STWW-Programma

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

Component-Oriented Design of Common Test Case

Deliverable D3.5

02 October 2001

Contents

1. Introduction

In this deliverable we discuss the analysis and design of the SEESCOA test case (*SEESCOA camera surveillance system*). The main purpose of this test case is the validation of the SEESCOA methodology and technologies: the design is based on the proposed component based approach (see [14]) and associated methodology. It also includes results from other work packages of the project, like the work package on user interfaces. Since the work described in this deliverable is mainly design oriented, not much will be said about debugging. This will however become important in the months to come, when the test case will be implemented and tested.

Before reading further it is recommended to read the deliverable D1.3 titled *Common Test Case* (see [1]); it gives a general description of the SEESCOA camera surveillance system and the overall (physical) architecture.

In chapter 2, some existing camera surveillance systems are described, with special attention for *next generation* systems. Chapter 3 prescribes the general requirements for the SEESCOA camera surveillance system[1], without being constrained to a particular design. Chapter 4 gives an overview of the system and terminology. Chapter 5 discusses the user interaction with the system by mains of use cases. Chapter 6 deals with the architecture of the system and the interactions that occur in this architecture. In Chapter 7 the core components will be described in detail.

The work in this deliverable is the result of 6 meetings with all university partners:

Meeting July 16th 2001 (at RUG): discussion on existing systems, goals of the test case, the requirements and definition of an initial architecture.

Meeting July 23rd 2001 (at VUB): overview of use cases and interaction scenarios, refinement of the architecture: defining important components.

Meeting August 2nd 2001 (at LUC): definition of services and plug-ins, user interface component, further refinement of architecture in components: every partner works out the specification of one or more components.

Meeting August 16th 2001 (at KULeuven): discussion on the different component specifications made by the partners, determination of the responsabilities of the controller and the user interface. Partners further refine component specifications.

[1] This will be abbreviated SCSS in the rest of the document.

Meeting September 5th 2001 (at RUG): discussion on refined component specifications, integration of all components into one system, synchronizing component specifications. Discussion on the format and contents of the deliverable for the test case design.

Meeting September 20th 2001 (@ LUC): final discussion on the format and contents of the test case design deliverable.

2. Existing Camera Surveillance Systems

Before we explain the requirements of the SCSS, it is important to have a look at existing surveillance systems. Doing so, will aid us to formulate the requirements for the SCSS. This chapter is partly based on material found in [2] and descriptions of existing commercial systems and projects.

The automatisation of the surveillance process and the reduction of video data have become the main drivers for the advances in video surveillance technology. However, new requirements are emerging: advanced motion detection, interoperating camera's, encryption and authorization, network integration, …

2.1. First systems

The very first systems were quite primitive but also very simple to operate: they made use of a photographic film and took photographs of the environment at certain moments in time (time-triggered systems). An important property of such a system was the fact that it was *unattended*: the film was manually collected and reviewed by inspectors. These inspectors had to review the pictures by hand which caused this surveillance process to be very time consuming.

Also, the more cameras a particular site needed, the more complex this became. A more automated (re)view process was necessary.

2.2. Analog systems

A new step in surveillance technology was the use of analog cameras and recording devices. Analog surveillance systems are still common today and the technology that has been developed for it is quite advanced. Analog systems are often *assisted* systems, this means that a guard can be watching the images in real time.

The analog output of camera's can also be recorded on tape, but this is not necessarily always the case. Some analog systems have extra functionality built into them: images can be multiplexed on the screen or tape, camera's can be attached to particular physical triggers or time triggers, … However, an analog system has no image processing intelligence; a guard is still needed to detect anomalies.

2.3. Hybrid systems

Thanks to advancements in digital technology, new types of surveillance systems were born. They were not fully digital, but processed the output of several analog camera's and stored this output in a digital format.

Often, a hybrid system consists of some nodes (*processing* nodes) that can accept 8 to 16 analog camera inputs and perform some basic motion detection processing on these video inputs. When motion is detected, the system can start recording the images or raise some alarm or any other trigger. Other nodes of the hybrid system are *recording* nodes: they record the digital output of processing nodes or analog output of the camera's into a digital compressed format. Some commercial systems (see [3]) store the new images (eg. last 24 hours) on a hard disk, while old images are moved to a digital tape.

Hybrid systems can be accessed via workstations and such that more advanced operations are becoming possible. Adding motion detection and digital compression functionality automates the surveillance process and also reduces the amount of video data to store.

One of the advantages of a hybrid system is the fact that it enables users to move from their old analog systems to digital systems without having to invest extensively in advanced surveillance (they can still use their existing analog camera's and cabling). Examples of existing hybrid systems can be found in [4] and [5].

2.4. Digital systems

The next generation of camera surveillance sytems is fully digital. The cameras produce digital video output and have processing facilities. Also, to enable remote monitoring, they can be attached to a network (Ethernet).

The processing facilities of a camera are used to perform some advanced motion detection processing (see the part on motion detection in chapter 7 for more information about this), encoding of the images, integration with other camera's and sensors in the environment, … This extra intelligence enables:

- the further automatization of the surveillance process (eg. motion detection, automatic generation of reports, alarm triggering,…).
- inter-camera cooperation.
- a reduction in the amount of video data that is transmitted or stored (eg. record images when motion is detected, advanced encoding and compression algorithms, …).
- remote operation and monitoring.

The fact that cameras are becoming more intelligent puts also some requirements on the hardware: a processor, memory and a network interface are needed. But, the price tag of the hardware may not be too high (since the price of a camera will also influence the total price of the system) and the energy consumption is also an issue. This is especially the case if the systems are completely or even partially battery operated.

2.5. Current status

The fully digital surveillance system is clearly *the system of tomorrow*, but some issues are still open. There exist some *hardware* issues, since these systems need processing power and network connectivity. The hardware may not be expensive and its energy consumption must be limited. There also exist some *software* issues, especially in the field of image processing. Current image processing algorithms are still quite basic or very specific (see the intermezzo *Specialized Surveillance*). Advanced motion detection algorithms are still a research topic. Examples of issues that are being looked at today are the detection of persons and objects ([6]), and the combination of several inputs, like sound and images for assisting in the surveillance process ([7]).

Most systems today are analog or hybrid systems. The hybrid systems can perform some processing on the images, but this processing is often constrained to centralized basic motion detection.

However, there exist a limited number of systems that can be categorized as fully digital (see [8] and [9]) with some available processing power.

Intermezzo: Specialized Surveillance

Today, there exist some surveillance systems that can be categorized as 'intelligent'. The image processing that is performed by these systems goes beyond the standard motion detection: it detects particular actions or situations. The drawback of these systems is the fact that they are highly specialized and centralized. They are not usable for general purpose surveillance, but can eventually be integrated in an existing surveillance system.

This section discusses two existing systems: an airlock surveillance system [10] and a fire detection system [11].

An airlock is a small room that is used for securing the access between two rooms: it gives access from the outside world to a secured area. Airlocks are often used in buildings for employee access, airports, banks,… Access to an airlock can be controlled by some device like biometric or magnetic card locks. The airlock video surveillance system can be used in combination with these access control devices. The airlock system detects if two persons are in the same room, if two doors are opened at the same time, if the person stays too long or even if the person wants to touch the camera. To be useful, the system has some installation guidelines: the airlock room must have contrasted and patterned walls (horizontal lines on the walls) and floor (chessboard pattern), it is also recommended to use neon tubes installed in a particular way to avoid shadows. It is clear that this type of system is very specialized and only useful in specific situations.

Another system, is a fire and smoke detection surveillance system. This system is often used in trafic situations (tunnels, car parks,…) or industrial applications. It can be connected to existing cameras and makes use of an image processing algorithm that is specialized for fire and smoke detection.

3. Requirements

This chapter describes the requirements for the SEESCOA Camera Surveillance System. Our purpose is to build a fully extensible digital system, with the ability to put (a part of the) processing activities on the camera.

3.1. Description

The SCSS offers a distributed architecture in which digital cameras can be put and interoperate with other cameras and nodes in the system. These cameras deliver their images to other nodes in a compressed digital format. The nodes, in turn, process the video data and can eventually control other parts of the system. At first, the system offers basic functionalities like storage and motion detection processing. However, new functionalities can be added easily over time.

To be more specific, the system that we are specifying has following characteristics:

- **Digital**: the system is fully digital. This is necessary if complex computations have to be performed on the image data.
- **Local intelligence**: cameras are not seen as passive image generating devices; they *can* also have processing possibilities. This enables the local processing[2] of images.
- **Services**: the system also offers services to the user. Some of these services are quite important, like the storage of images and the automatic monitoring and alarm reporting.
- **Networked**: all cameras are connected to a network. This means they can connect to each other and other nodes in the system. As a result, they can coordinate their actions (eg. coordinated zooming, send images (for recording), …)
- **Dynamic**: cameras can be disabled, or services can disappear. The SCSS must be able to handle changes in its environment.
- **Evolvable**: cameras can be added, services installed and updates of functionality must be possible.

[2] "Processing" has a wide meaning: motion detection, encoding, compression, … are all processing activities

3.2. Detailed requirements

The characteristics mentioned on the previous page are quite general. In what follows, we will describe the specification of the SCSS in more detail. This specification describes the basic requirements that must be met by the system.

- Connectivity
  - The cameras and the system must make use of a standard network: Ethernet (IP). This will reduce the overall installation cost of the system.
  - All nodes in the system make use of the *component system* [12] for their interaction.
  - Connecting a camera is straightforward and uniform.
- Camera[3]
  - A camera must be able to perform local processing. Some cameras could have more or less resources than other cameras, but every camera must be able to run a component system instance.
  - The software of a camera can be updated through the network.
  - Extra functionality can be added to the camera through the network. The ability to execute this functionality is dependend on the amount of local processing power.
  - A camera can deliver its images on the network to other parts of the system.
  - A camera is fully or partially (remote) controllable by other parts of the system.
  - A camera can support multiple video formats.
  - If a camera uses a new video format, then support for this new format must be added easily. Old parts of the system must be able to process the new video format without changes. This ensures the evolvability of the system.
  - Cameras can be inserted and used uniformly by the system (including new and old parts of the system).

[3] Note that it is dangerous to put heavy requirements on the camera's, since this will increase the cost of these camera's. The requirements mentioned here are the minimal requirements for a camera.

    - Cameras can offer a user interface through which they can be controlled.

- Services
    - Services can be added uniformly.
    - Services can run in a distributed manner: on the cameras and/or other nodes.
    - Services can be relocated if necessary.
    - Basic services that have to be offered:
        - **Storage**: images and events (like alarms) can be stored and retrieved.
        - **Motion Detection**: images of one or more cameras can be analysed in real-time for motion detection.
        - **Zoom Control**: cameras can coordinate their zooming activities.
    - Services can offer a user interface through which they can be controlled.
- Users
    - Users have access to the system based on their access rights. Basically, a distinction between *administrators* and *operators* has to be made.
    - Administrators are able to:
        - perform system management:
            - add/remove/configure the basic properties and infrastructure of the system.
        - perform user management:
            - add/remove/configure user information.
        - perform camera management:
            - add/remove/configure camera's in the system.
        - perform services management:
            - add/remove/configure services.

        - all activities that operators are allowed to do.
    - Operators are able to:
        - perform allowed[4] camera configuration.
        - perform allowed services configuration.
        - view allowed camera and services output.
- Dynamism and fault tolerance
    - Cameras and services can be added, configured and removed while the system is running.
    - Interactions can occur at any time, between any cameras and services.
    - System functionality can be distributed over the entire network.
    - It must be possible to let the system continue[5] its operations in case of network, node and camera failures (best effort continuation).
    - The SCSS provides a way for accessing and browsing the cameras and services available in the system (like a directory service) at running time.
    - Support for remote updates of parts of the system.
- Security
    - no requirements yet (security is important, but will not be added in a first version of the case).
- Interface
    - The interface should be automatically adaptable to the visualisation device[6] and user profile.
    - It should be possible to change, extend or model the interface dynamically (integration of new services offering an interface will allow that interface to be directly available and usable).

[4] An action is allowed if the access rights of the operator have been set accordingly by an administrator.
[5] This is also based on the network infrastructure, installation and configuration of the system.
[6] Visualisation can be done on a workstation but also on other types of devices with limited screen capabilities.

The requirements that were described on the previous pages are general requirements and will be worked out in the remainder of this document.

As will become clear in the next chapter and chapter 6, the SEESCOA Camera Surveillance System will consist of several components. All these components will also have some requirements associated to them. These requirements are discussed in detail in chapter 7.

4. Overview

This chapter describes the core concepts of the SCSS. It also shows a high-level view of the system architecture. This view is necessary to understand the use cases in chapter 5. The architecture will be futher elaborated in chapter 6. We will also define some common terms used in this deliverable, to eliminate misunderstandings.

4.1. Elements of the Camera Surveillance System

Figure 1 shows an overview of the architecture. It consists of a *controller*, *cameras, services* and *user interfaces*:

- **Controller**: the controller is responsible for the core management of the system. Its responsabilities are constrained to the strict minimum:
    - Detection of new/removed cameras and services.
    - Notification of new/removed cameras and services.
    - Upload of functionality to network nodes (eg. the camera hardware).

  In fact its main task is to act like a *lightweight directory service.*
- **Camera**: represents an image generating device[7]. The camera can also be more or less controlled: zoom, focus, light balance, … depending on the properties of the hardware.
- **Service**: represents a particular functionality added to the camera surveillance system. A service can consist of other services that work together. Some services will be image processing related (like the motion detection service), while other services will have an administrative nature.
- **User Interface**: represents an interface for access to the system by human users. Different 'types' of access exist: viewing of outputs from particular cameras, retrieving stored image sequences, configuration of the system, … The representation of a user interface is not constrained to a workstation; it is also viewable on other devices like PDA's.

[7] Here, the camera has to be seen as a real-world camera: an image generating device. Wether or not this camera can execute code does not matter at this moment.

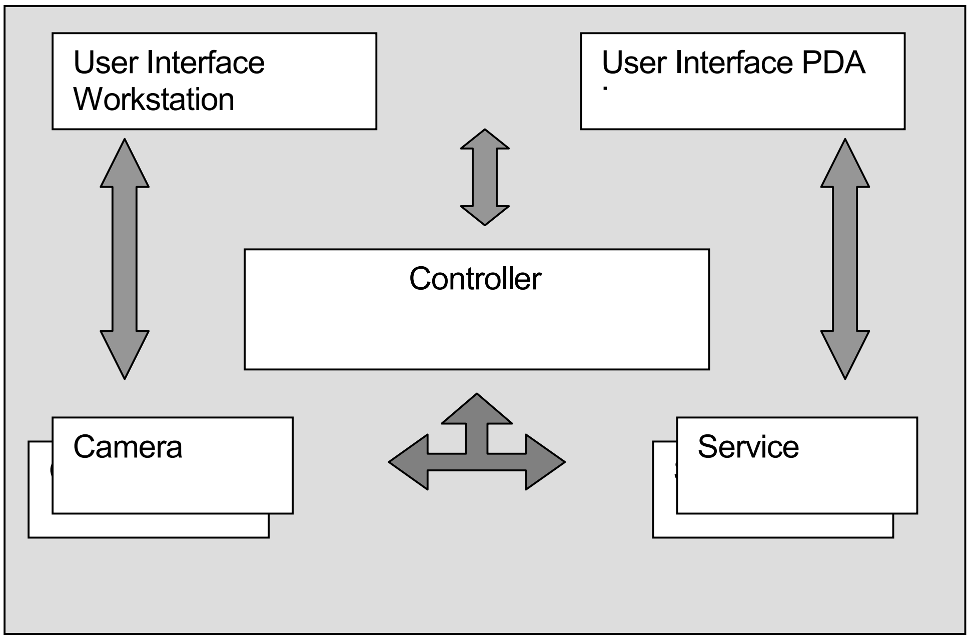


Figure 1: overview of the SCSS

4.2. User Interface

For human operators to have access to the surveillance system, it is important to provide these with a suitable user interface. Besides the fact the system is "living" on itself (camera's and services interact directly with each other), there has to be a way for users to operate the system. Some user tasks are: managing and configuring the system, view camera output, browse the storage,... All these services require user interaction and therefore there should be a user interface offering the functionality to the user.

The user interface is not bound to a particular type of device. This ensures that the SCSS is controllable from a standard desktop computer, the Internet or by using portable devices. Also, user interaction is not necessarily done using a graphical interface, speech or other non-traditional interaction modalities can also be an alternative.

The basic idea is that each service (this can include a surveillance camera for example) which offers functionality for a human user to interact with, has to provide an abstract description of the user interface. This abstract user interface is then interpreted on a client and visualized in a way that is dependend on the available resources and the constraints of that client. A client can reside on different kinds of devices: a desktop PC, PDA, Internet client via Web Browser, Mobile Phone,...

4.3. Core services

Till now, everything that was a camera and not a part of the SCSS, was defined as a service. There exist however different kinds of services:

- Some services are related to *image processing* tasks: they perform some algorithmic processing on images to extract information. A popular image processing functionality in a camera surveillance system is of course the automatic detection of motion in a sequence of images. This automatic motion detection can help a guard in his surveillance job, but image detection can also help in constraining the needed bandwidth for recording images by only recording images when motion has been detected. As a consequence, we include a motion detection service in the SCSS. The SCSS will also include a Zoom Control service or Zoom Behaviour service. This service is responsible for the management of zoom activities between a set of camera's.

- Other services are used to store image sequences. Most camera surveillance systems do have the possibility to store images on tape and/or[8] on disk. A storage service will also be provided by the SCSS. We extend this storage concept to the storage of *events* that occur in the CSS. An event is a particular occurrence of some activity that is of importance to an operator or administrator. In the case of motion detection, we could have a *motion detected* event. The storage is then responsible for storing this event, eventually with some information associated with it (time of the event, room where the motion was detected, the image sequence that contains the motion information, and so on).

- Services for administration of the system. Administration of a system is also an important aspect of a camera surveillance system. It concerns the management of users (administrators and operators) and their access rights. It also concerns the configuration of camera's and services.

The first two services mentioned above are part of the SCSS's core. The third service (user administration) is currently not a part of the SCSS.

It is possible to add or remove additional services to the system. Because of this extension capability of the system, a service is also called a *plugin* or a *coordinator*.

[8] Digital systems often store fresh image sequences (eg. last 24 hours of recording) on a disk for fast retrieval, while older image sequences (eg. last 7 days) are stored on tape, for slower retrieval. See [3].

5. Use cases

This chapter describes how a user would interact with the SEESCOA Camera Surveillance System. It does not describe a particular interface to use, it only describes the interactions between a user and the system. These interactions can be implemented in various ways (by means of a GUI, speech renderer and recognition interface, textual interface, …).

To get a first impression of how the user interface looks we have chosen to do a Use Case design and make paper mockups related to the use cases. Some use cases are followed by an illustration of a possible user interface. To give an impression of how this might look on different device types some images are palm mobile device based.

The format used for describing the interactions is based on the format defined in [13].

5.1 Manage Users

5.1.1 Manage Users

Use Case	Manage Users
Actors	Administrators
Purpose	Manage the user list for the camera system

Actor Action	System Response
1. The administrator has indicated the need to manage the user list	2. The system shows a list with managing options
3. The administrator selects the add user option	4. The system switches to the add user use case

Alternative

3. The administrator selects the change user option	4. The system switches to the change user use case

Alternative

3. The administrator selects the delete user option	4. The system switches to the delete user use case

Alternative

3. The administrator selects the quit option	4. The system switches to . . . (stop)

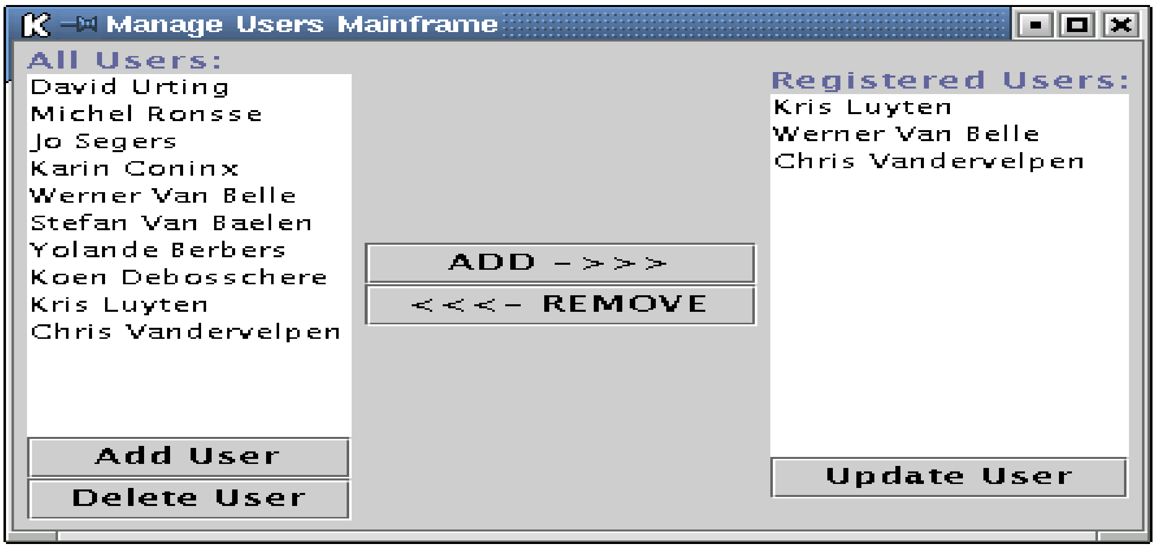


Figure 2: a possible desktop version for user management

5.1.2 Add User

Use Case Add User

Actors Administrator

Purpose Add a new user to the camera system user list

Actor Action	System Response
1. The administrator has indicated the need to add a user to the list	2. The system shows a screen asking for information about the new user

3. The administrator enters all user data	4. The system saves all the information concerning this user
	5.the system asks if another user should be added
6. The administrator wants to add another user	7. The systems starts again with 2.

Alternative

6. The administrator does not want to add another user	7. The system returns to the user managing use case

5.1.3 Delete User

Use Case Delete User

Actors Administrator

Purpose Remove an existing user to the camera system user list

Actor Action	System Response
1. The administrator has indicated the need to remove a user from the list	2. The system shows the list of current users
3. The administrator selects the user to be deleted from the list	4. The system removes all data concerning this user
	5. The system asks if another user should be deleted
6. The administrator wants to delete another user	7. The systems starts again with 2.

Alternative

6. The administrator does not want to delete another user	7. The system returns to the user managing use case

5.1.4 Change User Settings

Use Case Change User Settings

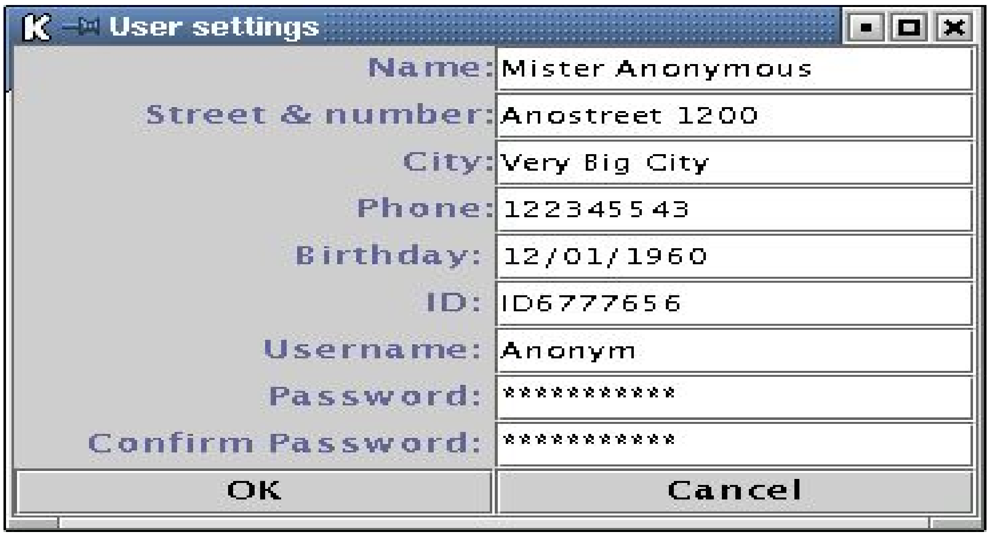
Actors Administrator

Purpose Change the settings of a user

Actor Action	System Response
1. The administrator has indicated the need to change user data	2. The system shows the list of current users
3. The administrator selects the user the data should be changed for	4. The system shows the requested user data
5. The administrator changes the user data	6. The system saves all data concerning this user
	7. The system asks if another users data should be changed
8. The administrator wants to change another users data	9. The systems starts again with 2.

Alternative

8. The administrator does not want to change another users data	9. The system returns to the user managing use case



Name:	Mister Anonymous
Street & number:	Anostreet 1200
City:	Very Big City
Phone:	122345543
Birthday:	12/01/1960
ID:	ID6777656
Username:	Anonym
Password:	************
Confirm Password:	************

Figure 3: a desktop version to enter user settings

5.2 Camera Plugin Manager

5.2.1 Manage Bounds Settings

Use Case Manage Bounds Settings

Actors Administrator

Purpose Manage the bounds settings for the different camera options

Actor Action	System Response
1. The administrator has indicated the need to manage the bounds settings for the camera	2. The system shows the bounds setting interface
3. The administrator sets the wanted bounds	4. The system saves the changed data for the bounds
	5. The system returns to the main interface …

Alternative

3. The administrator does not want to add another plug	4. The system returns to the main interface …

5.2.2 Add Coordination Plug

Use Case Add Coordination Plug

Actors Administrator

Purpose Add a plug that coordinates the focus of a camera

Actor Action	System Response
1. The administrator has indicated the need to add a control plug	2. The system shows the list of plugs
3. The administrator selects the wanted plug	4. The system installs the selected plug
	5. The system asks if another plug should be selected
6. The administrator wants to add another plug	7. The systems starts again with 2

Alternative

6. The administrator does not want to add another plug	7. The system returns to the main interface …

5.3 Camera Plugin Management

5.3.1 Manage Camera

Use Case	Manage Camera
Actors	Administrator, operator
Purpose	Control a camera

Actor Action	System Response
1. The administrator/operator has indicated the need to manage the camera	2. The system shows a list with managing options
3. The administrator/operator selects the zoom option	4. The system switches to the zoom use case

Alternative

3. The administrator/operator selects the change frame rate option	4. The system switches to the change frame rate use case

Alternative

3. The administrator/operator selects the set image compression option	4. The system switches to the set image compression use case

Alternative

3. The administrator/operator selects the set hotspot option	4. The system switches to the set hotspot use case

Alternative

3. The administrator/operator selects the settings option	4. The system switches to the settings use case

Alternative

3. The administrator/operator selects the quit option	4. The system switches to . . . (stop)

5.3.2 Zoom in/out

Use Case	Zoom in/out
Actors	Administrator, operator
Purpose	Manipulate the zoom-function of a camera

Actor Action	System Response
1. The administrator/operator has indicated the need to zoom the camera	2. The system shows the zoom interface
3. The administrator/operator makes the wanted zoom settings	4. The system adjusts the camera to the wanted settings

Alternative

3. The administrator/operator does not want to change the zoom settings	4. The system returns to the camera managing use case

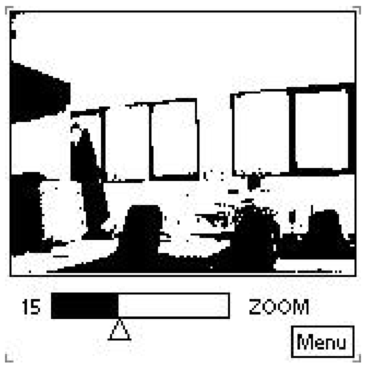


Figure 4: a possible zoom user interface on a Palm

5.3.3 Set Hotspot[9]

Use Case Set Hotspot

Actors Administrator, operator

Purpose Indicate an area of extra importance on the total camera surveillance view

Actor Action	System Response
1. The administrator/operator has indicated the need to set the hotspot	2. The system shows the set hotspot interface
3. The administrator/operator makes the wanted hotspot settings	4. The system adjusts the camera to the wanted settings

Alternative

3. The administrator/operator does not want to change the hotspot settings	4. The system returns to the camera managing use case

[9] For a definition of hotspot see 7.5.1.

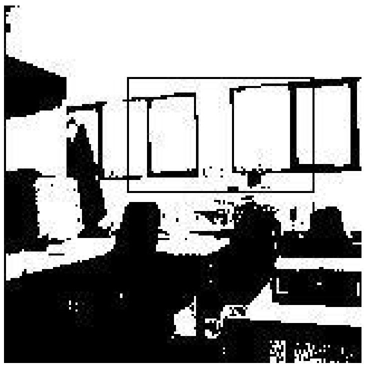


Figure 5: the stylus can be used to select a hotspot region

5.3.4 Change Frame Rate

Use Case	Change Frame Rate
Actors	Administrator, operator
Purpose	Change the frame rate of a camera

Actor Action	System Response
1. The administrator/operator has indicated the need to change the frame rate	2. The system shows the change frame rate interface
3. The administrator/operator makes the desired frame rate changes	4. The system adjusts the camera to the desired changes

Alternative

3. The administrator/operator does not want to change the frame rates	4. The system returns to the camera managing use case

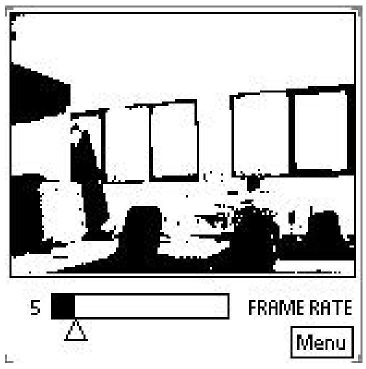


Figure 6: palm user interface to change the frame rate

5.3.5 Set Image Compression

Use Case Set Image Compression

Actors Administrator, operator

Purpose Select the image compression settings for the video stream or camera snapshots

Actor Action	System Response
1. The administrator/operator has indicated the need to set the image compression	2. The system shows the image compression interface
3. The administrator/operator makes the wanted image compression settings	4. The system adjusts the camera to the wanted settings

Alternative

3. The administrator/operator does not want to change the image compression settings	4. The system returns to the camera managing use case

5.3.6 Store Settings

Use Case Store Settings

Actors Administrator, operator

Purpose Look at the camera settings and store them

Actor Action	System Response
1. The administrator/operator has indicated the need to retrieve or store the camera settings	2. The system shows the camera settings interface
3. The administrator/operator stores the camera settings	4. The system saves the camera settings

Alternative

3. The administrator/operator does not want to change the camera settings	4. The system returns to the camera managing use case

5.3.7 Load Stored Settings

Use Case Load Stored Settings

Actors Administrator, operator

Purpose Loads previously stored settings

Actor Action	System Response
1. The administrator/operator has indicated the need to load stored camera settings	2. The system lists the previously stored settings
3. The administrator/operator selects a listed camera setting	4. The system loads the selected camera settings

Alternative

3. The administrator/operator does not want to change the camera settings	4. The system returns to the camera managing use case

5.4 Storage viewing

5.4.1 View Events

Use Case	View Events
Actors	Administrator, operator
Purpose	Observe what happens or what has happened

Actor Action	System Response
1. The operator/administrator has indicated the need to view an event	2. The system shows the events interface
3. The operator/administrator wants to view a (set of) particular event(s)	
4. The operator/administrator selects a (set of) event(s)	5. The system shows the particular events

Alternative

3. The administrator/operator does not want to view the events	4. The system returns to the camera managing use case

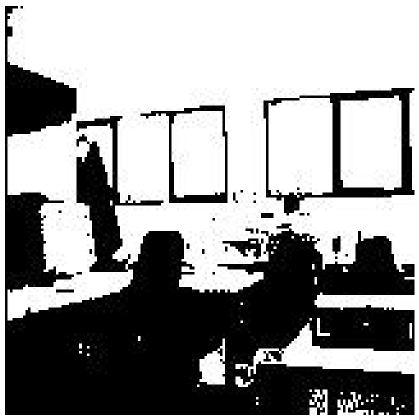


Figure 7: a full screen palm view

The different types of use cases that require a user interface have been covered in this section. The next step is to transform these use cases into real user interfaces. ConcurTaskTrees were used in the development and the way that was done is covered in the next section.

5.5 Task analysis using ConcurTaskTrees

An important step in designing a user interface for embedded systems is task analysis. These days most embedded systems are dedicated for doing a particular task (like an ATM). By consequence it becomes important to analyze this task. We use ConcurTaskTree to analyze the tasks for this case study.

ConcurTaskTree is a notation designed by Fabio Paternò, offering symbols to describe concurrent tasks. The hierarchical notation can be unambiguously expressed in XML, which eventually can be mapped to the XML used by the UIRenderer (see section 7.6) in a later stage. This provides us with a context in which the user interface is working in, enabling the UIRenderer component to make smarter decisions when building a platform dependent presentation of the abstract interface. Also, this notation could provide use with an idea of the possible sequences of dialogs in a dialog-based user interface.

ConcurTaskTree provides us with the following properties:

- **a graphical syntax** : more intuitive to use
- **concurrent notation** : expressing temporal ordering
- **focus on activities** : focus on relevant aspects, which are not system-related
- **hierarchical modelling structure** : allows clean problem decomposition

There are several possible tokens available in ConcurTaskTree. Here is a short explanation for each of them:

T1 ||| T2 : task T1 and task T2 are interleaving (concurrent) tasks;

T1 [] T2 : one of the tasks T1 or T2 will be selected (“or"-ed) to be executed;

T1 |=| T2 : the execution order of T1 and T2 is independent of T1 and T2;

T1 |[]| T2 : T1 and T2 are “synchronized"; this means there is information exchange between the two tasks;

T1 [> T2 : T1 will be deactivated if T2 terminates;

T1 |> T2 : T1 will resume if T2 terminates

T1 >> T2 : T2 only starts if T1 is deactivated;

T1 []>> T2 : T2 only starts if T1 is deactivated and information is exchanged;

T * : T is performed repeatedly, until it is deactivated by another task;

T(n) : T is performed n times;

[T] : T is an optional task;

Given these tokens, we can describe a wide range of tasks (there is no need for explicit interaction; the task could very well happen without human interaction). It gives us an overview on the temporal dependencies and illustrates the context in which an individual task resides.

This section shows ConcurTaskTree diagrams to model the interaction with the system when doing administrative tasks. The following tasks are included:

- Adding a new user to the system: figure 8
- Changing the frame rate of the camera: figure 9
- Setting the hotspot: figure 10
- Setting the image compression: figure 11
- Viewing the events: figure 12
- Zooming: figure 13
- Store settings: figure 14
- Loading stored settings: figure 15

With these ConcurTaskTrees some possibilities for real user interfaces were created. Images of some of them can be found in the previous use case section. The design of user interfaces cases for the camera surveillance system case is explained further on.

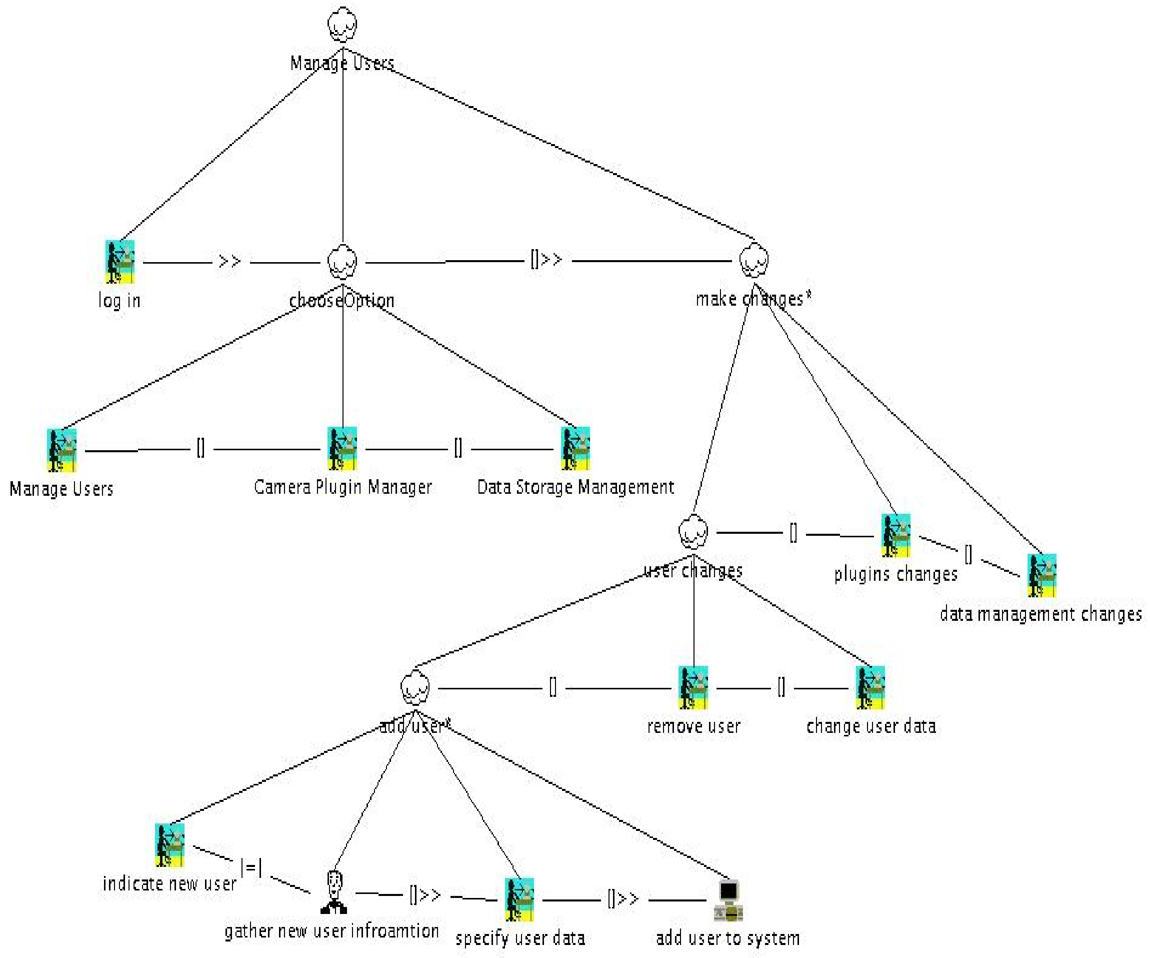


Figure 8: Adding a new user to the system

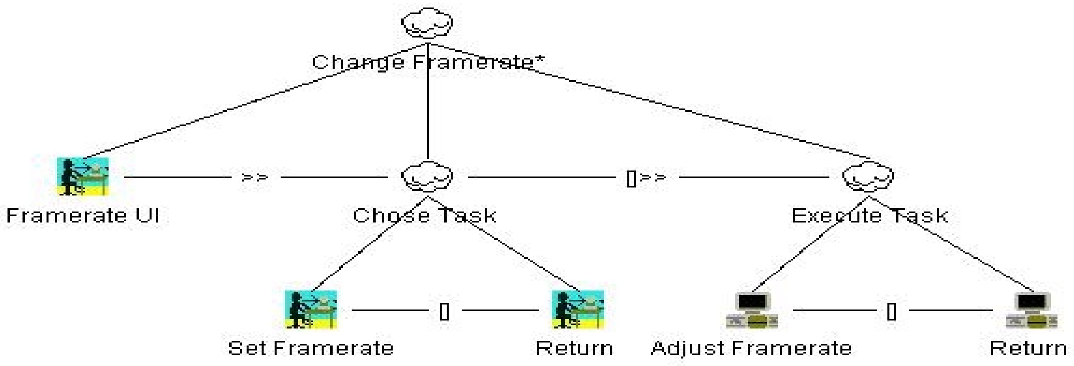


Figure 9: Change the frame rate

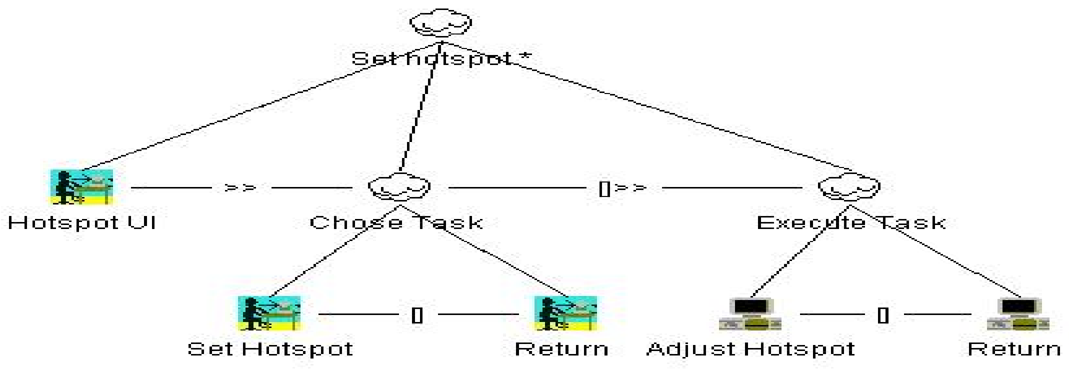


Figure 10: Set a hotspot

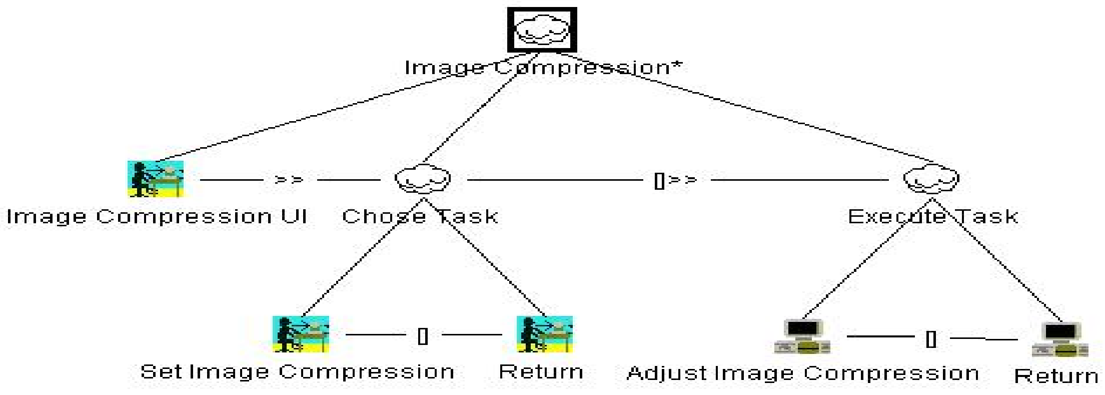


Figure 11: Set the image compression

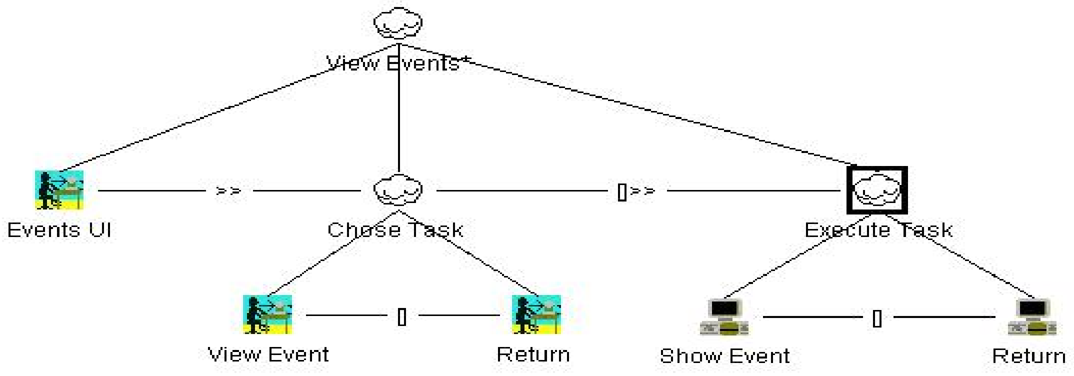


Figure 12: View the events

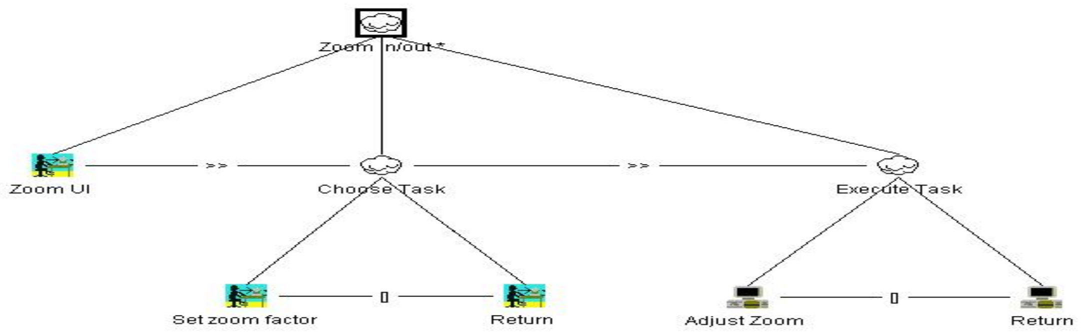


Figure 13: Zooms in/out the system

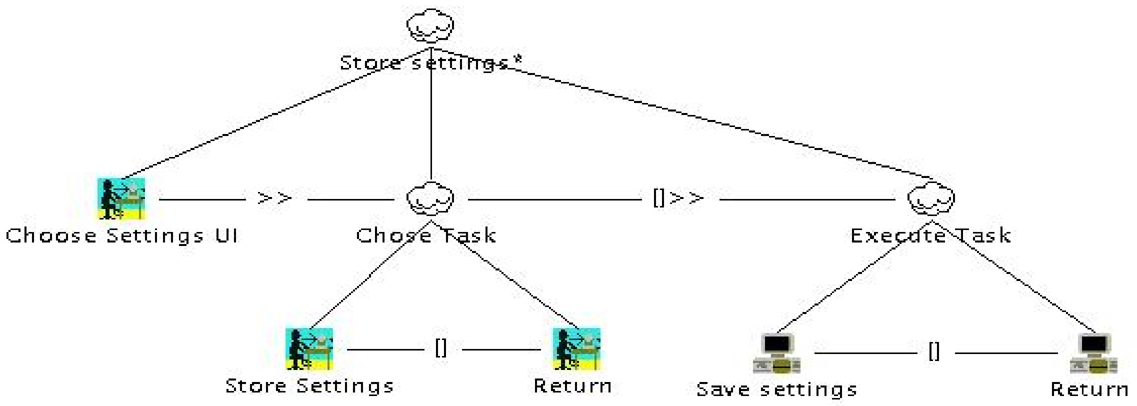


Figure 14: Store settings

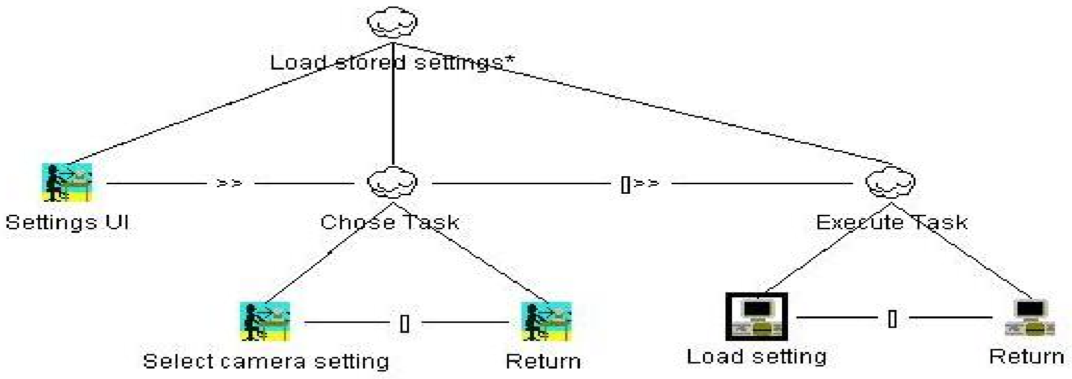


Figure 15: Load stored settings

6. Architecture

This chapter gives a description of the architecture of the SCSS. The architecture is described by means of *blueprint* and *instance diagrams*. A detailed definition of these diagrams can be found in [14].

Blueprint and instance diagrams do not prescribe any particular physical distribution of the components. Distributing these components is done at *system deployment* time. The best way to deploy the components is of course dependend on the particular (network) architecture of the site where the SCSS has to be installed.

Reading this chapter is sufficient to understand the design and architecture of the complete system. Chapter 7 will elaborate, in detail, on the different parts of the SCSS, here we will only give a high-level description.

6.1. Overview of the SCSS

Before discussing the architecture, some essential concepts are repeated here:

A **component instance** is an autonomous piece of code that can exchange messages via its **ports instances**. The behaviour of a component instance is described in a **component blueprint**. The specification of a port instance is described in its corresponding **port blueprint**. The port blueprint describes the signatures of the incoming and outgoing messages (**syntactic interface**), the semantics of these messages and the parameters of these messages (**semantic interface**), the sequential order in which the messages have to be sent (**synchronization interface**) and eventual quality of service parameters (**qos interface**).

Component instances can exchange messages with each other when they are connected by means of ‘compatible’ port instances; this connection is called a **connector**. Two port instances are said to be ‘compatible’ if their corresponding port blueprints are compatible. This means that they have to understand the same messages, and they have to adhere to the same synchronization protocol.

Note: it is also possible to connect more than 2 port instances with one connector. However, we will not need this type of interconnection here.

Figure 16 shows a zoomed out view of the complete SCSS architecture. It shows the core parts of the system. The diagram does not show the components yet, these will be shown in subsequent (more detailed) diagrams.

As one can see, multiple cameras, storages, … can exist in the system. The system can be accessed via a number of user interfaces. There is only one controller in the entire system.

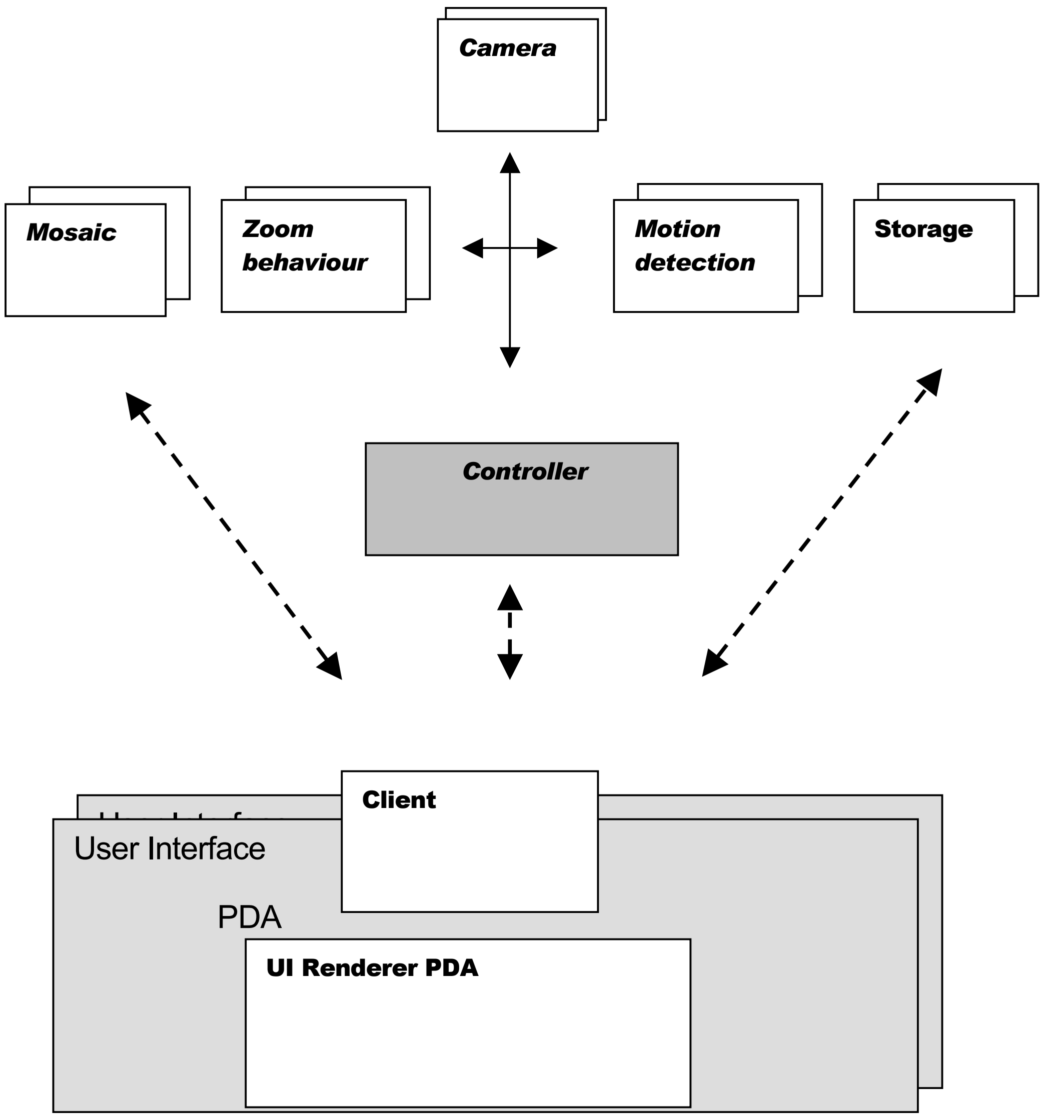


Figure 16: overview of the SCSS core components

The ***camera*** component represents a physical camera in the system. The ***mosaic*** component multiplexes several video outputs from the camera into one new video output. The ***zoom behaviour*** component contains logic to control the zoom of one or more cameras. Next, the ***motion detection*** component analyses the video streams of the cameras and raises an alarm when it detects motion. The ***storage*** component is responsible for the storage of events (eg. alarms) and video sequences.

The figure also shows a ***controller*** component. This component is responsible for the bootup of the system and the management of the

SCSS components. It can also be used by SCSS components to be notified of the activation or deactivation of other SCSS components. In fact, it acts like a directory service.

Next, we have the ***user interface***. The user interface is used to interact with the SCSS from the outside world. There can be different types of devices on which the user interface has to be shown, so there are different types of ***UI Renderer*** components. An UI Renderer component is responsible for the representation of interfaces on a particular device and it is *orchestrated* by means of the *client* component.

A ***client*** component is independent[10] of the particular type of user interface and is used to implement the *application logic*. The application logic decides how a user can interact with the system, and when particular interfaces for devices or services must be shown to the user[11]. In fact, one should see this client component as being the component that implements the behaviour described in the use cases (chapter 5).

6.2. Description of the SCSS components

This section describes the core components of the system by means of blueprint and instance diagrams. For a more elaborate specification of the components we refer to the next chapter. This section can be read if you are not interested in the detailed specification and design of every component.

For each component, a blueprint model is shown that contains this component and all other components that are related to it. Eventually, an instance model will be used to illustrate how these components behave at runtime.

[10] It is likely that all user interfaces will make use of the same client component. The user interfaces can reuse this component since it contains no device specific logic.
[11] That is why we say that this component ‘orchestrates’ the interaction.

6.2.1. Controller

Figure 17 represents the controller component blueprint.

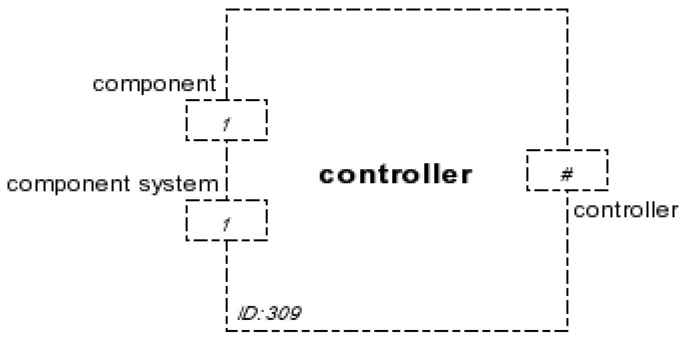


Figure 17: the controller component blueprint

The controller has two ports that are used in relation to the component system: the ***component*** and ***component system*** ports:

- **Component port**: is used by the component system for initializing components. Every component has such a port.
- **ComponentSystem port**: can be used by the controller to be notified of the presence or absence of other component systems.

A controller has also a port that is used in relation to other SCSS components:

- **Controller port**: is used for SCSS management purposes (eg. it checks if SCSS components are still alive). SCSS components can also use this port to receive notifications of join or disjoin actions of other SCSS components.

Important remarks:

Most SCSS components will have a *controller* port, therefore we will not draw this port in subsequent diagrams.

Every SCSS component also has a *component* port and eventually a *component system* port. We will not draw the component port in subsequent diagrams. If a SCSS component makes use of a component system port then this will be indicated.

6.2.2. Camera

Figure 18 represents the camera component blueprint and four additional component blueprints that are related to it.

The ***camera*** component represents the physical camera and has following ports:

- **UIRenderer port**: this port is used by the user interface to retrieve the user interface description for this camera.
- **ZoomRequest port**: this port is used by the camera to interact with the zoom controller (= *zoom behaviour* component). It notifies the zoom controller of zoom requests. The zoom controller can then decide to allow this zoom request or to change this zoom request into a new zoom action (see also 6.2.3.). If no zoom controller is present, then all zoom actions are allowed.
- **Settings port**: this port can be used by an unlimited number of other components for sending zoom, balance, … requests to the camera. It can also be used to retrieve these settings. The settings port will also notify subscribed components of changes in the camera settings.
- **VideostreamDecoderFactory port**: this port is used by other components to obtain a ***videostream decoder*** component. A video stream decoder component is able to decompress (or decode) the video stream of the camera. It can also be used to change the camera settings or to receive notification of changes.
- **VideostreamUpdate port**: this last port is attached to the videostream decoders. It is used to send out encoded image updates. The port has an unlimited multiplicity since there may be several components (and associated decoders) interested in the video stream of this component.

The ***videostream decoder*** component is used by any component that is interested in the video output of a particular camera. It transforms a compressed format in a standard RGBa format. The decoder can also be used to send zoom, focus, … requests to the camera. For each component that is interested in this camera, a videostream decoder is instantiated. To instantiate a videostream decoder for a particular camera, the *videostream decoder factory* port has to be used. The ports of a videostream decoder are:

- **VideostreamUpdate port**: via this port, the decoder receives updates from the camera. These updates are encoded.
- **Settings_out port**: this port is used by the videostream decoder to set the properties of the camera (zoom, balance, focus,…). It forwards requests that arrive at the ***settings in*** port if it can not handle these requests itself.
- **Settings_in port**: the component that makes use of the decoder component, can also make use of this decoder to set the

properties of the camera or to receive notifications of changes in the settings of the camera. As such, a component will never set the properties of a camera directly; it will always use a decoder component instead. The decoder will apply the setting locally if possible. In other cases, the request will be forwarded to the camera.

- **OutputStream**: this port is used to retrieve the RGBa images from the decoder. These RGBa images are generated from the stream of image updates received at the *videostream updates* port.

The ***delay*** component can be used to add a delay in a particular stream. This component has two stream ports: one port where the decoded videostream enters (in RGBa format) and one port where the videostream leaves the component after having experienced a particular delay. The delay can be set via the *delay settings* port.

The ***switch*** component can be used to switch on or off a particular videostream. The component has a port where the videostream enters, and a port where the videostream leaves the component if the switch is in the *on* state. If the switch is in the *off* state, nothing happens: there is no output stream. The switch component has also a port to control the switch state.

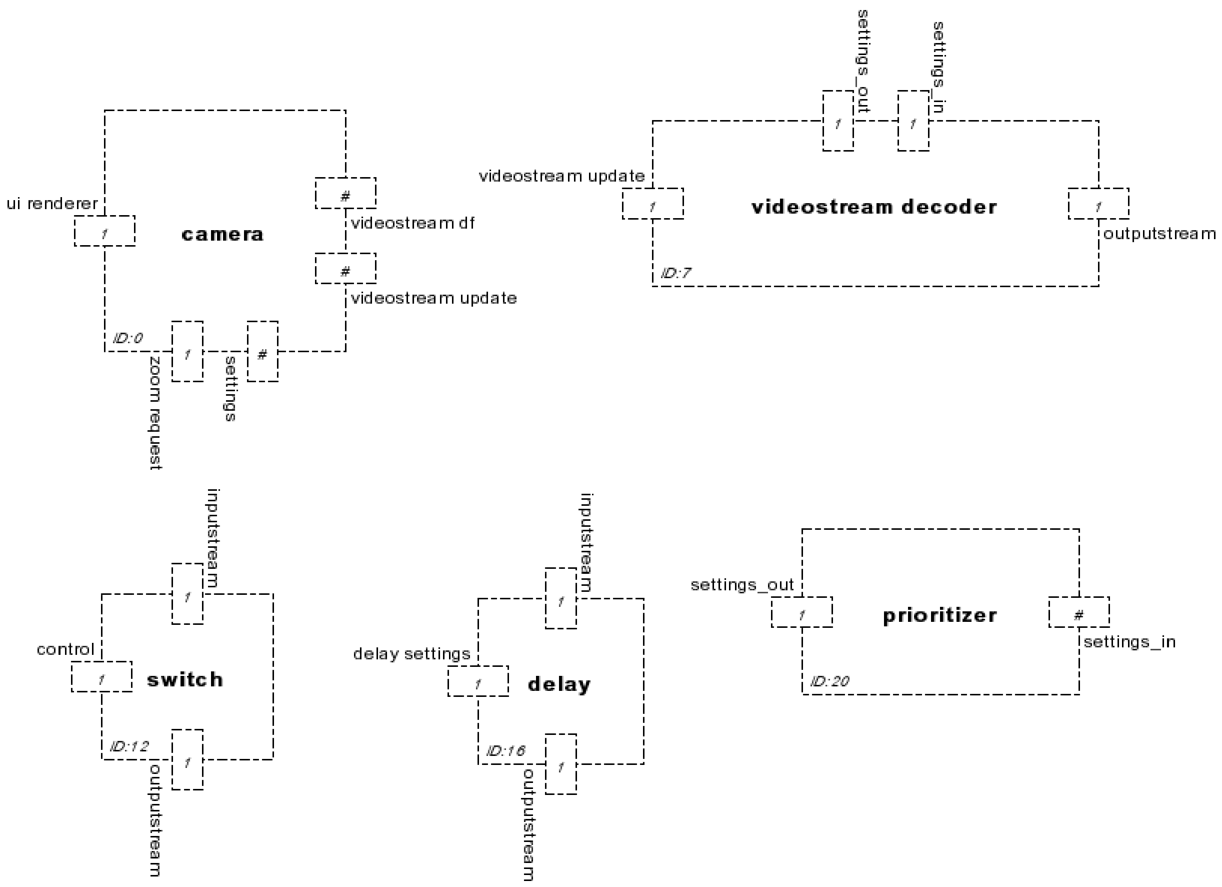


Figure 18: camera and related components

Finally, there is the **prioritizer** component. This component is put between the camera and all decoders for that camera. Sometimes, decoders will send conflicting *setting* requests to the camera (eg. one decoder wants to zoom out while the other wants to zoom in). The prioritizer takes care of these conflicting requests.

The *switch* and *delay* components are 'utility' components: their use is optional. We will show an example of their usage in section 6.2.6.

Figure 19 shows an instance model containing a camera instance that is connected to a video decoder instance. The output of the video decoder is connected to a switch. The *switch operator* component controls the switch and can thus enable or disable the flow of images to the *stream consumer* component. The *switch operator* and *stream consumer* components are undefined at the moment. The prioritizer component is also shown in the figure, but it has no important meaning in this situation since there is only one videostream decoder. We will leave the prioritizer component out in the remainder of the document.

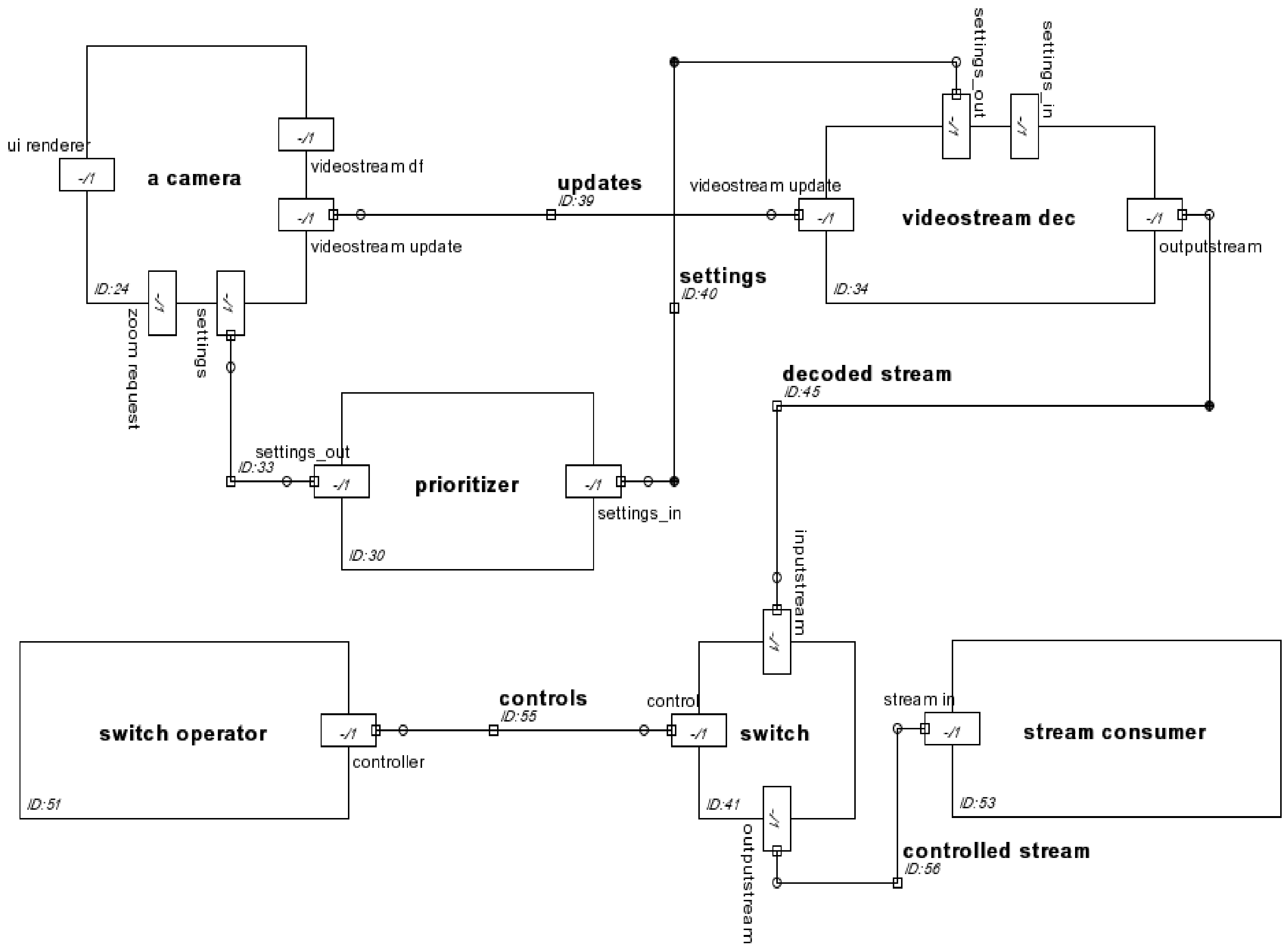


Figure 19: switch operator controls the stream directed to a stream consumer.

6.2.3. Zoom Behaviour

This component is also named the *zoom controller* component. It controls the zoom behaviour of a set of cameras based on a configurable zoom behaviour formula. A possible situation where this component can be used is to control the zoom activity of two camera's that are located in the same room: when the first camera zooms in, then the zoom controller will zoom out on the second camera. This can for instance be used to maintain complete coverage of a particular area in the presence of zoom activity.

Figure 20 shows the zoom behaviour component blueprint.

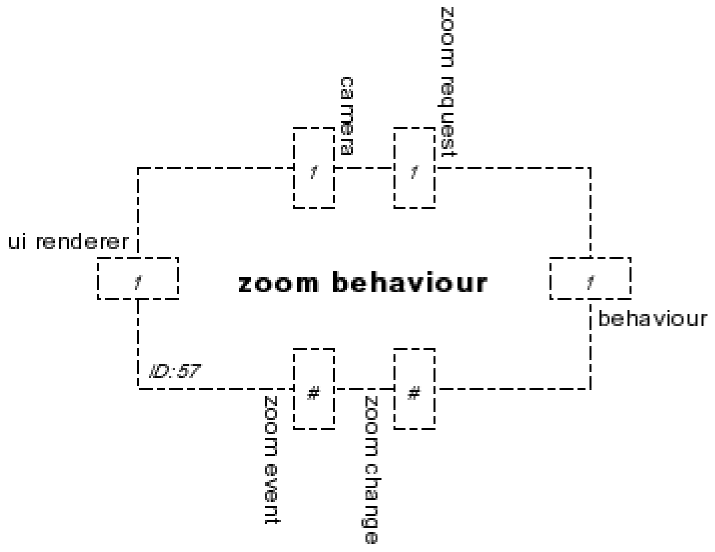


Figure 20: zoom behaviour component

The important[12] ports of the zoom behaviour component are:

- **ZoomRequest port**: this port will be attached to the *zoom request* port of the camera component. The camera can report zoom requests via this port. The controller can use the port to send zoom actions to the camera.
- **Behaviour port**: this port is used by any component that wants to set a particular behaviour formula for the component. The formula is based on a lineair equation. It indicates how the zoom controller should behave in the event of zoom changes on one or more cameras.

[12] Not all ports are shown here: the zoom behaviour component has also a *component* port like all other components.

- **ZoomEvent port**: this port is used to receive notifications of zoom changes on other cameras. Based on these notifications the zoom controller will decide to take action or not.
- **ZoomChange port**: this port is used to send out camera zoom change notifications to other components (eg. the port can be connected to the *zoom event* port of another zoom controller).
- **UIDescription port**: this port is used to output the user interface description of the zoom controller.
- **Camera port**: this port is used to obtain the minimum and maximum zoom parameters of the camera.

6.2.4. Mosaic

The mosaic component combines several video input streams into one new video output stream; all inputs are multiplexed[13] on the output stream.

The mosaic component has two ports:

- **InputStream port**: this port can be attached to every component that generates an RGBa image stream.
- **OutputStream port**: the mosaic component multiplexes all input streams onto a new output stream. This port can be used to read out this stream.

A visual representation of this component is not given here. More information can be found in 7.3.7.

6.2.5. Storage

The storage is also an important part of the SCSS. The storage can be used to store image sequences (video) or events (eg. alarms). The storage can also be used to retrieve image sequences and events.

The storage part of the SCSS consists of two components: the ***storage controller*** and ***storage*** components. The *storage controller* is connected to other components in the system and contains the storage logic, while the *storage* component is responsible for the storage and retrieval of events and images in a database. The *storage* component is not connected to other components in the system, and has no logic associated with it. In fact, it can be seen as an adapter to a database system.

The important ports of the *storage controller* are as follows:

[13] This multiplexing can be done by splitting the available resolution in blocks. An input stream is then downsized to fit in one of these blocks.

- **UI Description port**: this port is used to retrieve the user interface description of the storage subsystem.
- **Video Out port**: this port is connected to a *UI renderer* component for visualisation of recorded image streams.
- **Data Out port**: this port is connected to a *UI renderer* component for visualisation of recorded events.
- **Query In port**: the *client* (see 6.1.8.) component is connected to this port to send queries about events or recorded image sequences to the storage controller. Based on the query, the storage will search the corresponding image sequence or event and output it via its video out and/or data out ports.
- **Event Log In port**: an unlimited number of components can make use of this port if they want to store events on the storage.
- **Video Record In port**: components that generate image streams can be connected to this port if they want to store the images.
- **DB Out port**: this port is used to connect to the *storage* component. It is used as the main link between the *storage controller* and the *storage*.

The one and only port of the *storage* component is the **InOut port**. This port is connected to the *DB out* port of the *storage controller*. It is used to exchange image sequences and events between both components.

Figure 21 shows the storage controller and storage component blueprints.

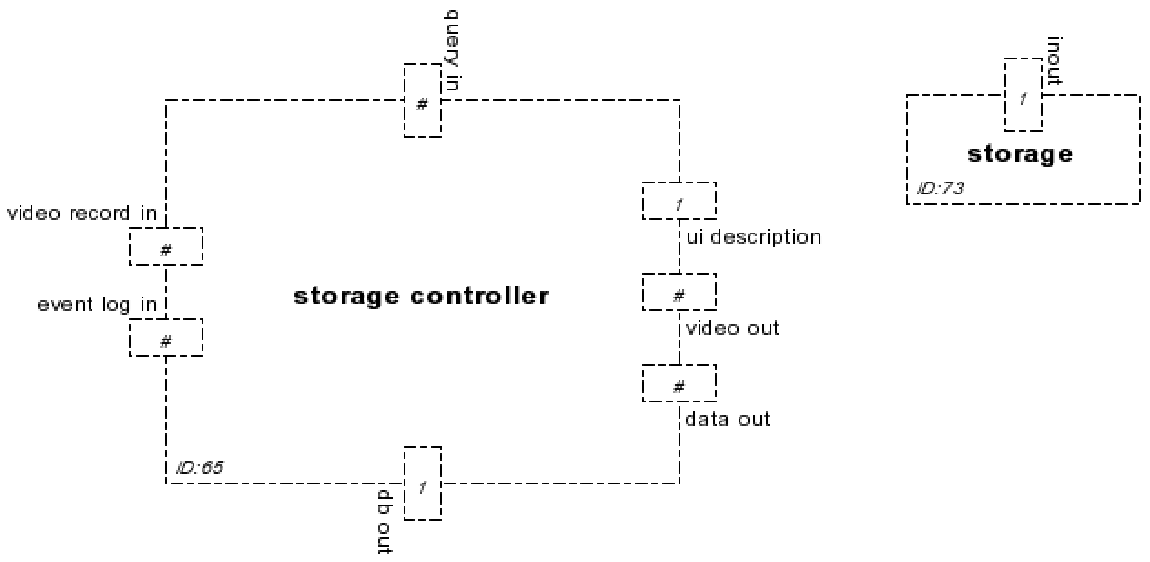


Figure 21: storage controller and storage component blueprints.

Let's illustrate the use of these components by means of an instance diagram: suppose one wants to record two video streams, each one coming from a switch component. There is also another component that generates events that have to be stored on the storage. Figure 22 shows how this can be done.

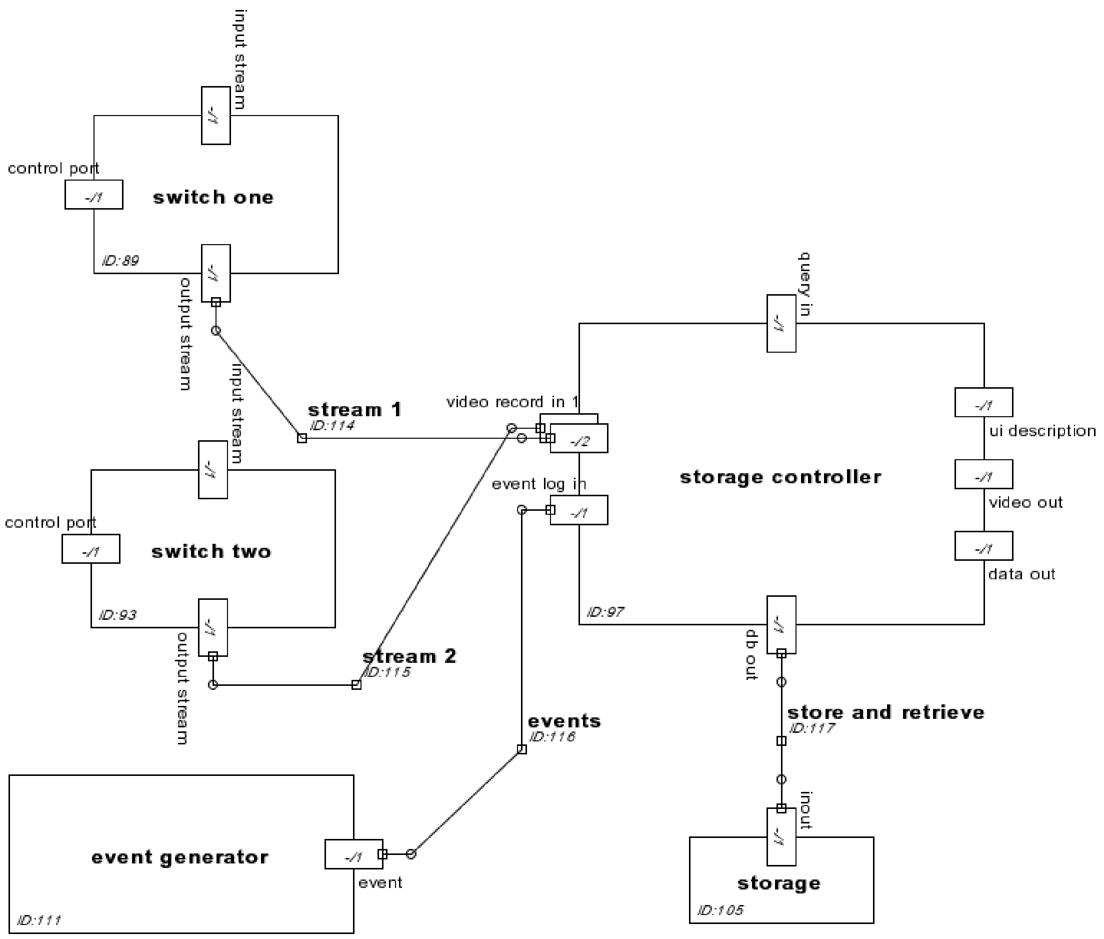


Figure 22: instance model of storage.

The *output stream* ports of both switches are connected to *video record in* ports of the storage controller. The *event* port of the *event generator* is connected to the *event log in* port of the *storage controller*. The *DB out* and *InOut* ports are also connected; this *internal* connector is used by the *storage controller* to store the images from the switches and events from the generator, into the storage database.

6.2.6. Motion Detection

The main task of the motion detection subsystem is to control the image output for a set of cameras and to analyse the image stream for eventual motion. Once motion is detected, this motion event (or *motion alarm*) must be stored on the storage. The motion detection subsystem can also decide to start the recording of the involved camera where motion was detected.

The motion detection subsystem consists of a set of ***camera motion detection*** components: for each camera there will be one such component. The *camera motion detection* component continuously applies a motion detection algorithm on the image stream coming from the camera.

When the component detects motion, it will store a *motion* event on the storage and it will eventually start the recording of the video stream coming from this camera.

The ports of the *camera motion detection* component are:

- **Stream Input port**: via this port it receives the RGBa image stream from a camera (decoder). This stream will be analysed for motion.
- **Camera Settings port**: this port is connected to the *settings* port[14] of the camera. If the zoom status of the camera changes, then the camera will notify[15] the *camera motion detection* component of this event.
- **MD Settings port**: this port can be used to set or get properties of the *camera motion detection* component. Possible properties are: enable/disable motion detection, set hotspot, …
- **UI Description port**: the motion detection component can send a user interface description of itself to interested parties for visuatization.
- **Motion Event port**: this port can be connected to the storage. Every time that motion is detected, a *motion detected* event will be sent through this port.
- **Switch Trigger port**: this port can be attached to one or more switches for starting or stopping a video stream.

Figure 24 shows the *camera motion detection* component blueprint.

[14] It is also possible to define a port that only listens to zoom events from a zoom behaviour component (see 7.2.). But this would imply that every camera needs a zoom behaviour (which is not the case).
[15] The motion detection process will stop during the zooming period. This is important, since zooming will result in a huge amount of motion alarms.

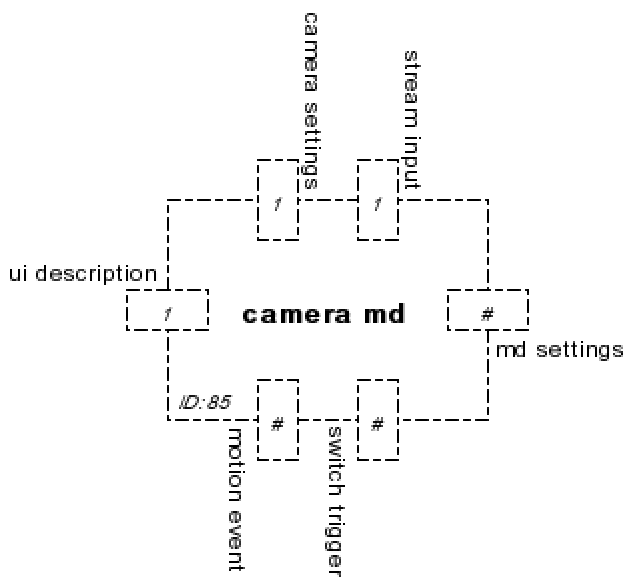


Figure 24: the camera motion detection component.

How will this component be used? A possible usage pattern for this component is to let it record a particular video stream when motion is detected.

A (wrong) solution is the folowing: Use a switch component to enable or disable a particular stream. The stream output of the switch is connected to the storage *video record in* port. When motion is detected, the motion detection component can trigger the switch to the *on* state. At that moment, the stream flows to the storage.

There is a problem with this approach: the motion detection component needs some time to process images and detect motion. If motion is detected, the switch will be triggered, but then it will be too late: the images containing the motion are already gone. Therefore, a *delay* component is put between the camera and the storage. This delay component buffers the stream for some time. The delay duration must of course be longer than the time it takes to perform the motion detection process[16]. Using a delay component ensures that the images containing the important information are stored on the storage.

Figure 25 shows an instance diagram with a *camera* component that is capturing images and sending these images to two *videostream decoder* components. The first videostream decoder is connected to the *camera motion detection component.* The stream coming from this decoder is thus the one that is being analyzed. The second videostream decoder is connected to a *delay* component. The delay value of this component is chosen larger than the processing time needed by the *motion detection* component. The *delay* component is connected to a *switch* that is

[16] This situation could be represented by means of a timing contract. It can however not be represented with a basic timing contract (for more information about timing contracts, see [14]).

operated by the *camera motion detection* component. Finally, the stream output of the switch is connected to the *storage controller* for recording the images.

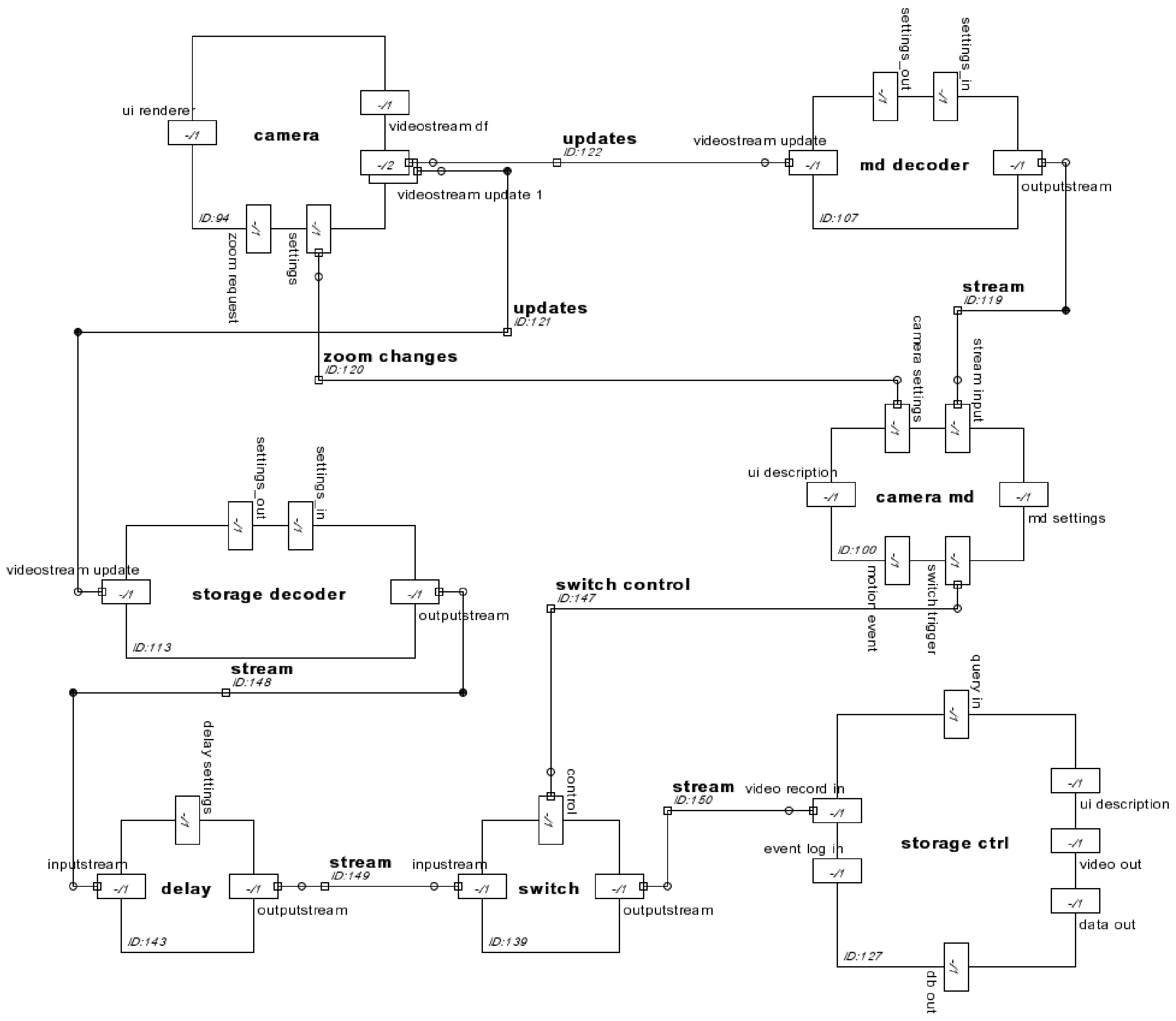


Figure 25: a camera motion detection component that controls a switch

Notice that the *camera motion detection* component is also attached to the *camera* component. When the camera is zooming, the camera motion detection will stop processing the images until the zooming has ended.

To summarize … The resulting behaviour, of the example given above, will be the following:

- *When there is motion, then the videostream will be recorded (including the images containing the motion).*

- *When no motion is detected, then no videostream is recorded.*

The main advantage of this approach is the fact that images are only recorded when there is motion. This reduces the amount of needed storage space.

6.2.7. Client

It is not a good idea to put the application logic in the user interface. Therefore, we make use of a ***client*** component. This *client* component contains the application logic for the SCSS and is not bound to any particular type of device. As was discussed earlier (in section 6.1.), this client component orchestrates the interaction between a user and the system.

The client component is connected to all SCSS components that have a port that can provide a user interface description. It is also connected to the *storage controller* by means of a query port. This is done to let the user perform queries on the storage via the client. At last, the client is also connected to the *UI Renderer* component. Via this binding, the client tells the *UI Renderer* component what to show and the *UI Renderer* informs the *client* from user interface events.

The *client* has following ports:

- **UIDescription port**: this port can be connected to all SCSS components that have to visualize their user interface.
- **QueryStorage port**: via this port, the client sends queries to the storage controller. These queries are initiated by the user.
- **UIOut port**: this port is connected to the *UI Renderer* component. It is used to send user interface descriptions for visualization.
- **EventIn port**: via this port, the client receives user interaction events that occur in the interface (eg. clicking on a button).

To summarize: the UI Renderer visualises the interface and the client tells the UI Renderer when to visualize something (based on client actions).

Figure 26 shows the client component blueprint.

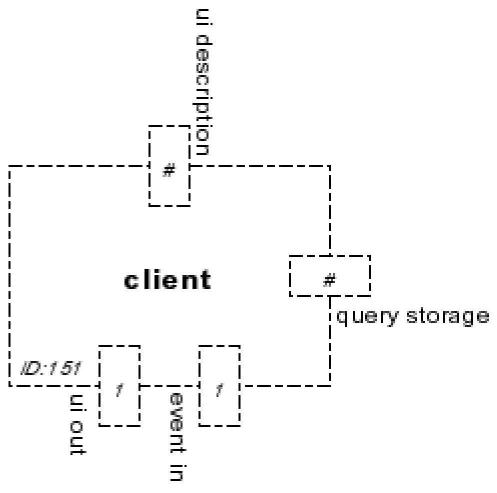


Figure 26: the client component

6.2.8. User Interface

The user interface is an important part of the SCSS. Depending on the type of device and device output capabilities, the user interface will be represented in different ways. As such, there will exists several user interface renderers. But all user interfaces will make use of the same application logic (the *client* component).

The ***UI Renderer*** component is responsible for rendering the user interface descriptions of SCSS components on the screen. As input it takes an XML description and renders it in a way that is dependent on the device output capabilties. This rendering constrains itself to UI widgets (buttons, lists, …). The UI Renderer can also be connected to data ports for extra visualisation possibilities that go beyond widget visualisation[17].

The UI Renderer has following ports:

- **UIDescription Requires port**: this port is connected to components that can export user interface descriptions (eg. the client component). These components can send XML descriptions that the renderer must visualise.
- **DataIn Provides port**: this port can be connected to all components that have some data to visualize. For instance, the storage can send event data to the renderer, like video streams.

[17] This can for instance be used to show streaming video on the screen.

- **EventOut Provides port**: this port will be connected to the *event in* port of the client component. It is used to report user interaction events (eg. a button that has been clicked) to this client.
- **UIStateDescription Provides port**: see 7.6.2.2.
- **PutUITree Provides port**: see 7.6.2.2.
- **PutUITree Requires port**: see 7.6.2.2.

The last three ports will not be used in the examples here, so we refer to section 7.6.2.2. for more information about them. Figure 26 represents the ui renderer component.

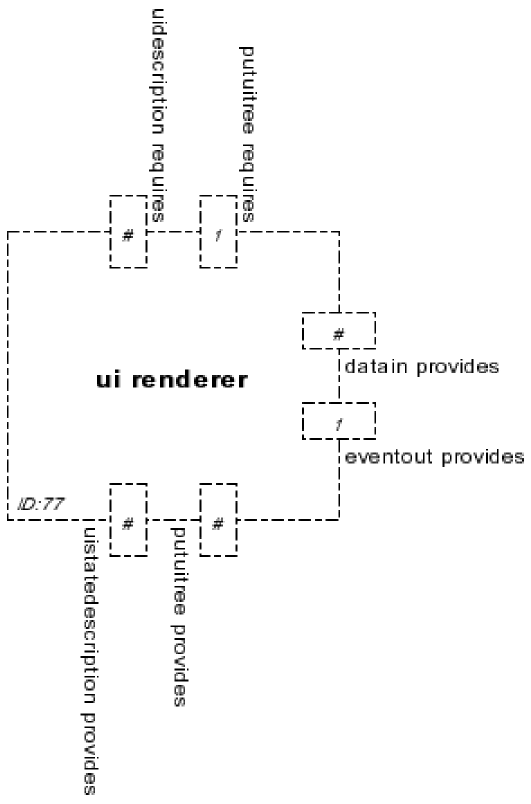


Figure 26: UI renderer component blueprint

Figure 27 illustrates the use of a *ui renderer*, *client* and *storage controller* component. The client is connected to SCSS components that can export a user interface (*component 1*, *component 2* and the *storage controller*), via their *ui description* ports. It is also connected to the storage controller via a *query storage* port. The *ui renderer* component is connected to the *ui out* and *event in* ports of the client. It is also connected to the storage *video out* and *data out* ports.

The *client* component tells the *ui renderer* component what to show. This is done via the *ui out* port. The *ui renderer* reports user interface events to the *client* component (via the *event in* port), who can then decide to render another interface or to perform any other action (eg. changing a camera setting). If the user wants to query some images or events from the storage, then the *client* will issue these requests via the *query storage* port. It is clear that the *client* plays a central role.

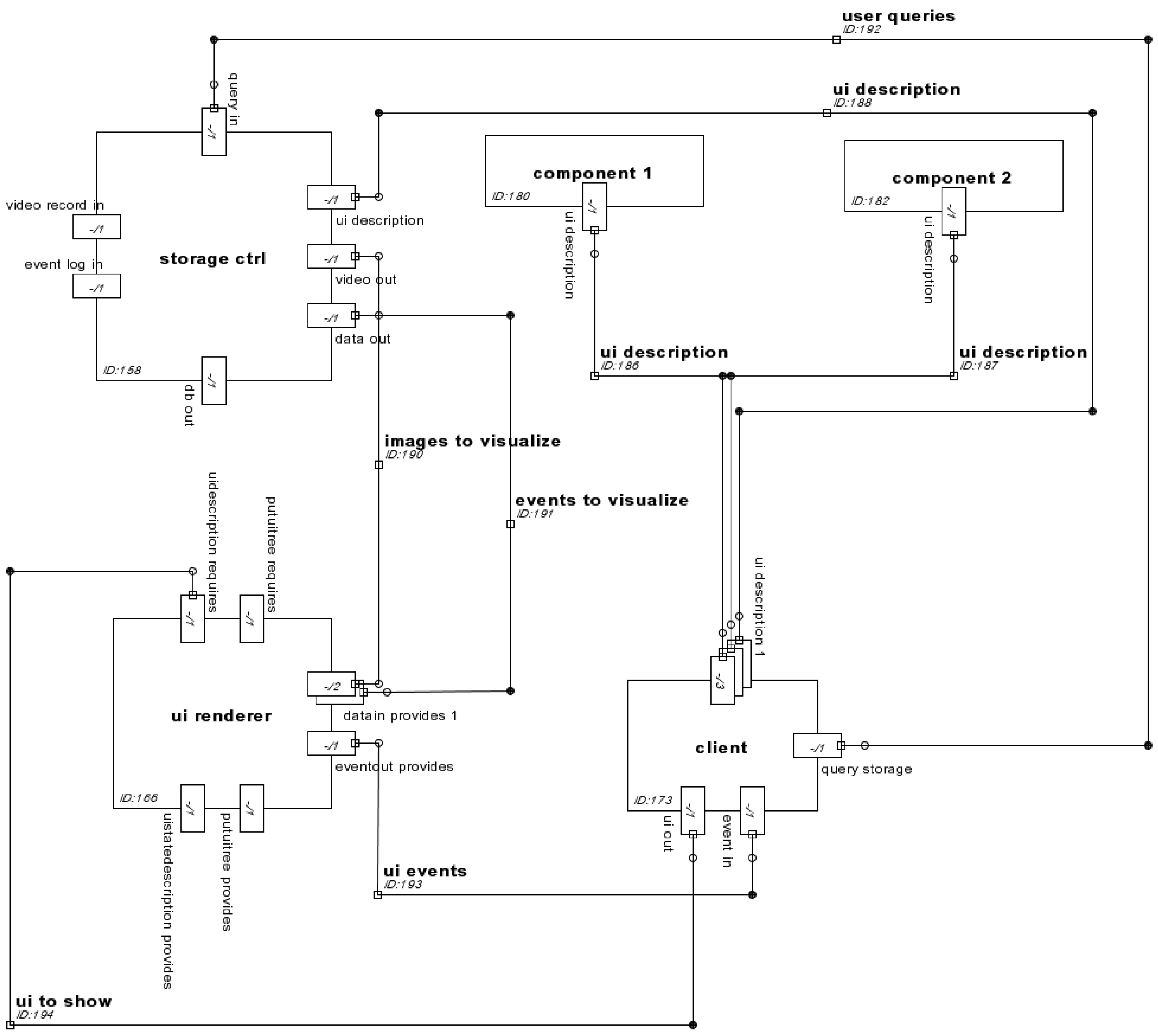


Figure 27: interconnection of client, ui renderer and storage controller.

6.3. Example

The previous section discussed the different types of components that can be used to make up the entire system. Every component was described on its own, sometimes its use was illustrated by means of an instance diagram.

This section will show some instance diagrams for a particular SCSS configuration (without deployment information). Some details will not be shown if they do not add any information[18].

6.3.1. Cameras

First of all, the example SCSS consists of two *cameras*. Both cameras have each a *zoom behaviour* component that is controlling their zoom behaviour. Both *zoom behaviour* components are connected to each other:

The behaviour of both zoom behaviour components is as follows: "When a camera zooms out for x % then the other camera has to zoom in for x % and vice versa". As such, each zoom behaviour component listens to zoom events coming from the other zoom behaviour component. That is why both components are connected to each other. Figure 28 shows the situation.

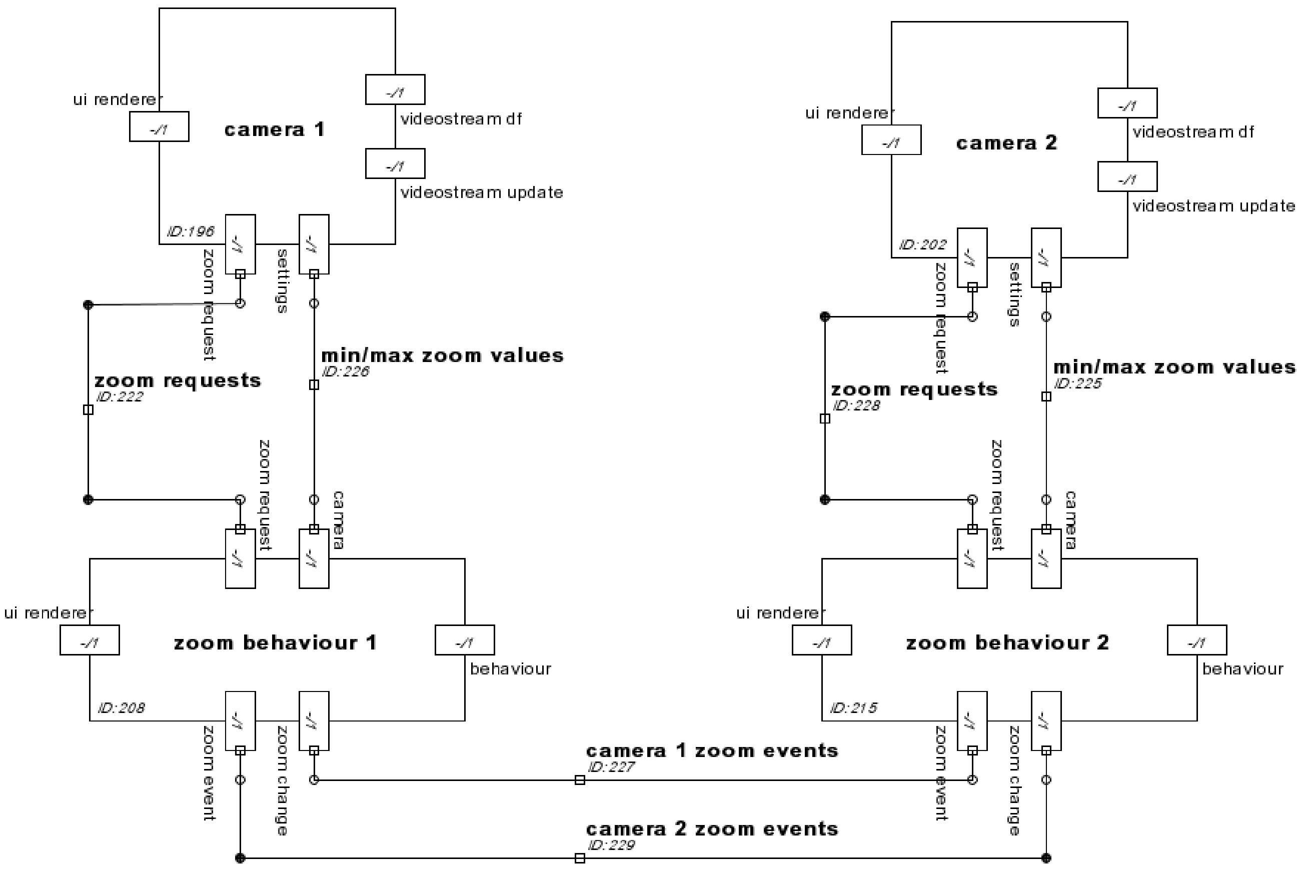


Figure 28: the SCSS consists of 2 camera and 2 zoom behaviour components

[18] All SCSS components have a controller port that is attached to the controller component and a component port that is attached to the component system component. We do not draw these components and ports in order to not overwhelm the diagrams with information.

Every zoom behaviour component receives zoom requests from its associated camera on its *zoom request* port. It can then decide if the request should be granted or not. If the request is granted, the camera is informed of this and can start zooming. The zoom behaviour component that granted the zoom request will send a zoom event to the other zoom behaviour component. This last component will then send an opposite zoom action to its camera.

The zoom behaviour components are also connected to their camera via the *camera settings* port. This is necessary since the zoom behaviour components need to know the minimal and maximal zoom values of the camera.

6.3.2. Storage

The example SCSS has a *storage controller* component (and associated *storage* component, but we will not show it here). The storage controller component is connected to video stream decoders for the cameras through delays and switches. *Decoder 1* is a VideoStreamDecoder for *camera 1* and *decoder 2* is a VideoStreamDecoder for *camera 2*. The switch components enable the control of the video streams coming from the decoders. These will be connected to the camera motion detection components (section 6.3.3.). The delay components slow down the video streams coming from their *inputstream* ports. The reason why we have put delay components in this diagram was already explained earlier (section 6.2.6.)

Figure 29 shows the storage controller component and all the switches and delays between the decoders and the storage controller.

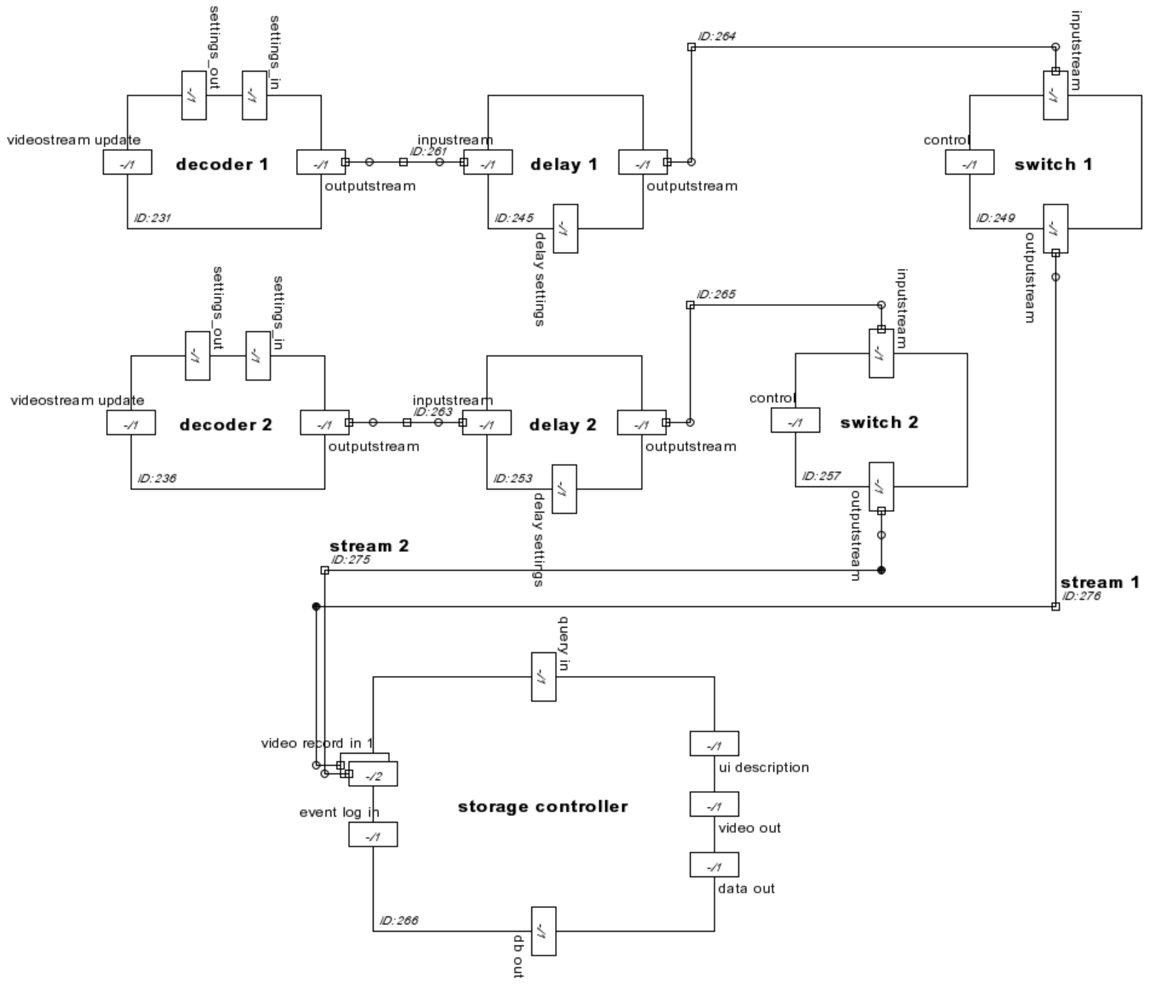


Figure 29: the video output from both cameras is routed to the storage.

6.3.3. Motion Detection

The SCSS consists of two *camera motion detection* components. Each camera motion detection component analyses the output of one camera. To do this, they also use a VideoStreamDecoder component.

If motion is detected, then this is reported to the storage by sending a motion detected event. The *camera motion detection* components also control the switches shown in figure 29: if motion is detected, then these switches are opened. If no motion is detected, then the switches are closed.

Figure 30 shows the situation for one camera motion detection component (namely for camera 1). As one can see, the camera motion detection component is connected to the *storage controller* via its *motion event* port. If motion is detected, then the camera motion detection component will send a motion event to the storage. The camera motion detection component is also connected to the *control port* of the *switch*, via its *switch trigger* port. This enables the motion detection component to open or close the switch in order to start or stop the recording of the images.

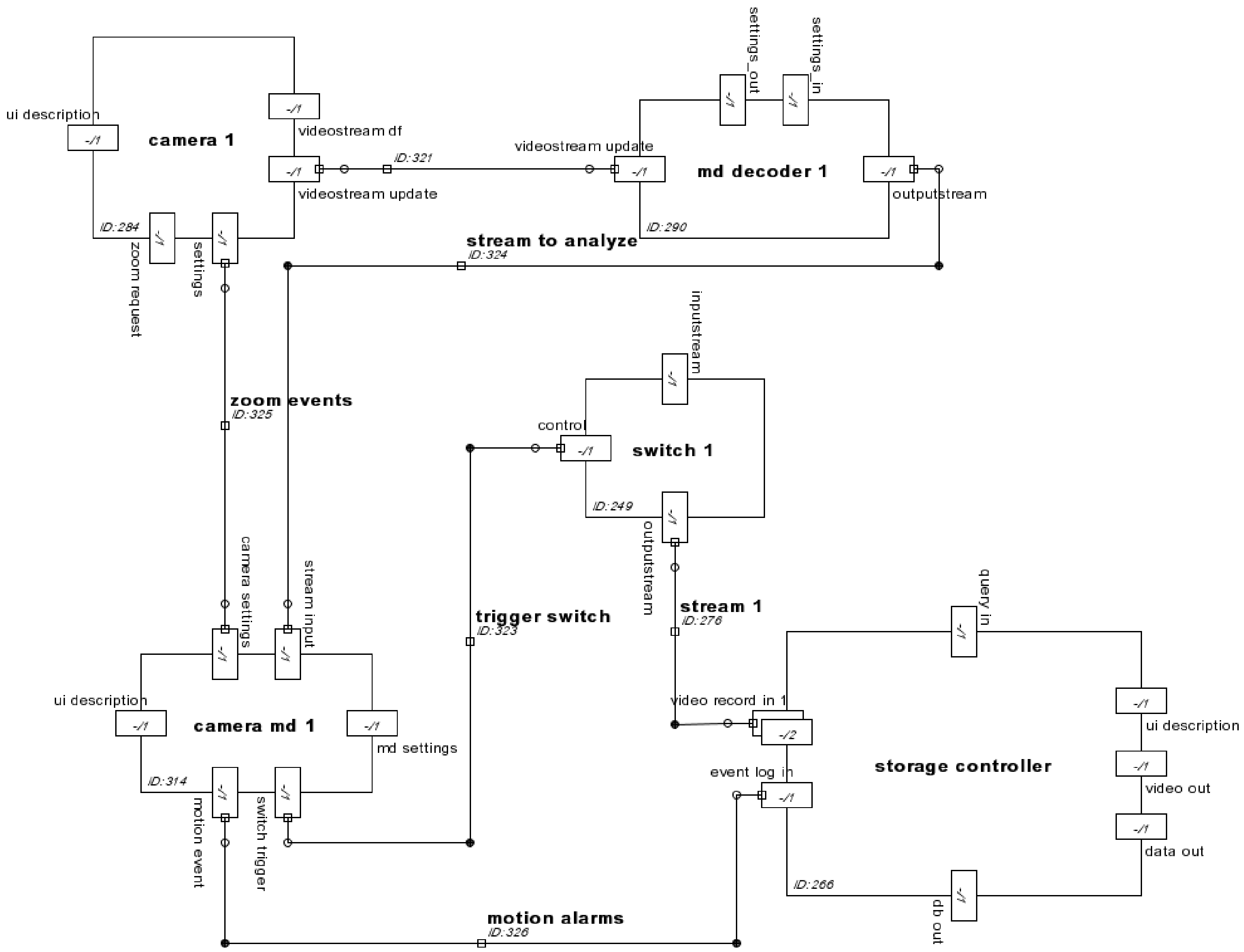


Figure 30: motion detection for camera 1

The *switch 1* and *storage controller* controller are the same components as in figure 29. Recall that it is possible to reuse component instances in different models. Connectors from figure 29 are visible if the components to which they are attached are also visible (this is the case with the *stream 1* connector).

6.3.4. Mosaic

Another important component is the *mosaic* component. This component combines the stream outputs of both cameras into one new video stream Figure 31 is an illustration of this mosaic component.

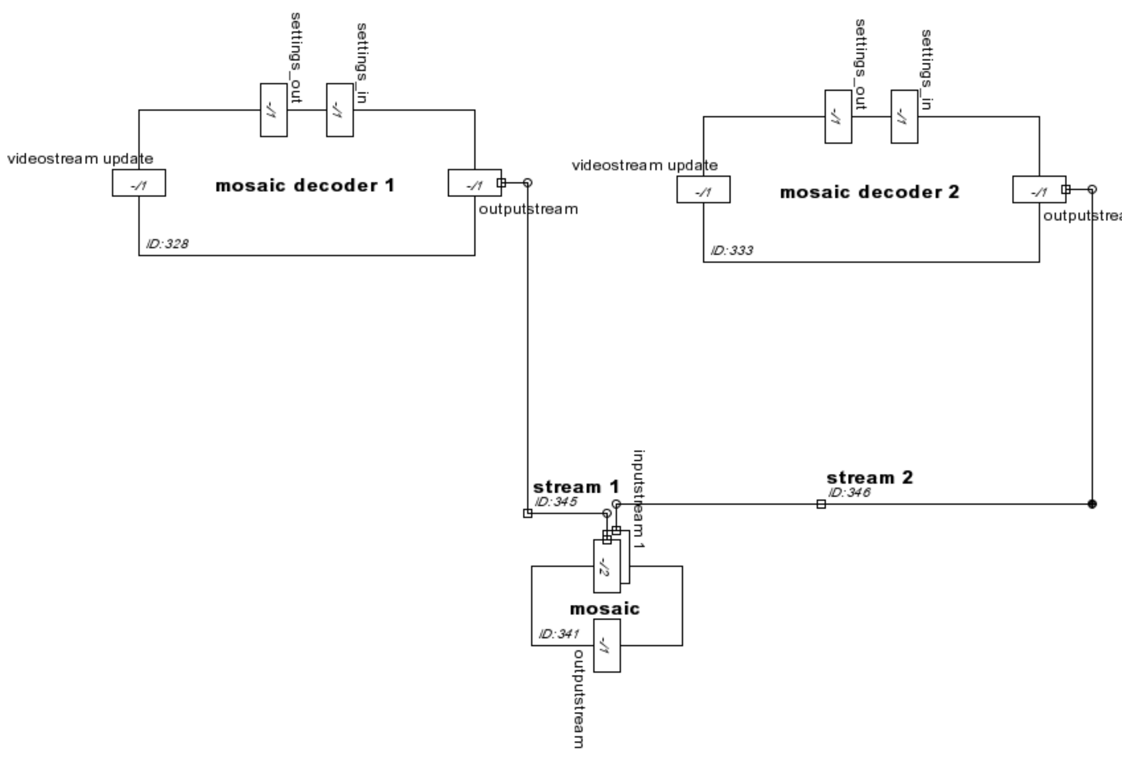


Figure 31: the mosaic component combines the streams of both cameras.

6.3.5. Client and UI Renderer component

Recall that the *client* component is responsible for the user interaction orchestration and that the *ui renderer* component is responsible for the rendering of user interfaces (on a screen for instance).

The client sends user interface descriptions to the ui renderer (via the *ui out* port) and is connected to all SCSS components that can export a user interface (via the *ui description* port). In the example, several components have a user interface: *camera 1, camera 2, camera md 1, camera md 2, storage controller, zoom behaviour 1* and *zoom behaviour 2*. We will only connect the client to the cameras and the storage controller; this enables us to visualize the user interface of the cameras and the storage. Other user interfaces will not be visualized in the example[19].

The client is connected to the storage controller via its *storage query* port. This means that a user is able to perform queries on the storage.

The ui renderer is connected to the *video out* port of the storage. This enables the ui renderer to show stored images. Finally, the renderer is

[19] In real situations this will not be the case: everything that should be visualized must be connected to the client.

connected to the *data out* port of the storage, to render events stored in the storage (like motion alarms). Figure 32 illustrates this.

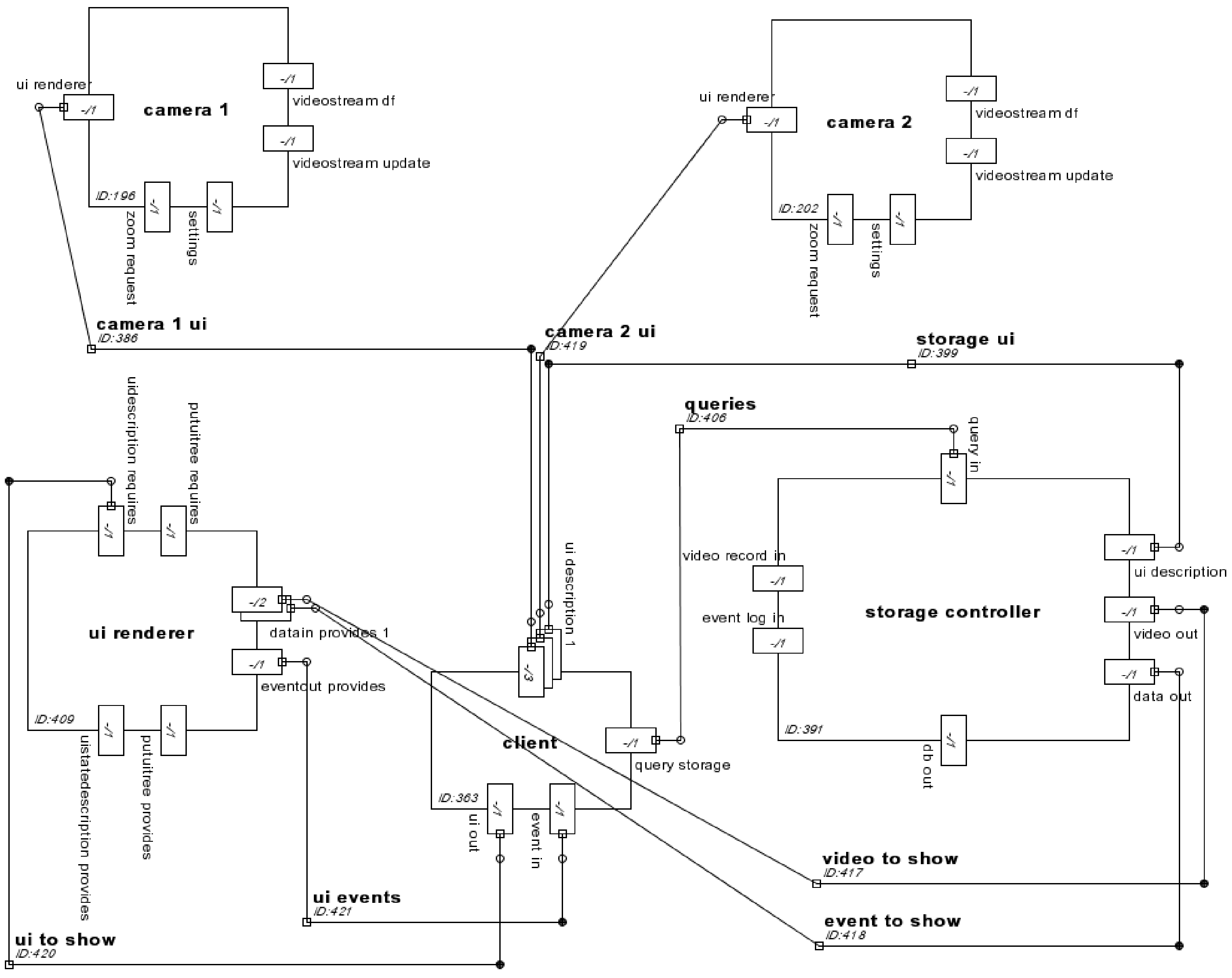


Figure 32: client orchestrates the user interactions.

The configuration shown above allows one to:

- Visualize the user interface of the cameras.
- Visualize the user interface of the storage.
- Visualize recorded video and events.
- Submit queries to the storage.

It is clear that this is only basic functionality. We also need to add a possiblity to show live video streams from the cameras, mosaic output, visualize the motion detection component, and so on.

7. Component Specification

In chapter 6 the core components of the SCSS were introduced and a short description of their behaviour was given. In this chapter, the core components are specified in more detail: context information is given, the interfaces are described formally and use cases are included to give information about their use.

The chapter is subdivided as follows:

Section 7.1: the **controller** component.

Section 7.2: the **zoom behaviour** component.

Section 7.3: the **image generating** components.

Section 7.4 the **storage** and **storage controller** components.

Section 7.5: the **camera motion detection** component.

Section 7.6: the **UI renderer** component (also see [15]).

Section 7.7: the **client** component.

The *Component Composer Tool* deliverable ([14]) handles about an XML format that can be used to specify the components. The XML format is not applied to the remainder of this chapter; a more informal descriptive language is used instead. XML specifications would make the specifications difficult to read[20].

[20] For a human.

7.1. Controller Component

This section describes the design of the Controller component. This component takes care of setting up interconnections between components.

7.1.1. Description

The controller his functionalities are described as follows:

- If new components connect or reconnect, all interested components will be notified.
- If components die or disconnect, the interested components will be notified.
- The controller takes care of distribution errors.
- The controller takes care of booting the system.
- When new component systems connect to the component infrastructure, the controller will upload the necesarry components to the target.

The controller is *not* responsible for the following:

- The controller does ***not*** handle runtime events and does not synchronise between components, nor does it contain any application logics.
- The controller is ***not*** responsible for security, account management, nor choosing between different events.

The controller in the second period of this project (year 2 to 4) can be extended to

- Offer component migration
- Remote update of components

7.1.2. Use Cases

7.1.2.1. Boot up

The boot up of the camera system contains two phases. The first phase covers the startup of the master-server, which also boots the controller. In the second phase, all cameras are looked up and new component systems can connect.

Phase 1: Master Component System Boot up

- [MCS]starts up with on the command line as only argument the controller component
- [MCS]loads the controller component
- [C]reads the initialisation file which contains a list of components to load
- [C]sends a create message to the MCS for all listed components
- [MCS]loads all components and initialises them. Normally, these component will not make connections with other components.
- [C]reads from the initialisation phase which components should connect to whom.
- [C]sends 'faked' Connect messages to all those components.

Phase 2: Client boot up

- [CCS]starts up with only the IP-address of the Master Component System on the command line.
- [CCS]connects to the MCS on the given address. Also connects to the Master Component System component.
- [MCS]accepts the connection, sends out a connect message to all interested parties (In this case, the controller)
- [C]looks up which components should exists at the given address.
- [C]sends out creation messages to the CCS for theses components
- [CCS]creates the components (loads the classes from the server)
- [C]after creation of the new components, the controller will send the correct 'faked' connect messages.

7.1.2.2. Component Join & Disjoin

Components can subscribe themselves to retrieve notification of certain new components within the system. For example, a user interface component would like to subscribe to new cameras. The user interface component will receive from the controller a connect message when a new camera joins (or is created).

Disconnecting component can be initiated from anywhere. Every delete command must be send to the controller, which will ask all components to disconnect themselves. If they don't the controller will take action. We will now illustrate connecting and disconnecting

Connecting

- [UI]is loaded and requests the controller a join for 'cameras'
- [CC]The camera component is loaded (by the controller)
- [C]Sends a HasJoined message to the UI
- [UI]Sends a connect message to the camera
- Disconnecting
- [UI]wants to disjoin, sends a message to the controller
- [C]sends a disconnect to the UI
- [UI]disconnects all its ports

7.1.2.3. Unwanted Component Disconnection & Component System Disconnection

At the moment a component disconnects because there was an error (crash in the component) or a network failure, all dependent components will be notified with a HasDisjoined message. The messages which trigger such an action are

- ComponentSystemDisconnect
- ComponentSystemFailed
- ComponentFail
- ComponentDisconnected

7.1.3. Interfaces

7.1.3.1. port Component (noi=1)

- [**in Init()**] is send by the component system. This is guaranteed the first message a component will receive
- [**in Connect**(**Port:<String>**, **With:<String>**)] connects the sender his port With with the port Port of the receiver.
- [**in Disconnect**(**Who:<String>**, **From:<String>**)] disconnects Who from port From at receiver side

7.1.3.2. port ComponentSystem (noi=1)

- [**in ComponentSystemDisconnect(ComponentSystem)**] see description of the ComponentSystem component[21]
- [**in ComponentSystemFailed(ComponentSystem)**] see description of the ComponentSystem component
- [**in ComponentSystemConnect(ComponentSystem)**] see description of the ComponentSystem component
- [in ComponentSystemQueueOverflow(ComponentSystem, Reason)] see description of the ComponentSystem component
- [**in ComponentDisconnect(Component)**] see description of the ComponentSystem component
- [**in ComponentConnect(Component)**] see description of the ComponentSystem component
- [**in ComponentFailed(Component)**] see description of the ComponentSystem component
- [**out CreateComponent(BluePrint,Name)**] is used to create all the necesarry components

7.1.3.3. multiport Controller (noi=1)

- [**in LookingFor**(**NameSubstring)**] Requests the controller to look for components with a name which contains the substring Name. In response to this message, the controller will send back all existing components and will from then on notify the requester of new components mathcing the given name.
- [**out HasJoined(Who)**] is send to notify everybody that Who has joined. This message is only send to all people subscribed to the given NameSubstring.
- [**out HasDisjoined(Who)**] is send as a notification of a disjoin. Is send only to subscribed components.
- [**out Alife(Who)**] is send out to check whether other component systems are still alive.
- [**in AreYouAlife**()] response of previous message

[21] In deliverable xyz

7.1.4. Message Traces

7.1.4.1. Master Component System Boot Up

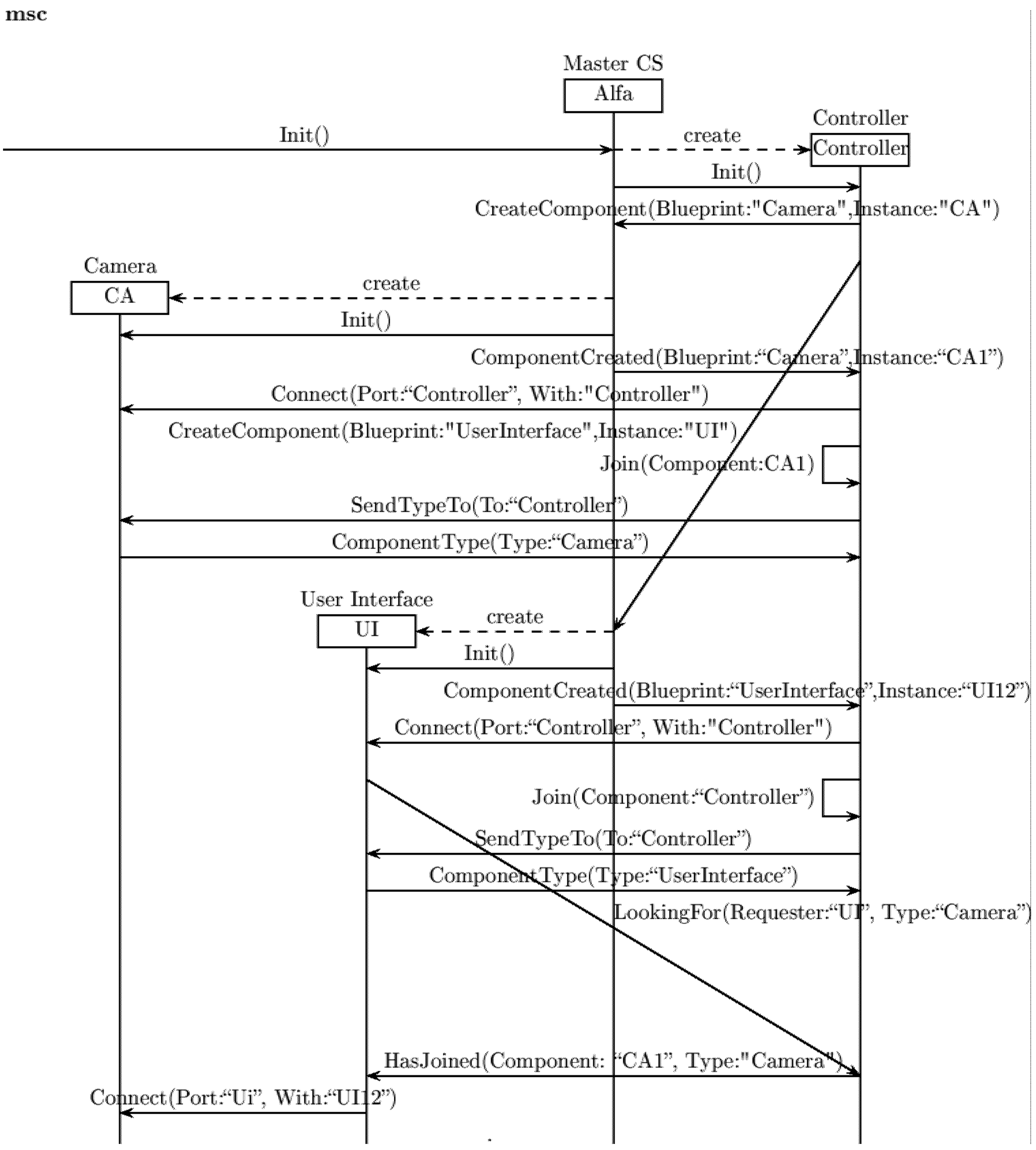

msc
Master CS
Alfa
Controller
Controller
Init()
create
Init()
CreateComponent(Blueprint:"Camera",Instance:"CA")
Camera
CA
create
Init()
ComponentCreated(Blueprint:"Camera",Instance:"CA1")
Connect(Port:"Controller", With:"Controller")
CreateComponent(Blueprint:"UserInterface",Instance:"UI")
Join(Component:CA1)
SendTypeTo(To:"Controller")
ComponentType(Type:"Camera")
User Interface
UI
create
Init()
ComponentCreated(Blueprint:"UserInterface",Instance:"UI12")
Connect(Port:"Controller", With:"Controller")
Join(Component:"Controller")
SendTypeTo(To:"Controller")
ComponentType(Type:"UserInterface")
LookingFor(Requester:"UI", Type:"Camera")
HasJoined(Component: "CA1", Type:"Camera")
Connect(Port:"Ui", With:"UI12")

Figure 33: master component system boot up

7.1.4.2. Camera Component System Boot Up

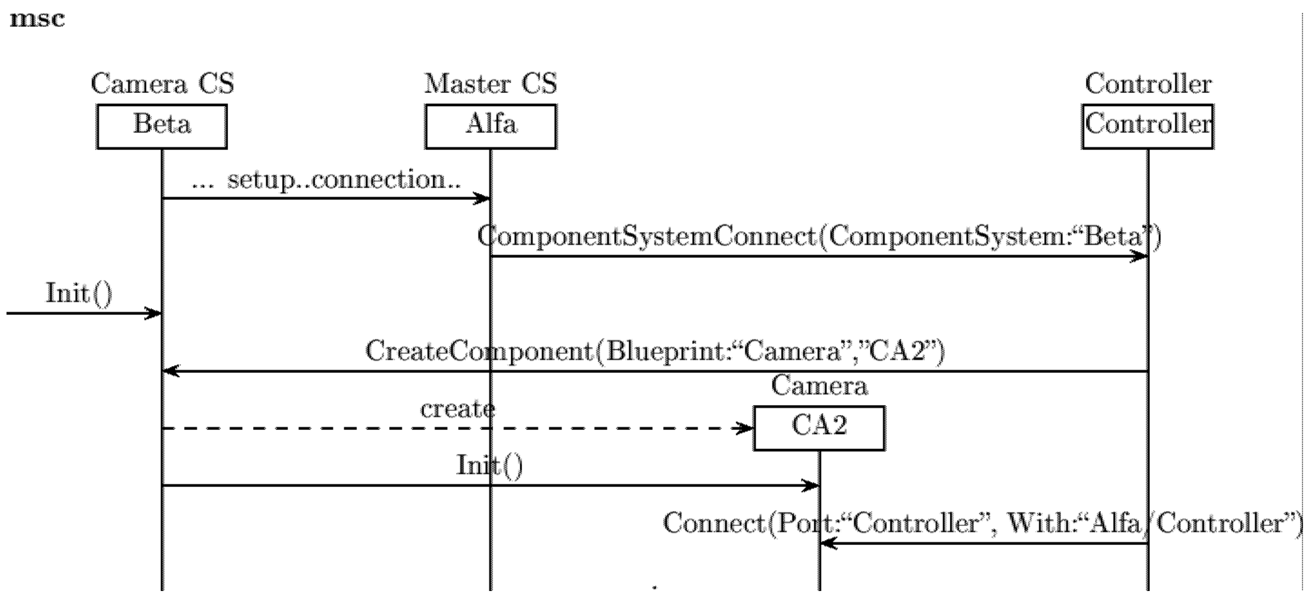

msc
Camera CS
Beta
Master CS
Alfa
Controller
Controller
... setup..connection..
ComponentSystemConnect(ComponentSystem:"Beta")
Init()
CreateComponent(Blueprint:"Camera","CA2")
Camera
create
CA2
Init()
Connect(Port:"Controller", With:"Alfa/Controller")

Figure 34: camera component system boot up

7.2. Zoom Behaviour Component

This section describes the design of the CameraZoomBehaviour component. This component takes care of zooming cameras and coupling the zoom behaviour of a number of cameras.

7.2.1. Description

This is one of the plug in components, which is added to the system to show its flexibility.

- The zoom behaviour component is able to notify all subscribed components of a zoom changed notification.
- Zoom behaviour components can be coupled such that one camera zooms in while another zooms out and vice versa.

7.2.2. Use Cases

7.2.2.1. Logging of zoom-events

It is possible to log zoom events. If we want this we should connect to the zoom behaviour zoom changed port.

7.2.2.2. Selection of zoom events

It should be possible to select (prioritise) between a number of camera zoom events. If for example, 2 users are zooming in at the same time, together with an operator on the camera, the zoom behaviour controller should select one suer and stick to him during a certain period of activity.

7.2.2.3. Coupling of Zoom Behaviours

It should be possible to couple two (or more) cameras, such that the first camera zooms in while the second one zooms out. The constraints placed upon the zoom will be simple linear equations.

7.2.3. Interfaces

7.2.3.1. port Component (noi=1)

- [**in Init()**] is send by the component system. This is guaranteed the first message a component will receive
- [**in Connect**(**Port:<String>**, **With:<String>**)] see ComponentSystem component
- [**in Disconnect**(**Who:<String>, From:<String>**)] see ComponentSystem component

7.2.3.2. port Camera (noi<=1)

We will use a port to the camera status to obtain information about the maximum zooming parameters. This port is temporary. After obtaining the values this port is not used anymore.

- [**out GetMax**(**Type**:**<String>**)] requests the maximum zoom
- [**out GetMin**(**Type**:**<String>**)] requests the minimum zoom
- [**in Value**(**Type:<String>, Value:<Integer>**)] answers to both questions above

7.2.3.3. port ZoomRequest (noi=1)

This is the port offered by all cameras, which we have to implement in order to intercept certain zoom events.

- [**in ZoomChangeRequest(From:<String>, Value:<Integer>)**] incoming message from the camera, to ask whether the zoom can be changed to the given value. If this is allowed this component will send back a ZoomChangeAction.
- [**out ZoomChangeAction(NewValue:<Integer>)**] this message is send back to the camera when the zoom should actually change.

7.2.3.4. multiport ZoomChange (noi=1)

This port is necessary for anybody interested in zoom-change events. This is a stripped down interface of the camera-settings port and will be used by other ZoomBehaviour components.

- [**out ZoomChanged(Camera:<String>, Value:<Integer> ,>InitiatedFrom<:[])**] send out whenever the camera reports a change of value of the zoom. Camera is the name of the controlling camera. InitiatedFrom is an array with cameras, which has already send out a ZoomChanged.

7.2.3.5. port ZoomEvent (noi>=0)

This port is necessary for anybody who wants to report a ZoomChanged event at this controller.

- [**in ZoomChanged(Camera:<String>, Value:<Integer>, >InitiatedFrom<:[])**] Camera is the name of the camera for which the zoom has changed to value Value. In response to this message, this camera can change its own zoom factor. InitiatedFrom is an array of names which already sent a ZoomChanged.

7.2.3.6. port Behaviour (0<=noi<=1)

This port is used to change the formule used in each controller to react to ZoomChanged events.

- [**in SetFormule(A:<Integer>[], B:<Integer>)**] The formule is a linear equation. The integer array contains the coefficients. The integer B contains the extra value added to all this.

7.2.3.7. port UiDescription (0<=noi<=1)

This port can be used by anybody to receive an XML description of the behaviour port.

- [**in GetUiDescription()**] received from somewhere. In response we have to send back an XML tree which contains the possible behaviour of this component.

- [**out PutUiDescription(Xml:<String>)**] send out. Will probably contain the parameters of the linear equation.

7.2.4. Example Message Sequences

7.2.4.1. Logging of Zoom events

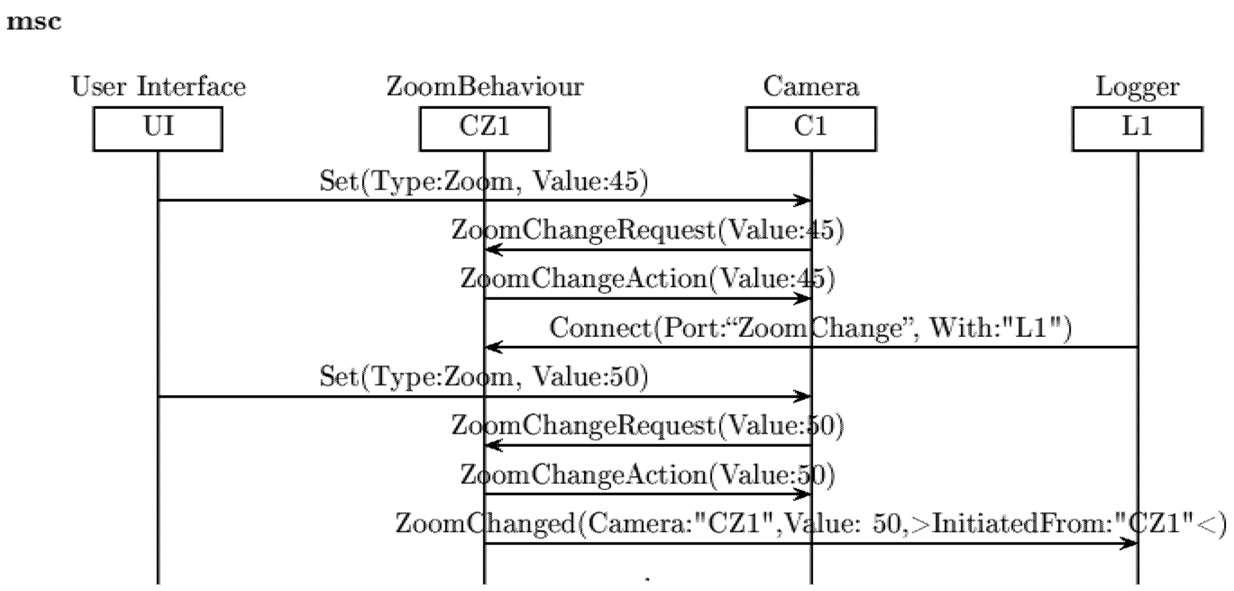

msc
User Interface
UI
ZoomBehaviour
CZ1
Camera
C1
Logger
L1
Set(Type:Zoom, Value:45)
ZoomChangeRequest(Value:45)
ZoomChangeAction(Value:45)
Connect(Port:"ZoomChange", With:"L1")
Set(Type:Zoom, Value:50)
ZoomChangeRequest(Value:50)
ZoomChangeAction(Value:50)
ZoomChanged(Camera:"CZ1",Value: 50,>InitiatedFrom:"CZ1"<)

Figure 35: logging of zooming events

7.2.4.2. Coupling of 2 cameras

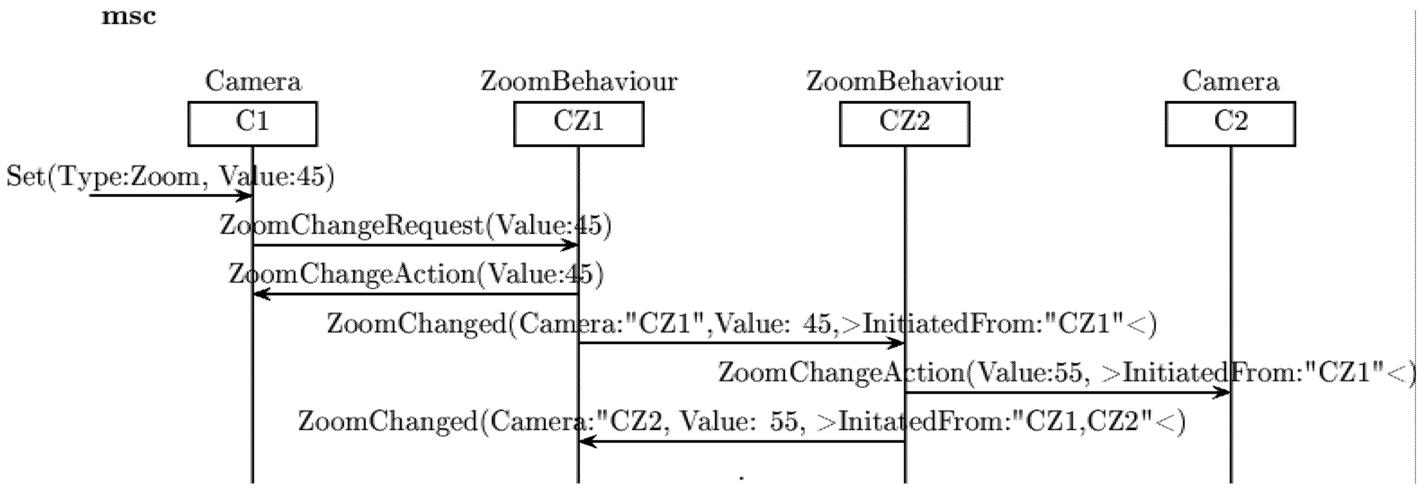

msc
Camera
C1
ZoomBehaviour
CZ1
ZoomBehaviour
CZ2
Camera
C2
Set(Type:Zoom, Value:45)
ZoomChangeRequest(Value:45)
ZoomChangeAction(Value:45)
ZoomChanged(Camera:"CZ1",Value: 45,>InitiatedFrom:"CZ1"<)
ZoomChangeAction(Value:55, >InitiatedFrom:"CZ1"<)
ZoomChanged(Camera:"CZ2, Value: 55, >InitatedFrom:"CZ1,CZ2"<)

Figure 36: coupling of 2 cameras

7.2.4.3. Coupling of 3 cameras, A with B, B with C

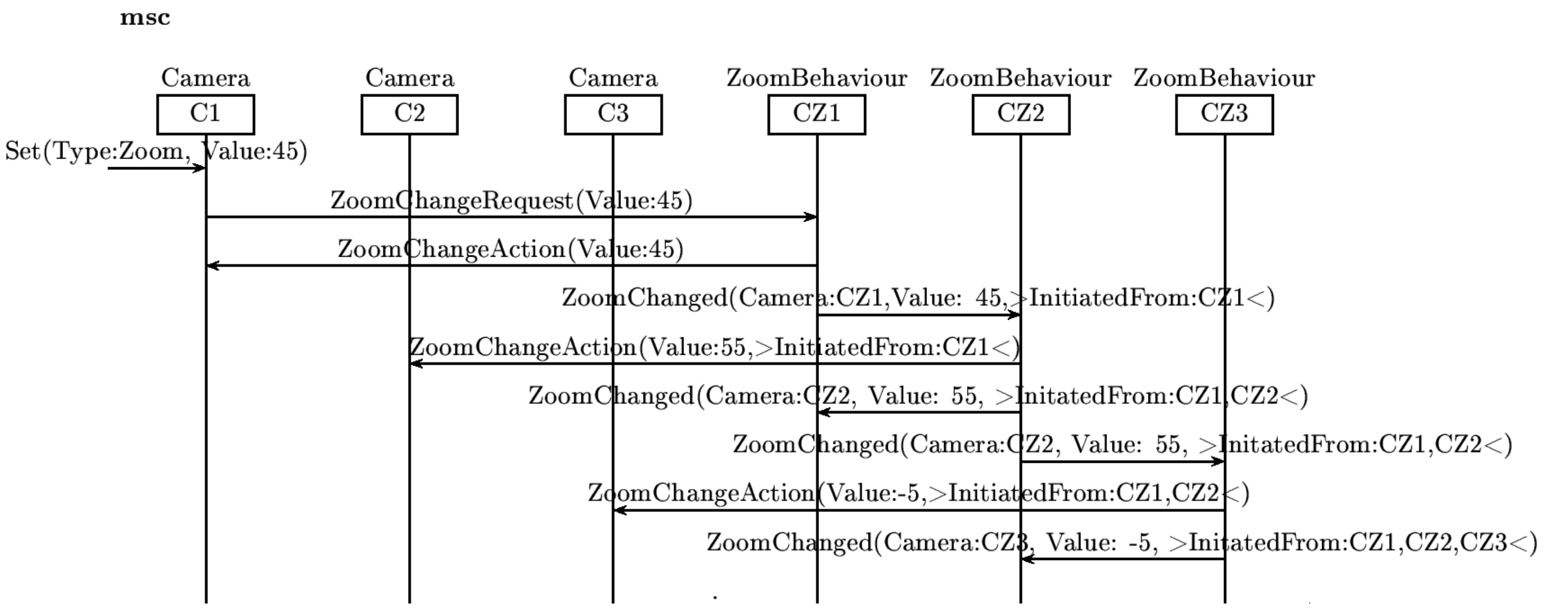

msc
Camera
C1
Camera
C2
Camera
C3
ZoomBehaviour
CZ1
ZoomBehaviour
CZ2
ZoomBehaviour
CZ3
Set(Type:Zoom, Value:45)
ZoomChangeRequest(Value:45)
ZoomChangeAction(Value:45)
ZoomChanged(Camera:CZ1,Value: 45,>InitiatedFrom:CZ1<)
ZoomChangeAction(Value:55,>InitiatedFrom:CZ1<)
ZoomChanged(Camera:CZ2, Value: 55, >InitatedFrom:CZ1,CZ2<)
ZoomChanged(Camera:CZ2, Value: 55, >InitatedFrom:CZ1,CZ2<)
ZoomChangeAction(Value:-5,>InitiatedFrom:CZ1,CZ2<)
ZoomChanged(Camera:CZ3, Value: -5, >InitatedFrom:CZ1,CZ2,CZ3<)

Figure 37: coupling of 3 cameras

7.3. Image Generating Components

This section describes the components that can generate video images: the **Camera**, **VideoStreamDecoder**, **Delay**, **Switch** and **Mosaic** component. An additional component (the **Prioritizer**) is introduced for handling conflicting user requests.

7.3.1. Introduction

The Camera and the VideoStreamDecoder play an important role in the test case as they are responsible for capturing video images and delivering them to the other components. The camera component captures the actual images. However, this component is not intended to deliver the actual images to an application component, instead a VideoStreamDecoder component is needed for this. The Camera generates image information that can be used by one or more VideoStreamDecoder components. Such a VideoStreamDecoder is a component that is provided by the camera itself. As such, a VideoStreamDecoder provided by a Camera can only be used to decode information delivered by this Camera; for decoding video information delivered by another type of camera, another VideoStreamDecoder is necessary. The format used for transmitting information between a Camera and an associated VideoStreamDecoder is only known by these two components.

A Camera should be able to provide two VideoStreamDecoders: one for producing an RGBA (RGBA is the natural choice as it is the native image format used by Java) and one for producing a B/W image.

The format in which data is transmitted from a Camera to a VideoStreamDecoder can change over time, e.g. a Camera can decide to switch from color to B/W images or use some kind of compression if the network load becomes too high.

The reason we have chosen a Camera/VideoStreamDecoder combination is fourfold:

- A new hardware camera using a new video format can be added very easily; it suffices to write an appropriate VideoStreamDecoder.
- If, for whatever reason, we would decide that the application needs another image type (e.g. YUV), it suffices to add VideoStreamDecoder components to the different Camera types.
- The VideoStreamDecoder is intended to run on the (embedded) system where the images it generates are consumed.[22] This allows us to limit the high bandwidth between the

[22] This is not forced by the system.

VideoStreamDecoder and its consumer to this system only, without using valuable network bandwidth. Using some kind of compression, we should be able to limit the bandwidth needed for the communication between the actual Camera and the VideoStreamDecoder.

- A VideoStreamDecoder is able to alter the image in a limited way (see later). This becomes handy when two or more components request images from the same Camera, but with different settings (e.g. another brightness). As a VideoStreamDecoder component can change the brightness of an image, this relieves the Camera of generating two different images. As such, VideoStreamDecoders allow us to distribute the computational load over the entire system while limiting the network bandwidth at the same time.

7.3.2. The Camera Component

7.3.2.1. Description

The Camera is the only component that actually communicates with the hardware camera. Its main purpose is to grab images from the camera and broadcast them to other components.

7.3.2.2 Interfaces

- port Component(noi=1)

Port required by the component system.

in Init()

in Connect(Port:<String>, With:<String>)

in Disconnect(Who:<String>, From:<String>)

- port VideoStreamDecoderFactory (noi>=0)

Using this port, a VideoStreamDecoder component can be requested by an application.

in GetFormats()

A request for a list of video formats (resolution and color depth) supported by the VideoStreamDecoder that can be provided by the camera. The answer is sent back using FormatList().

out FormatList(videostreamformat: <String>[])

The answer to a GetFormats request.

in GetDecoder(VideoFormat:<String>)

A request for a VideoStreamDecoder that is able to deliver video images in the format as described by VideoFormat. The camera reacts to this message by sending a Decoder(), a FormatNotSupported() or a ResourceViolation() message.

out Decoder(VideoStreamDecoder:<Component>)

The VideoStreamDecodercomponent sent as a response to a GetDecoder() query. As soon as a component receives the decoder, this connection with the VideoStreamDecoderFactory port can be closed as the VideoStreamDecoder will handle all future communication with the Camera.

out FormatNotSupported()

Answer to a GetDecoder() message requesting a decoder for an unsupported VideoFormat.

out ResourceViolation()

Answer to a GetDecoder() message requesting a decoder that is not supported at *this* moment, e.g. because the requested VideoFormat is not compatible with formats requested by other components.

- multiport VideoStreamUpdate (noi=1)

This port is used for sending image information to one or more VideoStreamDecoder components. This port can be a multiport or multiple single ports. In the latter case the camera can send different images to the different VideoStreamDecoder components, probably resulting in a higher processor load for the system running the Camera component.

out VideoStreamUpdate(VideoStreamUpdate:<java.lang.Object>)

Image information sent to a VideoStreamDecoder. As only a VideoStreamDecoder provided by this Camera is able to use the information in a useful way, it makes no sense to send this message to another component.

out Stalled()

Message sent by the camera if it is unable to generate new images for whatever reason. Is equivalent with SettingsChanged(FRAMERATE, 0) (see later).

- port Settings (noi>=0)

This port is used for getting/setting certain properties of the image requested. These settings can be the zoom factor, the brightness, the sharpness, etc. The only setting that cannot be changed using the port is the resolution of the requested video images. Changing the resolution can only be done by requesting an appropriate VideoStreamDecoder.

This port connects (normally) only with a VideoStreamDecoder provided by the same camera. However, one can imagine components that would connect to this port without using a VideoStreamDecoder component. An example could be a component that checks the processor load and decides to lower the framerate of the images generated.

The only reason why a message reaches the Settings port of the Camera is that a VideoStreamDecoder forwarded the request because it is unable to fulfil the request itself. This happens in three cases:

- the request can never be fulfilled by a VideoStreamDecoder; e.g. a zoom out request.
- the request can not be fulfilled by a particular VideoStreamDecoder because the decoder is too dumb.
- the VideoStreamDecoder has the required functionality but it refused to use it because it would damage itself; e.g. because it would require too much processor cycles.

In all these cases, the Camera must fulfil the request, unless it is an impossible request, e.g. asking for a negative zoom.

In all following messages ‘Type’ represents a particular setting, e.g. brightness, zoom, …

in Supports(Type: <String>)

Request whether the camera supports a certain setting. The answer (yes or no) is returned using Value().

in GetMax(Type: <String>)

Request for the maximum value of Type. The result is sent back using Value().

in GetMin(Type: <String>)

Request for the minimum value of Type. The result is sent back using Value().

in GetCurrent(Type: <String>)

Request for the current value of Type. The result is sent back using Value().

in Set(Type: <String>, Value: <Integer>)

Change Type to Value. As a result, we send Value() back to the requester and SettingsChanged() to all other VideoStreamDecoders that receive our images. The latter is not necessary if VideoStreamUpdate is not a multiport, but multiple single ports. Note that the Camera can decide at its own discretion to change a certain setting, as long as it informs all VideoStreamDecoders using SettingsChanged().

The zoom setting is handled in a special way (see later).

If we receive multiple conflicting requests (e.g. ComponentA requests 15 frames per second while ComponentB needs 30 frames per second) the last requested value is chosen. Using Value(), the chosen value is sent back. If the setting is changed at a later time, this is sent back using SettingsChanged, e.g. SettingsChanged(Type=FRAMERATE, NewValue=60).

The idea is that components that don't get the requested value, deal with it themselves (e.g. dropping half of the frames they receive) or notify the application component using a SettingsChanged (this can be regarded as a Java exception thrown form the bottom and forwarded until a component deals with it. In the worst case, the exception reaches the user application). Example:

- ComponentA and ComponentB receive images from the same CameraAB using VideoStreamDecoderA and VideoStreamDecoderB.
- ComponentA requests a brighter image, so it send Set(GAIN, 10) to VideoStreamDecoderA. There are now two possibilities:
    - VideoStreamDecoderA is intelligent enough to do this itself (multiplying each pixel value with a constant)
    - VideoStreamDecoderA is unable to do it himself. The decoder then forwards the request to the camera. CameraAB reacts by changing the gain of the camera images (either in hard- or software). As this also changes the image generated for VideoStreamDecoderB (if VideoStreamUpdate is a multiport) this decoder is notified using SettingsChanged(GAIN, 10). VideoStreamDecoderB can react to this in two ways:
        - It can take countermeasures (e.g. by changing the pixel values again). In this case ComponentB will still receive images with the original gain settings.
        - It does not take countermeasures. In this case, ComponentB is notified of the fact that the images are

taken with a changed gain setting by sending it SettingsChanged(GAIN, 10).

To conclude: if we envision the components in a treewise structure with a camera on top, than a Set() travels upwards until it is dealt with, and at this point in the tree SettingsChanged() is sent downward, again until it is dealt with.

in SetAuto(Type: <String>)

Request to set Type to automatic mode.[23] The result (succeeded or not) is sent back using Value() The camera leaves automatic mode after a successful Set().

in GetAuto(Type: <String>)

Check whether the camera supports an automatic mode for Type. The result (yes or no) is sent back using Value().

out Value(Type: <String>, Value:<Integer>)

Answer to a Get*() or Set*(). For a Set*(), the chosen value is returned, which is not necessarily the value requested.

out SettingNotSupported(Type: <String>)

Response to a message involving a setting that is not supported by the camera.

out SettingsChanged(Type: <String>, Value: <Integer>)

Notification that subsequent images will be made with changed camera settings. For zoom requests, three messages are sent; one message notifying that the camera starts zooming (SettingsChanged(ZOOMING, 1)), one notifying the end of the zooming (SettingsChanged(ZOOMING, 0) and a final one for broadcasting the final value (SettingsChanged(ZOOM, 1234)).

◗ port ZoomRequest(0<=noi<=1)

This port is intended to be used by the ZoomBehaviour component. All zoom requests are forwarded to the latter component. If there is no ZoomBehaviour component attached to this port, the Camera does not wait for a ZoomChangeAction, but fulfils the zoom request right away. The component supports two messages:

[23] The Camera or VideoStreamDecoder component is allowed to do this in software!

out ZoomChangeRequest(From: <String>, Value: <Integer>)

Message sent to the ZoomBehaviour component (if attached) if the camera receives a Set(ZOOM, X) request.

in ZoomChangeAction(NewValue: <Integer>)

Answer from the ZoomBehaviour component. This is the zoom value that will be used by the camera.

- port UIRenderer (0<=no<i=1)

This port can be used in order to receive an XML description of the camera.

in GetUiDescription()

Request for an XML description

out PutUiDescription(XML : <String>)

XML description of the camera.

7.3.3. The VideoStreamDecoder Component

7.3.3.1. Description

This component receives image information from a Camera and produces an RGBA or B/W image.

7.3.3.2. Interfaces

- port Component(noi=1)

Port required by the component system.

in Init()

in Connect(Port:<String>, With:<String>)

in Disconnect(Who:<String>, From:<String>)

- port VideoStreamUpdate(noi=1)

This port receives information from the camera.

in VideoStreamUpdate(Update: <java.lang.Object>)

This message is sent by a camera and contains the (compressed) information needed to generate an image. Only the VideoStreamDecoder and the Camera need to understand the Update information.

◗ port OutputStream(noi=1)

This port is used by the video consumer for retrieving the new image (either in RGBA or B/W format)

out NewImage(Image:<Videoimage>)

The new image.

out Stalled()

Message sent when the VideoStreamDecoder is unable to generate (momentarily) new images. This can be caused by a stall of the camera (notified with Stalled()), by memory constraints, …

◗ port Settings_in (0<=noi<=1)

Used for changing/requesting the value of a certain type. This port connects with an image consumer. The VideoStreamDecoder will deal with the request or forward the request to the camera using the Settings_out port.

in Supports(Type: <String>)

in GetMax(Type: <String>)

in GetMin(Type: <String>)

in GetCurrent(Type: <String>)

in Set(Type: <String>, Value: <Integer>)

in SetAuto(Type: <String>)

in GetAuto(Type: <String>)

out Value(Type: <String>, Value:<Integer>)

out SettingNotSupported(Type: <String>)

out SettingsChanged(Type: <String>, Value: <Integer>)

◗ port Settings_out (noi=1)

Used for changing/requesting the value of a certain type. The VideoStreamDecoder will deal with it or forward it to the camera.

out Supports(Type: <String>)

out GetMax(Type: <String>)

out GetMin(Type: <String>)

out GetCurrent(Type: <String>)

out Set(Type: <String>, Value: <Integer>)

out SetAuto(Type: <String>)

out GetAuto(Type: <String>)

in Value(Type: <String>, Value:<Integer>)

in SettingNotSupported(Type: <String>)

in SettingsChanged(Type: <String>, Value: <Integer>)

7.3.4. The Prioritizer Component

7.3.4.1. Description

The Camera, as described above, will fulfil all requirements submitted using a Set() message through the Settings port. This can lead to conflicts: e.g. one user wants to zoom in while another wants to zoom out. In order to deal with this, priorities and a Prioritizer component are introduced. However, the use of a Prioritizer is optional. A Prioritizer is used as follows:

- The Prioritizer is placed between a Camera and all its VideoStreamDecoders.
- The Prioritizer offers two Settings port: a multiport that connects to the VideoStreamDecoder and a normal port connected to the Camera.
- Each time a request enters the multiport, the prioritizer forwards the request to the Camera if the priority of the component/user submitting the request is equal or higher than the last request for that particular setting.

7.3.4.2. Interfaces

- port Component(noi=1)

Port required by the component system.

in Init()

in Connect(Port:<String>, With:<String>)

in Disconnect(Who:<String>, From:<String>)

◗ multiport Settings_in (noi=1)

in Supports(Type: <String>)

in GetMax(Type: <String>)

in GetMin(Type: <String>)

in GetCurrent(Type: <String>)

in Set(Type: <String>, Value: <Integer>)

in SetAuto(Type: <String>)

in GetAuto(Type: <String>)

out Value(Type: <String>, Value:<Integer>)

out SettingNotSupported(Type: <String>)

out SettingsChanged(Type: <String>, Value: <Integer>)

◗ port Settings_out (noi=1)

out Supports(Type: <String>)

out GetMax(Type: <String>)

out GetMin(Type: <String>)

out GetCurrent(Type: <String>)

out Set(Type: <String>, Value: <Integer>)

out SetAuto(Type: <String>)

out GetAuto(Type: <String>)

in Value(Type: <String>, Value:<Integer>)

in SettingNotSupported(Type: <String>)

in SettingsChanged(Type: <String>, Value: <Integer>)

7.3.5. The Delay Component

7.3.5.1. Description

This component can be used to introduce a delay in a video stream.

7.3.5.2. Interfaces

◗ port Component(noi=1)

Port required by the component system.

in Init()

in Connect(Port:<String>, With:<String>)

in Disconnect(Who:<String>, From:<String>)

◗ port InputStream(noi=1)

in Image(Image:<Videoimage>)

◗ port OutputStream(noi=1)

out Image(Image:<Videoimage>)

◗ port DelaySettings(noi=1)

out SetDelay(Delay:<Integer>)

7.3.6. The Switch Component

7.3.6.1. Description

This component can be used to switch a video stream on or off.

7.3.6.2. Interfaces

◗ port Component(noi=1)

Port required by the component system.

in Init()

in Connect(Port:<String>, With:<String>)

in Disconnect(Who:<String>, From:<String>)

◗ port InputStream(noi=1)

in Image(Image:<Videoimage>)

◗ port OutputStream(noi=1)

out Image(Image:<Videoimage>)

◗ port Control(noi=1)

in Controlport(Switch:<String>)

Used for switching the port on or off.

7.3.7. The Mosaic Component

7.3.7.1. Description

A Mosaic component receives video streams from several producers and combines them in a new video stream.

7.3.7.2. Interfaces

- multiport InputStream(noi=1)

A multiport that can receive images from multiple VideoStreamDecoders (using NewImage()).

in Image(Image:<Videoimage>)

- port OutputStream(noi=1)

out Image(Image:<Videoimage>)

Example Message Sequence Charts

The following message sequence charts show four possible interactions involving a Camera and one or more VideoStreamDecoders:

1. Setting up the communication between a Camera and a UserInterface using a VideoStreamDecoder.
2. Changing a Camera setting.
3. Trying to change a setting that is not supported by a particular Camera.
4. Changing the zoom of a Camera.

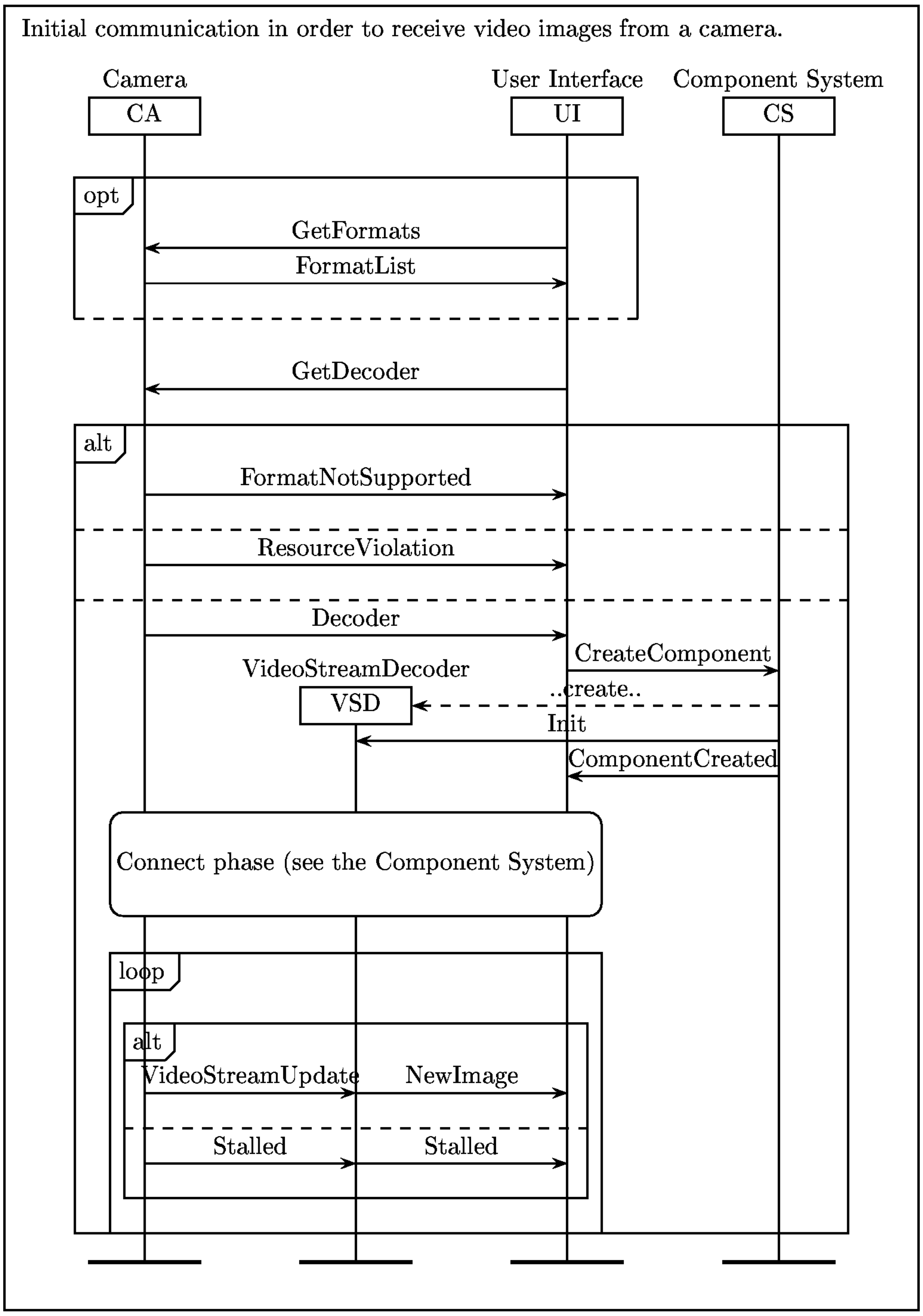


Figure 38: setting up the communication

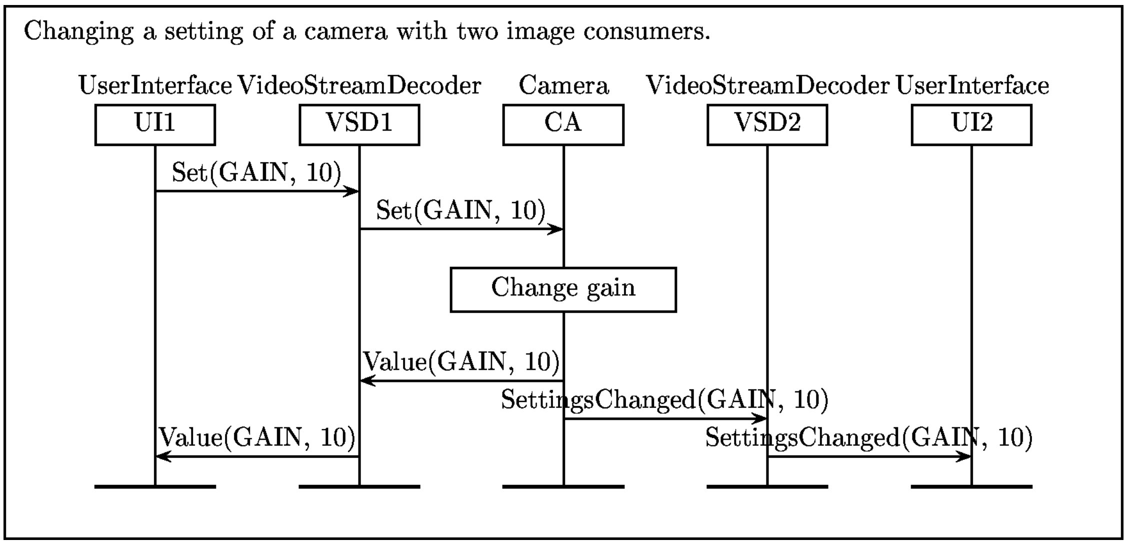


Figure 39: changing a camera setting

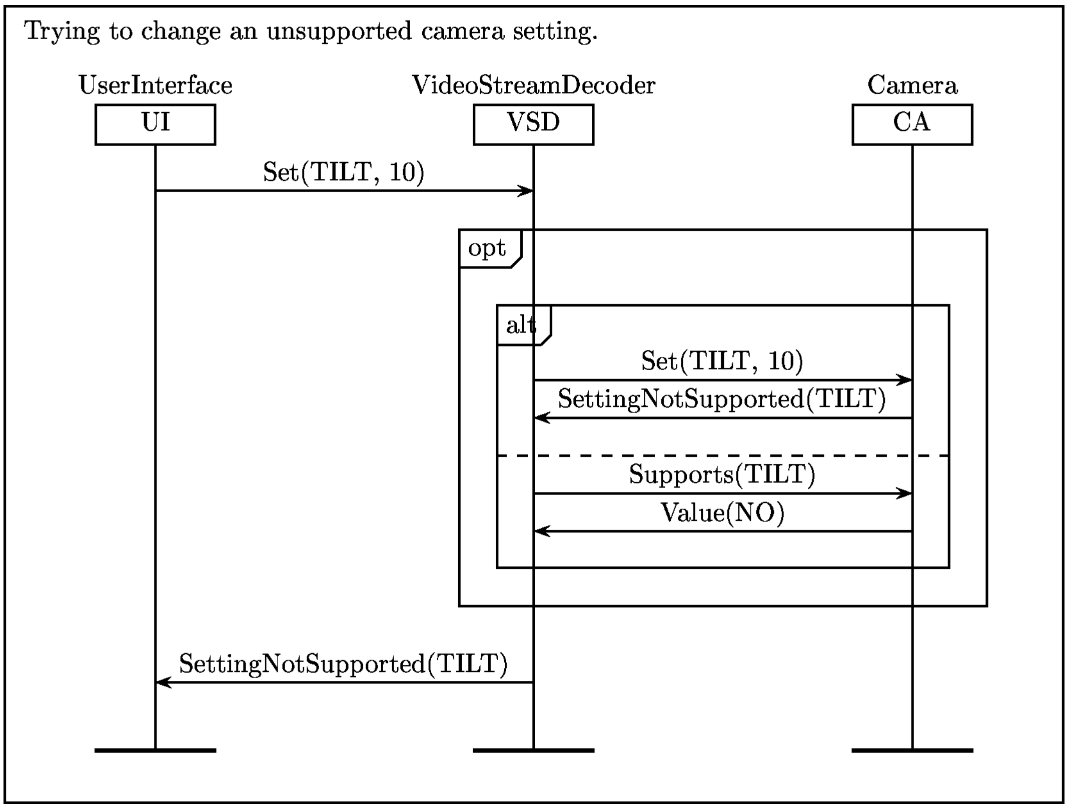


Figure 40: changing a setting that is not supported

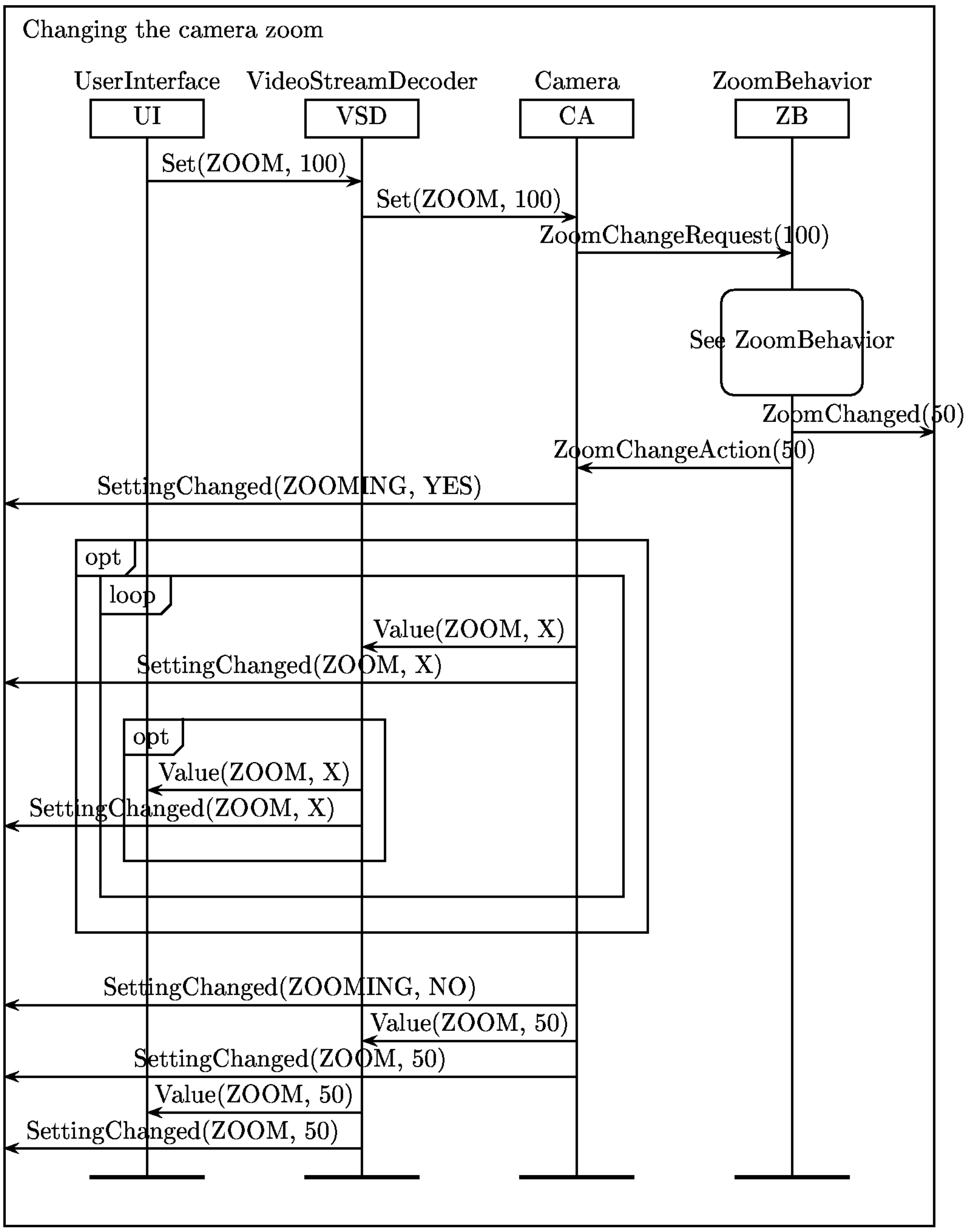


Figure 41: changing the zoom of a camera

7.4. Storage and Storage Controller Components

This section describes the design of the storage functionality required by the SCSS case study of the SEESCOA project. The functionality for the storage will be divided in two big components: the `StorageController` and `Storage`. The former connects to the rest of the components of the application and has some application logic included. For instance, it receives the events sent by other components and tells to the component `Storage` to put it in the database. Also, it receives queries entered by the user for retrieving events from the database. The latter is in charge of storing the data it receives from the `StorageController` and retrieving the events that are asked by the user through this last component. The `Storage` component does not have any application logic, it is just a dummy component that sends and retrieves data from a database. Of course both components have the interfaces requiered by the `Controller` component.

7.4.1. Component StorageController

7.4.1.1. Description

The `StorageController` is a component that connects the `Storage` with the rest of the application. It is in charge of sending the received events to the `Storage` component and thus any component of the system can potentially be connected to it. The `Camera Motion Detection` components must store an alarm when they detect some movement in a given place. We must say here that every alarm has associated an image, i.e., there are no alarms without images. They, images and alarms, are linked by camera id and a time stamp. The `Switch` components send images to the `StorageController` that have to be stored in the database. Also, the `StorageController` has ports for connecting to the client in order to push a determined XML description of an interface and also a port for connecting to the `UI Renderer` component in order to push the desired data (images, alarms, events).

The application logic that the `StorageController` possesses is related mainly to the recognition of different kind of events and the corresponding visualization. For instance, if the `StorageController` receives a query for some event stored in the database, it must first communicate with the `Storage` in order to extract the desired event from the database, and once retrieved, it must push the interface to the `Client` component in order to display the information in a coherent way.

It is necessary to remark that the `StorageController` does not have any connection with the database. It only communicates (sends and receives information) with the `Storage` component. Furthermore, it is not responsibility of this component to send notifications to the user about events that have ocurred in the system, with exception with the events that have been produced by the same `StorageController` component (see Use Cases).

In summary, the `StorageController` component provides the following ports for connecting with the rest of the components of the system:

- UI Description: it connects to the `Client` component.
- Video Out: it connects to the `UI Renderer` component.
- Query In: it connects to the `Client` component in order to receive the queries from the user.
- Controller: it connects to the `Controller` component.
- Event Log In: it connects to any component of the system.
- Video Record In: it connects to the `Switch` components.
- DB Out: it connects to the `Storage` component.
- Data Out: It connects to the `UI Renderer` component.

The required ports are:

- UI Description: in `Client`.
- Video In: in `UI Renderer`.
- InOut: in the `Storage` component.
- Component: in the `Controller` component.

7.4.1.2. Component Description

We now give the description of the `StorageController` component.

Port: UI Description (single)

```
Out putUITree(interface: XMLDescription)
```

It sends an XML description of the interface that has to be shown to the user of the system.

Port: Video Out (single)

Out sendImage(img: ImageData)

It sends the image and its related data to the UI Renderer in order to be shown to the user.

Port: Query In (single)

In queryImage(time: Time; cameraId: CameraID)

It requests an image from the database.

In queryEvent(time: Time; componentID: ComponentId)

It requests an event from the database.

In queryAlarm(time: Time; cameraId: CameraID)

It requests an alarm from the database.

Port: Controller Info (single). Description is given in the Controller component specification.

In sendTypeTo(type:String)

Out ComponentType(component: String; type: String)

In HasJoined(Who)

In HasDisJoined(Who)

In AreYouAlive()

Out Alive(Who)

Port: Event Log In (multi)

In saveEvent(event: EventData)

It stores the event into the database.

Port: Video Record In (multi)

In saveImage(img: ImageData)

It stores the image and its related info into the database.

Port: DB Out (single)

Out saveImage(img: ImageData)

It stores the image and its related info into the database.

Out saveEvent(event: EventData)

It stores the event into the database.

Out queryAlarm(time: Time; cameraId: CameraID)

It requests an alarm (special kind of event) from the database.

Out queryEvent(time: Time; componentID: ComponentId)

It requests an event from the database.

Out queryImage(time: Time; cameraId: CameraID)

It requests an image from the database.

In putImage(img: ImageData)

It receives an image from the database.

In putEvent(event: EventData)

It receives an event from the database.

Port: Data Out (single)

Out sendData(data: Data)

It sends data that has been retrieved from the database.

7.4.2. Component Storage

7.4.2.1. Description

The `Storage` component receives data from the `StorageController` and stores it in the database. This data can be images, alarms or any other kind of events that a component desires to store. This component also can receive queries from the `StorageController` in order to retrieve events from the database. The retrieval can be made by time, component, camera ID (in the case of alarms or images), etc.

The `Storage` is only connected to the `StorageController` component, and of course to the `Controller` component as well. This component does not have any application logic. It is just a component that stores and retrieves to and from a database.

The ports that this component has are:

- InOut: it connects to the `StorageController` component.

- Controller: it connects to the `Controller` component.

7.4.2.2. Component Description

The description of the `Storage` component is as follows:

Port: InOut (single)

In saveImage(img: ImageData)

It stores the image and its related info into the database.

In saveEvent(event: EventData)

It stores the event into the database.

In queryAlarm(time: Time; cameraId: CameraID)

It retrieves an alarm (special kind of event) from the database.

In queryEvent(time: Time; componentID: ComponentId)

It retrieves an event from the database.

In queryImage(time: Time; cameraId: CameraID)

It retrieves an image from the database.

Out putImage(img: ImageData)

It sends a retrieved image.

Out putEvent(event: EventData)

It sends a retrieved event.

Port: Controller (single). Description is given in the Controller component specification.

In sendTypeTo(type:String)

Out ComponentType(component: String; type: String)

In HasJoined(Who)

In HasDisJoined(Who)

In AreYouAlive()

Out Alive(Who)

7.4.3. Use Cases

7.4.3.1. Actors

- `Client`: It asks for images, alarms and/or events that have to be retrieved from the database, and it receives the corresponding interface.
- `UI Renderer`: It receives the retrieved data that has to be shown to the user.
- `Camera MD`: it sends to the database alarms that have occurred and have to be stored.
- `Switch`: It sends images to the `StorageController`.
- `Any Component`: It sends to the database events that have to be stored.
- `Storage Subsystem`: It gathers the funcionality for storing and retrieving alarms, images and other events to and from the database.

7.4.3.2. Query Event Use Case

1. `Switch` asks the `Storage Subsystem` for an specific event giving some parameters (time and/or component id).
2. `Storage Subsystem` retrieves the corresponding event from the database.
3. `Storage Subsystem` chooses the adequate interface description that has to be sent to `Client`.
4. `Storage Subsystem` sends the chosen interface description to `Client`.
5. `Storage Subsystem` finally sends the retrieved data to `UI Renderer`.

Alternative: Data cannot be retrieved

At step 2, `Storage Subsystem` fails to retrieve the desired data from the database. The causes can be of several kinds:

- Cause 1: There is a problem with the connection with the database.
- Solution 1: Send a warning message to `Client` notifying this problem and recomending to call the system administrator.
- Cause 2: There is no data corresponding to the parameters given by `Client`.
- Solution 2: Notify `Client` that there is no data corresponding to the given parameters. Allow `Client` to retry the operation.

7.4.3.3. Query Alarm Use Case

1. `Client` asks the `Storage Subsystem` for an specific alarm giving some parameters (time and/or camera id).
2. `Storage Subsystem` retrieves the corresponding alarm and its corresponding image from the database.
3. `Storage Subsystem` chooses the adequate interface description that has to be sent to `Client`.
4. `Storage Subsystem` sends the chosen interface description to `Client`.
5. `Storage Subsystem` finally sends the retrieved data to `UI Renderer`.

Alternative: Data cannot be retrieved

The same as Query Event Use Case.

7.4.3.4. Query Image Use Case

1. `Client` asks the `Storage Subsystem` for an specific image giving some parameters (time and/or camera id).
2. `Storage Subsystem` retrieves the corresponding image from the database.
3. `Storage Subsystem` chooses the adequate interface description that has to be sent to `Client`.

4. `Storage Subsystem` sends the chosen interface description to `Client`.

5. `Storage Subsystem` finally sends the retrieved data to `UI Renderer`.

Alternative: Data cannot be retrieved

The same as Query Event Use Case.

7.4.3.5. Store Image Use Case

1. `Switch` sends an image and its enclosed information that have to be stored to `Storage Subsystem`.

2. `Storage Subsystem` stores corresponding image.

Alternative: Data cannot be stored

At step 2, Storage Subsystem fails to store the image because:

- Cause 1: There is a problem with the connection to the database.
- Solution 1: Send a warning message to `Client` notifying this problem and recomending to call the system administrator in order to solve the problem.

7.4.3.6. Store Alarm Use Case

This is the a special use case as Store Event. Alarms can be sent only by `Camera MD` components.

1. `Camera MD` sends an alarm and its corresponding image that have to be stored on the `StorageSubsystem`.

2. `Storage Subsystem` stores corresponding image.

Alternative: Data cannot be stored

The same as Store Image Use Case.

7.4.3.7. Store Event Use Case

1. `Any Component` sends an event and its enclosed information that have to be stored to `Storage Subsystem`.

2. `Storage Subsystem` stores corresponding event.

Alternative: Data cannot be stored

The same as Store Image Use Case.

7.4.4. MSCs

In this part we present some MSCs showing the interaction among some of the components of the system involved in the storage or retrieving from the database.

There are 6 scenarios that can occur:

- **MSC1**: An alarm has happened and the `MD Alarm` component sends it to the `StorageController` in order to be stored in the database.
- **MSC2**: In some component an event has happened which has to be stored in the database.
- **MSC3**: An image that has been captured by a camera has to be stored in the database.
- **MSC4**: The user needs to retrieve an image giving some parameters. Images can be retrieved by giving the time it occurred, the camera in which was taken, the alarm with which it is associated (if it exists).
- **MSC5**: The user needs to retrieve an alarm and its enclosed information that has been raised at a given.
- **MSC6**: The user needs to retrieve an event that has occurred in a given component at a given time.

The abbreviations we use for the components is the following:

- **UI**: `UI Renderer` component.
- **SC**: StorageController component.
- **S**: `Storage` component.
- **CMD**: `Camera MD` component.

- **AC**: Any component.
- **C**: `Client` component.
- **SW**: `Switch` component.

MSC1

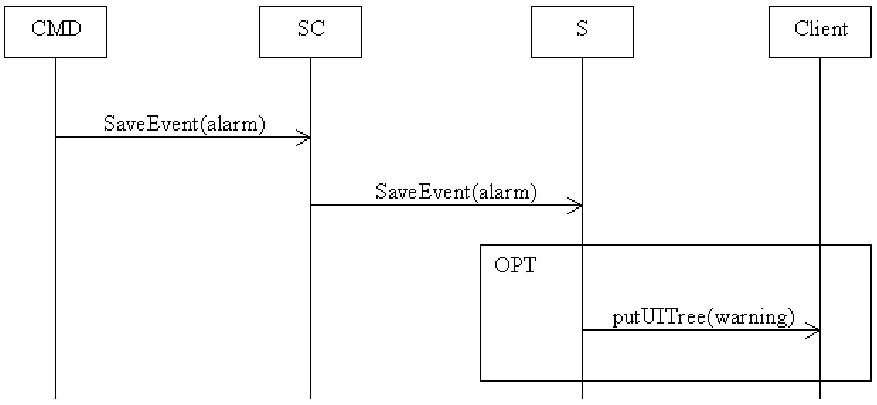


Figure 42: storing a motion alarm

MSC2

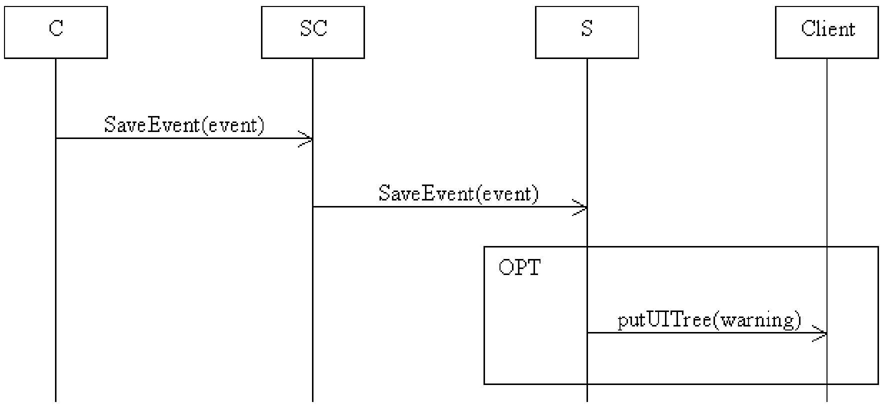


Figure 43: storing an event

MSC3

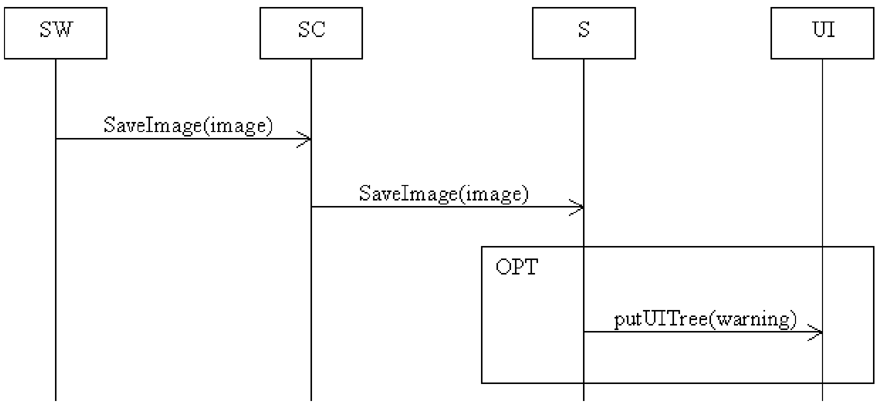


Figure 44: storing an image

MSC4

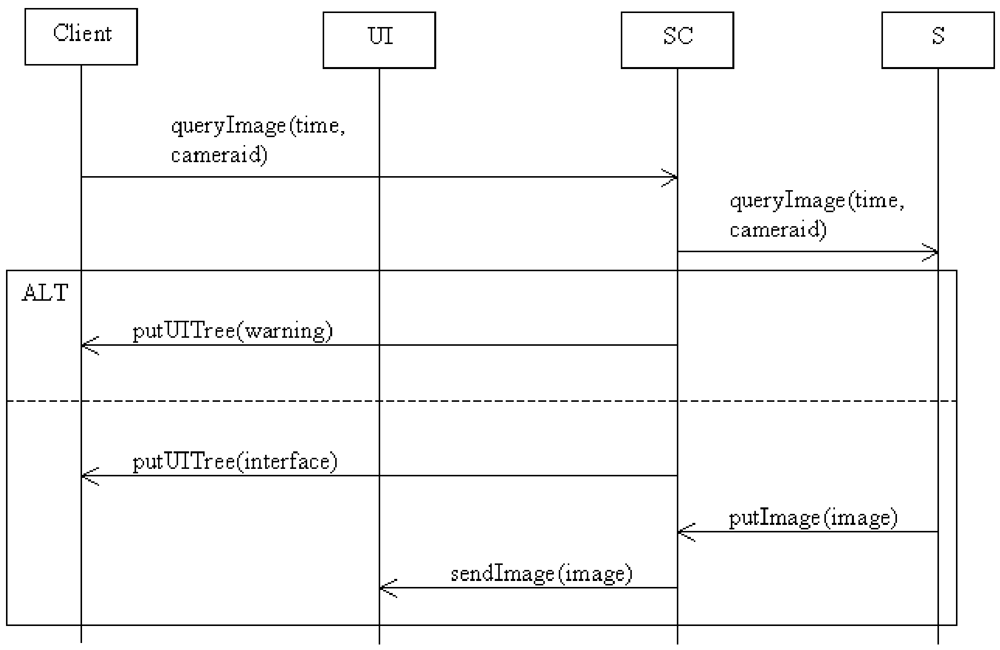


Figure 45: querying for an image

MSC5

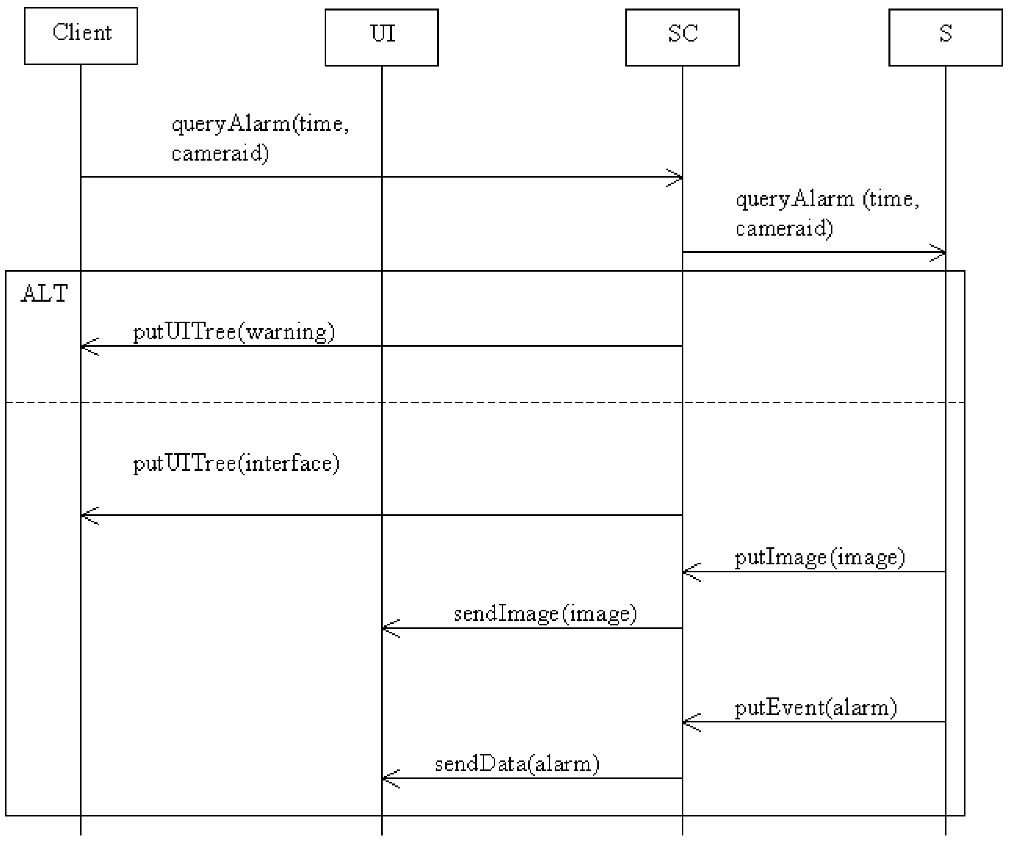


Figure 46: querying for an alarm

MSC6

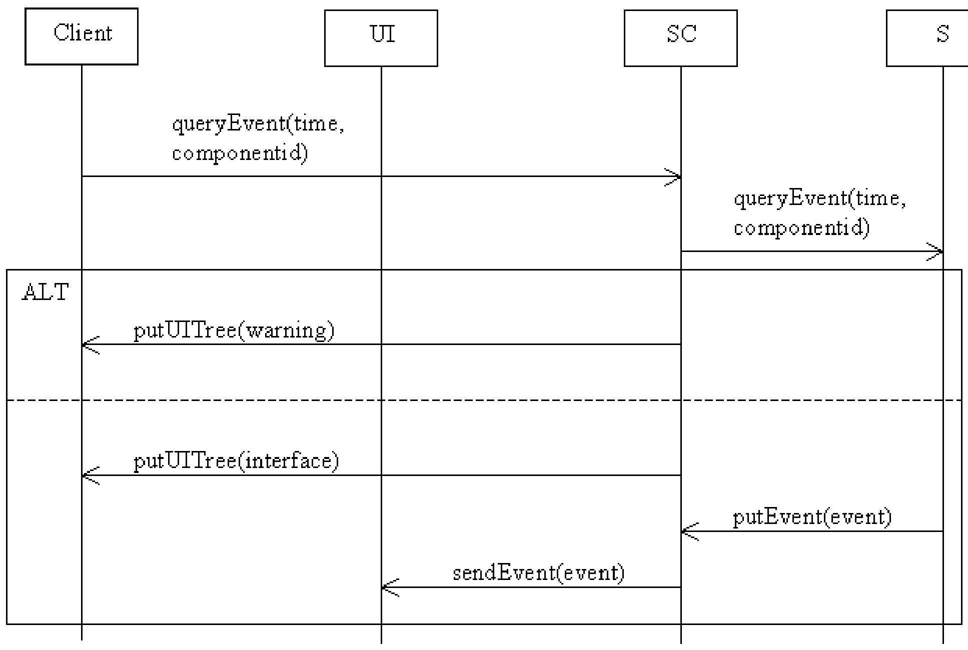


Figure 47: querying for an event

7.5. Camera Motion Detection Component

The motion detection service is an important part of a modern camera surveillance system. It facilitates the work of the guard (operator) considerably, since now a part of his job is done automatically. Also, adding motion detection to a system enables us to record only important video sequences (when there is motion). This is an advantage for the network load (a camera will only send images when needed) and the storage capacity (only image sequences with motion will be recorded).

As mentioned before (chapter 6), the motion detection service will consist of **camera motion detection** components. Each camera will have one corresponding *camera motion detection* component instance associated with it. This instance will analyse the video stream of the camera and report motion[24] events to the storage and eventually trigger a switch for recording the images.

Note: in this section, the term ‘MD’ will be used extensively. This is an abbreviation of ‘Motion Detection’.

7.5.1. Introduction on Motion Detection

This section is an introduction to the field of *motion detection*. It discusses some important concepts related to this topic.

There exist different types of motion detection, some can be called intelligent (they can spot the size, position and velocity of an object) while others just compare two images and see if they are different.

Common terms are:

Reference frame: MD algorithms often compare the current image to a reference frame. If they differ too much, an alarm is raised. The reference frame could be static: this will work wel in environments with a scene that does not change (like no lighting changes). But most algorithms adapt their reference frame to the current frame. So, if it is getting darker, then no alarm will be raised, since the reference frame will also get darker. Of course, if the light is switched off, then an alarm will be raised since the difference with the reference frame will be much bigger. An evolutive reference frame adapts itself to a gradually changing environment.

Frames per second: MD algorithms are also characterized by the number of frames they compare to the reference frame per second. A good value seems to be between 6 and 8 frames per second. If you go lower than that then some movements could go undetected. But if you go

[24] *Motion* is a complex term, since not all motion is *alarming* motion. Most motion detection algorithms have some treshold that can be set, this will ensure that there will not be an alarm for a bird that is flying by, but that an alarm will be raised if a human walks by.

higher than that (like 30 frames per second), then even a moving object could go undetected, because the difference between two frames will be small (at 30 fps, the moving object is moving very slowly, as such that the alarming treshold will not be reached).

Pixel: MD algorithms often compare pixels and see if there is a difference between two pixels on the same position (same x and y coordinates). Of course this could be generalized to only looking at some pixels, and not all pixels (like a grid of pixels applied to the image). It is not a good idea to do MD on compressed images since we need to decompress them first to get access to the pixels. As such, it is not good to put the compression between[25] the camera and the MD component, since the MD component will need to decompress the image first before being able to perform its algorithm.

Hot spot: some MD algorithms can also be configured to only monitor a part of the entire field of view of the camera. This part is also called a hotspot. The advantage of a hotspot is that the MD algorithm will be much faster since it only looks at a smaller area of the complete view.

Sensitivity: MD algorithms can be configured to be much or less sensitive to changes. This is also called a treshold: if the change in an image is greater than a particular treshold, then an alarm is raised. This change can be expressed in pixel change (if the pixel becomes a bit darker, this could or could not raise an alarm) and/or in the number of pixels that have to change (if 20 pixels have changed then this could or could not raise an alarm).

Object "recognition"[26]: Sometimes you are not interested in lighting events, but only in moving objects or persons. With a basic Motion Detection algorithm lighting changes will raise an alarm since the complete image will have changed.

Therefore, more intelligent algorithms are being researched: they can detect moving objects. Besides detecting moving objects, they can also measure their size and position. This is not really difficult: the computation is based on comparing two images and the creation of an image change matrix. The moving object will then be characterized as a "cluser" of changes. This cluster is then "the moving object". In the next step the algorithm could calculate the position of the cluster and also its size. This enables one to specify detection constraints as "detect moving objects of a size between 20% of the complete viewing area and 80 % of the viewing area". Such constraints ensure that small objects will not be detected and that big changes (like a lighting change) will not trigger the alarm.

[25] In some cases this will however be necessary to limit the bandwidth between the camera and the motion detection components (if they reside on different nodes).

[26] Sensitivity and Object "recognition" are not the same. You could for instance have a MD sensitivity setting that is very sensitive (the smallest change in pixel color is detected) but that will not raise an alarm if the object causing the change is smaller than 20% of the viewing area.

There are still some other topics in the motion detection field, but the ones mentioned above are the important ones. Basic MD algorithms can only be configured by setting some treshold values, while more advanced ones can have hot spot configuration and "object detection"[27] features.

7.5.2. Requirements for the Camera Motion Detection Component

This section discusses the requirements of the Camera Motion Detection component of the SCSS. If a particular requirement is not clear, please refer to 7.5.1. where an introduction on motion detection is given.

- **Setting the reference frame**: The user should be able to choose if the reference frame is static or evolutive.
    - **Static**: the user must be able to specify the frame that has to be used as reference frame. To do this, he can tell the MD component that the current frame of the camera is the reference frame (so the MD component has to grab a frame from the camera component)
    - **Evolvable**: nothing has to be done. The MD grabs an image when it is started and uses this one as the reference frame. When running it will adapt the reference frame to the new frames.
- **Setting the frames per second**: The user can set the number of frames per second the algorithm has to process. A value of 6 to 8 seems good. Faster frame rates have a disadvantage (see the explanation above) and do also require more processing power.
- **Setting a hot spot**: The user can set a hotspot on the viewable area of the camera. The MD algorithm will then only process image information from this area. Setting a hotspot can be done by providing start_x, start_y coordinates with width and height information.
- **Sensitivity:** This is not an easy setting: what means sensitive and not sensitive? It is nice to have a treshold setting like "Detect humans but not animals", but of course this is not possible for a simple MD algorithm. Often MD algorithms offer some kind of slide bar that goes from "detects nearly never motion" to "detects every pixel change". Sensitivity is thus an intuitive value, that will have an impact on the MD algorithm.

[27] Detecting cars, humans, … and their properties: color, size, direction, … is a more advanced object recognition application and is not described here.

We could define two kinds of threshold: a treshold indicating how many pixels have to change, and another treshold indicating how much these pixels have to change (how light or dark should they get for instance). These two tresholds will then set the sensitivity of the MD algorithm.

We have decided to offer five predefined settings: *detect_no_change*, *detect_large_changes*, *detect_changes*, *detect_small_changes* and *detect_every_change*.

- **Object recognition:** Is there need for some advanced Motion Detection? Not at the moment, since it is not the goal of the project to design image processing algorithms. As such, the Camera Motion Detection will detect all changes, even changes in lighting conditions. A lighting change will probably raise an alarm since the whole image is changed (the treshold is likely to be exceeded). In a first implementation it will not be possible to select min and max sizes of objects that have to be monitored. So, a setting like "Do not detect small objects (animals)" will not be possible.

- **Disable motion detection while zooming**: This is a requirement that ensures that the MD algorithm will not raise a sequence of alarms if the camera is zooming. The MD algorithm has to be enabled again when the camera has stopped zooming.

- **Report motion events to the storage**: When the camera MD component has detected some motion then this has to be reported to the storage controller component. The camera MD component has to report the time when the motion was detected, with the two or more frames that contain the motion information. The user must be able to switch off motion event reporting.

- **Trigger a switch to enable recording of video**: The camera MD component must also be able to trigger a switch when motion was detected. This switch can be attached to the storage as such that the storage starts recording images it receives from the switch. When no motion is detected any more, the switch has to be closed. The time between the motion detection and the opening of the switch, and the time between the end of motion and closing of the switch must be set by the user.

- **User interface for configuration**: Since every important component should offer a way to access its user interface, we also need this for the camera motion detection component: it needs to export its user interface.

7.5.3. Use cases

In what follows we will sometimes say that the camera MD component receives images from the camera, but in fact one should read this as: *the* *camera MD component receives images from a VideoStreamDecoder* (see 7.3.3.) *for that camera*. As such, there is always a decoder[28] between the camera and the motion detection component.

7.5.3.1. Receive a new image from the camera

The camera MD component is connected to the *OutputStream* port of the VideoStreamDecoder. Via this port it receives image updates. If there are no images (the decoder sends *Stalled()* messages to the camera MD component) then the camera MD component stops processing images until new images arrive.

7.5.3.2. Zoom notification

The camera MD component is also connected to the *Settings_in* port of the VideoStreamDecoder. Via this port the camera MD component will receive notifications of the start and end[29] of zoom actions on the camera. When a zoom action is started, then the camera MD component will stop its motion processing. When the zoom action is ended, then the camera MD component will continue with its motion processing.

7.5.3.3. Store motion event (motion alarm)

The camera MD component can store motion alarms on the storage. Therefore it will be connected to the *storage controller* (see 7.4.1.). The camera MD component will attach following parameters to the event: a set of images that contain the motion information (this set will minimally contain two images), the time at which the motion was detected and its identifier.

7.5.3.4. Trigger a switch

The camera MD component can also interact with a switch (see 7.3.6.) to control the image flow between an image producer and an image consumer. When motion is detected the switch can be opened after a predefined duration. When no more motion is detected, the switch will be closed again (also after a predefined duration). This can be used to only record images if there is motion. See 6.2.6. for a detailed explanation on this subject.

[28] As was mentioned earlier, it is not a good idea to compress images and send them to a camera MD component that resides on the same node as the camera. Since the camera MD component performs its processing on uncompressed images, the images will need to be decompressed first. This is clearly a overhead penalty. If the camera MD component resides on another node, then it is necessary to compress images in order to keep the network bandwidth usage low.
[29] Optical zooming is a mechanical action and takes some time. Therefore we talk about the 'start' of a zoom action and the 'end' of a zoom action.

7.5.3.5. Change settings

Other components can change the settings of the camera MD component. Settings that can be configured are:

- **Framerate**: the framerate at which the images have to be processed.
- **Sensitivity**: one of five predefined sensitivity settings.
- **Hotspot**: x, y, height and width of the hotspot box.
- **Reference frame**: static or evolutive.
- **Enabling/disabling switch trigger**.
- **Duration for opening/closing switch**: duration setting for closing a switch after motion has been detected and for opening a switch when no motion has been detected any more.
- **Enabling/disabling motion alarm logging**.

7.5.3.6. Exporting user interface description

The camera MD component has also an interface that can be exported for (human) user interaction with the component.

7.5.4. Camera MD Specification

The camera MD component consists of 6 ports:

◗ **Stream Input port** (noi = 1)

This port is used to connect the camera MD component to the *OutputStream* port of the *VideoStreamDecoder* component. Via this port, the camera MD component receives image updates (see 7.3.3.).

in NewImage(Image:<Videoimage>)

Image: a new image from the camera.

in Stalled()

If (for any reason) the decoder is not able to deliver an image.

◗ **Camera Settings port** (noi = 1)

This port is used to connect the camera MD component to the *Settings_in* port of the *VideoStreamDecoder*. Via this port, the camera MD will receive notifications about zoom actions (see 7.3.3.).

out Supports(Type: <String>): not used

out GetMax(Type: <String>): not used

out GetMin(Type: <String>): not used

out GetCurrent(Type: <String>): not used

out Set(Type: <String>, Value: <Integer>): not used

out SetAuto(Type: <String>): not used

out GetAuto(Type: <String>): not used

out Value(Type: <String>, Value:<Integer>): not used

in SettingNotSupported(Type: <String>): not used

in SettingsChanged(Type: <String>, Value: <Integer>)

This message is sent from the decoder to the camera MD component to indicate that a zoom action has started or ended. During the zooming period, the camera MD component will not process the images.

- **MD Settings port** (noi >= 0)

The MD Settings port is used to set or get the properties of the camera MD component. Following messages are understood:

in GetCurrent(Type: <String>)

If a component wants to retrieve the current value for a property, it has to send this message. Type is an identifier indicating the property that is requested.

in Set(Type: <String>, Value: <Object>)

This message sets a particular property of the camera MD component. Depending on the type of property that is being set, the contents of the value object will differ. For instance, if the “hotspot” type is being set, then the contents of the value object are: x, y, width and height.

out Value(Type: <String>, Value: <Object>)

This message is sent back when a component requests the value for a particular property.

Supported Types:

“framerate”, “sensitivity”, “hotspot”, “reference”, “enable switch triggering”, “trigger durations”, “enable alarm logging”.

Supported Values: (Type dependent)

- *framerate*: value is an Integer object.
- *sensitivity*: value is String from the set {“detect_no_change”,”detect_large_changes,“detect_changes”,“detect_small_changes”,“detect_every_change”}
- *hotspot*: value is an array of Integer objects: [x, y, width, height].
- *reference*: value is String from the set {“static”, “evolvable”}.
- *enable switch triggering*: value is Boolean
- *trigger durations*: value is an array of Integer objects: [duration after start of motion, duration after end of motion].
- *enable alarm logging*: value is Boolean.

◗ **Switch Trigger port** (noi >= 0)

This port will be connected to the *Control* port of the *Switch* component (see 7.3.6.). It is used to trigger the switch when motion is detected.

out Controlport(Switch:<String>)

This message is sent from the camera MD component to a switch component, to open or close it. The camera MD component closes the switch if motion is detected and reopens it after no more motion is detected.

◗ **Motion Event port** (noi >= 0)

This port is used to connect the camera MD component to the *Event Log In* port of the *StorageController* component (see 7.4.1.). Through this port, the camera MD component can store motion alarms on the storage.

out saveEvent(event: EventData)

Message sent out to the CameraController to store an event.

◗ **UI Description port** (noi = 0)

Via this port the camera MD can export a user interface description in XML. More information on this can be found in 7.6. and 7.7.. This port will be connected to *UIDescription* port of the *Client* component.

7.6. UI Renderer Component

7.6.1 Requirements of the User Interface

In the design of user interfaces for embedded systems and real-time systems, several approaches are possible. Omitting the component-oriented approach in the first stage, we concentrated on Task design using ConcurTaskTrees. This notation enables us to model user interaction with the system on several levels (from coarse-grained to fine-grained).

Traditional systems exist out of the typical multiplexer views, in which a single screen is divided in several square boxes, each presenting a different camera. A typical camera surveillance system application contains hardware multiplexers to combine several video streams and present this on a single monitor. Figures 48 and 49 give an illustration of such a system. Also, in traditional systems there is a static user interface available, most of the time only accessible from a single point. Our system will provide a more flexible approach using the component system, enabling multiple points of access, which can be extended very easily.

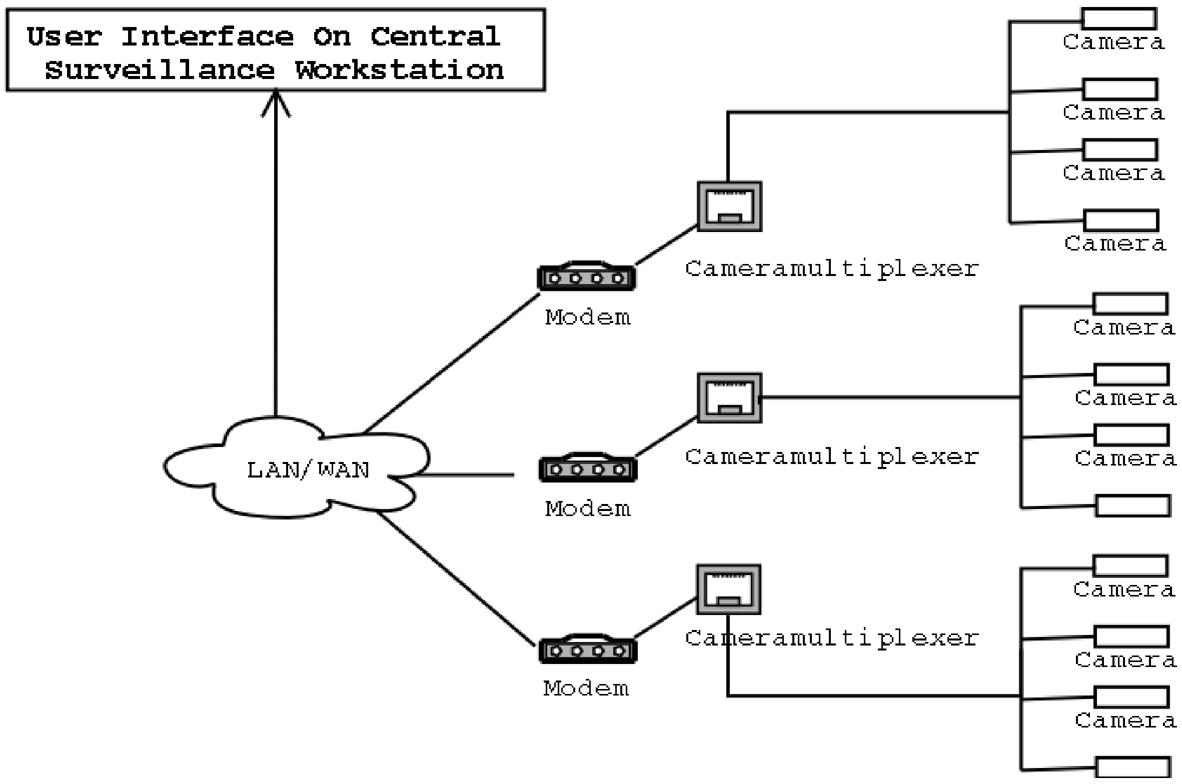


Figure 48: A typical camera surveillance system setup.

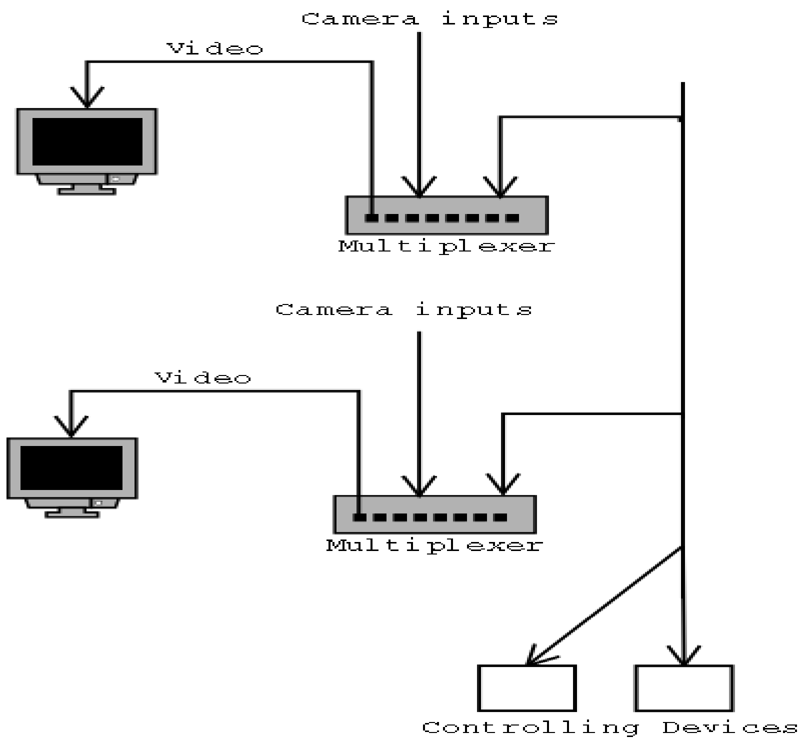


Figure 49: System layout with multiplexers

Using the user interface framework the component system provides, the importance of a ready to serve user interface is very small. Using a user interface description language the user can choose a custom made presentation of the user interface. Nevertheless, we will provide a standard user interface for the system. An example of a user interface controlling a single surveillance camera can be found in figure 50. This is an example of a user interface for the desktop PC. This user interface will be expanded to control several surveillance cameras, like shown in figure 51, which is a first draft version of a user interface. In section 7.6.4 there are some XML descriptions of this user interface.

The user interface for the camera surveillance system should be focused on the constant observation of a geographical place, enhanced with “intelligence" embedded in the cameras (the motion detection component for example). We can divide the functionality roughly in three user interfaces:

1. A user interface for controlling a single camera
2. A user interface for managing and controlling a group of cameras
3. A user interface for administration of the system.

User interface for controlling a single camera change zoom / focus / frame rate / hotspot for a single camera. Contains indicators when a suspicious event is detected.

User interface for controlling or managing a group of cameras control “group" behaviors of a set of cameras, select a particular camera for a closer look, view geographical setting of a (set of) camera(s),. . .

User interface for administration adding, removing and editing users, configuring surveillance system (cameras, plugins,. . .), managing bound settings for a (set of) camera(s), . . .

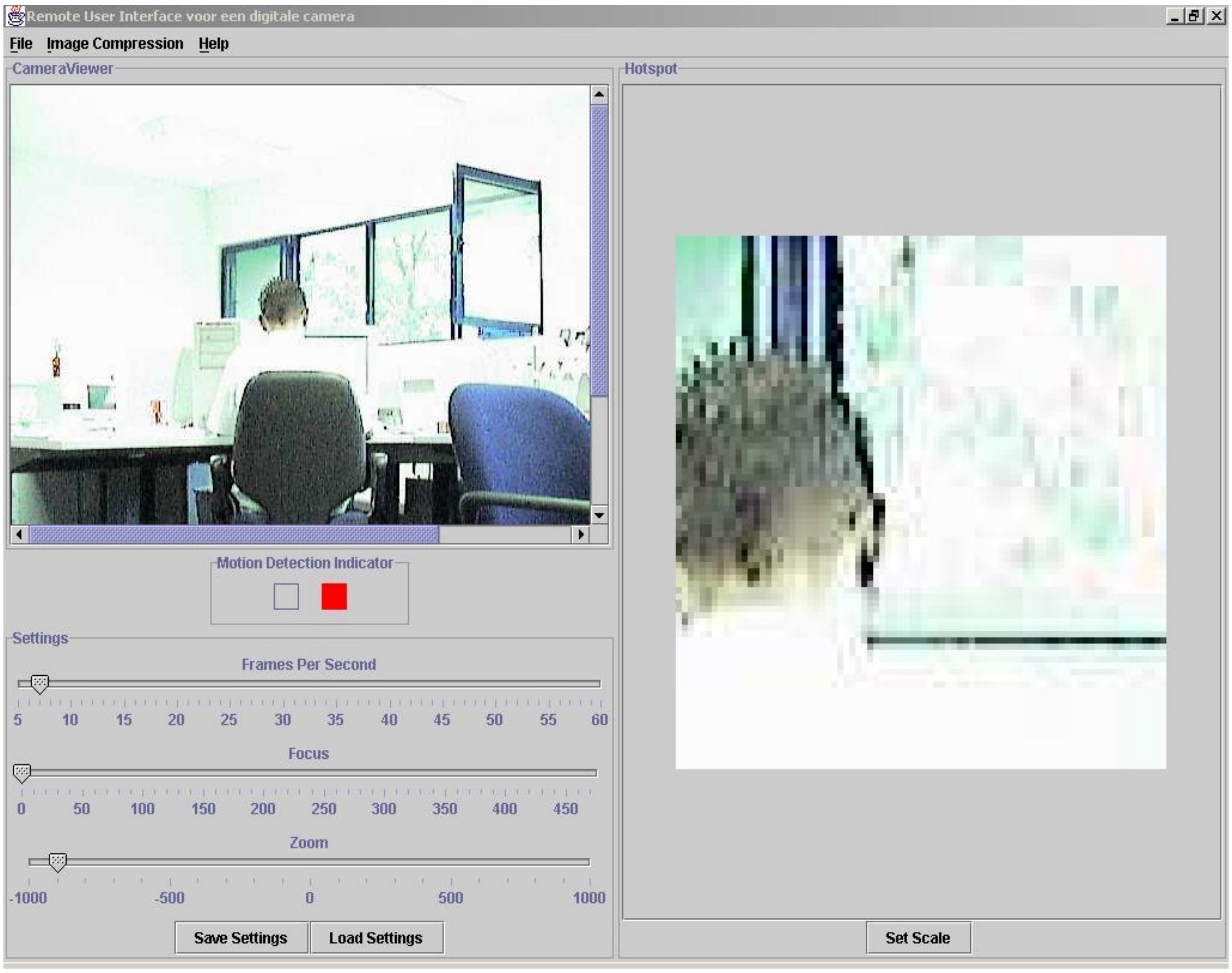


Figure 50: A user interface for controlling a single surveillance camera (Desktop version)

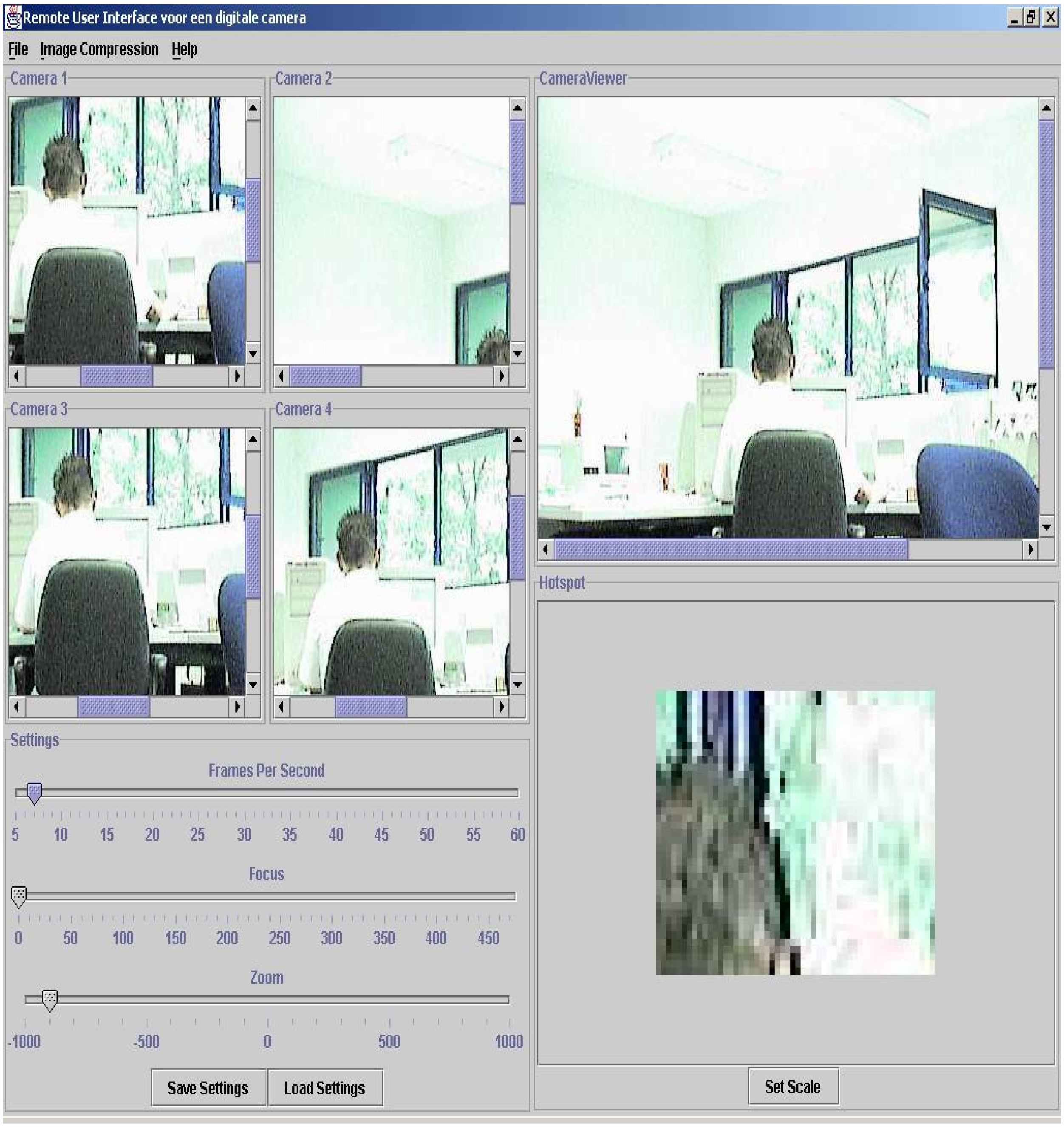


Figure 51: a user interface for controlling several surveillance cameras (Desktop version)

How such a user interface can be rendered onto a screen within the component system is the subject of the next section.

7.6.2 User Interface Components

7.6.2.1 Introduction

The User Interface Components in the component system are not static user interfaces. They are components that receive a user interface description and render this description to an output or interaction device. Two advantages of this approach are:

Flexibility: the user can compose his own user interface, and the user interface can be adapted to the target system.

Extensibility: if new components or plug-ins are integrated in the system, and these components support a user interface, they can be graphically controlled without further configuration. Components that are not meant to be controlled graphically do not offer a UI description.

A very simple example of the user interface description that a UIRenderer component can render one can find in listing 1. A more thoroughly discussion (of the used XML descriptions) can be found in deliverable 4.3: Generalization of a Component-Based User Interface.

Notice the “group" type of an Aio element indicates that the subtree of this element cannot be split up in the user interface: the child elements make up one logical part of the user interface. The Aio tag indicates an Abstract Interface Object. To enable interaction, the description also includes which port of which component must be notified when an event is generated by that widget. Listing 1 defines which action has to be taken by adding a specification for the port to be notified when the button “hotSpotButton" is pushed.

Listing 1: A user interface description with interactive capabilities.

```
<?xml version="1.0" encoding="UTF-8"?>
<Aio NAME="VideoControls1181232615"TYPE="testapp.VideoControls">
  <Aio NAME="hotSpotGroup" TYPE="group">
    <Property NAME="hotSpotButton">
      <Aio NAME="Button126489797" TYPE="BUTTON">
        <Property NAME="label">set Hotspot</Property>
        <Property NAME="actionCommand">activateHotspot</Property>
          <Events>
            <Out>
              <Component NAME="CameraController05">
                <Port NAME="HotSpot">
                  <In>SetNewHotSpot</In>
                </Port>
              </Component>
        </Out>
      </Events>
  </Aio>
```

```
      </Property>
    </Aio>
  </Aio>
```

This gives a “point-to-point" communication mechanism. It is also possible (but not necessary) to define more than one target port for the event. This can be done by adding several *Port* tags as children of the *Component* tags, or if the event should be delivered to several components. *Component* tags can be added as children of the *Events* tag.

7.6.2.2 The UIRenderer component

7.6.2.2.1 Description

The user interface component is a component that renders a user interface description to a particular output device. This output device is not necessarily the screen, but can also exist out of different modalities such as sound. To enable the user interface component to adapt to the target system constraints and modalities, we use a high-level user interface description language. The user interface is described in XML, like shown in listing 2. A graphical presentation of the UIRenderer component is given in figure 52.

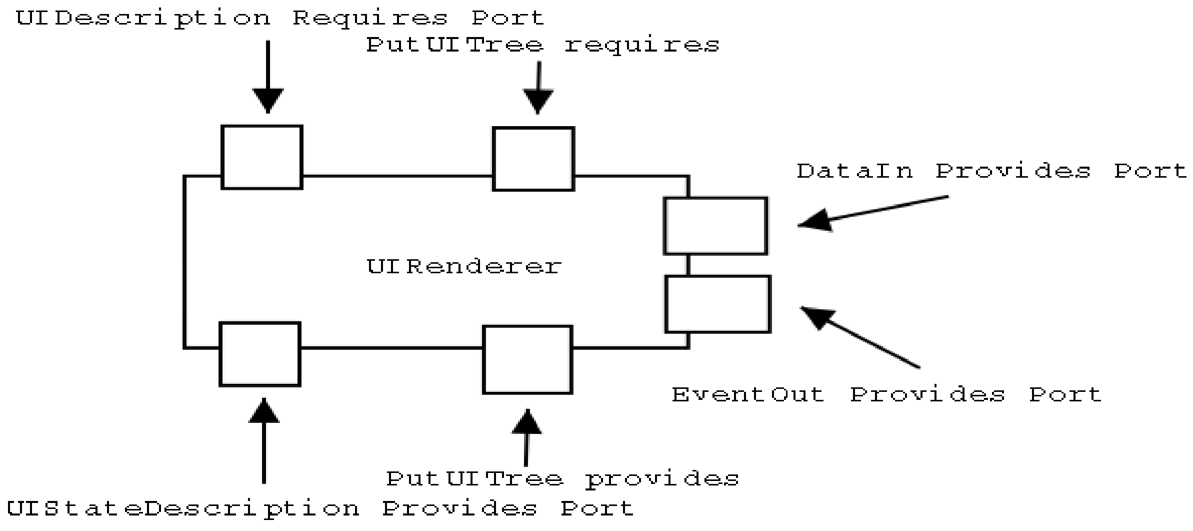


Figure 52: The UIRenderer component

The UIRenderer component uses a platform and output device specific mapping to map the abstract user interface to a concrete user interface, using the appropriate widget set. For now these mappings can be read from a file in which the mappings are described using XML, or from a precompiled Java class. For each different system the specific mapping has to be provided.

Internally the UIRenderer exists out of several other components: the 2DUIRenderer and the 2DlayoutManager. These two components will layout the several widgets using a constrained screen space, and render it choosing an appropriate widget set. An overview of the structure of the UIRenderer can be found in figure 53.

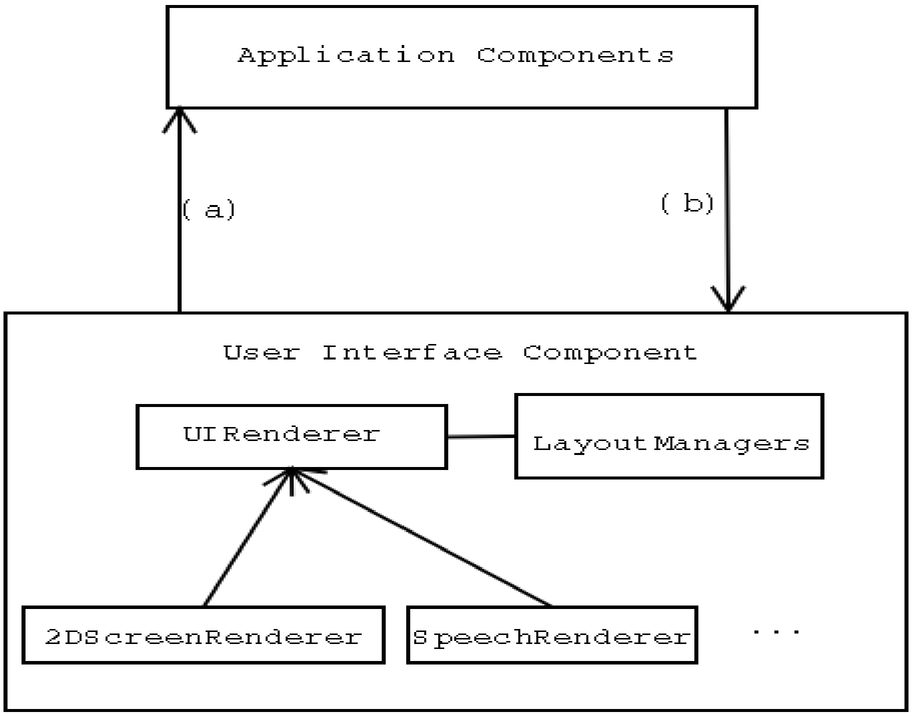


Figure 53: An overview of the UI Component (a) sends messages to the right components as reaction to user actions as described in the user interface description. (b) sends User Interface Descriptions and update messages to UI Component.

This component has no application logic, it only takes a user interface description and renders this to an output device. Application logic should be modeled by other components. For example, consider the user interface for a surveillance system, containing a mosaic view for 4 cameras. There should be a component defining a mosaic (taking the input streams of 4 cameras and combining them) and providing a “User Interface Port" which gives an XML description for the component. An example for such a description can be found in listing 2 and a schematic drawing in figure 54. Notice there exists a DataIn port in the UIRenderer where updates of visualized data can be submitted, for the User Interface to be updated.

Listing 2: A User Interface Description from the Mosaic Component

```
<?xml version="1.0" encoding="UTF-8"?>
<Aio NAME="MozaikControl1181232615" TYPE="MosaicControl">
  <Property NAME="Controls">
    <Aio NAME="ControlsGroup">
     <Property NAME="">
```

```
              . . .
      </Property>
    </Aio>
  </Property>
  <Property NAME="Screens">
    <Aio NAME="ScreenGroup " TYPE="group">
      <Property NAME="screen1">
        <Aio NAME="s c r e en " TYPE="canvas">
          <Property NAME=" label ">surveillance camera 1</Property>
          <Events>
            <Out TYPE=" select “>
            <Component NAME="">
              <Port NAME="">selectCamera</Port>
            </Component>
           </Out>
           <In TYPE="update”>
            <Component NAME="">
              <Port NAME="">setFrame</Port>
            </Component>
           </In>
          </Events>
        </Aio>
      </Property>
      <!—the same for screens 2,  3 en 4 -->
    </Property>
  </Aio>
</Aio>
```

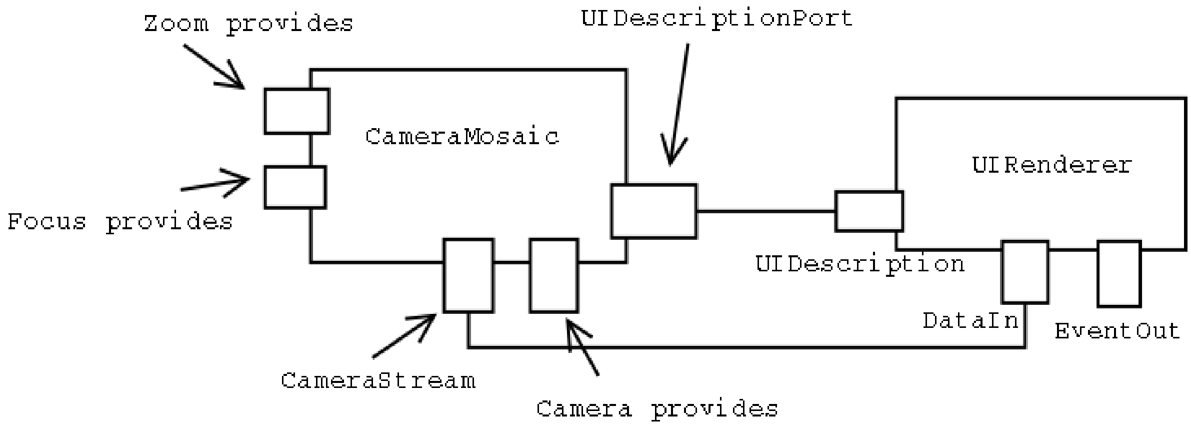


Figure 54: the UIRenderer and CameraMosaic components.

7.6.2.2.2 Ports

Init This will be used to initialize the UIRenderer, for example: the mapping definitions can be loaded on initialization. These mapping descriptions can be provided in a recompiled Java class or file(containing an XML description of these mappings).

UIDescription Requires Port This port will be connected to the components that can deliver a user interface description for their functionality. It receives the user interface description in XML.

UIStateDescription Provides Port This port provides a user interface description containing the current state of the user interface. For example, if a check-box is used and its state is “on", then this information will be added to the user interface description. Such an approach lets user interfaces migrate “on the fly", preserving the information contained in the user interface.

PutUItree Provides Port Provides a user interface description, expressed in an internal data structure. This is a port which will primarily be used by other, internal components of the UIRenderer component: for example a special purpose Layout Manager could use this to directly manipulate the User Interface structure.

PutUITree Requires Port Requires a User Interface description, expressed in an internal data structure.

DataIn Provides Port To Enable the user interface to be updated from an external data source, this port can be used. It will provide the user Interface with information and data of updated sources.

EventOut Provides Port This port connects to the EventsIn port of the Client component. It will send certain messages to the client when user interaction occurs. Which message will be sent is described in the user interface description. Once received the Client component will process the message and notify the appropriate component that the event has occurred.

7.6.2.3 The 2DScreenRenderer component

7.6.2.3.1 Description

This component renders a user interface description on a 2 dimensional screen, taking into account the constraints of target output device. This component will only be used by a UIRenderer component. Other components will have to communicate with the UIRenderer to get their interface rendered. Possible constraints for a screen are:

- The size (width and height)
- The color depth (for example: only 4 bit colors)
- The refresh rate
- The actual resolution

A graphical presentation can be found in figure 55.

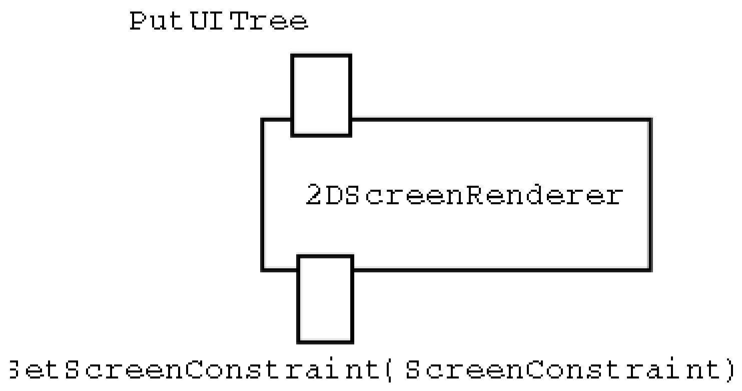


Figure 55: the 2DscreenRenderer component

7.6.2.3.2 Ports

Init This will be used for initialization. At this stage, default constraints can be set or screen constraints defined in an external file can be read.

PutUITree Provides Port This delivers an adapted UI description for presentation on a 2D screen medium.

PutUITree Requires Port This takes the portion of the user interface description that is meant to be shown on a visual 2D screen.

SetScreenConstraint(ScreenConstraint) Port This port is used to set the constraints which must be taken into account by this renderer.

7.6.2.4 The LayoutManager component

7.6.2.4.1 Description

It is important to state that the UIRenderer does not affect the structure of the user interface as it is described in the user interface description it receives. It only maps AIO's to CIO's and instantiates the UI on the target device. To transform the structure of the UI we make use of a LayoutManager component.

When the UIRenderer receives a user interface description it will store it in memory in the form of a tree. This tree in combination with platform constraint information will be forwarded to the LayoutManager component. This component will then transform the structure of the tree keeping the platform constraints in mind. In this way it tries to construct an optimal structure for the user interface for the target device. When this is done the tree will be send back to the UIRenderer for rendering it on the screen. More information of the internal working of layoutmanagers can be found in deliverable 4.3: “Generalization of a Component Based User Interface”.

7.6.2.4.2 Ports

Init This will be used for initialization. At this stage default constraints can be set or target constraints defined in an external file can be read.

PutUITree Requires Port This takes a tree represetation of the user interface which it can transform to an optimal structure for the target device.

PutUITree Provide Port Delivers an adapted tree representation of the user interface for rendering by the UIRenderer.

7.6.3 Palm mobile device related User Interfaces

Suppose an operator is walking around in a building and he or she wants to see an image from a camera on a Palm. Pointing the Palm to an infrared device, that is available on some locations, and starting a camera control application allows the operator to make contact with the system and ask for an image. The system returns the requested image and it is displayed on the screen of the mobile device. This section now describes what is involved to get a camera image on the Palm.

7.6.3.1 An XML based Palm User Interface

When the operator connects to the system, the user settings are initialised. Each Palm has a unique identifier that allows the system to recognize the user, assuming that it is not going to be shared among operators. According to the privileges of the user a XML description of the interface is send to the Palm device, which then will be rendered on the screen.

For now, an example application has been developed that has a user interface description in XML stored on the server. When the client connects to the server and makes a request, a user interface description is sent back. To cope with the system constraints of a mobile device the XML description has been kept as simple as possible (in particular to cope with the memory constraints of the PDA). Listing 3 shows such a description. This description is converted (“server-side”) by using an appropriate XSLT.

The XSLT stylesheet adapts the original XML description by replacing the tags with simpler tags, and removing unnecessary data out of the original description.

Listing 3: A XML camera control user interface description.

```
<UI name="CAMERA CONTROL">
<B name="Select camera"><trigger method="selectCamera()"/></B>
<B name="View image"><trigger method="viewImage()"/></B>
```

```
<B name="Set hotspot"><trigger method="setHotspot()"/></B>
<B name="Set frame rate"><trigger method="setFrameRate()"/></B>
<B name="Zoom camera"><trigger method="zoomCamera()"/></B>
<B name="Exit"><trigger method="exit()"/></B>
</UI>
```

This XML file is then parsed and rendered on the screen as in figure 56, including a desktop and a Palm version.

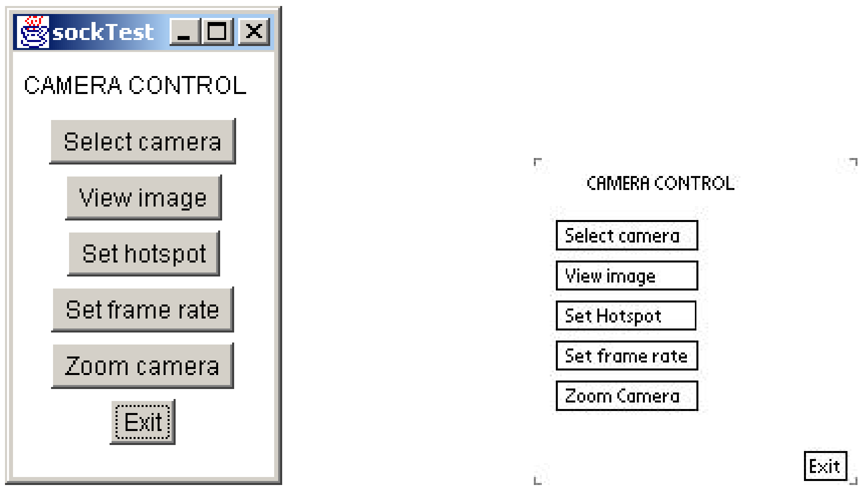


Figure 56: The rendered user interface, on the left side a desktop version and on the right side one for a PDA (Palm IIIc).

7.6.3.2 Getting a camera image to the Palm

Suppose the operator wants to use the zoom function of a camera. The specific camera can be selected. If the Zoom camera button is activated, the system receives the request for this action. From the storage the specified image will be retrieved. In this section it is a capture from our room.

Figure 57 shows the camera image. To get this image to be displayed on the Palm subsequent actions have to be taken. Complying with the system constraints for the Palm means converting the image to a 1-bit bitmap format[30], a size of 160*160 pixels and adding specific headers to the resulting bitmap file.

[30] This is a restriction forced by the used Java API for the Palm handheld device

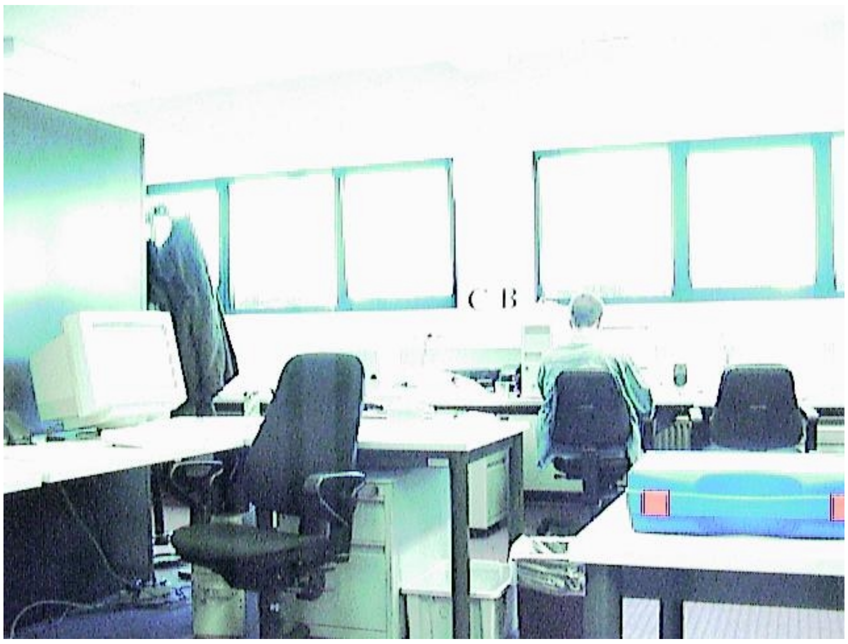


Figure 57: A captured camera image

A gray filter is used on the image to reduce the image to 8-bit format. Applying a dithering filter with a given treshold value creates the 1-bit black and white image.

This original sized image is then scaled to 160 * 160. The scaling factor in this example is chosen to use the total Palm screen size. Figure 58 shows the result of the image processing.

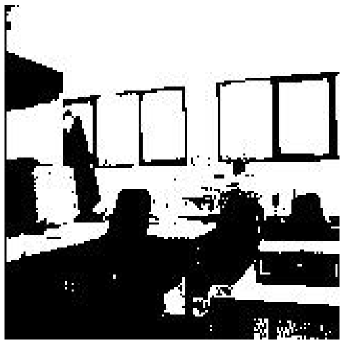


Figure 58: The original image scaled to Palm screen size

Then by adding the width and height to the hexadecimal notation of the image the bitmap file is ready to be transmitted back to the Palm. The Palm receives the file and adds it to the Zoom camera user interface. In order to show the zoom slider, the image size on the Palm can be reduced to 158*140 pixels for instance. The stylus can be used to drag the image up to see the lower part, see figure 59.

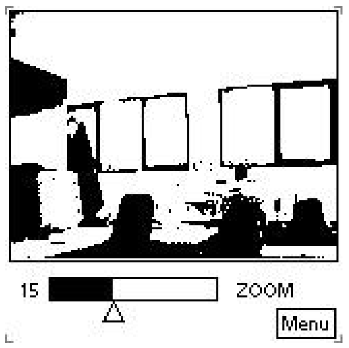


Figure 59: A converted camera image image in the zoom camera user interface

In the chapter about the use cases, chapter 5, there are more figures of other possible user interfaces.

7.6.4 Additional listings

The XML descriptions, of some parts of the user interface prototype of the surveillance system one can find in figures 50 and 51, are listed below. Listing 4 shows the menu section of the user interface while listing 5 covers the bottom left part with the sliders.

Listing 4: Description of the menu bar.

```
<AIO NAME="MenuGroup" TYPE="LineGroup">
  <Property NAME="CameraAppMenu">
    <AIO NAME="JMenuBar44555" TYPE="MenuBar">
      <Property NAME="CompressionMenu">
        <AIO NAME="JMenu4565454" TYPE="Menu">
          <Property NAME="Text">Image Compression</Property>
            <Property NAME="JPEGMenuItem">
              <AIO NAME="JMenuItem54554645"  TYPE="MenuItem">
                <Property NAME="Text">JPEG</Property>
                  <Events>
                    <Out>
                      <Component NAME="CompressionComponent">
                        <Port NAME="CompressionType">
                                    <In>chooseJPEGCompression</In>
                        </Port>
                      </Component>
                    </Out>
                  </Events>
                </AIO>
                </Property>
                  <Property NAME="NoneMenuItem">
                    <AIO NAME="JMenuItem454545" TYPE="MenuItem">
                      <Property NAME="Text>None</Property>
                        <Events>
                          <Out>
                            <Component NAME="CompressionComponent">
                    <Port NAME="CompressionType">
                      <In>chooseNoneCompression</In>
                    </Port>
                  </Component>
                </Out>
              </Events>
            </AIO>
          </Property>
           ...
         </AIO>
       </Property>
        ...
  </AIO>
 </Propert>
</AIO>
```

Listing 5: Description of the panel with all the sliders.

```
<AIO NAME="slidersGroup" TYPE="Group">
 <AIO NAME="fpsSliderGroup" TYPE="Group">
  <Property NAME="fpsSliderLabel">
   <AIO NAME="Label123" TYPE="Label">
    <Property NAME="Text">Frames per second</Property>
   </AIO>
  </Property>
  <Property NAME="fpsSlider">
   <AIO NAME="Slider1234" TYPE="Slider">
     <Property NAME="Minimum">5</Property>
      <Property NAME="Maximum">60</Property>
       <Property NAME="Value">10</Property>
        <Events>
         <Out>
   <Component NAME="Client">
           <Port NAME="Settings">
             <In>setNewFPS</In>
           </Port>
          </Component>
         </Out>
        </Events>
       </AIO>
      </Property>
     </AIO>
     <AIO NAME="focusSliderGroup" TYPE="Group">
       <Property NAME="focusSliderLabel">
        <AIO NAME="Label4556546" TYPE="Label">
         <Property NAME="Text">Focus</Property>
        </AIO>
       </Property>
    <Property NAME="focusSlider">
      <AIO NAME="Slider1254515" TYPE="Slider">
        <Property NAME="Minimum">0</Property>
        <Property NAME="Maximum">450</Property>
        <Property NAME="Value">0</Property>
          <Events>
            <Out>
              <Component NAME="Client">
                <Port NAME="Settings">
                  <In>setNewFocus</In>
                                    </Port>
              </Component>
            </Out>
          </Events>
        </AIO>
    </Property>
  </AIO>
  <AIO NAME="zoomSliderGroup" TYPE="Group">
    <Property NAME="zoomSliderLabel">
      <AIO NAME="Label155454" TYPE="Label">
        <Property NAME="Text">Zoom</Property>
      </AIO>
    </Property>
    <Property NAME="zoomSlider">
      <AIO NAME="Slider545454" TYPE="Slider">
        <Property NAME="Minimum">-1000</Property>
        <Property NAME="Maximum">1000</Property>
        <Property NAME="Value">-900</Property>
          <Events>
            <Out>
              <Component NAME="Client">
                <Port NAME="Settings">
                  <In>setNewZoom</In>
                </Port>
              </Component>
            </Out>
          </Events>
    </AIO>
   </Property>
  </AIO>
  <AIO NAME="SliderButtonsGroup" TYPE="LineGroup">
   <Property NAME="saveSettingButton">
     <AIO NAME="Button56456564" TYPE="Button">
       <Property NAME="Label">Save Settings</Property>
        <Events>
          <Out>
            <Component NAME="Client">
              <Port NAME="Settings">
                <In>saveSettings</In>
```

```
                  </Port>
                 </Component>
                </Out>
              </Events>
            </AIO>
          </Property>
          <Property NAME="loadSettingsButton">
            <AIO NAME="Button5456564" TYPE="Button">
              <Property NAME="Label">Load Settings</Property>
              <Events>
                <Out>
                  <Component NAME="Client">
                    <Port NAME="Settings">
                      <In>loadSettings</In>
                    </Port>
                  </Component>
                </Out>
              </Events>
            </AIO>
          </Property>
      </AIO>
  </AIO>
```

7.7. Client Component

The ***client*** component is connected to all SCSS components that have a port that can provide a user interface description. It is also connected to the *storage controller* by means of a query port. This is done to let the user perform queries on the storage via the client. At last, the client is also connected to the *UI Renderer* component. Via this binding, the client tells the *UI Renderer* component what to show and the *UI Renderer* informs the *client* from particular user interface events.

The Client component contains part of the application logic: it is a kind of glue to coordinate and connect different parts of the system. Although it looks like a monolithic blok (a single component) it can be subdivided into several components, controlling more specific parts of the application logic.

The *client* has following ports:

- **UIDescription port**: this port can be connected to all SCSS components that have to visualize their user interface.
- **QueryStorage port**: via this port, the client sends queries to the storage controller. These queries are initiated by the user.
- **UIOut port**: this port is connected to the *UI Renderer* component. It is used to send user interface descriptions for visualization and to get user interface events.
- **EventIn port** this port is connected to the Event out port of the UI Renderer. The messages sent from that port are related to the user interface description.

References

[1] *Common Test Case*, SEESCOA Deliverable D1.3, April 2000

[2] *New Demands in Safeguards Surveillance Systems*, M. Ondrik, S. Kadner, J. Beckes, http://www.canberra.com/literature/technical_ref/safeguards/demands.htm

[3] *CCTVware*, Loronix Information Systems, USA, http://www.loronix.com/

[4] *VideoSafe+*, Security Management Systems, Denmark, http://www.safecon2000.com

[5] *EDR1600*, EverFocus Electronics Corp., USA http://www.everfocus.com/

[6] *Intelligent Camera Project*, LANL and Motorola, http://www.nis-www.lanl.gov/~bschlei/labvis/lanlmoto.html

[7] *Multimedia Sensor Fursion for Intelligent Camera Control and Human-Computer Interaction*, Mindspring, http://www.mindspring.com/~sggoodri/dissintro.htm

[8] *IQeye3*, IqinVision, http://www.iqinvision.com

[9] Model 2420 Network Camera, Axis Communications, http://www.axis.com

[10] SMACS (Smart Airlock Control System), Fastcom Technology, http://www.fastcom.ch/security/products/SMACS.htm

[11] SFA (Smoke and Fire Alert), Fastcom Technology, http://www.fastcom.ch/security/products/SFA.htm

[12] *Component System*, SEESCOA Deliverable D 3.3.b

[13] *Applying UML and Patterns*, Craig Larman, Prentice Hall, ISBN 0-13-748880-7, 1998

[14] *Component Composer Tool*, SEESCOA Deliverable D 2.3

[15] *Component-based UI development, generalization,* SEESCOA Deliverable D 4.3