) with the help of a component runtime system. RTOS resource managers and the IDL-extensions are based on the same mathematical foundation. Thus, the component runtime system can use information expressed in a component-oriented manner in the extended IDL to derive parameters for the task-based admission and scheduling in the RTOS. Once basic realtime properties can thus be guaranteed, runtime support can be extended to more elaborate schemes that also support adaptive applications (*container-managed quality assurance*).

We claim that this study convincingly demonstrates how component-based software engineering can be extended to build systems with non-functional requirements.

## 1 Introduction

The complexity of modern software systems is continuously increasing. For quite some time already, component-based software engineering (CBSE) (McIlroy 1968; Szyperski 2002) has been considered to be a good way to cope with this increasing complexity by dividing software systems into manageable parts which can be developed largely independently and reused many times in different application contexts. However, so

far CBSE has only provided a means to cope with the *functional* complexity of software systems. Component-based support for non-functional properties of such systems is still very much lacking. For example, realtime properties such as response time or jitter are typically guaranteed by realtime operating system (RTOS) schedulers, which use a task-based terminology and mindset not directly usable for a component-based application. This paper presents *enforceable component-based realtime contracts*, a detailed approach extending specification languages, repositories and the runtime environment to support component-based applications that have to meet realtime requirements. Continuous support of realtime properties from specification to execution is our main contribution in this paper. We have described the individual elements of our approach to a large extent in other publications (Aigner et al. 2003; Göbel 2004; Göbel et al. 2004; Göbel et al. 2004; Hamann et al. 2001; Härtig et al. 1998; Röttger and Zschaler 2003; Röttger and Zschaler 2004). In this paper we focus on their combination and interaction, which lead to continuous support of realtime properties from system inception to system runtime. To the best of our knowledge, this is the first comprehensive solution providing support for component-based systems from design to execution.

In order to raise support for realtime properties to the level of component-based applications, various sub-problems need to be solved, and the provided solutions must be designed so as to interact closely. These issues and our respective solutions will be discussed in this paper, but we first will outline the requirements we derived from the problem domain. The following list is meant to provide guidance regarding the way the individual elements of our approach fit together and complement each other in fulfilling these requirements:

1. We require a component-oriented development process, which is tailored both for the specific issues related to CBSE and the explicit consideration of non-functional properties of the system to be developed.

2. During development of a component-based application following our process, application developers need to specify non-functional properties of their application. They then need to select components based on their non-functional properties such that the non-functional properties of the components contribute appropriately and positively to the required non-functional properties of the application as a whole. Component developers providing component implementations and wishing to sell them to application developers need to describe the non-functional properties of their component implementations. We thus need a component-oriented specification technique that allows the expression of non-functional properties of individual components or applications as a whole.

3. We require an RTOS as the technological basis for guaranteeing realtime properties. The RTOS needs an underlying mathematical model upon which the derivation of realtime properties of applications can be based.

4. The component-oriented specification of non-functional properties must be mapped into the task-based parameters expected by the RTOS's schedulers. Component runtime environments, also called containers, can provide an implementation of such a mapping. The mapping is simplified by basing the component-oriented specification language on the same mathematical model as the RTOS schedulers.

The container can use arbitrarily complex algorithms for mapping component resource requirements to RTOS resource reservations—for example by placing buffers between components or instantiating multiple copies of the same component to improve throughput. What decisions the container takes is hidden from the component developer and the client of the system, giving service providers more control and flexibility when managing the non-functional properties of the services they provide. We refer to this as *container-managed quality assurance*. Another example of this is that the container can adapt non-functional properties of components and applications already running, if resource availability or client requirements change.

The remainder of this paper is structured as follows: Section 2 presents a small video-player application, which we are going to use as an example throughout the paper. We then begin by reviewing briefly some core concepts of CBSE in Sect. 3 as a basis for the discussion in the following sections. As we have indicated above, it is useful to use the same mathematical foundation for both the component-based specification technique and the RTOS schedulers. Therefore, we introduce our underlying mathematical foundation in Sect. 4. Section 5 then discusses the component-oriented specification of non-functional properties. Additionally, this section outlines a development process for component-based systems with support for non-functional properties. The component runtime environment that realises the mapping from the component-oriented specification to the task-oriented mindset of the RTOS is presented in Sect. 6. Section 7 discusses advanced techniques of container-managed quality assurance, which support quality adaptation at runtime. The paper concludes with a review of related work and an outlook.

# 2 An Example System

Multimedia applications are good examples for systems with realtime requirements. We will therefore use a video player software as a running example in this paper. The player should in particular be able to display fluid video (i.e., at least 25 frames per second), accompanied by the corresponding audio samples.

The functionality of a video player can be divided naturally into the following five parts (cf. Fig. 1[1]):

1. A *demultiplexer* (dubbed demuxer in the figure), which analyses video file data and provides separated video and audio streams.

2. An *audio decoder*, which transforms compressed audio frames into suitable data for a sound card.

3. A *video decoder*, which transforms encoded video frames into the display format. Video is processed in two steps: First, the encoded data is decoded, and then a set of post-processing filters are applied, removing compression and motion artefacts, adapting contrast and brightness, or even watermarking the decoded frames.

[1] In this figure, we already use parts of an extension of UML we have developed to model component-based systems and their non-functional properties. Some more details of the notation will be explained in Sect. 3.2; for now just consider the white boxes to indicate components, the double lines connecting them to represent stream-based communication, and the single lines to stand for request–response communication.

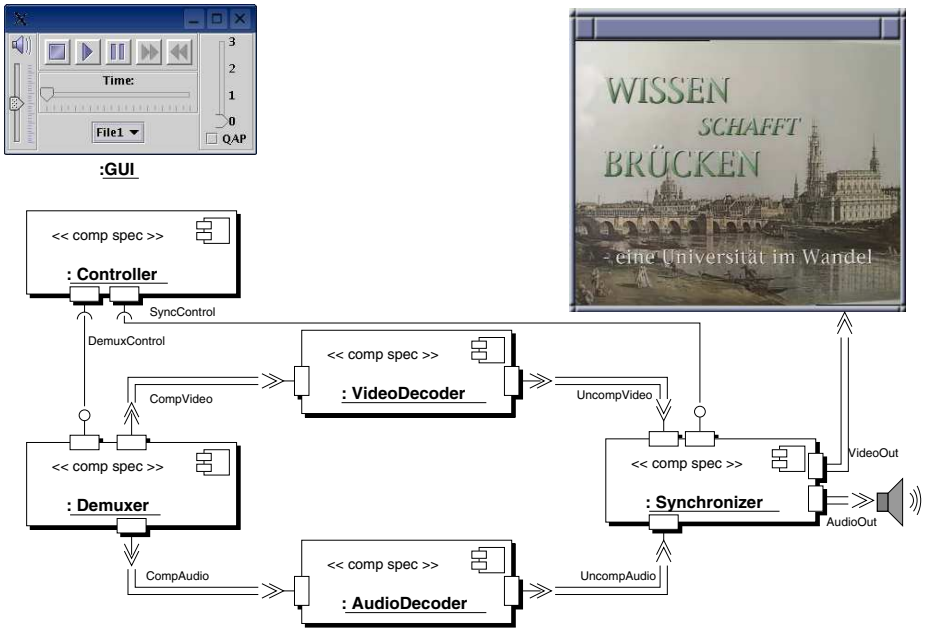


Figure 1: Overview of the sample system

4. A *synchronizer*, which synchronises the audio and video output. The synchronizer delivers the video and audio streams to appropriate rendering devices (display and sound card, respectively).

5. Users control playback via a *graphical user interface (GUI)*, which sends commands, such as `start,` or `stop,` via the controller to the demultiplexer and synchroniser in response to user actions.

If we consider the realisation of such a video player, some technical requirements become obvious. We will focus on the two most prominent ones in this paper:

Firstly, to enable reuse of existing implementations of video and audio decoders, we would like to structure our application into interacting, independently exchangeable components. The five parts identified above are a natural starting point for such a component structure. These components need to communicate in two ways: (i) they exchange control information on a request–response basis, but (ii) they also exchange streams of video and audio data. Streams are different from request–response communication, because they are packet-based and a direct response is not required.

Secondly, the video player has to guarantee the realtime properties of playback. This requires reservation and scheduling of resources, in particular the CPU. The protocols supporting such reservations are platform-specific, and resource reservation can be a complex issue in its own right. Thus, integrating resource reservation code into the components themselves prevents reuse on different platforms, and additionally makes component development even more complex. Therefore, we would like resource reservation and realtime guarantees to be managed as much as possible by the runtime environment. We refer to this as *container-managed quality assurance*. Of course, we need to give some information about the realtime behaviour of the components, but, in order

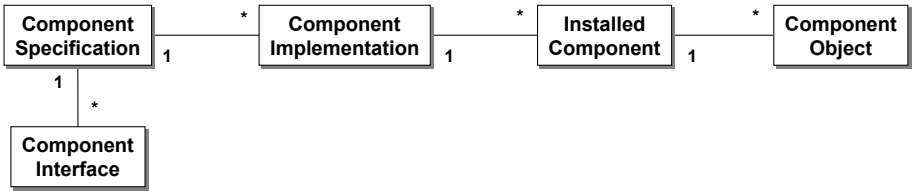


Figure 2: Component forms identified in (Cheesman and Daniels 2001)

to separate this concern as much as possible from the core business logic, we want this to be in a declarative form clearly separated from the component implementation. Specifically, in our example, we need to know the distribution function of execution times for each decoding, demultiplexing, or synchronisation step. From this information, we want the runtime environment to derive a CPU allocation scheme, which uses the resource as efficiently as possible, including potential overbooking. The system will then not be able to give absolute guarantees for all cases. Therefore, it should provide ways to adapt to missing resources dynamically, so that a certain quality level can be maintained.

The following explains how we structure our runtime system, and how we declaratively describe the realtime behaviour of our application to support these requirements. We will come back to the video player to give concrete examples of our technologies.

# 3 Component Concepts

Software components are seen as an important way to structure applications, helping application developers cope with the complexity inherent in large systems. In this section we give a short overview of the important concepts in this area.

## 3.1 Introduction to CBSE Concepts

The definition of a *component* most widely accepted in the CBSE community is

> "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." (Szyperski 2002)

Software components are elements of software that need to be composed with other components to form an application. The different components in an application may be developed by different component developers. The components can interact because there exists a standardised component runtime environment offering a space to live in to the components.

(Cheesman and Daniels 2001) introduce another important notion into the world of CBSE, namely the idea of *component forms*. They argue that the concept of a component varies depending on the current step in the software project life cycle. Cheesman and Daniels distinguish the following four major component forms (cf. Fig. 2):

**Component Specification** The specification of the behaviour of a unit of software without reference to implementation decisions. Cheesman and Daniels use component interfaces and interface models to define component behaviour.

**Component Implementation** An actual implementation of a component specification. There can be various implementations for the same component specification as long as they exhibit the same externally observable behaviour.

**Installed Component** Component implementations are installed into a runtime environment where they will be executed. Although Cheesman and Daniels do not mention it, at this level it would be possible to further distinguish different configurations of the same component implementation, even installed into the same runtime environment.

**Component Object** Installed components are instantiated to create component objects, which are the entities actually handling any requests. The important thing here is that component objects can have a runtime state, so that two component objects of the same installed component can be distinguished by their respective runtime state.

The view taken by Cheesman and Daniels is an extension of the definition given by Szyperski. Szyperski's components are a mixture of the first three component forms (i.e., component specification to installed component), exhibiting properties of each. Szyperski explicitly states that components have no persistent state. Cheesman and Daniels clarify this concept by introducing the Component Object form, which represents an instantiated component having *runtime* state.

In this paper, we follow the definitions and concepts explained above, and add consideration of non-functional properties from design to runtime. We associate non-functional properties with component implementations. However, as will be seen, functional specifications of applications are completely written at the level of component specifications. The runtime environment then selects appropriate implementations to instantiate based on the non-functional properties to be guaranteed. The next section shows how our example fits into this terminology.

## 3.2 Revisiting the Example

In our example, we can identify five components: controller, demultiplexer, video decoder, audio decoder, and synchronizer. They have been composed as shown in Fig. 1 to form an application which provides the service of rendering videos.

We will identify the various component forms using the video decoder as an example. First, there is the video decoder component specification, which expresses that a video decoder component has two streaming ports `compVideo` and `uncompVideo`. The former is a stream sink (indicated by ≫-)—a port through which data packets flow into the component—and the latter is a stream source (indicated by -≫)—a port through which data packets flow out of the component. The functionality of the video decoder, also described in its component specification, is that it takes streams of compressed video material in some form (MPEG4 in our case), and decodes it into a stream of raw video data fit to be presented on screen.

There can be multiple implementations of such a specification. We consider two implementations: the Verner video decoder (Rietzschel 2003), which supports adaptation of post-processing steps, and a standard MPEG 4 decoder without support for

such adaptation. These component implementations provide different realtime properties: Verner can consistently provide 25 frames per second of output, but it may drop post-processing steps. The standard decoder will not drop post-processing steps, but this means it may not achieve an output of 25 frames per second.

Providing these component implementations to a component container makes them installed components. Some additional configuration settings happen in this step, the most notable among them is the setting of a name under which the component can be located in the component container's name server. When a user wants to view a video, the component container will select appropriate implementations from the installed components available, instantiate them, and connect them according to the specification of the application. The main distinction between an installed video decoder and an instantiated video decoder is that the instantiated decoder processes a specific video for a specific client. At any point in time, the instantiated decoder processes a specific frame of the video, which may be different from the frame decoded by other instances processing the same video for a different client.

# 4 The Mathematical Model: Schedulers for Tasks with Mandatory and Optional Parts

In this section we discuss the mathematical model that forms the foundation of enforceable component-based realtime contracts. We first explain our task model, followed by the presentation of a simple scheduling scheme for this task model.

## 4.1 The Task Model

The following concepts are based on the theory of *Imprecise Computation* (Chung, Liu, and Lin 1990) applied to CPU scheduling. Given a suitable application or algorithm, such as a video player, it is possible to divide work into a single mandatory part—the decoding of a picture—and one or more optional parts—the post-processing of the picture.

This is reflected in the task model we use in our system. We split resource demands into two parts: a mandatory part and an optional part. The mandatory part must always be completed by its deadline. Consequently, it must be scheduled according to its worst case execution time. At least a certain requested percentage of the optional parts should be completed before their respective deadlines. Our task model $T$ comprises a set of independent tasks:

$$T \quad = \quad \{T_1, \ldots, T_n\},$$

where each task $T_i$ is defined as follows:

$$T_i \quad = \quad (X_i, Y_i, w_i, q_i, d_i),$$

such that:

$X_i$ is a random variable for the execution times per period of the mandatory part,

$Y_i$ is a random variable for the execution times per period of the optional part,

$w_i$ is the worst-case execution time of the mandatory part,

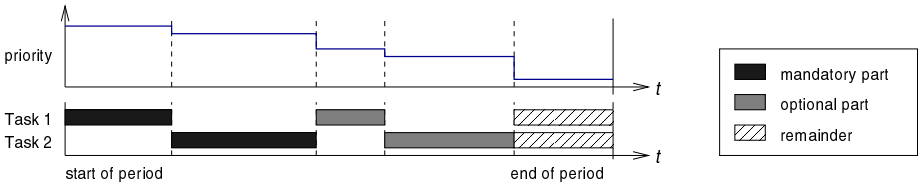


Figure 3: Working principle of schedulers with mandatory and optional jobs for each task

$q_i$ is the quality of the optional part. Quality is defined as percentage of parts completed before the deadline.

$d_i$ is the relative deadline. For simplicity, we use deadlines which are equal to a task's period.

The use of distributions for execution times combined with splitting the work into the two described types, allows us to achieve much higher resource utilization for (i) systems with hardware components classically not suited for realtime tasks due to varying execution times (e.g. hard disks), and (ii) software problems with varying amounts of work in each iteration—for example, video decoding. Based on this task model $T$, we describe one possible admission and scheduling model in the following.

## 4.2 Admission and Scheduling

Quality Rate-Monotonic Scheduling (QRMS) is a method to compute reservation times. Information about QRMS has not been published so far, however, we have previously introduced the more complex Quality-Assuring Scheduling (QaS) (Hamann et al. 2001), a predecessor to QRMS, and applied it to disk request scheduling in (Reuther and Pohlack 2003) and (Reuther 2005).

For QRMS we compute the reservation time $r_i$ for the whole task $T_i$ as follows:

$$r_i{}' = \min(r \in \mathbb{R} | \mathcal{P}(X_i + Y_i \leq r) \geq q_i) \quad (1)$$

$$r_i = \max(r_i{}', w_i) \quad (2)$$

In Equation 1 we determine a reservation for both, the mandatory and the optional parts, such that the required quality $q_i$ is met, by simply adding the distributions. With Equation 2 we ensure that there is enough reservation for the mandatory part by potentially increasing the reservation to the worst case execution time of the mandatory part.

Since mandatory parts do not usually run for their worst case execution time, we use the distributions of both parts for the computation of the execution times. These times are mapped to time slices with certain lengths and priorities for our CPU scheduler. An example schedule for two tasks with mandatory and optional parts and corresponding global priorities is depicted in Fig. 3.

The admission test for QRMS is relatively simple; the only requirement is that the utilization summed up for all tasks in the system must be less than 1:

$$\sum_{i=1}^{n} \frac{r_i}{d_i} \leq 1 \quad (3)$$

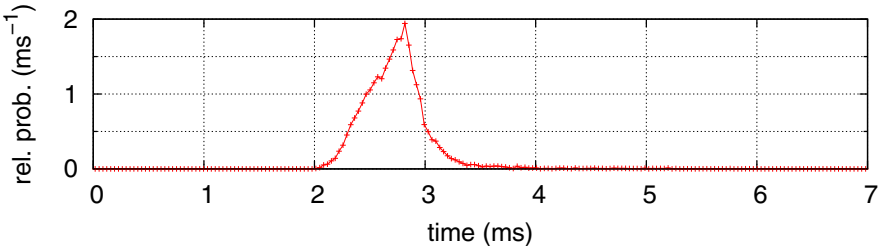


Figure 4: Distribution of video decoding times

### 4.3 Revisiting the Example

In this section we explain how the example system detailed in Sect. 2 fits together with our scheduling technology as described above.

An overview of the example application is given in Fig. 1. All component activities are realized as periodic and independent processes. The audio decoder component fits easily into our task model. It continuously runs a loop, which fetches new packets from the demuxer component, decodes them, and finally delivers the resulting raw samples to the synchronizer component. Computation time requirements for audio decoding (typically MPEG-1 Layer 3 or similar formats) are mediocre for current desktop systems. Furthermore, computation time for decoding a single frame of a given data format is nearly constant as frames have similar sizes. Thus, we modelled the audio decoding part as a simple periodic process, with fixed period and deadline. All the computation is done in the mandatory part. The distribution of the execution times for this component mostly consists of a single sharp spike.

In contrast to audio frames, video frames vary in size and complexity. Decoding video frames is, therefore, harder to accomplish in realtime systems than decoding audio frames. As mentioned previously, we implemented frame decoding as mandatory parts and post-processing as optional parts; that is, all frames are always decoded, but some frames are not post-processed. Video decoding times fluctuate as shown in Fig. 4, whereas simple post-processing algorithms run in approximately constant time since they always have to process the same amount of data (one decoded frame). Post-processing filters remove compression and motion artefacts, adapt contrast and brightness, or even watermark decoded frames, and are typical for current video codecs such as MPEG 4. Post-processing has a CPU-time demand in the range of the frame decoding time itself.

Returning to the task model introduced in Sect. 4.1 our video player would consist of four realtime tasks $T_1$ to $T_4$ (demultiplexer, audio decoder, video decoder, synchronizer), where $T_3$ would describe the video decoder component. The distribution of a typical execution time $X_3$ for the mandatory decoding step is shown in Fig. 4. The distribution for $Y_3$ would be more narrow but in the same order of magnitude for the times, as post-processing is more steady in CPU usage than decoding. The worst case execution times $w_3$ were around 10 ms on one of our machines. We used a quality $q_3$ of 95% for the optional parts. The relative deadline and period length $d_3$ would be 40 ms for a video with 25 frames per second.

Psychologically, it is not desirable to have a flickering video where post-processing

is skipped occasionally. Instead, it is preferable to adjust the quality of the displayed video slowly and in small steps. Therefore, we defined several setups of post-processing filters, each setup differing in resource demand. Of course, it is only useful to decrease image quality in a setup if the corresponding resource demand is decreased as well. We call these setups post-processing *levels*. Currently, we use four different levels, where level 0 means *no post-processing* and level 3 means *very high quality post-processing*. In the running system, the current CPU demand is monitored. In case of resource availability changes the video decoder is notified and it adapts post-processing levels.

# 5 Specifying Enforceable Realtime Contracts of Component-Based Software

After we have presented the mathematical foundations of our approach, we now explain important design-time issues for component-based software with enforceable realtime contracts. We discuss our method of specifying both the application and its realtime contracts, and present an overview of a development process for such software. We begin by explaining the specification of our sample system.

## 5.1 Revisiting the Example

In this section we explain the specification concepts for our example system. First, we discuss how non-functional properties of a component implementation are specified, using the Verner video decoder as an example. Afterwards, we show our method of specifying the functional application structure; that is, how the components interact to provide the application's functionality.

On the non-functional side, the key characteristic we are interested in for the video decoder is the frame rate that it will deliver. To this end, we first need to define what we mean by frame rate. Listing 1 shows a CQML$^+$ (Extended Component Quality

```
quality_characteristic frame_rate (f: Flow) {
  domain: numeric real [0..) frames_per_second;

  values: f.events.eventsInRange (1000);
}
```

Listing 1: CQML$^+$ definition of frame rate.

Modelling Language (Röttger and Zschaler 2003)) specification defining the quality characteristic (or measurement) `frame_rate` for a flow `f` of frames. The domain clause tells us that frame rates are expressed as non-negative real numbers with a (informatory only) unit of measurement of frames per second. The `values` clause tells us that frame rate is defined, at any moment in time, as the number of events registered for the flow `f` in the last 1000 milliseconds.

Actually, in order to interpret the `values` clause, there are two things that one needs to know: First, the `values` clause is written in a language called the Object Constraint Language (OCL) (Object Management Group 2003b) a textual sub-language of the Unified Modelling Language (UML) which allows designers to add first-order predicate logic constraints to UML models. OCL expressions are always written with respect to some UML model—most importantly a static structure model

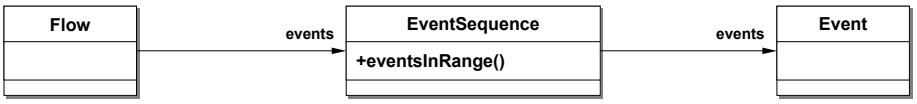


Figure 5: Context model for frame rate definition

(e.g., a class diagram)—called the *context* of the expression. Therefore, we need to know, secondly, the static structure model forming the context of the `values` clause. In CQML$^+$, each characteristic is defined relative to a context model, which defines the system elements relevant for the characteristic, as well as their interactions. This context model also provides the context for the `values` clause. A representation of the static structure of the context model for the definition of frame rate can be seen in Fig. 5. There are also some definitions of the dynamic behaviour in the context model. They state that, for each frame that is put into the flow, an event is placed in the associated event queue, marked with the time at which the frame was put into the flow. The operation `eventsInRange (x)` is defined to return the number of events in the last `x` milliseconds.

So far, our specifications have been independent of a specific application. Now, we begin to model our application. As a first step, we define (in Listing 2) a frame rate of 25 frames per second or more to be good. And, last but not least, we spec-

```
quality good_rate (f: Flow) {
  frame_rate (f) >= 25;
}
```

Listing 2: CQML$^+$ definition of good frame rate.

ify that the Verner video decoder can provide such a frame rate. In order to apply a characteristic to a concrete application model, we need to define a mapping from the concrete modelling elements to the elements of the context model. We define that video stream connections can be viewed as flows in the context model. Thus we can apply `good_rate` to any video stream connection in our model. As explained in Sect. 2, Verner is implemented so that it can decode videos in two modes: one with an additional post-processing step, and one without this step. In both cases, Verner will output 25 frames per second, provided it received at least 25 frames per second as input. However, its CPU demand differs for the two cases. Listing 3 gives the corresponding CQML$^+$. The resource demand is specified in two steps. First, a model of the resource itself is specified, using characteristics to model attributes of resource requests. This model is currently very platform-specific; that is, the structure of the model is shaped exactly as required by the resource management of the target platform. For a CPU which can schedule tasks with one mandatory and one optional part as described in Sect. 4, the corresponding CQML$^+$ specification is given in Listing 4. Based on these definitions, `decoder_good_cpu` can then be defined as in Listing 5. where `vdecoder_mand.dist` and `vdecoder_opt_q3.dist` reference tabular representations of the distribution functions of execution times for the mandatory and the optional part, resp.

After having shown the non-functional part of the specification, we specify the application structure of the example next. Our component model contains separate

```
profile decodingBehaviour for VernerVideoDecoder {
  profile good_work {
    provides good_rate (uncompressedVideo);
    uses good_rate (compressedVideo);
    resources decoder_good_cpu;
  }

  profile ok_work {
    provides good_rate (uncompressedVideo);
    uses good_rate (compressedVideo);
    resources decoder_ok_cpu;
  }
}
```

Listing 3: CQML$^+$ profile for the Verner video decoder component implementation.

```
resource cpu {
  quality_characteristic task_quality {
    domain: numeric [0..100] percent;
  }
  quality_characteristic execution_time {
    domain: distribution;
  }
  quality_characteristic task {
    domain: tuple {
      task_quality,
      execution_time
    };
  }
  quality_characteristic mandatory : task {
    invariant: task_quality = 100;
  }
  quality_characteristic optional : task {}
  quality_characteristic period {
    domain: numeric microseconds;
  }
  quality_characteristic demand {
    domain: tuple {
      period,
      mandatory,
      optional
    };
  }
}
```

Listing 4: CQML$^+$ resource CPU specification.

```
quality decoder_good_cpu {
  cpu.period = 40000;
  cpu.mandatory.task_quality = 100;
  cpu.mandatory.execution_time = "vdecoder_mand.dist";
  cpu.optional.task_quality >= 95;
  cpu.optional.execution_time = "vdecoder_opt_q3.dist";
}
```

Listing 5: CPU demand of Verner in the `good_work` case.

descriptors for component specifications and component implementations, but these are very simple and straight-forward, so that we will omit them here for brevity. The most interesting functional descriptor is the assembly descriptor, which defines the structure of an application.

The XML snippet from the assembly descriptor of our example in Listing 6 illustrates the definition of component networks using `template` elements. It shows the wiring of the component assembly from Fig. 1. The template is triggered when a client wants to create a `Controller` instance via its home interface. The declaration part of

```
<template homeplacementref="Controller">
  <instance id="sync" homeplacementref="Synchronizer"/>
  <instance id="vdec" homeplacementref="VideoDecoder"/>
  <instance id="adec" homeplacementref="AudioDecoder"/>
  <instance id="mux"  homeplacementref="Demuxer"/>

  <connectinstance id="this">
    <connect type="call"
             usesport="SyncControl" providesport="SyncControl"
             instanceref="sync"/>

    <connect type="call"
             usesport="DemuxControl" providesport="DemuxControl"
             instanceref="mux"/>
  </connectinstance>

  <connectinstance id="sync">
    <connect type="stream"
             usesport="UncompVideo" providesport="UncompVideo"
             instanceref="vdec"/>

    <connect type="stream"
             usesport="UncompAudio" providesport="UncompAudio"
             instanceref="adec"/>
  </connectinstance>

  ...
</template>
```

Listing 6: Assembly descriptor excerpt for the example application.

the template consists of several `instance` elements assigning an ID (`sync`, `vdec`, `adec`, and `mux` in the example) and a component type to each instance. The actual type is determined by indirectly referencing component specifications via *homeplacements* (similar to those of CCM), which are also described in the assembly descriptor, but omitted for brevity.

A `connectinstance` element for each component instance including the instance that triggered the creation of the component net (represented by `this`) forms the wiring part. Several contained `connect` elements specify which `providesport` of which component instance must be connected to the given `usesport` of the instance referenced in the surrounding `connectinstance` element using the names of the ports specified in Fig. 1.

## 5.2 Specification Language Concepts

After the discussion of a concrete example, we now explain the general structure of CQML$^+$—our language for specifying non-functional properties—and the various descriptors for the specification of the functional make up of an application in more detail.

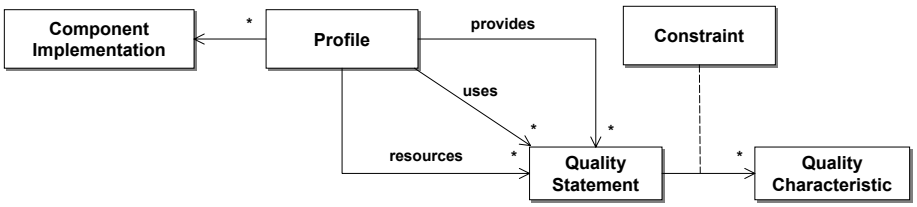


Figure 6: Abstract syntax of CQML$^+$

### 5.2.1 CQML$^+$: A Language for Specifying Non-functional Properties of Component-Based Systems

To specify non-functional properties of component-based software (and in particular realtime properties), we use CQML$^+$ (Röttger and Zschaler 2003), an extension of the Component Quality Modelling Language (CQML) (Aagedal 2001) defined by Aagedal. CQML's terminology is based on the ISO QoS Framework (International Standardisation Organisation 1998).

Figure 6 shows the conceptual structure of CQML$^+$ represented as a UML class diagram. The basic building block of a CQML$^+$ specification is the quality characteristic. It represents an entity to be constrained by the specification. Quality characteristics have a name, a domain, and a semantic, given by the `values` clause—an expression in OCL specifying how values of the characteristic can be determined in a running system. Examples for characteristics are frame rate (see the example CQML$^+$-specification of this characteristic in Listing 1 on page 10), jitter, screen resolution, but also—in a different context—learnability. Based on these quality characteristics, quality statements are used to specify constraints on quality characteristics. We have seen an example of such a specification in Listing 2 on page 11. Because both quality characteristics and quality statements are parametrised, they allow for reuse of parts of the specification in different contexts.

The specification is completed by associating quality statements with components of the system. For this, CQML$^+$ offers the concept of quality profiles. Listing 3 on page 12 shows a concrete example of a quality profile. Here, the formal parameters of the quality statements are replaced by actual elements (e.g., operations, streams) of the component for which the QoS constraint is to be specified. There are three ways in which a quality statement can be associated with a component: `provides`, `uses`, and `resources` clause (compare the corresponding keywords in Listing 3). A QoS offer (`provides`) describes the quality a component offers to its environment, provided it receives services with a certain quality from other components (`uses`) and it is allocated a certain amount of resources (`resources`). Resource specification can vary from very simple specification in a name–value style to highly complex specification including distribution functions. The latter is the case for our example CPU demand specification, which can be seen in Listings 4 and 5. As explained in (Röttger and Aigner 2002), resources can be viewed as system-level components, in contrast to normal application-level components.

A profile, therefore, describes the relationship between the quality of services received by a component from its environment, the resources allocated to the component, and the quality of services the component provides. Profiles can contain multiple sub-

profiles (not shown in the figure, but Listing 3 shows an example), which essentially represent different working regions—or modes—of the component. Sub-profiles are used as a means to express adaptivity by allowing the runtime environment to switch between profiles—for example, when resource availability changes. For this purpose a profile can have a `transitions` clause, which defines all allowable transitions between sub-profiles, and specifies an operation in the component's interface which the runtime environment calls to trigger the transition. This will be explained in more detail in Sect. 7.

Finally, CQML$^+$ specifications can be structured using quality categories, which essentially provides name spaces to the specifier. We have not explicitly mentioned them in the example.

### 5.2.2 Specifying Functional Contracts and Component Assemblies

Although we mainly focus on the description of non-functional properties of components, functional aspects must still be considered. We use three XML-based descriptors to describe properties of specifications, implementations, and component assemblies, respectively. The descriptors are bundled with other constituents, such as binary code, in component archives that can be deployed into the runtime environment.

**Specification Descriptor** A component specification basically consists of a set of named used and provided ports, and a home interface. Additionally, each component offers an *equivalent interface* similar to the same concept in CCM that allows clients to navigate among the component's ports and that represents the component identity. Streaming ports are a special type of component interfaces, which we introduced in our component model (Göbel et al. 2004). They are connected using the same mechanisms as for common ports.

**Implementation Descriptor** A component implementation is always bound to exactly one component specification. This is done by referencing the unique specification ID. The implementation descriptor also contains mappings from all interfaces defined in the specification descriptor—equivalent, home, and port interfaces—to actual implementation classes. The definition of non-functional properties also belongs to a particular component implementation but following the separation of concerns principle, it is specified in the CQML$^+$ descriptor (cf. Sect. 5.2.1). The container uses information about both functional and non-functional properties to create component instances at runtime.

**Assembly Descriptor** Component assemblies need to be defined declaratively because components must not explicitly create or destroy new instances and component references cannot be exchanged between components. This is necessary to ensure that the container can take into account all component instantiations, connections, and required resources when determining whether to accept an additional request. The *assembly descriptor* defines which components need to be created and how they are interconnected by employing concepts from Architecture Description Languages (ADLs) (Medvidovic and Taylor 2000). Listing 6 on page 13 shows an excerpt from the assembly descriptor for the example, namely from the template section.

## 5.3 A Software Development Process for Component-Based Software with Non-functional Properties

Non-functional properties must be considered throughout the complete process of developing software. Figure 7 shows an overview of a development process which explicitly takes into account non-functional properties and components. We have reported on earlier forms of this process in previous publications (Aigner et al. 2003; Röttger and Zschaler 2004; Röttger and Zschaler 2004).

After the requirements analysis the *application designer* begins to model the system. This includes modelling of non-functional properties by specifying non-functional constraints and attaching them to components and connectors. Our approach separates specifying non-functional constraints from the definition of the underlying measurements (i.e., `quality_characteristic`s in CQML$^+$). Measurement definitions can be very complex, but on the other hand will be developed only once. Therefore, we separate the roles of *measurement designer* and *application designer* in our process. Their combined efforts lead to a specification of the system including its non-functional properties.

Our process comprises the following major actions (cf. Fig. 7):

1. Definition of measurements at different levels of abstraction by the measurement designer. The measurement designer can do so independently of application development and even at a far earlier time.

2. Use of measurements during the specification process by the application designer. The application designer constrains measurements and binds these constraints to elements of the functional model to indicate expectations on the component implementations to be used.

3. Tool-supported refinement of measurements. The application designer chooses one out of several kinds of provided refined measurements. These have been previously provided by the measurement designer together with an informal description of each measurement. Tool-supported refinement is discussed in more detail in (Röttger and Zschaler 2004).

4. Implementation of components by *component developers*. These can either be components developed by third parties and selected to be placed into the application under development, or components implemented specifically to fill gaps in the application design. Component developers use a test container (Meyerhöfer and Neumann 2004) to analyse the non-functional properties of their component implementations and provide a corresponding specification—for example, the execution time distribution functions—to the application designer. Component implementations are maintained in a component repository for later reference.

5. Assembly of the application from components from the component repository. At this stage, compatibility and schedulability analysis can be performed to validate the feasibility of the design.

The resulting non-functional specification is used for a variety of purposes. Besides generating code for runtime monitoring of QoS parameters, its main use is in providing a base for QoS contract negotiation and resource reservation in the running system, as described in the following section.

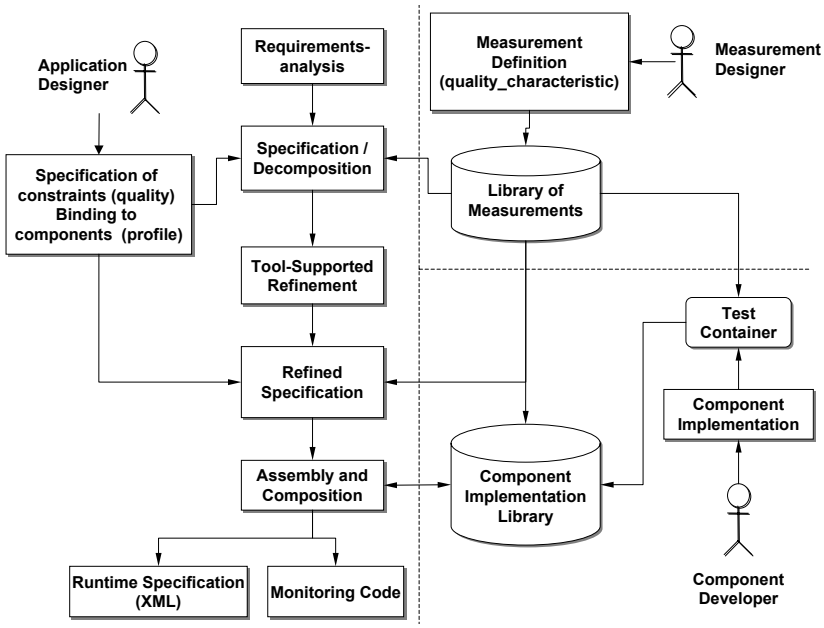


Figure 7: Overview of the development process

# 6 A Runtime Environment for Component-Based Software with Enforceable Realtime Contracts

This section introduces the runtime environment for our components consisting of both the realtime operating system DROPS and the component container CONQOS with its split architecture. We also explain how our sample application, specified in CQML$^+$ and our functional descriptors, is set up and executed by our runtime environment; that is, the steps the runtime environment takes to translate the component-based specification into a task-based specification for the RTOS.

## 6.1 DROPS – The Dresden Realtime Operating System

DROPS is a realtime-capable operating system based on the L4 microkernel (Liedtke 1995), which provides fast inter-process communication (IPC). Device drivers are implemented as user-level servers to provide isolation and fault-tolerance for the OS. To demonstrate the flexibility of L4, Linux has been modified to run as a user mode server on top of L4: L$^4$Linux. This approach demonstrates that in order to provide features such as realtime or security, only a few parts—for example, device drivers—have to be ported, while the majority of the former application, which in our case means the major portion of the Linux code, continues to run unmodified.

DROPS includes a realtime window manager, DOpE (Feske and Härtig 2004), which can guarantee refresh rates for its realtime clients. Non-realtime clients of DOpE are drawn whenever there is time left. DOpE also contains features to hide the content of sensitive windows from untrusted windows.

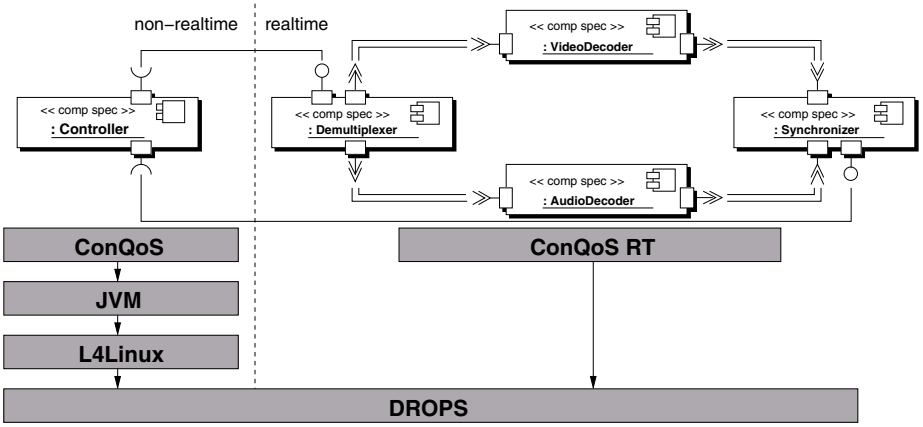


Figure 8: The split container architecture running the example application

Resources in DROPS are managed by resource managers (Härtig et al. 1999). Resource managers can form hierarchies to either combine different resources into a higher-level resource, or, if managing the same resource, to build resource domains. Resources can be reserved by an application in advance to ensure their availability when they are used. This includes the reservation of computation time, memory, and so on.

## 6.2 A Split Architecture for a Runtime Environment for Realtime Components

Based on our experience with the development of DROPS we have concluded that realtime capabilities are often not necessary for large applications, but only for small parts of them. Consequently, we applied the same philosophy to the component container and designed it as a split architecture where we have a large *non-realtime container* (CONQOS [2]), which is based on JBoss (Fleury and Reverbel 2003), and a small *realtime container* (CONQOS RT), which runs directly as a user-level server on the microkernel. By doing so, we both simplify the development by reusing existing software and minimize the amount of realtime-capable program code. A communication interface connects the two parts and makes them appear as one component container for applications. Figure 8 depicts the general structure of our split container architecture.

The following sections describe design and function of the non-realtime and realtime container. More details on the communication between containers and components can be found in (Göbel et al. 2004; Pohlack, Aigner, and Härtig 2004).

**Non-realtime container** CONQOS implements all functions that do not need to provide realtime guarantees. This includes the deployment process of components, the management of component specifications and implementations, as well as the initialization and startup phase of applications. In general, the CONQOS acts as a controller

[2] *Con*tainer supporting *Q*uality of *S*ervice

of CONQOS RT. In Sect. 6.3.3 we describe the functions in more detail ordered by their usage in the component life-cycle.

CONQOS deploys component specification and implementation archives containing binary code and descriptors (see Sect. 5.2.2) at startup or at runtime triggered by a deployment tool. Descriptors are processed and information about both functional and non-functional properties is stored in an internal repository. A name server enables clients to obtain initial references to home interfaces. They are bound to name server entries during component deployment as specified by the assembly descriptor.

Clients create actual component instances via the home interface and transmit realtime requirements together with the `create` request. The actual contract negotiation is then performed by the contract manager in three steps:

1. *Component assembly computation:* A request is typically not handled by a single component, but by a network of cooperating components. In this step, the contract manager recursively derives a representation of this component assembly from information specified in assembly descriptors. The component assembly representation exists entirely at the level of (functional) component specifications, no implementations have been selected so far.

2. *Component implementation and profile selection:* Based on the initial requirement of the client, only component implementations and profiles are selected that mutually fulfil the required and offered quality statements for each connection between used and provided ports in the component assembly.

3. *Resource reservation:* The contract manager transmits the resource demand of the concrete component assembly found in the previous step to the resource manager (cf. Sect. 6.3.2). If the resource reservation has been completed successfully, the contract manager returns a reference to the requested component instance to the client. Otherwise, the contract manager must return to the previous step and try to find another component configuration.

**Realtime container** For the realtime container we have identified a minimal set of necessary services. It contains a simple instance repository, communication infrastructure, resource managers, and a small framework for components, consisting of interfaces and base classes for component instances, and helper functions.

## 6.3 Revisiting the example – Running the Video Player Application

In this section we walk through an example of how the runtime environment sets up and runs the video player application for a specific video.

### 6.3.1 Configuring the Video Player Application

The client uses the name server of CONQOS to acquire a reference to the `Controller` home interface. It then calls the `create` method, passing the required non-functional properties as arguments. In our example, the client might demand videos to be displayed with at least 25 frames. This method call is processed by CONQOS and triggers the contract negotiation process as illustrated in Sect. 6.2.

The *component assembly computation* step of the contract negotiation process in CONQOS forms the component assembly (only at the level of component specifications) as depicted in Fig. 1 using the assembly descriptor from Sect. 5.1. In the next

step the contract manager selects appropriate implementations and QoS profiles for all component specifications of the component assembly determined. In our example we have two implementations for the `Videodecoder` and one implementation for `Audiodecoder`, `Synchronizer`, `Demuxer`, and `Controller` specifications, resp. One `Videodecoder` implementation only provides 20 frames per seconds and is therefore not selected. Additionally, the contract manager selects the QoS profiles with the maximum quality if more than one profile is available.

In the final step of the contract negotiation process, the contract manager adds up all necessary resources of the component assembly and sends it to the resource manager in the realtime container. If the resources are reserved successfully, as described in the next section, a reference to the `Controller` component is returned to the client.

### 6.3.2 Reserving Resources

Resources are managed by *resource managers* as described in (Härtig et al. 1999). If a component intends to use a resource with a specified quantity, it must make a reservation with the resource manager. The resource manager checks if the requested quantity of the resource is available and either grants or rejects the request.

Because a client may request multiple resources, the reservation has to have transactional semantics: all resources have to be granted or rejected. Therefore, we use a centralized *QoS manager* that receives a combined reservation request for all requested resources. The QoS manager iterates each of the requested resources and tries to make a reservation with the respective resource manager on behalf of the client. If all resource requests are granted, the request of the client is granted as a whole.

A request for a resource contains a generic part—the name of the resource—to allow the QoS manager to identify the responsible resource manager. The rest of a request for a resource is resource-specific and has to be interpreted by the respective resource manager only. The container can generate the complete resource request from the CQML$^+$ resource specification of all component implementations in the network, because we took care to base the CQML$^+$ specification on the same mathematical model, described in Sect. 4, as our schedulers.

The resource-specific part of the request describes the quantity to be reserved. Using this information, the resource manager performs an admission test and, if the request can be admitted, returns a handle to the reserved quantity. When all handles are collected, the QoS manager returns these handles to the caller, which uses these handles to access its reservation.

To minimize the modifications to components required to access reserved resources with handles, we use resource proxies. Resource proxies intercept resource accesses, identify the accessing component, and add the respective resource handle to the access call. In this way, the accessed resource can identify the reservation and grant or deny the access.

If a component can cope with different resource constellations, such as a large amount of processing time and a small amount of memory or less processing time but more memory, its specification contains alternatives. The QoS manager can then use a global cost evaluation to determine an alternative that provides the best utilization of the resources. If more than one alternative provides equally good utilizations, the specification contains preferences (declared using the `number` of `alternative`).

### 6.3.3 Starting, Running and Stopping the Application

After all the resource reservation steps have been successfully completed, ConQoS asks ConQoS RT to create instances of the realtime components—demultiplexer, audio and videodecoder, and synchronizer instances in our example as depicted in Fig. 1. Afterwards these instances are connected and started, again by sending appropriate commands from ConQoS to ConQoS RT. From then on, the instances are only controlled and executed by ConQoS RT.

At runtime the component uses the resources reserved for it. Furthermore, the application can be influenced via the GUI. Typical video playing tasks, such as starting and pausing the video and seeking in the video are handled this way.

After completion of the work or on user request (i.e., a `remove`-call on the controller's home interface) the whole video playing application is terminated and all reserved resources are freed.

# 7 Adaptation to Changes in Resource Availability

Environment conditions or resource availability may change during application runtime. To deal with such situations, enforceable component-based realtime contracts enable adaptation of an application to changing resources and other environment conditions.

We deploy resource reservation mechanisms to guarantee realtime properties for applications. Thus, it might appear that adaptation techniques are unnecessary or even mutually exclusive to our approach. On the contrary, we argue that adaptation can improve overall quality by making better use of temporarily available resource surpluses and by compensating for unforeseen long-term environment changes. Adaptation and reservation work together in our architecture, which allows for adaptation on four different levels:

1. Omitting optional parts. This covers adaptation as described by resource demand specifications and special structural conventions of our component model.

2. Adapting by adjusting parameters

3. Adapting by adjusting parts of an application's structure. Together with the previous item, this covers adaptation described by profiles in the CQML$^+$ specification.

4. Finally, adapting statically by selecting an appropriate component implementation while setting up an application. This has already been covered in the previous section, but we list it here because it also constitutes a form of adaptation. This covers adaptation which can be described either by profiles in CQML$^+$, or by providing different implementations for the same component specification.

A key concept in our approach is that we want to separate the actual adaptation logic as much as possible from the business logic constituting the components' code. Adaptation should be managed by the runtime environment, it should not be something application or component developers need to worry about.

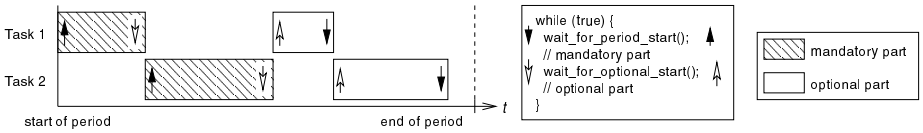


Figure 9: Invoking methods to start period or optional parts.

## 7.1 Omitting Optional Parts

The most basic form of adaptation is to omit work for which there is no time left in favour of other work which may thus be finished on time. This can be easily supported within our task model (cf. Sect. 4.1): Parts of work which can be omitted without compromising the essential functionality and consistency of the application are represented as optional parts while the remaining work is represented by the mandatory part of a task.

Components must follow some conventions so that the container can recognise their mandatory and optional behaviour, and can thus, together with the underlying platform, control when to leave out optional parts. Specifically, each component has to provide a `run_mandatory()` and a `run_optional()` operation. The container calls these operations when appropriate and provides the necessary interaction with the platform. Specifically, the container provides a loop in which it first waits for a notification from the scheduler to begin work. It then invokes `run_mandatory()`, negotiates with the scheduler about the start of the optional part, and finally makes a conditional call to `run_optional()`. Listing 7 presents this *work loop* in pseudo code. The oper-

```
while (true) {
  wait_for_period_start (theComponent.task_id);
  theComponent.run_mandatory();
  if (run_optional) {
    wait_for_optional_start (theComponent.task_id);
    theComponent.run_optional();
  } else { run_optional = true; }
}
```

Listing 7: Central loop for active components.

ations `wait_for_period_start` and `wait_for_optional_start` are part of the scheduler API, signalling that a task is ready to start its period—and therefore its mandatory part—and its optional parts, respectively. These operations work as depicted in Fig. 9. When invoked (represented by an arrow pointing downward in the figure), they do not return (represented by an arrow pointing upward) until the scheduler determines that the task has become ready to work. Thus, when the operations return the mandatory or optional work can be started.

The CPU scheduler calls the *notification interface* responsible for the component to indicate when no computation time is available for the optional part. The container implements this interface for all its components and uses the information to set the `run_optional` variable to the appropriate value. Listing 8 shows the corresponding pseudo code.

For our video player application, good work could be quantified by post-processing 95% of the pictures. This means that the mandatory part of the corresponding task

```
while (true) {
  message = receive_preemption_msg(theComponent.task_id);

  if ((message.type == MISSING_NEXT_RESERVATION) ||
      (message.type == MISSING_NEXT_PERIOD)) {
    run_optional = false;
  }
}
```

Listing 8: Central loop of the container's notification interface implementation.

consists of decoding the frames, while the optional part consists of the post-processing steps. Thus, `run_mandatory()` would be implemented to invoke the decoding routine, and `run_optional()` would call the post-processing routines in the appropriate order.

## 7.2 Parameter-Based Adaptation

For several real-world problems skipping a part of the work as described in the previous section is not always possible because the algorithms and data to be processed are not well suited to this kind of adaptation.

For example, in case of memory shortage in a system it will not be helpful if every second frame would be skipped or if a higher percentage of disk requests would be omitted. Instead it is necessary to reduce the resolution of a video stream, which is processed by a chain of filters, so that in every component less buffer memory is used.

This kind of adaptation is called parameter-based adaptation, as parameters of the involved components are adapted at runtime. In order to be adaptable to new resource situations each component must implement an *adaptation interface*. The component container uses this interface consisting of methods defined by the CQML$^+$ specification to switch QoS profiles. The parameters and the values to set in order to switch to a certain profile are defined in the `transition` clause of the specification as described in Sect. 5.2.1. Additionally, the *adaptation manager*, as part of the container, perceives changes in the amount of available relevant resources and notifies components to be adapted using their adaptation interface. To summarise, the *adaptation manager* decides when to adapt and which components to adapt to new resource situation based on runtime information about the system.

To collect the data to be used by the adaptation manager we use a *runtime monitoring system*. This system typically consists of several sensors or feedback mechanisms implemented in the different resource providers, that is, the system collects information about the global resource usage situation in the system. Imagine a memory provider that simply provides information about the amount of available and reserved memory, a network driver with a realtime protocol stack that provides information about the reserved bandwidth, or a hard disk driver that provides information about the amount of disk request and distribution of execution times, which may vary highly, depending on the current load. Furthermore, specific applications may provide feedback information specific for their domain. For example, our video decoder monitors its CPU demand for decoding single frames and makes this information available.

We want to illustrate parameter-based adaptation with one implementation detail in our system. As mentioned, the video decoder monitors CPU time necessary to decode video frames over a time period of several seconds and feeds this information to the runtime monitoring system. Based on this information and the global CPU usage the

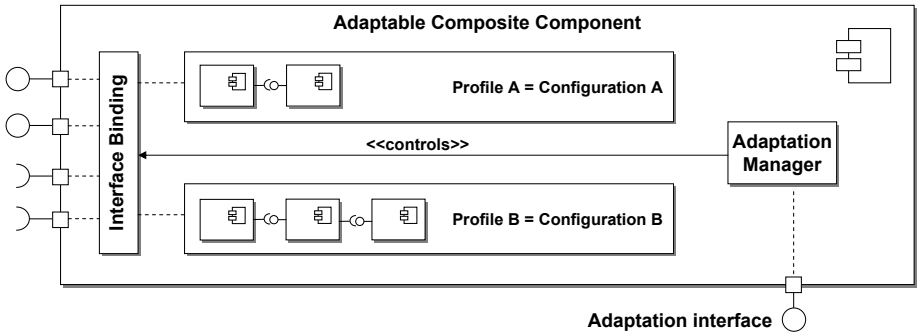


Figure 10: Internal structure of a composite component

adaptation manager signals the video decoder to adjust the post-processing algorithm, thereby trading computation time for quality. Using this mechanism we achieve a more constant CPU usage for otherwise much more erratic CPU demands.

## 7.3 Adaptation by Structural Modification

The previous section illustrated how a component implementation with different QoS profiles can be utilized to adapt to environment changes. Still, the question remains how components that support different QoS profiles can be designed and developed efficiently. Switching the QoS profiles of components requires internal changes in the component—for example, omitting some processing steps or choosing different algorithms for calculations. A possible implementation strategy is to manually write special code, like `if...else` or `switch(mode)` constructs, in the component. However, this process hides information about structural adaptation and, moreover, it is tedious and error-prone for the developer. Instead, the necessary reconfigurations of the component should be made explicit and untangled from the program logic. Thus, we adopt the concept of composite components to encapsulate structural adaptation and hide it from other components (Göbel 2004). Each QoS profile is mapped to a different internal configuration of the composite component and the definition of these configurations is part of the composite component.

Our model of composite components supporting different QoS profiles is depicted in Fig. 10. Composite components like all our components have fixed sets of provided and used interfaces, even if the internal configuration is changed in the course of a profile switch. This means that the reconfiguration process is transparent to other components of an application. Only contracts for non-functional properties to interconnected components are changed together with the QoS profile.

A particular configuration of a composite component consists of a set of subcomponents, connections between them, and a binding from external interfaces of the composite component to internal interfaces of subcomponents. Configurations are defined by a descriptor based on the assembly descriptor (cf. Sect. 5.2.2) enhanced by information about interface binding. Additionally, information about non-functional properties required for contract negotiation is attached to each configuration. This data must be determined by the component developer using appropriate measuring tools (Meyerhöfer and Neumann 2004).

The interface binding, being the central hub, forwards request from external interfaces to interfaces of subcomponents and also wires internal subcomponents. The adaptation manager controls the interface mapping and the reconfigurations using the adaptation specification. Both interface binding and adaptation manager are only conceptual constituents of our composite components. The component container actually implements their functionality using the particular component descriptor.

# 8 Related Work

The OMG's RT/CORBA specification (Object Management Group 2003a) defines realtime extensions for the CORBA middleware platform. CORBA belongs to the older generation of *explicit* middleware; that is, it provides a set of middleware services, which have to be used explicitly by application programmers. However, our platform follows the approach of *implicit* or *descriptive* middleware; that is, the use of such services is not directly implemented within components' application code but provided implicitly by the container runtime environment according to additional component descriptors. This is the philosophy behind most modern component-oriented middleware platforms—for example, CORBA Components or Enterprise JavaBeans.

The OMG's CORBA Component Model (CCM) (Object Management Group 2001) forms the basis for many functional concepts of our component model, but it does not address special problems related to non-functional properties—for instance, dynamic selection of implementations at runtime. Like Sun's Enterprise JavaBeans (EJB) (DeMichiel 2003) component model, CCM supports only a limited, fixed set of non-functional aspects like persistence, access control, transactions, and so on.

CIAO (Wang et al. 2003; Gokhale et al. 2002), another related project, builds a QoS-enabled CCM implementation on top of TAO (Schmidt, Levine, and Mungee 1998). The project's philosophy is a strong adherence to existing OMG specifications such as RT/CORBA and CCM, and the extension of those. In contrast, we decided to focus on the challenges of supporting non-functional properties. Hence, we have tried to keep the functional part of our component model as lean as possible while still adopting tried and tested concepts. The considerable overhead of implementing or extending a fully compliant CCM infrastructure would have been counter-productive to a prompt realization of our main targets.

The project QuA (Staehli and Eliassen 2002) aims at precisely defining an abstract component architecture, including the semantics for general QoS specifications. While the abstract QuA architecture could theoretically be implemented on top of any realtime-capable combination of operating system and middleware, our approach is more closely tied to DROPS (Härtig et al. 1998). This allowed us to fully leverage all advantages of this platform and thus to realize our goals within a relatively short time frame.

The Real-Time Specification for Java (RTSJ) (RTJ 2001) introduces the concepts of timeliness, schedulability, and realtime synchronization to Java-based applications. One of the biggest challenges in this context is to prevent Java's garbage collector from interfering with realtime task scheduling. Furthermore, resource reservation is not addressed by this specification, which would prevent an implementation of our concepts on top of this platform. The reference implementation of Real-Time Java is based on TimeSys Linux (TimeSys Corp. 2004), which ensures dependability of realtime applications by running critical sections in kernel mode. In contrast, our platform DROPS (Härtig et al. 1998) follows the philosophy of running as much code as possible in user

mode, thus increasing system stability and safety.

Requirements for realtime extensions to Java were defined in the NIST report (National Institute of Standards and Technology 1999). The NIST group proposes partitioning the execution environment into a realtime core providing the basic realtime functionality and a traditional JVM, which services normal Java applications. Based on these requirements, the J Consortium defined the Real-Time Core Extensions for Java (RTCE) (J Consortium 2000), which follows the idea of a separate core for realtime services. RTSJ, in contrast, provides all services in one JVM, thereby containing the realtime and the non-realtime applications. The architectural RTCE approach is similar to the design of DROPS, in that both run large and complex parts in a classic non-realtime environment and only small, predictable parts in a realtime environment.

The 2K Operating System (Kon et al. 2000) implements a resource management system targeted at distributed applications. It focuses on load balancing through global knowledge about resource utilization on local nodes (Kon et al. 2001). All local resources are monitored by a local resource manager, which is also responsible for admission, resource negotiation, reservation, and scheduling of jobs. Resources are described using a name–value pair: The name identifies the resource and the value contains a description of the resource properties. One reservation can contain several resource descriptions for different resources. In contrast, we use heterogeneous resource managers and a generic description for resources and reservation interfaces.

Other approaches have been proposed for the specification of timing properties of software systems. Most notably among them is probably the "UML profile for schedulability, performance and time specification" (Object Management Group 2002). This approach, however, is focused only on the specification of issues relevant to the performance properties of the system. In contrast, CQML$^+$ and its predecessors take a more generic view of things and allow arbitrary quality characteristics to be formally specified in the language.

# 9 Outlook and Conclusions

In this paper we have presented our approach to guaranteeing realtime properties of component-based software. It is a hallmark of our approach—and our main contribution—that we consider realtime properties in an integrated manner from the phases of inception and design to the actual execution of the final system. We refer to this approach as *enforceable component-based realtime contracts.*

There are at least two general conclusions that can be drawn from this work: (i) realtime component-based systems can be constructed from components whose realtime properties have been formally described, then, the resulting application's realtime properties can be checked at the design level already; and (ii) component runtime environments dynamically map component-oriented properties into task-based, resource reservation requests and manage adaptation of such reservations at runtime; we refer to this as *container-managed quality assurance.*

In particular, we have shown how application designers can specify realtime properties of components and applications, including the components' resource demands, using CQML$^+$, and how this is integrated into a development process specifically designed for building applications with realtime guarantees. We define two additional roles in this process, namely the measurement designer, who formally describes the properties of concern, and the component developer, who implements components and guarantees their non-functional contracts. Furthermore, we have shown how the func-

tional structure of an application can be described, and why standard architecture description language techniques are not sufficient in the presence of dynamic selection of implementations by the runtime environment.

Our capability to give realtime guarantees is based in the Dresden Realtime Operating System (DROPS), which provides schedulers for workloads that can be split into mandatory and optional parts. The gap between the component-based specification using CQML$^+$ and the task-based world of DROPS is bridged by a component runtime environment, the CONQOS container. This container decides how to use the available resources for the components to be executed. The container implementation we have presented is split into two parts: a) a large part based on JBoss, which performs complex component management, and b) a small realtime-capable part running directly on the realtime operating system and executing components with realtime demands. Using a video player application as an example, we demonstrated how the specifications developed in the design phase are used by the component runtime environment to select component implementations, create instance networks, reserve resources, and eventually execute an application guaranteeing realtime properties as specified. Although our approach is reservation-based, we strongly advocate the need for adaptation even in such systems. Hence, we present three strategies for dynamic adaptation of component-based applications to changes in resource availability at runtime: Omission of optional work loads, parametrised mode changes, and structural adaptation encapsulated by composite components.

Although we were able to show an integrated approach to providing realtime properties to component-based applications, areas remain open for further research:

- So far, our runtime environment can provide realtime guarantees, and it supports other non-functional properties as long as they do not imply specific requirements on the runtime environment. We are working to define a generalisation of our split-architecture approach to provide specific support for other non-functional properties. We have published a first paper on the relevant concepts (Aigner et al. 2004). The core idea here is to use aspect-weaving technology to generate a tailor-made container from different container strategies, each of which supports a specific (combination of) non-functional properties.

- At the moment, the semantics of CQML$^+$ specifications is defined only by natural language descriptions, and the behaviour of the runtime environment. To allow for CQML$^+$ specifications to be integrated into the development process even more tightly than they are now—in particular for partial specifications to be developed independently by different component developers and combined by application assemblers later on—the semantics of CQML$^+$ must be defined much more formally. We are working on a semantic framework for this purpose (Zschaler 2004).

- Not all resource requirements can be specified statically (e.g., with MPEG-4 part 10 post-processing is an integral part of the decoding step itself, so one cannot use this step for adaptation as we did with our video-player application). Here, resource usage is highly dependent on the *data* processed, which is not known at component deploy time. Additionally, sometimes resource requirements are stable on one machine but not constant over machine boundaries due to different hardware. For both cases we aim to determine resource usage at runtime—that is, after deploying components. The problem of different machine

configurations could be solved by running calibration tests after deploying components on the target machine. We work on defining sensible use cases stressing and monitoring newly deployed components with sample data sets. For the case of fluctuating resource requirements we work on deploying runtime monitoring with feed-back techniques.

- Based on a precise understanding of non-functional specifications we plan, in the longer term, to integrate more advanced analysis techniques into our approach. For example, we would like to integrate more intelligent container strategies that automatically determine the number of component instances to pre-create at system startup time, or the amount of buffer space to allocate for incoming service requests.

Enforceable component-based realtime contracts are a systematic approach for supporting realtime properties of component-based applications from design to runtime. Together with container-managed quality assurance they can be extended to support other non-functional properties, too.

## Acknowledgements

This work was partially funded by the German Research Council (DFG) in the research project COMQUAD (FOR 428) and by Intel Labs Hillsboro. The authors would also like to thank Sten Löcher, Uwe Aßmann, Ihor Kuz, and the anonymous reviewers, who provided important comments and helped us focus more clearly on the essential information.

## References

Aagedal, Jan Øyvind. 2001. "Quality of Service Support in Development of Distributed Systems." Ph.D. diss., University of Oslo.

Aigner, Ronald, Christoph Pohl, Martin Pohlack, and Steffen Zschaler. 2004, October. "Tailor-Made Containers: Modeling Non-functional Middleware Service." *Workshop on Models for Non-Functional Aspects of Component-Based Software (NFC'04), colocated with UML 2004*. Lissabon, Portugal.

Aigner, Ronald, Martin Pohlack, Simone Röttger, and Steffen Zschaler. 2003, December. "Towards Pervasive Treatment of Non-Functional Properties at Design and Run-Time." *Proc. Intl. Conf. on Software & Systems Engineering and their Applications (ICSSEA'03)*. Paris, France.

Cheesman, John, and John Daniels. 2001. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison Wesley Longman, Inc.

Chung, J.-Y., J.W.S. Liu, and K.-J. Lin. 1990. "Scheduling Periodic Jobs That Allow Imprecise Results." *IEEE Trans. on Computers* 39, no. 9.

DeMichiel, Linda G. 2003, 12 November. *Enterprise JavaBeans Specification Version 2.1*. Final Release. Sun Microsystems.

Feske, Norman, and Hermann Härtig. 2004, December. "Demonstration of DOpE — a Window Server for Real-Time and Embedded Systems." *24th IEEE Real-Time Systems Symposium (RTSS)*. Cancun, Mexico, 74–77.

Fleury, Marc, and Francisco Reverbel. 2003, 16–20 June. "The JBoss Extensible Server." Edited by Markus Endler and Douglas Schmidt, *Intl. Middleware Conf.*, Volume 2672 of *Lecture Notes in Computer Science*. ACM / IFIP / USENIX, Rio de Janeiro, Brazil: Springer.

Göbel, Steffen. 2004, October. "Encapsulation of structural adaptation by composite components." *ACM Workshop on Self-Managing Systems (WOSS'04)*. Newport Beach, CA, USA.

Göbel, Steffen, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. 2004, June. "The COMQUAD Component Container Architecture." *4th IEEE/IFIP Working Conf. on Software Architecture (WICSA-4)*. Oslo, Norway.

Göbel, Steffen, Christoph Pohl, Simone Röttger, and Steffen Zschaler. 2004, 22–26 March. "The COMQUAD Component Model—Enabling Dynamic Selection of Implementations by Weaving Non-functional Aspects." Edited by Gail Murphy and Karl Lieberherr, *3rd Intl. Conf. on Aspect-Oriented Software Development (AOSD'04)*. Lancaster, UK: ACM Press, 74–82.

Gokhale, Aniruddha, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. 2002. "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications." *Communications of the ACM*, vol. 45 (October). Special Issue on Enterprise Components, Service and Business Rules.

Hamann, Claude-Joachim, Jork Löser, Lars Reuther, Sebastian Schönberg, Jean Wolter, and Hermann Härtig. 2001, December. "Quality Assuring Scheduling - Using Stochastic Behavior to Improve Resource Utilization." *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS-XXII)*. London, UK.

Härtig, H., R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. 1998, September. "DROPS: OS Support for Distributed Multimedia Applications." In SIGOPS98 1998.

Härtig, H., L. Reuther, J. Wolter, M. Borriss, and T. Paul. 1999, June. "Cooperating Resource Managers." *5th IEEE Real-Time Technology and Applications Symposium (RTAS)*. Vancouver, Canada.

International Standardisation Organisation. 1998. Information Technology – Quality of Service: Framework. ISO/IEC 13236:1998, ITU-T X.641.

J Consortium. 2000, September. *Real-Time Core Extensions (RTCE)*. J Consortium. Available at http://www.j-consortium.org/.

Kon, F., R. H. Campbell, F. J. Ballesteros, M. D. Mickunas, and K. Nahrstedt. 2000, August. "2K: A Distributed Operating System for Dynamic Heterogeneous Environments." *9th Intl. Symposium on High Performance Distributed Computing*. IEEE, Pittsburgh, USA.

Kon, F., T. Yamane, C. K. Hess, R. H. Campbell, and M. D. Mickunas. 2001, January. "Dynamic Resource Management and Automatic Configuration of Distributed Component Systems." *6th Conf. on Object-Oriented Technologies and Systems (COOTS '01)*. USENIX, San Antonio, USA.

Liedtke, J. 1995, December. "On $\mu$-Kernel Construction." *15th ACM Symposium on Operating System Principles (SOSP)*. Copper Mountain Resort, CO, 237–250.

McIlroy, M.D. 1968. "Mass produced software components." *Proc. Nato Software Eng. Conf. (Garmisch, 1968)*. 138–155.

Medvidovic, Nenad, and Richard N. Taylor. 2000. “A Classification and Comparison Framework for Software Architecture Description Languages.” *IEEE Transactions on Software Engineering* 26 (1): 70–93 (January).

Meyerhöfer, Marcus, and Christoph Neumann. 2004. “TESTEJB – A Measurement Framework for EJBs.” Edited by Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, *Proc. 7th Intl. Symposium on Component-Based Software Engineering (CBSE’04)*, LNCS no. 3054. Springer, 294–301.

National Institute of Standards and Technology. 1999, September. *Requirements for Real-time Extensions for the Java Platform*. National Institute of Standards and Technology. Available at http://www.nist.gov/rt-java/.

Object Management Group. 2001. CORBA Components. OMG document. ptc/01-11-03.

———. 2002, March. UML Profile for Schedulability, Performance, and Time Specification. OMG Document. URL http:// www.omg.org/ cgi-bin/ doc?ptc/02-03-02.

———. 2003a, November. Real-Time CORBA Specification. OMG document. formal/03-11-01.

———. 2003b, October. UML 2.0 OCL Specification. OMG document. ptc/2003-10-14.

Pohlack, Martin, Ronald Aigner, and Hermann Härtig. 2004, February. “Connecting Real-Time and Non-Real-Time Components.” Technical Report TUD-FI04-01, Technische Universität Dresden.

Reuther, Lars. 2005, November. “Disk Storage and File Systems with Quality-of-Service Guarantees.” Ph.D. diss., TU Dresden, Fakultät Informatik.

Reuther, Lars, and Martin Pohlack. 2003, December. “Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS).” *24th IEEE Real-Time Systems Symposium (RTSS)*. Cancun, Mexico, 374–385.

Rietzschel, Carsten. 2003. “VERNER - ein Video EnkodeR uNd playER für DROPS.” Diploma thesis, Technische Universität Dresden. In German.

Röttger, Simone, and Ronald Aigner. 2002, October. “Modeling of Non-Functional Contracts in Component-based Systems using a Layered Architecture.” *Component Based Software Engineering and Modeling Non-functional Aspects (SIVOES-MONA), Workshop at UML 2002*.

Röttger, Simone, and Steffen Zschaler. 2003, June. “CQML$^+$: Enhancements to CQML.” Edited by Jean-Michel Bruel, *Proc. 1st Intl. Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*. Cépaduès-Éditions, 43–56.

Röttger, Simone, and Steffen Zschaler. 2004. “A Software Development Process Supporting Non-functional Properties.” *Proc. IASTED Intl. Conf. on Software Engineering (IASTED SE 2004)*. ACTA Press.

Röttger, Simone, and Steffen Zschaler. 2004. “Model-Driven Development for Non-functional Properties: Refinement through Model Transformation.” *Proc. UML Conf.* To appear.

RTJ (The Real-Time for Java Expert Group). 2001, 12 November. *The Real-Time Specification for Java*. v1.0. The Real-Time for Java Expert Group. http://www.rtj.org/.

Schmidt, Douglas C., David L. Levine, and Sumedh Mungee. 1998. "The design of the TAO real-time object request broker." *Computer Communications* 21, no. 4.

SIGOPS98. 1998, September. *Proc. 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications (Sintra, Portugal, Sept. 1998)*.

Staehli, Richard, and Frank Eliassen. 2002. "QuA: A QoS-Aware Component Architecture." Technical Report Simula 2002-12, Simula Research Laboratory.

Szyperski, Clemens. 2002. *Component Software : Beyond Object-Oriented Programming*. Second Edition. Component Software Series. Addison-Wesley Publishing Company.

TimeSys Corp. 2004. *TimeSys Linux*. TimeSys Corp. See http://www.timesys.com/.

Wang, Nanbor, Christopher D. Gill, Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. 2003. "Total Quality of Service Provisioning in Middleware and Applications." *Microprocessors and Microsystems* 27 (2): 45–54 (March). Special Issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet.

Zschaler, Steffen. 2004, September. "Towards a Semantic Framework for Non-functional Specifications of Component-Based Systems." *Proc. EUROMICRO Conf. 2004, track on Component-Based Software Engineering*. Rennes, France.