CHIL

Functional Requirements & CHIL Cooperative Information System Software Design

Deliverable D2.1 (Part 2: Cooperative Information System Software Design)
of the Project CHIL (Computers in the Human Interaction Loop)
IP 506909

14-July-2004

Fraunhofer-Institut für Informations- und Datenverarbeitung IITB
IBM Czech Republic
Kungl Tekniska Högskolan (KTH)
Laboratoire GRAVIR-UMR5527, INRIA
Research and Education Society in Information Technologies (RESIT)
Universität Karlsruhe (TH) - Institute IPD (UKA/IPD)

Project	CHIL – IP506909 – Computers in the Human Interaction Loop
Title	**Cooperative Information System Software Design**
Workpackage	WP2
Classification	Final
Dissemination level	PU: Public
Version	1.0
Date	14-July-2004
Number of pages	114
Document ID	CHIL-WP2-CooperativeInformationSystemSoftwareDesign-V1.0-2004-07-14-PU
Partners	Fraunhofer IITB IBM INRIA KTH RESIT/AIT UKA/IPD
Authors	Axel Bürkle (Fraunhofer IITB) James Crowley (INRIA) Jan Curín (IBM) Jens Edlund (KTH) Pascal Fleury (IBM) Honza Kleindienst (IBM) Jürgen Moßgraber (Fraunhofer IITB) Wilmuth Müller (Fraunhofer IITB) Michael Okon (Fraunhofer IITB) Alexander Paar (UKA/IPD) Uwe Pfirrmann (Fraunhofer IITB) Jürgen Reuter (UKA/IPD) John Soldatos (RESIT/AIT) Gábor Szeder (UKA/IPD) Martin Thomas (Fraunhofer IITB)
Contributors	Augustin Lux (INRIA) Lazaros Polymenakos (RESIT/AIT) Manfred Schenk (Fraunhofer IITB) Gerhard Sutschet (Fraunhofer IITB) Kostas Stamatis (RESIT/AIT)
Final editing	James Crowley (INRIA) Michael Okon (Fraunhofer IITB)
Synopsis	This document describes the software architecture and design of the CHIL system in all intended CHIL scenarios.
Key words	Architecture/Software Design
Project	CHIL – IP506909 – Computers in the Human Interaction Loop

Revision history:

Version	Date	Changes	Editor
1.0	2004-07-14	Creation of document	Michael Okon

Table of contents

0 Preface

The project CHIL – "Computers in the Human Interaction Loop" is an Integrated Project (IP 506909) funded by the European Union under its 6th Framework Program. The project started on January 1st, 2004 and has a planned duration of three years.

The CHIL team is a consortium of internationally renowned research labs in Europe and the US, who collaborate to bring friendlier and more helpful computing services to society. Rather than requiring user attention to operate machines, CHIL services attempt to understand human activities and interactions to provide helpful services implicitly and unobtrusively. The CHIL consortium is coordinated jointly by the Fraunhofer Institut für Informations- und Datenverarbeitung (IITB) and the Interactive Systems Labs (ISL) of the University of Karlsruhe.

Considerable human attention is expended in operating and attending to computers, and humans are forced to spend precious time on fighting technological artefacts, rather than on human interaction and communication. CHIL aims to radically change the way we use computers. Rather than expecting a human to attend to technology, CHIL attempts to develop computer assistants that attend to human activities, interactions, and intentions. Instead of reacting only to explicit user requests, such assistants proactively provide services by observing the implicit human request or need, much like a personal butler would. To achieve this goal, machines must understand the human context and activities better; they must adapt to and learn from the humans' interests, activities, goals and aspirations. This requires machines to better perceive and understand all the human communication signals including speech, facial expressions, attention, emotion, gestures, and many more.

Based on the perception and understanding of human activities and social context, a new type of context aware and proactive services can be developed. Within the first three years of the CHIL project, four instantiations of such CHIL services will be implemented:

- The Connector: This service attempts to connect people at the best time by the best media, whenever it is most opportune to connect them. In lieu of leaving streams of voice messages and playing phone tag, the Connector tracks and knows its masters' activities, preoccupations and their relative social relationships and mediates a proper connection at the right time between them.
- The Memory Jog: This is a personal assistant that helps its human user remember and retrieve needed facts about the world and people around him/her. By recognizing people, spaces and activities around its master, the Memory Jog can retrieve names and affiliations of other members in a group. It provides past records of previous encounters and interactions, and retrieves information relevant to the meeting.
- Socially supportive workspaces: This service supports human gathering. It offers meeting assistants that track and summarize human interactions in lectures, meetings and office interactions, and provide automatic minutes and create browseable records of past events.
- The Attention Cockpit: This agent tracks the attention of an audience and provides feedback to a lecturer or speaker.

CHIL represents a vision of the future - a new approach to more supportive and less burdensome computing and communication services. The research consortium includes 15 leading research laboratories from 9 countries representing today's state of the art in multimodal and perceptual user interface technologies in European Union and the US. The team sets out to study the technical, social and ethical questions that will enable this next generation of computing in a responsible manner.

The CHIL results will be disseminated and made available to a wide community of interested parties. Several major deliverables, including this document, will be placed in the public domain to promote an active exchange of ideas.

For further information on CHIL refer to the project web site at http://chil.server.de.

This document deals with the second part of deliverable D2.1, the *Cooperative Information System Software Design* of the CHIL system.

1 Introduction

The ultimate goal of the CHIL project is to develop and integrate perceptual technologies and multi-modal interfaces for services or providing assistance to human activities in indoor environments (e.g., in the scope of lectures, meetings, offices, presentations) without disruption. Technology components contributed and developed by several CHIL partners (in the scope of WP4, WP5 and WP6) provide the foundation of these services. The CHIL project requires design, implementation and experimentation with useful services as part of WP3 and WP7 as well as the development of new technologies (WP4, WP5 and WP6). Realizing the goal of the project demands that perceptual interfaces are integrated according to the design, purpose and objective of the targeted services.

Rather than focus on an ad-hoc implementation of particular services, the CHIL project will proceed by specify a structured method for interfacing with sensors, integrating technology components, processing sensorial input and ultimately composing non-obtrusive services as collections of basic service capabilities. Characteristic examples of such basic service capabilities include:

- Identification of people (i.e. users) and artefacts.
- Recognition of user actions and activities, as well as situations.
- Understanding and anticipation of user needs during task performance.
- Information presentation to users.
- Logging of activities, essential facts and deliberations for future use.

Achieving structured integration and composition of sensors, technology components and service elements requires the specification of a set of structuring principles enabling the overall CHIL system to be comprised by these individual components that feature atomicity, while being compatible with the overall system. These structuring principles constitute the *architecture.* This architecture will guide the integration of components by providing a definition of the interfaces between sensor processing, technology and service-oriented components. The CHIL architecture will comprise a rich set of middleware services establishing a continuous, ubiquitous and autonomic sensing infrastructure, along with utilities facilitating the management, configuration and operation of the CHIL services.

The design principles as well as the above set of middleware services and the associated software infrastructure are referred to as the *CHIL architecture*[1]. The main objective of WP2 is to design and implement a prototype version of the CHIL architecture. In the scope of large projects an architecture facilitates development, integration, debugging, while minimizing re-engineering and dependencies on specific technologies and platforms. The vast majority of ubiquitous and pervasive computing projects dealing with sophisticated services based on multiple sensors and perceptual components (e.g., [2], [3]), rely on such architectures. The need for establishing an architecture is particularly compelling for the CHIL project. This is

[1] Following paragraphs of this document elaborate on the properties of the CHIL architecture, as well as the requirements from this architecture

due to the scale of the project, the anticipated sophistication and complexity of the services, the number of partners and demonstration sites, as well as the diversity of technology components contributed to and developed within the project. The CHIL architecture is therefore expected to facilitate integration and implementation of the service prototypes.

The CHIL partners have performed a thorough review of existing architectures for ubiquitous, pervasive and context aware computing (e.g., [4], [5], [6], [7], [8], [2], [3], [9]) and have concluded that none of these architectures meets the needs and requirements for the CHIL project. The sophistication of the CHIL prototypes, as well as the nature and scale of the project impose a wide range of challenges not addressed by any of these existing architectures. In most cases, these architectures integrate a significantly smaller number of technological components than CHIL (e.g., focus solely on vision technologies such as [10], [9], [5]), while targeting less sophisticated services (e.g., [11]).

It can be noted that that several partners of the CHIL consortium have developed their own architectures in the scope of other projects (e.g., [10]) and thus have experience with the challenges entailed in architecture design and implementation. Even though the CHIL architecture is not based on any existing project, CHIL will include use of software components developed in previous projects, provided that these components comply with the objectives and requirements of the CHIL architecture. Such reuse of existing components will accelerate the bootstrap implementation of the CHIL system, while at the same time allowing other CHIL WPs to evolve without tight dependencies to the ultimate output of WP2.

The purpose of this document is to specify the software design of the CHIL architecture, while complying with requests described in ***Functional Requirements*** [1]. Starting from a high level description of the CHIL system and an identification of the functional requirements, this document provides a specification for the basic object-oriented model that will be used to drive these requirements to implementation detail. However, being the first official deliverable of WP2, it also illustrates the concept of the CHIL architecture and explains the rationale behind a developing a new architecture and middleware for the CHIL project. Moreover, it exemplifies the functionality and utilities of the CHIL architecture. This document is strongly connected to the document mentioned above, which provides a through analysis of the functional requirements of CHIL. This functional requirements document accompanies the software design document, since it establishes and illustrates the full range of requirements addressed by the current software design. It is no accident that the present deliverable references the functional requirements document multiple times.

Following this introductory section, section 2 elaborates on general principles adopted for producing the software design. These principles address requirements for software design, organization and maintenance, along with technical guidelines for actually producing the software design. The software organization and maintenance part elaborates on the processes guiding configuration of the software, production of portable software, production of documentation, bug tracking, conduction of revisions, security implications, but also privileged access to software by various groups committed to designing, developing and maintaining the CHIL software. On the other hand, technical guidelines include best practices, blueprints and software design rules taken into account in the scope of the CHIL software design.

Following the general guidelines driving the software design, section 3 presents the overall picture of the CHIL system, and establishes the CHIL architecture as a distributed system. The main types of both hardware entities (e.g., sensors, controlling workstations) and software entities (e.g., perceptual components) are described. This overall view of the CHIL ser-

vice system and its components is followed by an analysis of the 'software' layers constituting the CHIL architecture. This analysis is referred to as the 'CHIL Architecture Framework' and provides a classification of the software components of the CHIL architecture according to their functionality and role in the overall CHIL system. In particular the following layers are specified: Low-Level Distributed Data Transfer (LDT), Metadata (MD), Control (C), Logical Sensors/Actuators (LSA), Perceptual Components (PC), Situation Modelling (SM), Services (S), Service Access and Control (SAC), User Front-End (UFE), CHIL Utility (U).

The notion of layering in the specified component hierarchies implies that components of a certain layer make use of services exposed by components falling in lower layers of the hierarchy. Layering allows for a functional separation of the main building blocks of the CHIL architecture, as is also illustrated in the scope of sample use cases that address indicative uses of the system. Apart from a functional separation, the CHIL architecture framework offers possibilities for modularising the software design towards distributing, managing and auditing the software design efforts.

Following the description of the CHIL Architectural Framework, Section 4 maps the functional requirements of the CHIL architecture to one or more of the layer. This ensures that software designers entailed in certain layers design, take into account the full set of requirements associated with the particular layer. This section also provides an initial prioritisation / classification of the various requirements according to their importance to the overall CHIL system and its goals.

Section 5 addresses the design of the user interface for end users of the CHIL system. Although CHIL is primarily concerned with the design and development of a transparent sensing infrastructure, as well as services, there is necessarily a need for users to activate, configure and interact with CHIL Services. Therefore, a portion of the CHIL architecture addresses the interaction of the CHIL spaces and services (as described in section 5) with users.

Section 0 constitutes the heart of this deliverable. Based on the structure established in the scope of the CHIL architecture framework, this section provides the software design as a set of layers and their interfaces. This section also provides a coarse granularity software design, providing UML diagrams illustrating the main class level entities.

While this document provides a comprehensive software design outlining the core software objects of the CHIL architecture, the design is neither exhaustive nor final. This is intentionally done to avoid unnecessary reverse engineering cycles that could have a hindering impact on the implementation of the CHIL architecture. The current level of detail is mainly imposed by dependencies and interactions of the CHIL architecture to other technical groups of the project. A detailed specification of the composition of service elements into CHIL services depends on the specification of the CHIL services, while a detailed specification of the interfacing and integration of perceptual components, demands a mature description of the technologies to be used. Because the service design and technology components prototyping is in progress it is not realistic to provide a detailed design of software entities down to the attribute and operation granularity. Thus, in the scope of this deliverable, software design is provided at a coarser granularity (i.e. Class / Entity level). However, a basic set of methods and attributes are specified as well.

This will remain a *living* document that will be extended as appropriate to address the whole range of technology components and services to be used within the project, as well as (additional) requirements imposed in the scope of their operation and integration.[2]

[2] See therefore also the chapter about *Open Issues*.

2 General Design Principles

CHIL is a research project. As such, it is not the objective of the CHIL work plan to deliver a ready-to-market commercial product. At first glance, it would appear that a relaxed approach to software engineering principles might be appropriate for the CHIL project. However, one of the main challenges of CHIL is to integrate a heterogeneous collection of software and hardware contributions provided by research groups scattered all over Europe into a single system. Such an effort requires somewhat more rigid rules for smooth cooperation than it would be necessary for a small local group of researchers. The rules, guidelines, and suggestions developed in this document are primarily introduced to ease cooperation between participating research groups, thus facilitating the integration of all submitted contributions into a sound CHIL software environment.

2.1 Keywords

For use in working documents, we define capitalized key words to indicate requirement levels, following the definitions in RFC 2119 (see http://www.ietf.org/rfc/rfc2119.txt). The key words are: "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL".

2.2 Configuration and Dependencies

For each published component, tools, packages, libraries, and required runtime environments, on which components depend, MUST be provided for both design time and runtime as described below. This policy is to enable CHIL partners to reproducibly compile and run the software. Each description MUST at least contain the following items.

Design time configuration ("The software is known to build under…"):

- Platform (operating system; i.e. kernel version, distribution vendor)
- Programming language (i.e. version, dialect)
- Compiler, interpreter (i.e. build options)
- Dependencies (packages, libraries)
- …

Runtime configuration ("The software is known to run under…"):

- Operating system (i.e. localization adaptations, little-endian / big-endian issues)
- Runtime environment (e.g. Java Runtime Environment)
- Dependencies (packages, libraries, required back office software)
- …

Template forms will be published to contain this information. Each contribution to the CHIL software environment, whether it is binary executables or software source code, MUST be labelled with such informative tags.

To be able to tell if include headers, libraries, binaries, etc. can be considered to be part of a typical CHIL operating system installation, all CHIL partners SHOULD announce which operating systems they use. If a CHIL partner ("user partner") succeeds in compiling and/or running a software on a different platform or in a different environment than has been developed by another CHIL partner ("developer partner"), the user partner SHOULD inform the developer partner that the user partner's environment MAY be added to the list of plat-

forms/environments that the software "is known to run under". When a user partner fails to compile or run the developer partner's software, the user partner SHOULD contact the developer partner to solve the problem. Often, a different environment causes such issues and it may turn out that, for example, updating a specific library will fix the problem. In this case, the developer partner SHOULD document the user partner's software environment and required changes. This policy may also help to more efficiently track issues with particular versions of headers, libraries, binaries, etc. during compilation or runtime. Template forms will be published to facilitate standardized configuration communication.

Portability of source code such that program code can be compiled on different architectures without any modifications SHOULD be pursued whenever possible. Possible exceptions are the usage of operating system- and hardware specific functionality. In those cases, platform dependent code SHOULD be encapsulated in separate files such that when targeting a new platform only platform dependent files need to be changed.

For Unix/Linux specific software tools like `configure`, `make`, and `make install` SHOULD be used. A configure script or any other utility which provides equivalent functionality MUST be included with every package. Thus, the presence of all dependencies can be detected. This allows some sort of compile-time configuration of the software. In case dependencies are missing, the script SHOULD give useful error messages (e.g. “You need the gtk2 development libraries to build this software.” instead of “Error: foogbar.h not found. Aborting.”). Makefiles SHOULD have the following conventional targets: `all`, `install`, `uninstall`, `clean`, `distclean` (see the GNU `make` manual for the purpose of these targets and further conventions.

Developers (see *Levels of Commitment*) MAY decide to use commercial software (i.e. integrated development environments, CASE tools) for building contributions to the CHIL software environment. However, it MUST be feasible to build the CHIL software environment from source code without purchasing any commercial software except for proprietary hardware drivers and commercial operating system core libraries. Developers SHOULD use development environments and modelling tools, which are available for a variety of operating systems and which are able to build software for multiple target platforms. Data formats (e.g. source code, project files, MDA data, etc.) SHOULD be used which facilitate data interchange between development- and modelling tools.

Developers of contributions to the CHIL software environment SHOULD provide the tools necessary to build their components along with the components' source code. Both source code and build tools MUST be provided in order to have software classified as source code compatible (see *Levels of Portability*).

2.3 Interoperability and Extensibility

Interoperability and extensibility of the CHIL software environment will be heavily affected by data type systems that are used. Therefore, contributors to the CHIL software environment MUST agree on compatible type systems when data is to be communicated beyond address spaces. Primitive data types that may eventually be exchanged between different platforms (operating systems, processor architectures) MUST conform to the types defined in section 7.18 of the ISO Standard "Programming Languages - C", ISO/IEC 9899-1999 (a.k.a. "ISO-C99") or they MUST comply with the XML type system as described by XML Schema. These agreements are particularly important because built-in standard C data types such as `short`, `int`, `long` and `long` are platform dependent. For example, the data type `int` may

represent a 16-bit integer type on one platform and a 32-bit integer type on a different platform. State-of-the-art ANSI-C-compilers typically support ISO-C99 data types by providing proper header files (e.g. on UNIX platforms, the directive `#include <inttypes.h>` should suffice).

Defined data types include (among others):

`int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

POSIX Standard section 2.5 defines primitive system data types that are compatible with the C99 data types and therefore MAY be used alternatively to the C99 types. Under Unix, these types can by typically included with `#include <sys/types.h>`.

Under Microsoft Windows operating systems, the following definitions may suffice[3].

```
typedef signed char int8;
typedef unsigned char uint8;
typedef signed short int16;
typedef unsigned short uint16;
#ifdef _WIN32
typedef long int32;
#else
typedef signed int int32;
#endif
typedef unsigned int uint32;
```

Byte order of primitive data types MUST be considered when exchanging data between big-endian architectures (such as PowerPC, PA-RISC or SPARC64) and little-endian architectures (such as x86, Itanium or Alpha).

When writing and reading files that may be processed on different architectures, all readers of and writers of the file MUST agree on the byte order. CHIL software developers SHOULD either lay down the byte order in a fixed way as part of the data file format, or provide a field in the file format that allows specifying the byte order for each file individually. In order to keep the implementation of file readers and writers simple, the first approach is recommended. For example, although the .wav audio data file format allows both little endian and big endian encoding, it is recommended that CHIL partners agree on using either little endian or big endian encoding for .wav files, but not both.

When transferring data through a network connection (e.g. Ethernet) byte order MUST follow TCP/IP network byte order unless the sender and receiver explicitly agree on a data format that implicitly lays down byte order. Typical TCP/IP implementations provide functions or macros such as `hton`, `htons`, `htonl`, `ntohs`, `ntohl` to convert between host byte order and network byte order.

[3] Extensions for “int 64” and “uint64” are required.

2.4 Levels of Portability

Over the lifetime of the CHIL project, all architecture framework layers are likely to be populated with a large number of software components. Users may select a subset of the available components from a CHIL software repository and set up a CHIL environment that comprise those components. A software component may be categorized into one of five classes of portability depending on its platform requirements.

Binary compatibility is recommended when compilation of software components on target platforms is inappropriate. Examples are CHIL services running on client devices that users take into the smart room. Such services (e.g. GUI tasks) may for example be coded in a portable abstract machine code (e.g. Java or MSIL bytecode).

Alternatively to binary compatibility a set of pre-compiled variants for a set of different platforms may be provided such that a client may download and run the proper code variants of a CHIL service on the fly.

Table 2-1: Portability Levels

Level of portability	**Requirements**
Platform independent (binary compatible)	Script programs (e.g. Python, Tcl/tk) and managed code (e.g. Java, .NET-languages) can be run on several platforms without any modifications. Platform dependent native code MUST NOT be called by software of this category.
Platform independent (binary available for platform)	Unmanaged code may have been build for a number of platforms. Thus, it may be available in the CHIL software repository. Platform dependent libraries MUST NOT be called by software of this category.
Platform independent (source code compatible)	Software that is source code compatible SHOULD be build for all CHIL platforms and SHOULD be made available in the CHIL software repository.
Platform independent with simple source code modifications	This kind of software can be ported to all CHIL platforms with only simple source code modifications.
Platform independent with significant source code modifications	This kind of software requires significant source code modifications to be available to all CHIL platforms.
Platform dependent	This kind of software depends on one particular platform.

2.5 Device Ontology

Devices will be classified into a special ontology. Among other criteria, this classification will be based on the class of target machine. Three target machine classes are suggested:

- Core machines – These machines present the basic static part of the CHIL infrastructure and are crucial to a working CHIL environment. Software on these computers is supposed to be installed only once by knowledgeable technical personnel who are particularly familiar with setting up CHIL sensor networks and with constituting a sound CHIL software environment (i.e. *committers*, see *Levels of Commitment*). Core machines SHOULD NOT impose any portability requirements on the software they run (see *Levels of Portability*).
- Tentative workstations – These machines are temporarily part of a CHIL environment. However, they are not removed or replaced during an ongoing CHIL event (e.g. meeting). In most cases, workstations will outlast several CHIL meetings. Software on these computers is supposed to be installed and removed on demand by knowledgeable technical personnel who are familiar with installing CHIL software components (i.e. a*ssembling administrators*, see *Levels of Commitment*). Software running on these machines SHOULD be at least source code compatible (see *Levels of Portability*).
- Mobile-/personal devices – This device class comprises mobile and/or personal devices such as PDAs, cell phones, laptops. These devices may arbitrarily join or leave a CHIL software environment without prior notification. Software running on these machines MUST be platform independent with binaries available for all platforms (see *Levels of Portability*).

2.6 Levels of Commitment

The CHIL software environment is likely to continuously evolve over time. In order to ensure consistency and backward compatibility among all components both users and developers contributing to WP2 are supposed to comply with certain rules depending on their levels of commitment. Similarly to the open source project *Eclipse* (http://www.eclipse.org/) a commitment hierarchy is suggested as follows:

- End users – These are users who employ the CHIL software environment, as opposed to those who develop or support it. End Users may or may not know anything about the CHIL software environment, how it works, or what to do if anything goes wrong. End Users do not usually have administrative responsibilities or privileges. End Users are certain to have a different set of assumptions than the developers who created the CHIL software environment.
- Configuring users – Users who customize their experience of the CHIL software environment by configuring an available set of previously installed components. CHIL smart rooms are obliged to permit unauthorized changes only if the resulting configuration still constitutes a sound CHIL software environment. Configuring Users do not need to be granted security credentials in order to apply changes.
- Assembling administrators – Administrators who set up and customize a CHIL software environment, either by configuring installed functionality, or by adding or removing components. Configuration changes made by assembling administrators may have a global impact on a CHIL smart room and may require technical knowledge to preserve a valid state of the CHIL software environment.

- Extenders – Programmers who make changes not envisioned by the original CHIL software environment. In order to support extenders, CHIL software environment components SHOULD provide a rich set of places to "plug in" new functionality and an extensive help system that makes it easy to figure out quickly how to utilize these extension points. Extenders MAY NOT provide extension points and help information for their components. As a consequence, they SHOULD NOT distribute these components beyond the scope of their institution.
- Publishers – Once one has written something useful, other CHIL users may want it. In order to put someone in the position to incorporate published components documentation and reference material MUST be published along with the software component itself. Software components are classified according to their levels of portability and published to the CHIL software repository.
- Enablers – Once a contribution to the CHIL software environment has been published, the next step is to enable others to extend it in ways one does not foresee ("functionality-to-be-plugged-in"). This is accomplished through "places-to-plug-in-functionality". Enablers MUST publish those extension points.
- Committers – For special purposes such as demos or productive environments CHIL software environments may need to be set up that comply with certain requirements such as compatibility and portability. Committers incorporate trustworthy CHIL software components into global releases of the CHIL software environment.

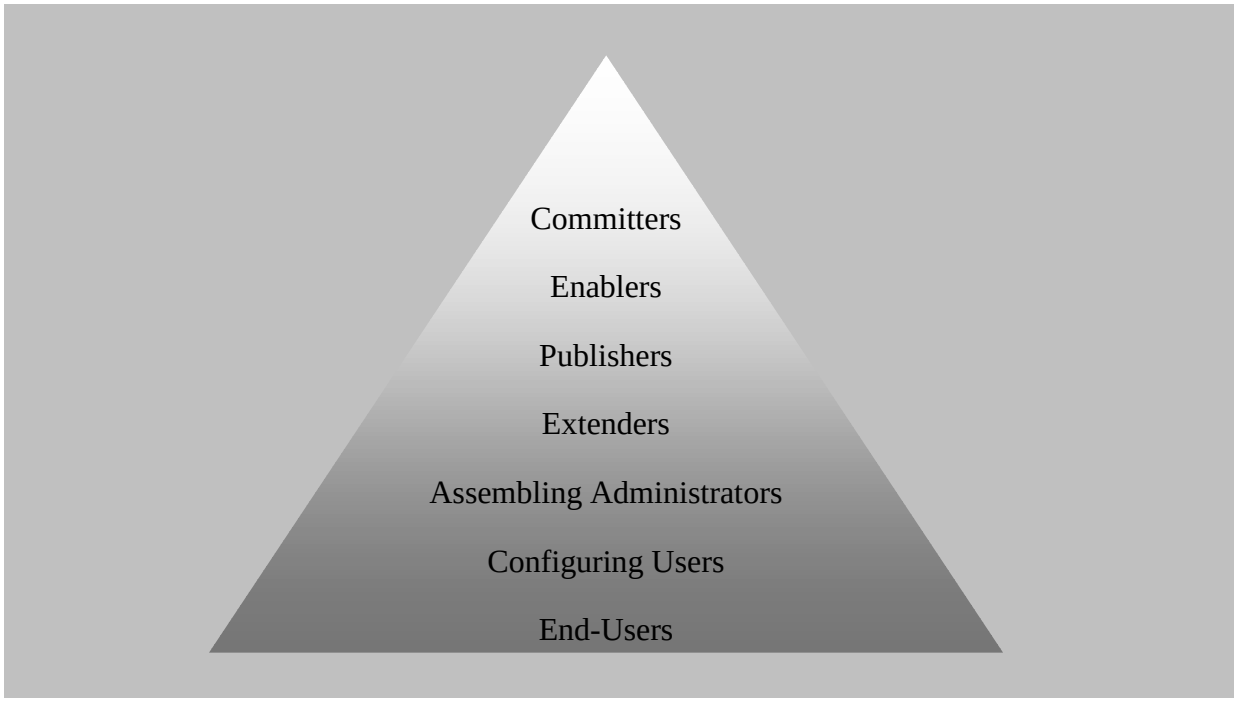

Figure 2-1: Commitment Levels

2.7 Design Rules

Software developers SHOULD share a consistent set of rules for contributing to the CHIL software environment. For each rule, the roles of CHIL contributors are given to specify to

whom the rule applies. Each CHIL contributor SHOULD obey all rules that pertain to his or her respective role. The first rule of the CHIL software environment is as follows.

Table 2-2: "Contribution"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Everything is a contribution.						

The CHIL software environment will not be a monolithic system. It will rather comprise a number of loosely coupled components that will be put together and configured depending on particular scenarios. With growing acceptance and maturity of the CHIL software environment there will be a lot of contributions. However, components that are not required for a particular scenario MUST NOT hamper the performance and the experience of the overall CHIL software environment. This leads to the second rule:

Table 2-3: "Lazy Installation"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Contributions to the CHIL software environment SHOULD be installed and started up only when a particular scenario explicitly requires them.						

Software components, which do not provide any functionality to a particular scenario, MUST NOT be required for mandatory components to run. Moreover, the presence of components that are not operational in the course of a particular scenario MUST NOT have an impact on the CHIL software environment as required by the following rule.

Table 2-4: "No Function No Impact"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Components that do not provide any functionality in course of a particular scenario MUST NOT have an impact on other components or on the overall CHIL software environment.						

During the evolution of the CHIL software environment, several releases of components may become available. Moreover, components that may be added later, may subsume functionality that has previously been provided by a number of distinct components. Since existing software may rely on particular configurations, both major and minor releases of components MUST NOT be replaced (see *Revision Control*).

Table 2-5: "Sharing"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Add, don't replace. Once having been published, both major and minor releases of components MUST NOT be replaced by obsolete nor by subsequent versions.						

In order to facilitate component interoperation contributions to the CHIL software environment MUST conform to expected interfaces. In particular, a component MUST declare explicitly where it can be extended.

Table 2-6: "Conformance"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Contributions MUST conform to expected interfaces.						

Table 2-7: "Explicit Extension"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Components MUST declare explicitly where they can be extended.						

In order to inspire developers with confidence in extending other components, the following three rules SHOULD be obeyed when components expose their programmatic interface for others to use.

Table 2-8: "Explicit API"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
The API SHOULD be separated from internals.						

Table 2-9: "Stability"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Once others have been invited to extend one's component, one MUST NOT change the rules.						

Table 2-10: "Defensive API"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Those parts of the API in which one does not have confidence MUST NOT be published.						

Once a contribution to the CHIL software environment has been installed and/or run in the context of a particular CHIL smart room configuration, the assembling administrator who conducted the installation SHOULD report both the success and the failure of the installation procedure to the responsible developers of the respective contributions. This will help to track installation and configuration issues.

Table 2-11: "Run It And Report It"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Once a contribution to the CHIL software environment has been installed and/or run in the context of particular CHIL smart room configuration, both the success and the failure of the installation procedure SHOULD be reported to the responsible developers of the respective contributions.						

An important rule to improve the serviceability and usability of contributions to the CHIL software environment is to take full responsibility for them and to provide means such as log files and error dialogs to make it particularly easy for users and developers to identify one's component as the source of a problem.

Table 2-12: "Responsibility"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
One MUST clearly identify one's contributions to the CHIL software environment as the source of problems.						

The next three rules apply to the etiquette concerning the invitation of others to make use of extension points.

Table 2-13: "Invitation"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Whenever possible, one SHOULD make minimize the effort required for others to extend one's published contributions to the CHIL software environment.						

Table 2-14: "Fair Play"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
All developers who contribute to the CHIL software environment MUST adhere to the same rules.						

Table 2-15: "Diversity"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Extension points MUST accept multiple extensions.						

Once a component has been incorporated into the CHIL software environment control may be passed to external code. In order to avoid malicious behaviour (i.e. caused by uncaught exceptions) potentially dangerous code MUST be wrapped by appropriate try-catch statements. In especially, when providing extension-points developers are responsible to protect their components against misbehaviour on the part of extenders

Table 2-16: "Good Fences"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
When control is passed outside a component, potentially dangerous code MUST be wrapped adequately.						

Table 2-17: "Safe Software Environment"-Rule

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
As the provider of extension points such as APIs for components developers MUST protect their contributions against misbehaviour on the part of extenders.						

2.8 Design Principles

This section provides suggestions on the principles to be applied towards developing the CHIL software environment. These design principles are in-line with the objectives of the CHIL architecture. In particular, they boost an architecture mainly facilitating: (a) standard, structured integration of components, (b) bridging between service constructs and technology components. Note that the ultimate design SHOULD take into account these principles, while it is quite difficult to use all of them with no deviation.

Table 2-18: Design Principle 1

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Adhere to distributed computing standards (e.g. W3C standards compliant Web Services).						

The CHIL architecture addresses the integration of numerous distributed components. In the scope of the resulting distributed system, the use of distributed computing standards (e.g. Web Services) facilitates the specification of integration mechanisms through levering existing transport schemes as well as service-oriented aspects (e.g. service registration and discovery). Basing the implementation on distributed standards will also facilitate the extensibility, evolution and longevity of the architecture.

Table 2-19: Design Principle 2

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Do not exclude particular platforms, operating systems, and sensor vendors.						

Develop the architecture independently of particular platforms, operating systems and sensor vendors towards assuring its generality and increasing the scope of the system.

Table 2-20: Design Principle 3

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Adopt object-oriented approaches.						

Modelling basic entities of the system as objects and adopting object oriented approaches to software design and implementation is expected to provide opportunities for reusability, encapsulation and polymorphism of architecture components.

Table 2-21: Design Principle 4

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Reuse existing middleware and software components.						

Existing components and platforms fulfilling the architecture requirements SHOULD be re-used. Such components SHOULD be incorporated into the overall design of the architecture. This will accelerate the design and subsequent implementation of the architecture. A characteristic platform that can be taken into account in the design and reused in the scope of the implementation is the "NIST Smart Flow System" middleware (www.nist.gov/smartspace/toolChest/nsfs/).

Table 2-22: Design Principle 5

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Define interfaces (APIs) between components. Develop component wrappers based on these APIs.						

APIs will facilitate the integration of heterogeneous components as specified in the scope of the CHIL architecture. Given a set of APIs, the whole range of components engaged in the CHIL systems (e.g., service constructs entities, perceptual interfaces) SHOULD be wrapped to conform to the APIs.

Table 2-23: Design Principle 6

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Separate configuration from application logic.						

Configuration parameters MUST NOT be hard coded within application logic (rules) driving the integration of components. Management of configuration logic MUST be allowed independently of the architecture, towards configuring CHIL system instances corresponding to different physical (smart) rooms, technology components, etc.

Table 2-24: Design Principle 7

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
Programming patterns and best practices used in the scope of similar pervasive / ubiquitous computing architectures SHOULD be exploited.						

The use of programming patterns (e.g. MVC) pattern will accelerate the design process. At the same time best practices applied in similar projects provide a sound basis for designing particular aspects of the CHIL architecture.

Table 2-25: Design Principle 8

End-Users	Configurers	Assemblers	Extenders	Publishers	Enablers	Committers
The produced design SHOULD be incremental and scalable.						

Given the scale of the CHIL architecture in terms of the number of technology components, complexity of services, and sensors an incremental and scalable design catering for the gradual addition of new components and features SHOULD be produced. This will allow for augmented early prototypes based on additional technology components and according to evolving service requirements.

2.9 Security Considerations

No critical changes to the CHIL software environment MAY be made by *end users* and *configuring users* (see *Levels of Commitment*). Significant changes to the CHIL software environment MUST be authorized based on sufficient security credentials. Security will be user- and role-based. In especially, configurable components that provide system critical configuration- and/or extension points MUST support the CHIL security system.

Security mechanisms do not necessarily need to be fully implemented for the prototype version of the CHIL software. Still, care MUST be taken for the design easily allowing for later adding elaborated security mechanisms. Taking into account at least rudimentary security considerations may help to fulfil this requirement and is therefore strongly recommended.

2.10 Documentation Guidelines

Contributions to the CHIL software environment are assumed to be bundled in software packages. A software package is a collection of files that is released as a single package (e.g. as a .tgz or .rpm file). Software packages SHOULD be accompanied with information about how to build, deploy, install, and operate published components. The collection of documents will comprise formal, semi-formal and informal unstructured material. External documentation is mainly targeted for CHIL partners as well as end-users using the CHIL software. Internal documentation helps a CHIL partner enhancing its internal software development process. External documentation typically includes application independent information such as a NEWS files, INSTALL notes, or README notes, an FAQ, as well as application dependent documents such as API documentation, manuals or tutorials. Internal documentation typically includes a ChangeLog file and application specific documents that specify or describe the software in parts or as a whole, typically in a functional manner.

2.11 External Documentation

External documentation of a software package MUST include a NEWS file that for each release summarizes the changes relative to the previous release. It is recommended to create a single NEWS file that is updated for each release by adding a new section for each new release.

INSTALL notes MUST contain all prerequisites (other software packages) that are needed to build or run the software package. Hardware/software configurations (“platforms”) that the software is known to run under SHOULD be collected in this document. Platform specific issues SHOULD be collected in the INSTALL notes.

The README file SHOULD contain general information about the software package (e.g. purpose and scope of the software, contact and origin information (institution and/or main author(s)), download link, links to mailing list, etc.) as well as general instructions such as pointing to the INSTALL or NEWS file.

Application dependent external documentation MAY contain e.g. API documentation, a programmer's manual or tutorial, and/or a user's manual or tutorial or quick reference.

2.11.1 API Documentation

API documentation is required in particular when a CHIL partner uses another partner's software package. Most cross-partner use of APIs is expected to occur along the borders of the CHIL architecture layers.

Integrated documentation systems should be used where applicable, such as doxygen or javadoc for automatic generation of API documentation. API documentation typically consists of a collection of classes and method signatures with a short informal, preferably functional description of their semantics, including the purpose of each parameter and return value (if applicable) as well as side effects, if any. In object-oriented languages, a UML class diagram may be a good choice to give an overview over the classes of an API.

2.11.2 Programmer's manual

A programmer's manual (or programming reference) adds to the core API documentation in so far as it describes the semantics of functions or methods in more detail and focuses on the underlying interface design concepts rather than on just listing all classes and methods or functions. For that purpose, the manual may have reordered or restructured its contents in order to fit the design concepts in a proper manner.

2.11.3 Programmer's tutorial

A tutorial serves as an introductory document. It does not need to be complete in the sense of containing all classes and methods or functions. Rather, it SHOULD focus on the essential functionality that most programmes be omitted or just mentioned. For further details and a complete reference, the tutorial should refer to the programmer's manual or API documentation.

2.12 Internal Documentation

Internal documentation consists of a ChangeLog file for internally tracking changes during development. Depending on the kind of software, there are various models for describing the architectural and semantics of software. Each CHIL partner should decide which model fits best to its software. Examples include but are not limited to UML models, data flow diagrams, entity relationship models, pseudo code, decision tables, rule-based models, finite automata / state machines, petri nets, etc.

2.12.1 ChangeLog

It is assumed that each partner uses a revision control and/or configuration management tool for software development, and that such tools are provided either by the partner or hosted by IPD/UKA, each check-in to the software repository SHOULD be accompanied with a proper ChangeLog entry describing informally which changes where applied to what files. Typi-

cally, the check-in date and author's name are automatically tracked by the revision control or configuration management system.

There MUST be a ChangeLog file for each software package. Each release of a software package (be it minor or major, stable or unstable) MUST be accompanied with a proper ChangeLog entry in the package's ChangeLog file that describes all relevant changes relative to the preceding release. An entry in the ChangeLog file consists of date, author and an informal summary of what has been changed on what files. An entry SHOULD be put into the software package ChangeLog file for each check-in, especially if releases occur infrequently.

2.13 Bug Tracking

Good bug reports are considered to be valuable contributions to the development of any software project. But just like writing good software, good problem reports involve some work.

2.14 Revision Control

A configuration management system SHOULD be used. The usage of *Subversion* (see http://subversion.tigris.org/) is recommended, as it offers a number of novel features compared to its predecessor CVS (see http://www.cvshome.org/). Developers MUST write descriptive and detailed log messages to each commit. A single commit MUST NOT contain more than one set of changes. Each new feature or bugfix MUST be committed separately (i.e. one commit MUST fix exactly one bug or add exactly one feature). Access to local repositories MAY be provided to other CHIL developers beyond the scope of one's own institution (i.e. one MAY consider to allow at least read-only access). Early and frequent releases are likely to be a critical part of the CHIL development model. Contributors to the CHIL software environment are encouraged to frequently release versions of their software components. Releases MUST be versioned similar to the versioning guidelines of the Linux kernel. Each version has three numbers X.Y.Z, where X is the major version number, Y is the minor version number and Z denotes the patch level. Major version numbers MUST start at zero. They MUST only be incremented as a result of a significant change (for example, changes such that software written for one version no longer operate correctly with the subsequent version). Differences between two different patch levels MUST be small (e.g. bugfixes and minor features).

2.15 Software Engineering for a cognitive CHIL software environment

The layering of the CHIL software environment and its explicit support for ontological knowledge bases is likely to have an impact both on how software contributions to the CHIL cognitive middleware will be developed and on how application developers will make use of the novel features of this architecture framework. Software libraries and ordinary software toolkits will probably not provide sufficient support for CHIL extenders, publishers, and committers (see *Levels of Commitment*). This is why development tools and methods will be devised to explicitly facilitate the evolution of the CHIL software environment and the creation of higher-level applications. For each layer of the CHIL software environment the most significant software development challenges will be scrutinized. In especially, the following issues need to be taken into account:

- Ontology engineering: Knowledge acquisition tasks will account for a significant part of the efforts to devise the CHIL software environment. Despite the fact that there are mature tools both for knowledge acquisition and software development there is mostly no strong link between software source code and the extensional and intentional knowledge the software is working on. The CHIL software toolkit should assist both application developers and contributors to the CHIL software environment in refining Ontologies and software components.
- Event correlation: Since the CHIL software environment will principally follow an event driven approach, event correlation will primarily define the control flow of applications. Software development tools should provide code generation capabilities that let the programmer specify the events to which his (or her) code will react. Tool support for event handling may automatically generate a framework comprising appropriate event handlers. Application specific program code will be complemented by an event correlation engine that delegates occurring events to appropriate event handlers.
- Real-time constraints: The quality of the user experience for software written in the CHIL software environment will be significantly influenced by how well real-time constraints are respected. In particular, constraints defined on different layers may be related. The CHIL software toolkit should assist developers in writing code that complies with real-time constraints taking into account the layered architecture of the CHIL software environment.
- Situation-/scenario awareness: Software contributions to the CHIL software environment may operate differently depending on the current situation or on a particular scenario. Assistance for location and scenario awareness programming should be a primary goal of the CHIL software toolkit.
- CHIL software repository: There will likely be a significant number of extenders, publishers, and committers (see Section *Levels of Commitment*) who will contribute to the CHIL software environment. Therefore, conventional configuration management approaches may prove to be unsatisfactorily when it comes to assembling a working CHIL software environment comprising contributions from a number of distributed sites. The CHIL software toolkit should assist users in managing compatibility issues.
- Code generation: In general, all contributors to the CHIL software environment should share the same coding conventions (see Section *Interoperability and Extensibility*). In order to make it particularly easy to follow these rules the CHIL software toolkit should support writing software according to defined conventions and advice programmers when software documents deviate from these conventions.

3 System Architecture

3.1 System Structure

The CHIL system consists of hardware and software components that work together to assist the user to accomplish tasks in the most efficient manner. Hardware components are described in section 3.2, while software components and architecture in section 3.3 and 0.

3.2 Hardware (sensors, devices)

The following hardware devices are part of the CHIL system infrastructure:

- A variety of sensors including cameras (fixed digital cameras, an analogue pan-tilt-zoom (PTZ) camera and a '*panoramic camera*'), microphones, close talking microphones, microphone arrays as described in ***Initial Specification of the Sensor Setup*** [12] [4];
- Controlling workstations (e.g. hosting sensor controllers and perceptual software);
- End user terminals (e.g. laptops, PDAs, smart phones);
- Auxiliary peripherals and devices (in particular a projection screen).

Sensors are controlled by appropriate software (e.g. the capture drivers of an audio or video stream, controlling software of PTZ camera), hosted in workstations (i.e. sensor controller).

Sensor Controllers are workstations hosting the sensor control software. These entities are usually directly connected (i.e. attached) to the sensors, and make available the sensors' raw data to other entities (Figure 3-1: Sensor Controllers and Device Controllers).

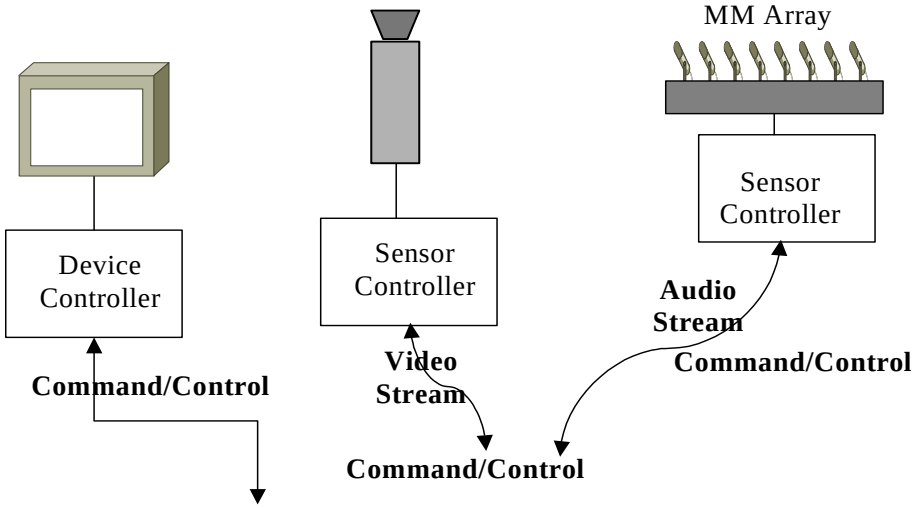

Figure 3-1: Sensor Controllers and Device Controllers

[4] Note that ***Initial Specification of the Sensor Setup*** [12] specifies a minimum sensor set for the CHIL system; additional sensors may be introduced in later stages of the project.

3.3 Software (low-level drivers, perceptual interfaces, services)

In terms of software components, the following types of distributed entities are assumed in the scope of this document:

Device/Peripheral Controllers:

Low-level software components (e.g. Drivers) used for interfacing controlling peripheral devices (e.g. for regulating the environment).

Perceptual Components:

These are software components realizing perceptual providing interpretations of sensor data. They receive sensor data from sensor controllers and produce output relating to who, where and what.

Context (Aware) Entities:

These entities process the output from perceptual components (from one or more perceptual entities) and encode contextual information at higher levels of abstraction than single perceptual entities.

Non-obtrusive service entities:

This class of components comprises applications exploiting perceptual components outputs, as well as higher layer information (e.g., context) towards providing information and automating tasks in a non-obtrusive (or non-disruptive) manner.

3.4 SW-Architecture (middleware)

3.4.1 CHIL agent based global architecture (high level)

A global architecture has been proposed in CHIL Annex I – "Description of Work". This architecture has been updated as follows:

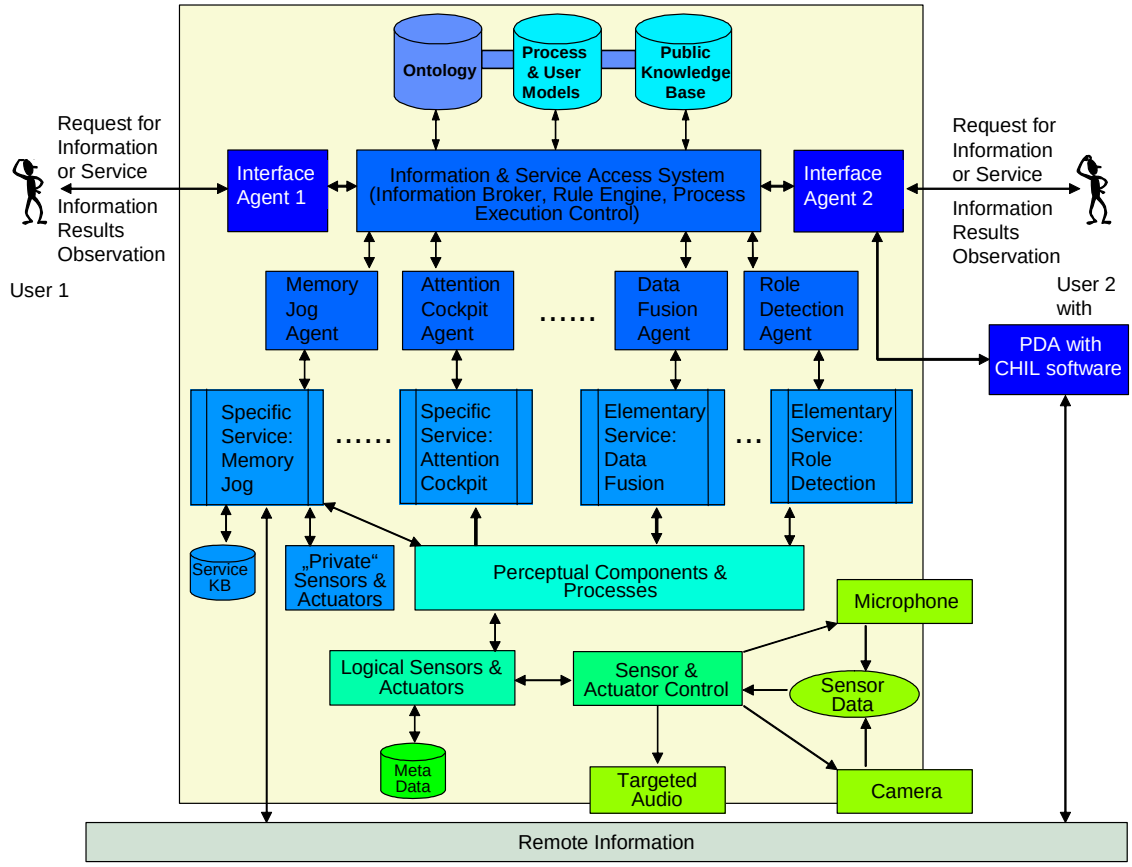

Figure 3-2: CHIL agent based architecture

Ontology:

The Ontology contains the common concepts, relations, entities and rules of order in the meeting room and lecture room domain.

Process & User Models:

These constitute the descriptions of the services & participants (roles, profiles, social aspects) in the conceptual context, which the broker merely understands.

Public Knowledge Base:

The Public Knowledge Base holds the dynamic knowledge (instances) for a planned or present lecture or some post-session actions like writing & disseminating the minutes, taking knowledge from inference into account.

Information and Service Access System:

The Information and Service Access System with its subparts Information Broker, Rule Engine, Process Execution Control is the central intelligence for decision support, resource planning, conflict and failure handling, process planning, execution and control. It acts as the central access point for the domain knowledge. Another task of this component is to manage the user profiles.

Interface Agents (IAs):

These are the agents, which act on behalf of the user via a multi-modal well adapted human computer interface.

Service Agents:

These are the bridges (cardinality n to m, in first approach 1 to 1) between the "lower" level services and the broker. These agents may assist in managing resources and conflict resolutions. Agents are communicating with the broker and with other agents on a conceptual base described in the common ontology. The interfaces to the services include the mapping from the syntactical level of the services to the semantically level of the agent.

Elementary Services:

Elementary Services are common services for domain tasks. They are software-oriented and include role detection, browse context, Internet access management, information retrieval services and further more.

CHIL Services:

CHIL Services as Memory Jog, Connector, Attention Cockpit, etc. consist of a set of Elementary Services. Special mechanisms control their semantic characteristics.

Service KB:

The Service knowledge base is private to the service, and may contain a vocabulary only known and handled by a specific service. There may exist a mapping to the common domain ontology (see Service Agents)

Private Sensors and Actuators:

These sensors are dedicated to one specific service and are only accessible and controllable by this service. For example a dedicated service containing speech recognition runs on the user's notebook and uses the notebook's internal microphone as sensor.

Perceptual Components and Processes:

Sensor Base (incl. Pre-processing) is the place where the incoming sensor data is stored. There are several Elementary and CHIL Services that make use of the data for interpretation purposes. The resulting information can be used by other services or can be the answer to user requests.

3.4.2 CHIL Layer model

The global architecture described above will be mapped to the following layered model. This model is described in the document ***Thoughts on CHIL-Architecture*** [14]. It shows 9 different horizontal layers of the proposed CHIL architecture. A vertical layer "CHIL utilities" is added to provide global timing and other basic services that are relevant to all layers. The ontology is also part of the whole architecture.

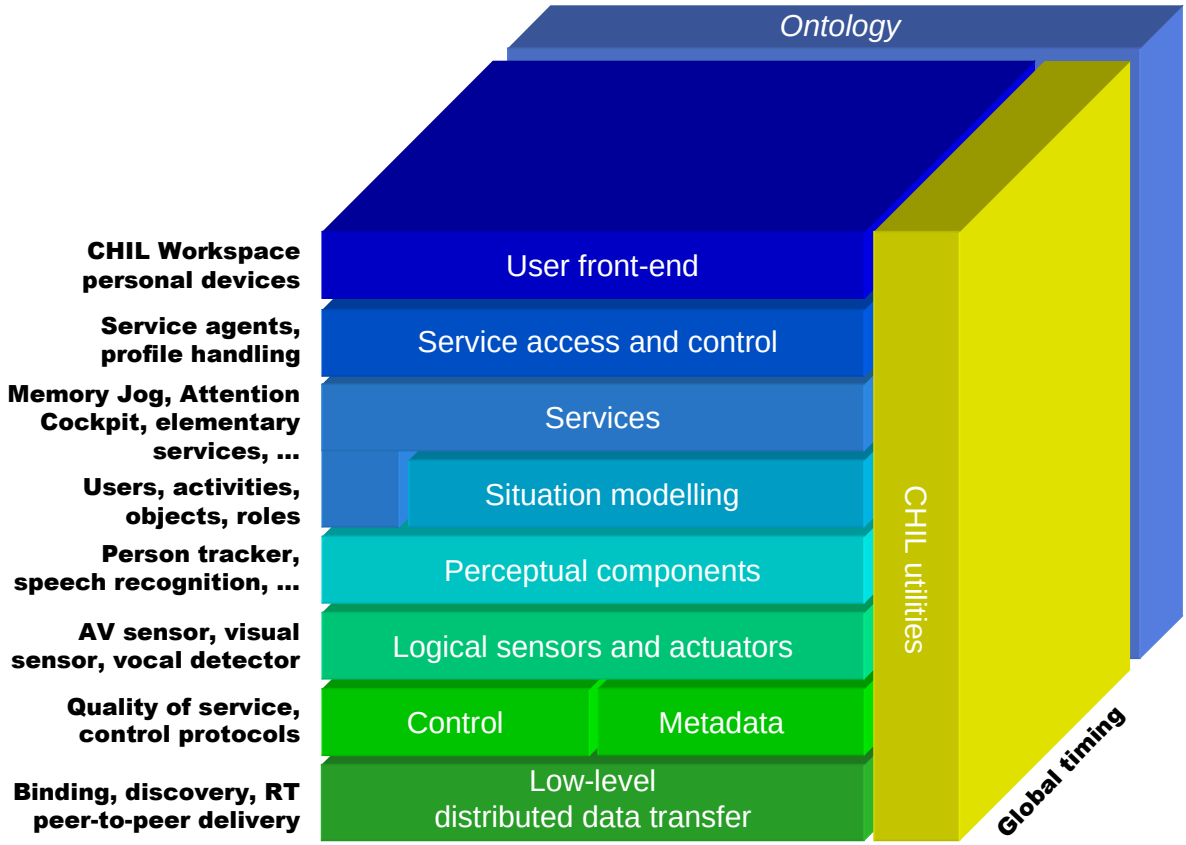

Figure 3-3: CHIL Layer Model

3.4.2.1 User front-end

3.4.2.1.1 Functionality

The user interface layer contains the InterfaceAgents, which act as personal assistants of users. InterfaceAgents take care of the demands of users by subscribing them to the CHIL-services they want to use. To do this, the InterfaceAgent interprets the current role profile of a user and its specific settings in the user profile or reacts due to a direct input of a user.

The CHIL-services for which a user is subscribed notify the InterfaceAgent about information that should be presented to the user. The InterfaceAgent then assures that the user gets the information in the way the user prefers by verifying the specific settings in the user and role profile.

3.4.2.1.2 Key abstractions

If a user is recognised when entering the room, a new InterfaceAgent is created to act as a personal assistant for the user. The InterfaceAgent queries the current role of the user to retrieve the role profile from the RoleDetectionAgent. If this user is already identified and he/she has a user profile, this is blended with the role profile. Otherwise the default settings of the role profile apply.

Thereafter, the InterfaceAgent subscribes to the CHIL services requested by the user by interpreting the blended user/role profile. For example, if a lecturer wants to be informed about an attention loss, his (or her) InterfaceAgent subscribes to the Attention Loss Service.

During a scenario, the InterfaceAgent must take attend to role changes by its user. If a role change occurs, the InterfaceAgent must again read the role profile for the new role of the user, blend it with the user profile, subscribe to new services and unsubscribe from services that are no longer required.

If a CHIL-service produces information for a user, the InterfaceAgent must retrieve the user's preferred means of notification by querying the user/role profiles and then use output components (like Targeted Audio) or a GUI on the users laptop/PDA.

3.4.2.2 Service access and control

3.4.2.2.1 Functionality

This layer comprises the service agents, and their management, by encapsulating/wrapping the underlying agent platform. This layer performs the “user specialized wishes” of the “User Front-end”-layer to the underlying “Services”-layer by agent-based mechanisms, e.g. user-triggered as described in the use case “BrowseContextInformation” or profile-interpreted as described in use case “NotificationAboutAttentionLoss” in ***Functional Requirements*** [1]. Inversely this layer is responsible for providing information from the “Services”-layer to the “User Front-end”-layer according to the manner in which a service is notified, e.g. synchronously or asynchronously.

The appropriate agent objects interact with objects of the proximate layers “User Front-end” and “Services”, in first approach not directly with objects of “Situation Modelling” This sort of interaction may be necessary later on, and with objects of their own kind, especially via a Broker/Facilitator agent. The interaction with other agent objects and the “User Front-end”-layer follows the communication mechanisms of the agent platform, while communication with the “Services”-layer follows internal mechanisms, which will be specified later.

Furthermore this layer provides access to the Ontology-based knowledgebase of the CHIL-System, so that a common ontology can be used for communication and interpretation of contents.

3.4.2.2.2 Key abstractions

General key abstractions/words are “Agent-Based Technology”, “Ontology”, “Knowledge-base” and “Wrapper Techniques”. CHIL-specialised key abstractions/words are the classes “ServiceAgent” and its derivations, e.g. “ConnectorAgent”, “AttentionCockpitAgent”, “MemoryJogAgent”, “ProfileHandlerAgent” and “RoleDetectionAgent”, as well as its common interface “CHILAgent”. Further on there exist an “OntologyAccessAgent” and a “CHILAgentManager”.

Our first approach is that in the beginning or during a CHIL-scenario every Service Agent attaches to one and only one Service, e.g. “ConnectorAgent” to “ConnectorService”, etc. A user driven request or subscription is handled via InterfaceAgent->CHILAgentManager (as Broker or Facilitator)->Special Service Agent->Special Service and backwards.

3.4.2.3 Services

3.4.2.3.1 Functionality

This layer provides reusable components that help users solve tasks quickly and easily. These reusable components constitute elementary service (constructs/entities) that can be aggregated to composite services representing the CHIL non-obtrusive services (e.g., the Memory Jog, the Attention Cockpit, the Connector, the Socially Supported Workspaces). Composition of elementary service elements into sophisticated (non-obtrusive) CHIL services will be realized based on appropriate control logic (e.g., an event-based triggering mechanism). As a result, the functionality of the services layer consists of:

- The set of elementary services comprising the CHIL services
- A Control logic guiding the aggregation and activation of particular elementary services in the scope of the CHIL service.

The service layer models the elementary service, the composite service and provides the means for implementing control logic.

3.4.2.3.2 Key abstractions

Key abstractions can be defined at the level of an elementary service, as well as at the level of composite service. At the level of elementary services abstractions represent individual service oriented functionalities, which are offered within or out of the scope of perceptual interfaces. Characteristic examples of elementary services relating to perceptual interfaces include:

- A TTS (text to speech) service
- A Targeted Audio Service
- A Summarization Service
- An IR Service
- A Keyword Extraction service

These services are abstracted based on appropriate service oriented wrapping of perceptual components. Characteristic examples of non-perceptual service abstractions include:

- Any database search service
- Any service regulating the environment (e.g., room temperature)

Key abstractions at the composite service level have already been defined in CHIL Technical Annex as an output of WP3 Services tasks, e.g. Memory Jog, Attention Cockpit, Socially Supported Workspace, Connector, etc.

3.4.2.4 Situation modelling

3.4.2.4.1 Functionality

This layer is a collection of abstractions representing the environment context in which the user interacts with the application. Ideally, it should act as a “database” that maintains up-to-

date state of objects (people, artefacts, situations) and their relationships. More insight will be needed to understand how short term and long-term memory capabilities could be modelled at this layer.

Additional function of the situation-modelling layer is to act as an "inference engine" that regularly deduces and generalises some facts during the process of updating the context models as a result of events coming from the underlying layer of Perceptual Components.

Once CHIL ultimately becomes a toolkit, this layer can be viewed as an Abstract Context Toolkit, an analogy to Abstract Window Toolkit in the visual world.

We expect this layer to be an inevitable part of any future application dealing with contextual information that captures models of users and environments, especially in cars, homes, and offices.

3.4.2.4.2 Key abstractions

The situation modelling process contains the set of context-related abstractions of the modelled universe. Therefore, there is a need to define the boundaries of the modelled world, which is embodied in form of a **Context Boundary Container**. In this container, models of abstractions may appear or disappear, as people or other artefacts enter or exit the modelled universe. For CHIL, this boundary will typically be carved out by the limits of a CHIL room. However, for other systems it may be something smaller like an interaction zone in front of an appliance or a cockpit, or something bigger like a building, a street, or a city.

In this context boundary container, real objects will be modelled and tracked for their intrinsic information and their relationships. We basically distinguish between three categories of objects:

- **Subjects**, modelling people and/or animals,
- **Artefacts**, which model relevant objects of the scene, like *whiteboard*, *tables*, *doors*, but also personal belongings like *notebook*, *PDA*, *mobile phone*
- **Situations/Roles/Tasks**, which models things like *meeting*, *presenter*, *participant*, and also lower-level tasks like *Person XY is using whiteboard*.

The situation modelling process will also track the **state** of these objects, and their relationships. The state of an object depends on the category to which the object belongs; location tracking makes sense for people and mobile artefacts, but not for the whiteboard. Level of attention, on the other hand, does make sense only for people. ID may be easily discovered from a device (like a PDA), but may need fusion of several inputs to specify a user's identity. And for devices or fixed artefacts, an *in-use* status can be monitored.

Relationships may be known in advance, or discovered. Two types of relationships are modelled in CHIL:

1. **Static relationships**, which represent the common knowledge of the modelled world, static facts like *a meeting is a collection of people and artefacts*, prerequisites like *a meeting has a presenter and an audience*, or embodiment of physical limits of the real world, as *to write on the whiteboard, a person must be close to it*. These can be represented with the help of Ontologies and rule sets. Some of these may contain a level of uncertainty, as in *people use mostly their own PDAs*.

2. **Dynamic relationships**, which represent the discovered relationships. This includes role assignment, like Person XY… *is the presenter*, *is part of the audience*, *is writing on the whiteboard*, relations with artefacts as *is using a PDA*, which may be extended based on the a priori knowledge to *probably owns a PDA*, or one-to-many relations, as the count of people in the meeting, the number of presenters, the number of mobile terminals, etc.

The situation modelling process will also keep track of the **history** of these different relationships and states, so that it is possible to ask for the sequence of presenters in a meeting, or to find out the content of the third question, or to ask for a screen shot of the content of the white board just before it was cleared.

3.4.2.5 Perceptual components

3.4.2.5.1 Functionality

This layer provides interpretation of data streams coming from various audio and video sources represented by the underlying layer of Logical Sensors. The Perceptual Component layer will be populated with components that wrap different technology engines (typically those provided as output of WP4, WP5, and WP6 CHIL work packages).

The interpretation of raw data input into higher-level semantics event output will, in most cases, be required to happen in real-time. It is expected that during the initial stages of the project, many components at this layer will be simulated.

Components at this layer will form interpretation hierarchies with many cross-links, feedback loops, dependencies, and aggregations. It will be very important to do the modelling right, so that existing component "wirings" can be captured and modelled. The existing systems maintained at various CHIL sites will be used both as use cases for the architecture design as well as reality-check data to assess modelling capabilities of the framework.

3.4.2.5.2 Key abstractions

The Perceptual Components layer provides a collection of well-defined APIs that allow replacing one technology engine for another. Its output is used by the Situation Modelling layer and/or by the Services Layer.

- A collection of recognition engines for a specific modality (speech, vision, haptic, etc.) and their combination (AVSR, etc.).
- A collection of output generation components (speech, GUI) and their combination is required (animated TTS).

We expect that the APIs at this layer will need to be "cultivated" via several iterations of API prototyping and via a "committee process" that will also involve CHIL technology providers from various sites. This intra-CHIL standardization process can lead to wider adoption of CHIL APIs and provide strong input into the proper standardization bodies.

This can be achieved by carving a set of *mandatory* functionality as an API for every component being part of a particular engine class, e.g. Body Tracker, TTSEngine, etc.

This API then becomes a contract for the other layers of CHIL architecture framework. Moreover, it will be used to determine CHIL compliance of third-party perceptual components.

Provider-specific extensions on top of the mandatory API will be supported through optional APIs that will be formally described using component introspection mechanism.

Detailed description of APIs is given in section 6.4.3 Perceptual Component APIs.

3.4.2.6 Logical sensors/actuators

3.4.2.6.1 Functionality

The components at Logical Sensors and Actuators layer are either sensors feeding their data to the Perceptual Component layer, or various actuators, such as output devices receiving data or steering mechanisms receiving control instructions from the Perceptual Components layer. The aim of this layer is to organise the sensors and actuators into classes with well-defined and well-described functionality. The sensors will be classified mainly based on the kind of output stream they produce and the actuators will be classified mainly based on the input stream they consume.

Another goal of this layer is to define means for labelling the sensors and actuators with logical names and to maintain the binding of namespaces to physical devices. Logical names are used by the upper layers to operate on the desired part of the observed space and select the proper signal source for its processing and the proper actuator for its output (e.g., “the door-facing camera”, “the table microphone”).

This layer makes possible the abstract control of steerable devices, e.g. an upper level component may say, "Turn the camera a little bit to the left".

This layer also provides the means for controlling access rights on the sensors. This includes, for example, the right to read data from a sensor or to control the steering of the sensor.

3.4.2.6.2 Key abstractions

There will be some categorisation work needed to come up with reasonable hierarchy of logical sensors to which real sensors could map.

The key features for this layer of abstractions comprise:

- Ability to emulate the sensor by repeatedly serving the same data,
- Ability to control quality of service of data acquisition and transfer,
- Support for storing/retrieving raw data to/from database.

3.4.2.7 Control/Metadata

3.4.2.7.1 Functionality

These two layers provide mechanisms for data annotation, synchronous and asynchronous system control, effective storing and searching multi-media content and metadata generated by data sources. Synchronization of various data streams is a very important capability of this layer (e.g., to allow speed to be traded off against accuracy).

3.4.2.7.2 Key abstractions

One of the key challenges will be to find proper abstractions to capture unstructured data as annotations of data streams.

3.4.2.8 Low-level distributed data transfer

3.4.2.8.1 Functionality

Components and technologies at this layer are responsible for transferring various low and high bandwidth data streams from producers to consumers, in real-time with a desired quality of service.

The usage of existing middleware "Smart Flow System" by NIST to bootstrap the development work on this layer has been agreed to.

3.4.2.8.2 Key abstractions

The two most important abstractions are data producers and data consumers.

3.4.2.9 CHIL utilities

3.4.2.9.1 Functionality

Most of the information that passes through the CHIL infrastructure comes from readily defined sources. Examples include audio and video streams provided by logical sensors, user models and context by the situation modelling, and so on. However, some system internal information has no obvious source in the framework, although it is needed by most or all components. In particular, utilities for over-all system monitoring are better provided by a layer transcending the layer model. Global time is a clear example; general system information and general system statistics may be others.

The CHIL architecture framework describes a distributed, event driven system, in which time-stamped messages and streams are sent through the infrastructure. In such a system, latency issues are inevitable. Tracking latency may serve several purposes, such as measuring quality-of-service on-line and off-line, but it is also essential for the on-line functionality of the controlling logic, which may use it to abort, skip, or amend operations if the delay is too long. Although CHIL components are expected to time-stamp their messages, receiving components have no means of determining whether the delivery of a message suffers from latency unless they have access to global time – the same *now* as the senders.

3.4.2.9.2 Key abstractions

Initially, the key abstractions for the CHIL utilities are synchronisation and GlobalTime. Although robust synchronisation in a large, distributed, asynchronous system is difficult at best, an estimation of synchronised global time can be achieved. Whenever a CHIL component starts up, it retrieves the system's *now* from the CHIL utilities, and any time-stamps subsequently sent by the component should be relative to this global CHIL time. Minimising latency in the communication between initialising components and CHIL utilities will improve the quality of this semi-synchronisation.

3.4.2.10 Ontology

3.4.2.10.1 Functionality

In order to devise cognitive capabilities the CHIL software environment will extensively use Ontologies to conceptualise entities and to formally describe relations among them. To support novel sorts of computation and to make it particularly easy for CHIL developers to make contributions, it is necessary to specify the meaning of the resources that occur in the CHIL software environment. CHIL software components will both know the meaning of the data they are operating on and they will expose their functionality according to a common classification scheme. Such self-description capabilities will facilitate the implementation of self-regulation features. Contributions that share common characteristics may be interchanged in order to speculatively configure new working sets of components. Different configurations may perform better under certain circumstances. Moreover, ontologically annotated CHIL software will be easier to discover both at runtime and at design time.

Additionally to the technical use of ontological concepts CHIL Ontologies are likely to help resolving ambiguities that may arise from multilingual issues. Instead of using a number of terms to talk about the same meaning one will use natural language terms to refer to ontological concepts instead. Since an arbitrary number of dictionaries may be attached to an ontology one can easily disseminate CHIL knowledge to various languages.

3.4.2.10.2 Key abstractions

CHIL Ontologies will contain concepts from the following domains of discourse.

Technical terms ontology (TTO): Technical terms, which are common among parts of the CHIL runtime environment, will be modelled as ontological concepts. This includes concepts like for example *frame* or *image resolution* and relations among those concepts like *a video recording has a frame rate*. A common set of concepts will help to make the CHIL software environment independent of a particular low level distributed data transfer layer middleware. Moreover, it will be easier to extend upon a middleware, which may already be in use.

Contributions ontology (CO): Contributions to the CHIL software environment will be classified according to a common classification scheme. Thus, one can replace contributions such as perceptual components or logical sensors by equivalent components that may be more applicable under certain circumstances. Semantically annotated components that adhere to a common classification scheme may be employed by optimisation strategies that may operate on knowledge such as *if there is an instance of a component operating and the component's throughput is insufficient replace it speculatively by an instance of a component that shares common characteristics but may perform better in this situation.*

Real life ontology (RLO): Real life concepts as mentioned in CHIL scenario descriptions. Such concepts mainly present non-technical entities like for example *users, user roles, meeting room equipment*. These concepts will be used to formalize CHIL scenario plots as devised during the planning- and design phase. Moreover, programs that operate on real life entities will be annotated with concepts that elucidate on what kind of input they depend and what kind of output they produce.

How these three domains of discourse fit into the CHIL layer model is depicted below.

Table 3-1: CHIL Ontology Layer Model

CHIL software environment layer	**Domain of discourse**		
Services			RLO
Situation modelling			RLO
Perceptual components	TTO	CO	RLO
Logical sensors and actuators	TTO	CO	
Control and metadata			
Low-level distributed data transfer	TTO		

3.4.3 Functional components mapping

Table 3-2: Functional components mapping

Layer / Functional Blocks	Low-level distributed data transfer	Metadata	Control	Logical sensors/actuators	Perceptual components	Situation modelling	Services	Service access and control	User front-end	CHIL utilities	Ontology
Ontology											
Process & User Models											
Public Knowledge Base											
Interface Agent											
Information & Service Access System											
Service Agents											
CHIL Services											
Elementary Services											
Service Knowledge Base											
Private Sensors & Actuators											
Perceptual Components & Processes											
Logical Sensors & Actuators											
Sensor & Actuator Control Unit											
Meta Data											
Sensors & Actuators											
Sensor Data											
Remote Information											
Personal Devices (PDA, Notebook)											
Global Timing											

The table above maps the functional components of the global architecture to the corresponding layers.

3.4.4 Use cases

With respect to the use cases contained in the Functional Requirements document (cf. Figure 3-3) some indicative use cases are detailed by sequence diagrams later on (see Section *Sequence Diagrams for Use Cases*). These use cases do not aim at providing an exhaustive walk-through to the functionality of the system; rather they illustrate the structure of the various components and how information flows between them.

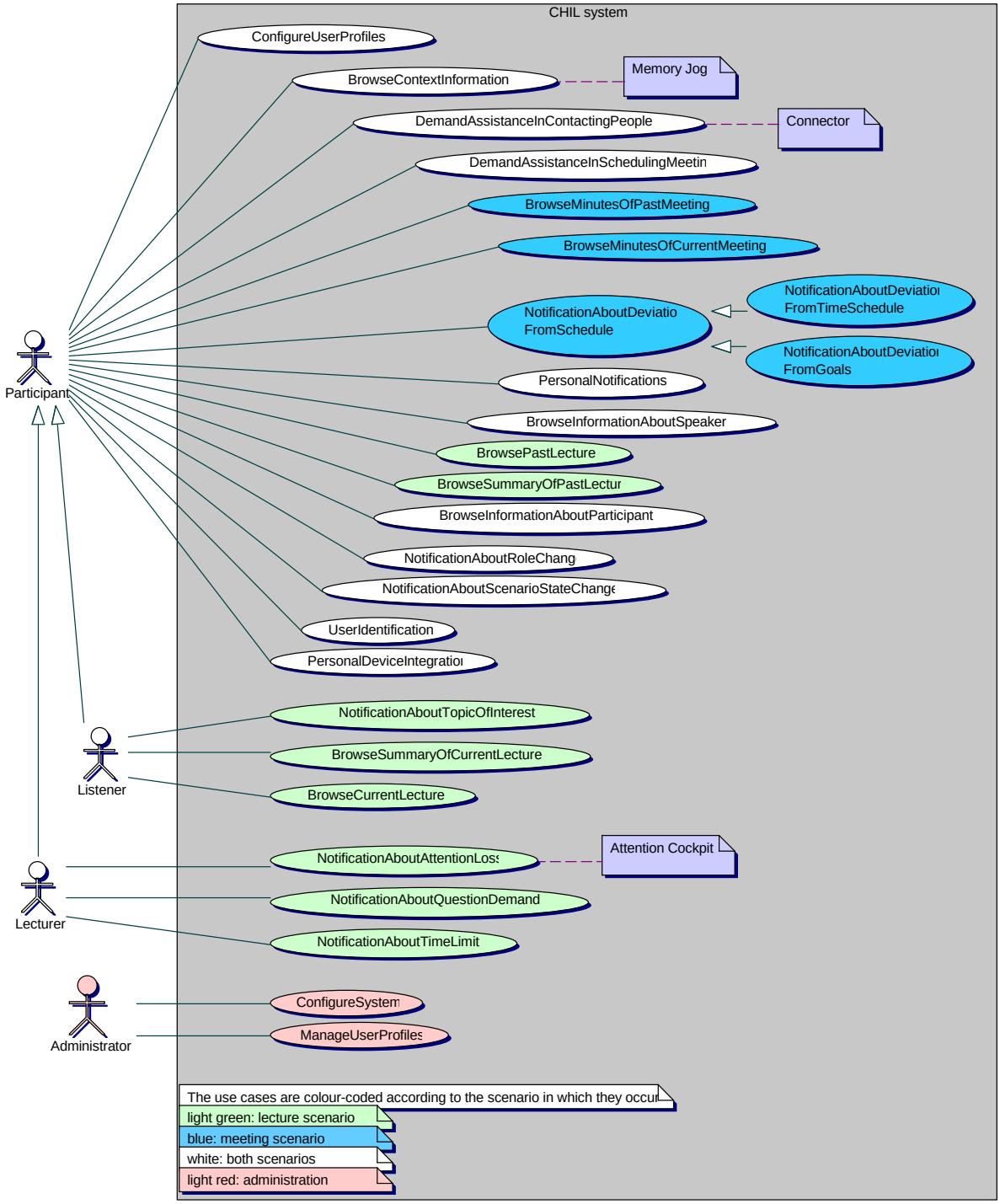

Figure 3-4: Use Cases in the CHIL System as in the Functional Requirements document

4 Mapping the Requirements to the Architecture Framework Layers Model

Table 4-1: Summary of Requirements

Code	Description	Importance	Met by									
		Priority	Low-level distr. data transfer	Metadata	Control	Logical sensors/actuators	Perceptual Com-ponents	Situation model-ling	Services	Service access and control	User front-end	CHIL Utilities
R1	Transparent Distributed Communications	High										
R2	Subscriptions	High										
R3	Polling	High										
R4	Synchronised Global Timing	High										
R5	Perceptual Components Simulation	High										
R6	Support for Heterogeneous Sensors and Perceptual Components	Medium										
R7	Situation Interpretation	Medium										
R8	Storage	Medium										
R9	Situation Tagging	Medium										

Code	Description	Importance	Met by									
		Priority	Low-level distr. data transfer	Metadata	Control	Logical sensors/actuators	Perceptual Com-ponents	Situation model-ling	Services	Service access and control	User front-end	CHIL Utilities
R10	Direct Access to Raw (Sensor) Data	High										
R11	Raw data Storage	High										
R12	Integration with 3rd party entities	High										
R13	Directory Services	Medium										
R14	Life supervision	High										
R15	User front-end integration	High										
R16	Scenario detection	High										
R17	Dynamic role assignment	High										
R18	Service handling	High										
R19	Profiling	High										
R20	Real time assembly and correlation of inputs	High										
R21	Heterogeneous Operating Systems Support	Low										

Code	Description	Importance	Met by									
		Priority	Low-level distr. data transfer	Metadata	Control	Logical sensors/actuators	Perceptual Com-ponents	Situation model-ling	Services	Service access and control	User front-end	CHIL Utilities
R22	Multi User Front-End Support	Medium										
R23	Failure Resilience	Low										
R24	Multiple Data Formats	Medium										
R25	System management	High										
R26	System configuration	High	?	?	?	?	?	?				
R27	System instrumentation	High	?	?	?	?	?	?				
R28	Resource control	Medium	?	?	?	?	?	?				
R29	Access security	Medium	?	?	?	?	?	?				
R30	Automatic client setup	Medium										

All requirements are summarised in Table 4-1, which also provides an initial indication on the importance and priority of requirements within the scope of the CHIL implementation. The last columns indicate the relevance for the positioning in the architecture layers model (see chapter 3.4.2). Possibly open meetings are labelled by '?'.

The following chapters summarise all requirements for each layer.

4.1 Requirements for the "Low-Level Distributed Data Transfer (LDT)"-Layer

Table 4-2: Summary of Requirements for the LDT-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R6	Support for Heterogeneous Sensors and Perceptual Components	Medium
R10	Direct Access to Raw (Sensor) Data	High
R11	Raw data Storage	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R20	Real time assembly and correlation of inputs	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R24	Multiple Data Formats	Medium
R25	System management	High

4.2 Requirements for the "Metadata (MD)"-Layer

Table 4-3: Summary of Requirements for the MD-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R6	Support for Heterogeneous Sensors and Perceptual Components	Medium
R10	Direct Access to Raw (Sensor) Data	High
R11	Raw data Storage	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R20	Real time assembly and correlation of inputs	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R24	Multiple Data Formats	Medium
R25	System management	High

4.3 Requirements for the "Control (C)"-Layer

Table 4-4: Summary of Requirements for the C-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R6	Support for Heterogeneous Sensors and Perceptual Components	Medium
R10	Direct Access to Raw (Sensor) Data	High
R11	Raw data Storage	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R20	Real time assembly and correlation of inputs	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R24	Multiple Data Formats	Medium
R25	System management	High

4.4 Requirements for the "Logical Sensors/Actuators (LSA)"-Layer

Table 4-5: Summary of Requirements for the LSA-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R5	Perceptual Components Simulation	High
R6	Support for Heterogeneous Sensors and Perceptual Components	Medium
R10	Direct Access to Raw (Sensor) Data	High
R11	Raw data Storage	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R20	Real time assembly and correlation of inputs	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R24	Multiple Data Formats	Medium
R25	System management	High

4.5 Requirements for the "Perceptual Components (PC)"-Layer

Table 4-6: Summary of Requirements for the PC-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R5	Perceptual Components Simulation	High
R7	Situation Interpretation	Medium
R10	Direct Access to Raw (Sensor) Data	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R20	Real time assembly and correlation of inputs	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R24	Multiple Data Formats	Medium
R25	System management	High

4.6 Requirements for the "Situation Modelling (SM)"-Layer

Table 4-7: Summary of Requirements for the SM-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R7	Situation Interpretation	Medium
R8	Storage	Medium
R9	Situation Tagging	Medium
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R15	User front-end integration	High
R16	Scenario detection	High
R17	Dynamic role assignment	High
R18	Service handling	High
R19	Profiling	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R25	System management	High

4.7 Requirements for the "Services (S)"-Layer

Table 4-8: Summary of Requirements for the S-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R8	Storage	Medium
R9	Situation Tagging	Medium
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R15	User front-end integration	High
R16	Scenario detection	High
R17	Dynamic role assignment	High
R18	Service handling	High
R19	Profiling	High
R21	Heterogeneous Operating Systems Support	Low
R22	Multi User Front-End Support	Medium
R23	Failure Resilience	Low
R25	System management	High
R26	System configuration	High
R27	System instrumentation	High
R28	Resource control	Medium
R29	Access security	Medium
R30	Automatic client setup	Medium

4.8 Requirements for the "Service Access and Control (SAC)"-Layer

Table 4-9: Summary of Requirements for the SAC-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R12	Integration with 3rd party entities	High
R13	Directory Services	Medium
R14	Life supervision	High
R15	User front-end integration	High
R18	Service handling	High
R19	Profiling	High
R21	Heterogeneous Operating Systems Support	Low
R23	Failure Resilience	Low
R25	System management	High
R26	System configuration	High
R27	System instrumentation	High
R28	Resource control	Medium
R29	Access security	Medium
R30	Automatic client setup	Medium

4.9 Requirements for the "User Front-End (UFE)"-Layer

Table 4-10: Summary of Requirements for the UFE-Layer

Code	Description	Priority
R1	Transparent Distributed Communications	High
R2	Subscriptions	High
R3	Polling	High
R12	Integration with 3rd party entities	High
R14	Life supervision	High
R15	User front-end integration	High
R19	Profiling	High
R21	Heterogeneous Operating Systems Support	Low
R22	Multi User Front-End Support	Medium
R23	Failure Resilience	Low
R25	System management	High
R26	System configuration	High
R27	System instrumentation	High
R28	Resource control	Medium
R29	Access security	Medium
R30	Automatic client setup	Medium

4.10 Requirements for the "CHIL Utility (U)"-Layer

Table 4-11: Summary of Requirements for the U-Layer

Code	Description	Priority
R4	Synchronised Global Timing	High

5 Interface Design

5.1 Overview

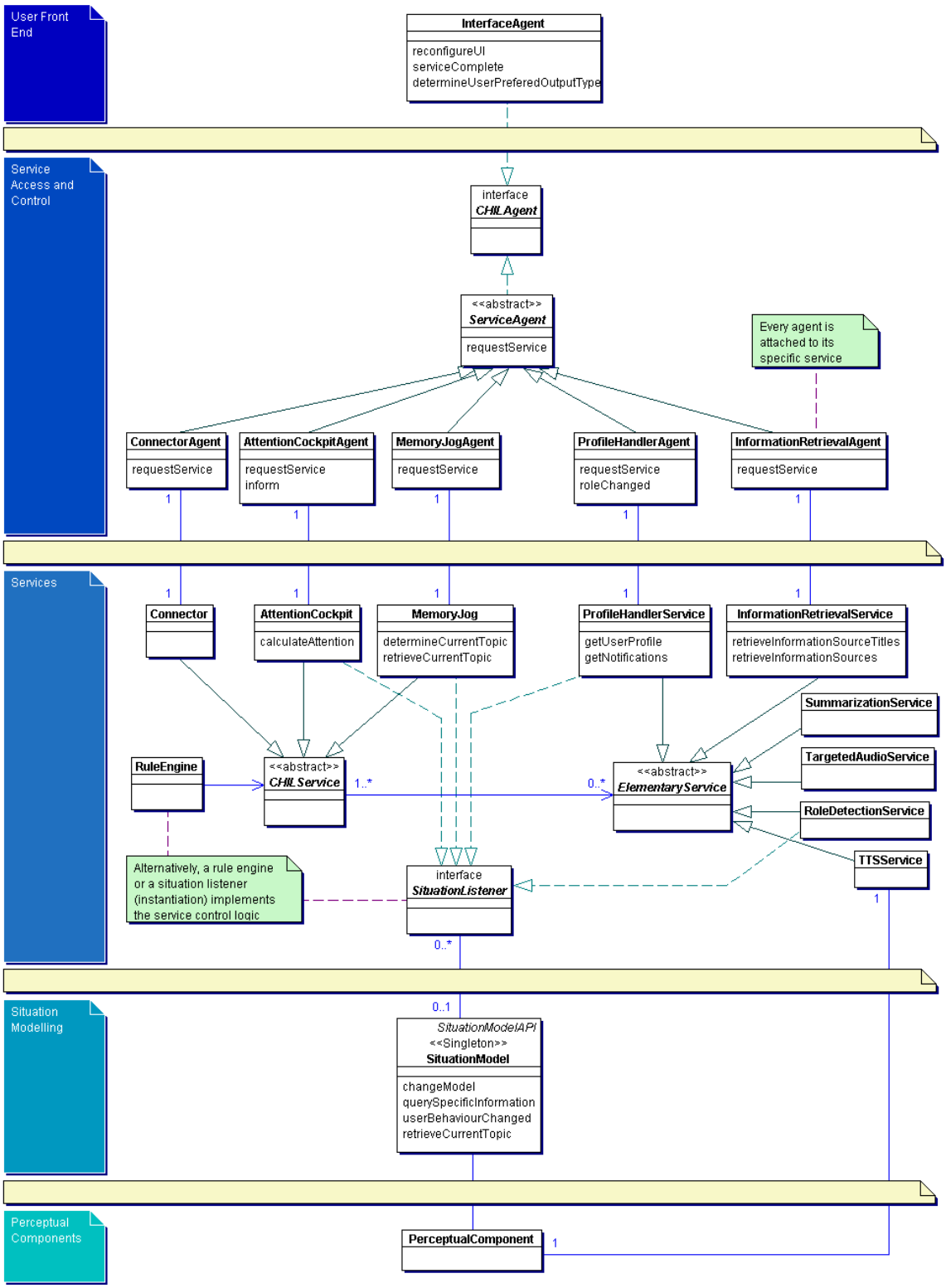

Figure 5-1: Interface Model Overview of the "Upper Layers"

The picture above gives an overview of the interfaces between the “upper layers” combining Agent Architecture, CHIL-Layer Model and layers themselves. To maintain consistency with the layer model, the layers are represented as vertical bars on the left side of the diagram in the same order and using the same colour as in the layer model. Interfaces between the layers are displayed as horizontal bars between the classes of each layer. To focus on the interfaces, the diagram has been designed on a higher, analysis level, hiding layer internal classes, attributes and operations.

Direct interaction with the user will be performed by InterfaceAgents. Every InterfaceAgent is dedicated to one specific user, managing his input devices such as keyboard and mouse or direct speech input and displaying information request results and notifications to the user. This will not comprise perceptual tasks like speaker recognition or identification, which will be handled by sensors and the components of the Perceptual Components layer in an unobtrusive manner.

The InterfaceAgent also represents the interface between the User Front End Layer and the Service Access and Control Layer by transmitting the user’s requests to appropriate services and receiving results and notifications from them, whereas communication between user and services will be organised by an Agent Management System.

Every service is associated with a specific agent that connects the service to the communication mechanisms of the agent manager. In a first approach, the connection between service and agent is specified as an association having cardinality of 1 to 1 and may be extended to a cardinality of N to M later. Information between agent and service may be exchanged by subscriptions, observer/listener pattern or any other mechanism. The final choice will depend on the concrete implementation of the services and their interfaces.

The services themselves may communicate with the underlying Situation Model Layer using subscriber mechanisms in order to get notifications about changes of the current situation and using control patterns to assign computational results to the situation model. Interfaces between the Situation Model Layer and the Perceptual Components will be described in detail in the corresponding chapter.

5.2 Sequence Diagrams for Use Cases

5.2.1 Use case NotificationAboutAttentionLoss

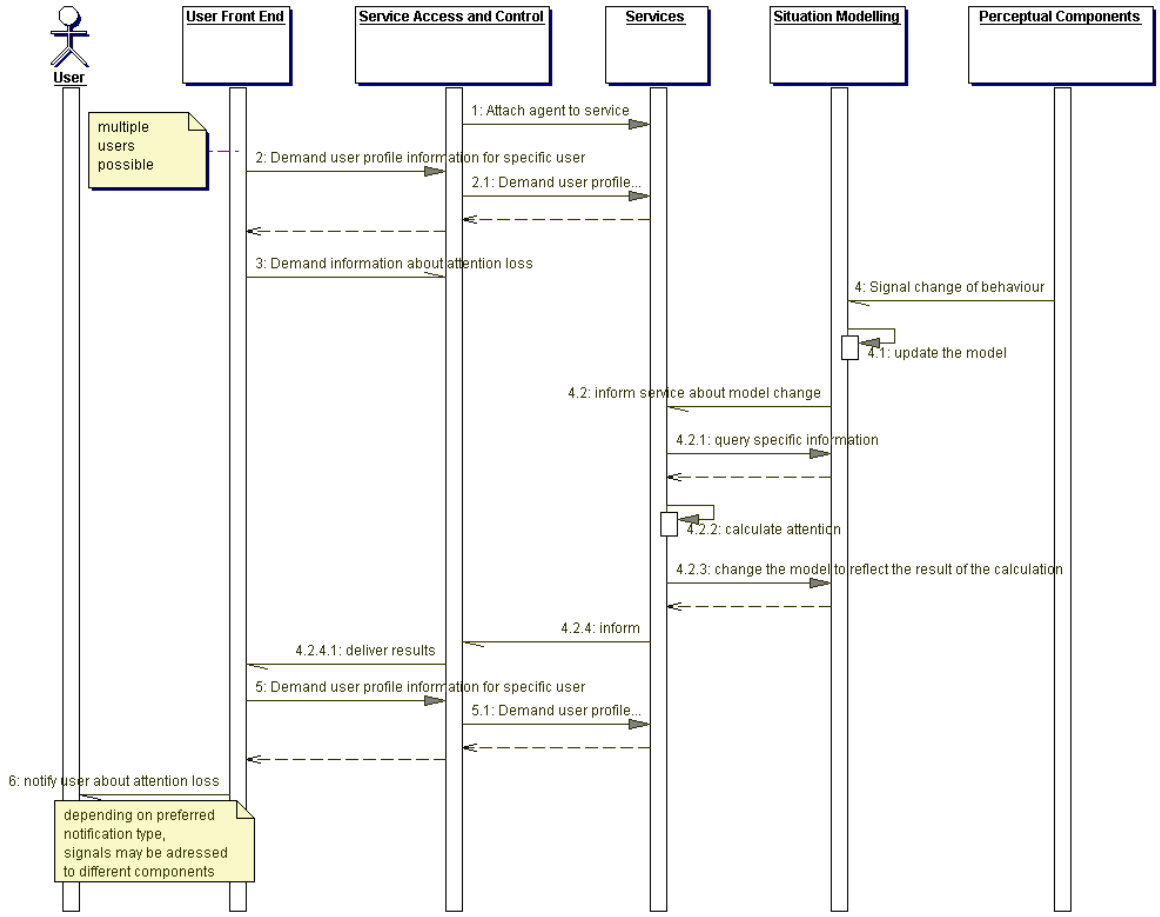

Figure 5-2: Sequence diagram for use case "NotificationAboutAttentionLoss"

This use case can be divided into two phases. The first part is the "initialisation and setup phase", the second may be called the "notification phase".

The first phase contains the attachment of an agent to the specific service as well as the registration for this service by a user (initiated by the User Front End according to the settings in the user profile).

The second phase shows the actions that take place when a change of behaviour is detected by the Perceptual Components Layer.

5.2.2 Use case BrowseContextInformation

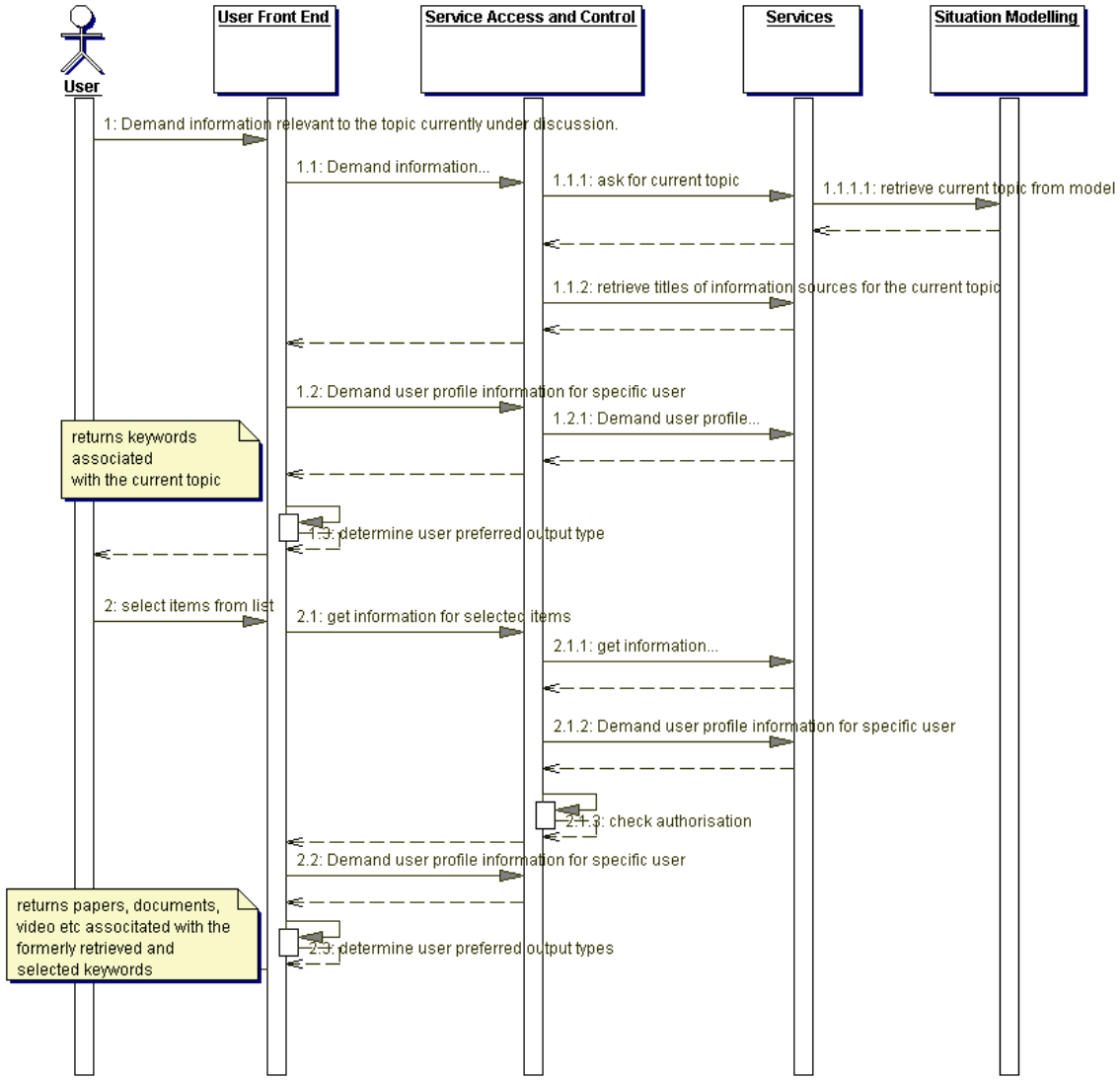

Figure 5-3: Sequence diagram for use case "BrowseContextInformation"

In this use case, the user demands information relevant to the topic currently under discussion. To satisfy the user's needs, the User Front End asks the Service Access and Control Layer for the titles of available information relevant to the current topic. Before the Service Access and Control Layer can build a list of these titles, it determines the current topic. This task is done by the Services Layer with the help from the Situation Modelling.

After the topic is known the Services Layer provides the requested titles. In the following step, the User Front End asks the underlying layers for the user profile information, in order to present the results to the user in the desired form.

Once this information has been obtained, the user can select an item and the User Front end requests the information from the other layers and presents the result to the user in the desired form. The authorization to view specific information is checked in the Service Access and Control Layer.

6 Detailed SW-Architecture

6.1 Upper Layers

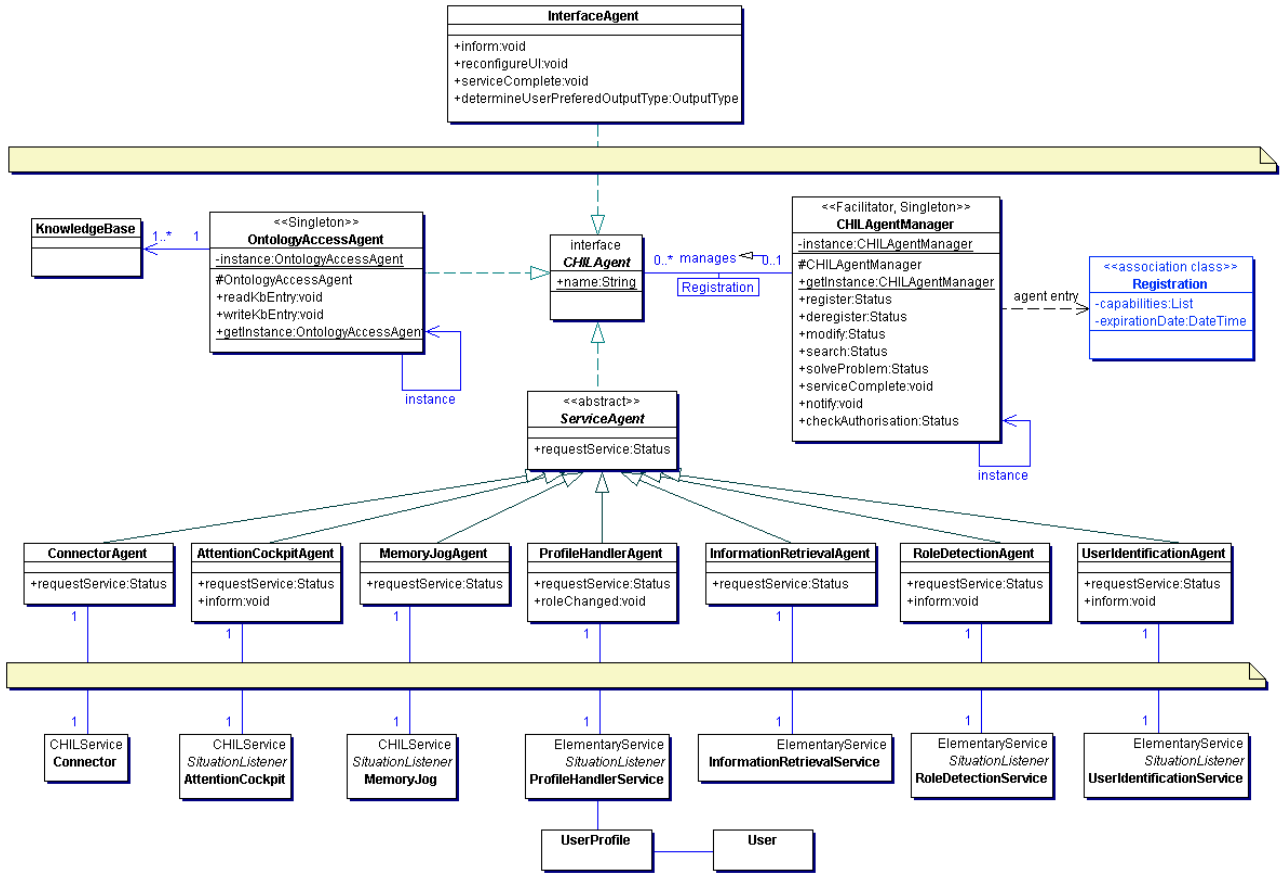

Figure 6-1: Detailed class model of the upper layers.

6.1.1 Use case NotificationAboutAttentionLoss

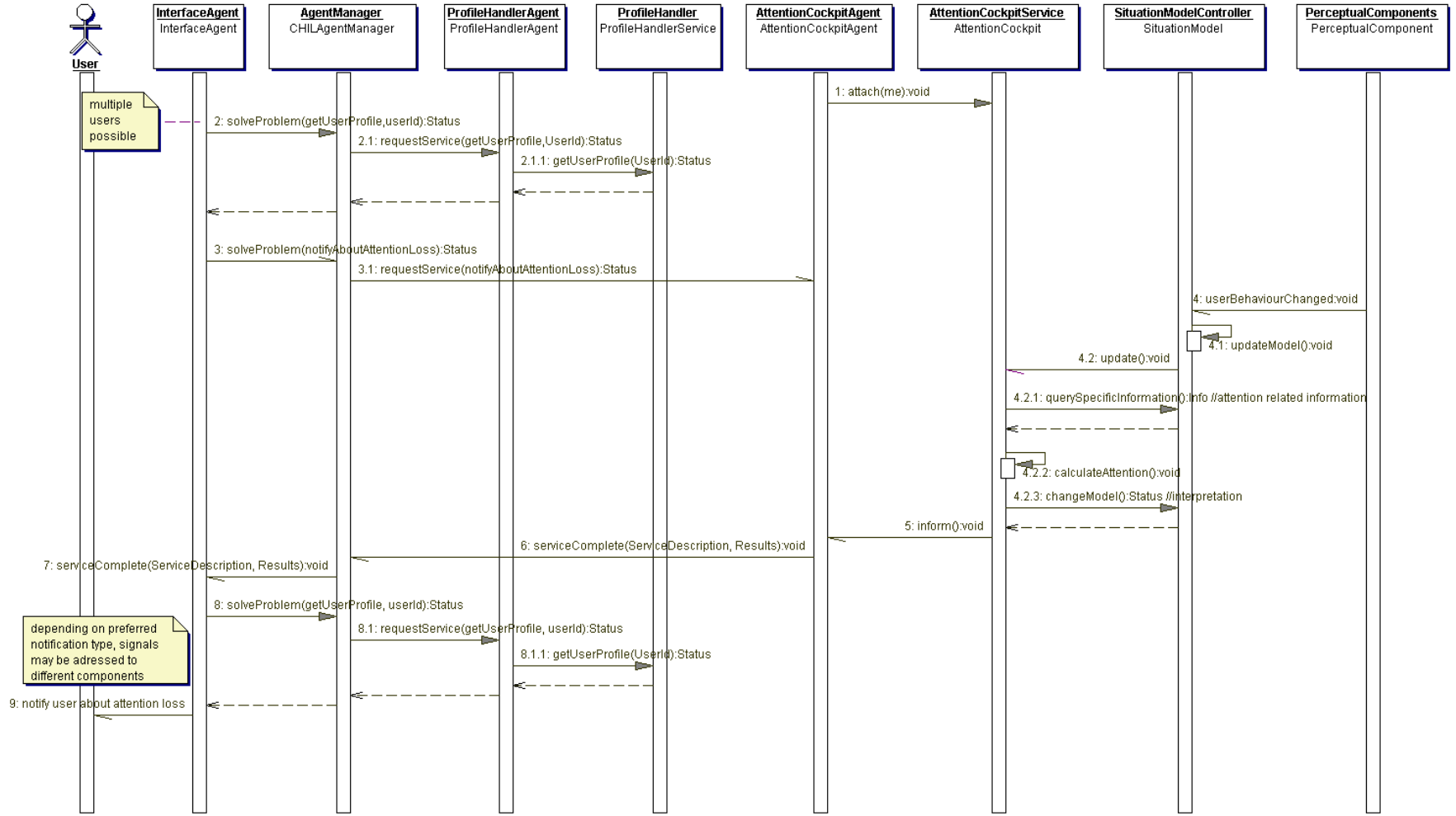

Figure 6-2: Sequence diagram for use case NotificationAboutAttentionLoss

This use case can be divided into two main parts: the initialisation phase and the notification phase.

In the first phase, the AttentionCockpitAgent is attached to the corresponding AttentionCockpit Service. The next step is the user's registration for this service: The InterfaceAgent requests the user profile from the ProfileHandlerAgent via the AgentManager. The ProfileHandlerAgent delivers the requested user profile with the help from the ProfileHandlerService. As soon as the InterfaceAgent knows the user profile it checks whether the user is interested in notifications in the current situation. If the user is interested in notifications the InterfaceAgent advises the AttentionCockpitAgent via the AgentManager to subscribe the user for notifications.

In the second phase, the Perceptual Components detect a change in a user's behaviour. For example, a user may be reading a newspaper or snoring. The SituationModel updates its model after getting this information from the Perceptual Components and sends a notification to the AttentionCockpit. Now the AttentionCockpit requests all the attention related information from the SituationModel that is necessary to calculate the actual attention (Example: participants look at their note pads and computer screens rather than the lecturer.). The result of this calculation is sent back to the SituationModel to be stored. Additionally the AttentionCockpit notifies the AttentionCockpitAgent. On receiving this notification the AttentionCockpitAgent sends a serviceComplete message to the InterfaceAgent via the AgentManager. Now the InterfaceAgent knows that it should inform the user and must determine the user's preferences for being informed. These preferences are stored in the user profile so the InterfaceAgent has to request the user profile from the ProfileHandlerAgent before the user can be informed.

6.1.2 Use case BrowseContextInformation

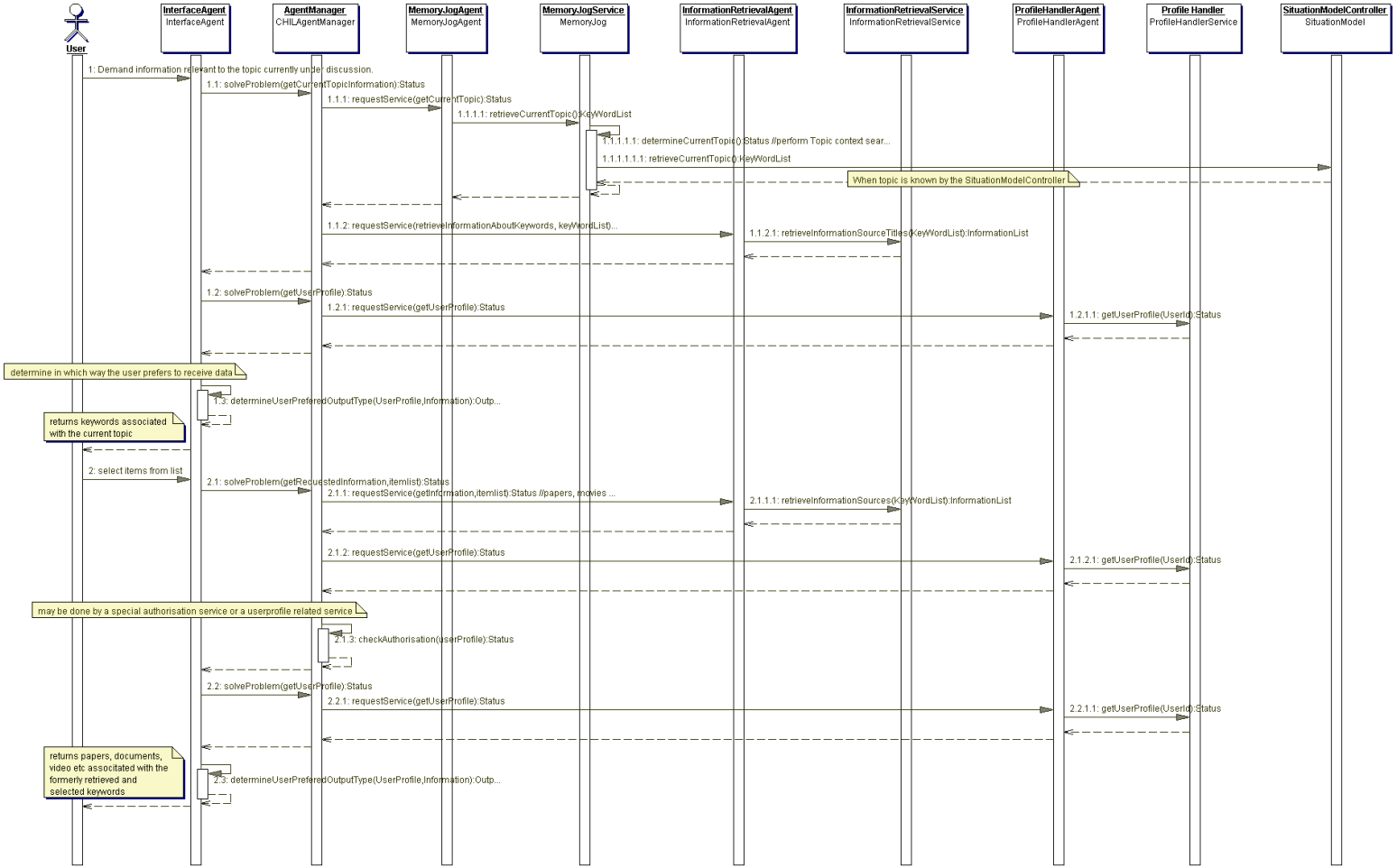

Figure 6-3: Sequence diagram for use case BrowseContextInformation

This sequence diagram models the use case BrowseContextInformation as described in the ***Functional Requirements*** [1] document. It is assumed that the situation model includes the topics currently under discussion.

The use case starts with the user's demand for information. The InterfaceAgent situated in the User Front End Layer then passes this demand to the AgentManager in the Service access and Control Layer. The AgentManager is responsible for routing the different subtasks to the specific agents for these tasks. The first task is the determination of the current topic, which is done by the MemoryJogAgent by using the MemoryJog from the Services Layer. On its way to determine the current topic, the MemoryJog uses the SituationModel that knows about the topic.

In the next task, the now known topic is passed to the InformationRetrievalAgent. This agent can now use the InformationRetrievalService to gather information relating to this topic and build a list of titles/headings. Before the InterfaceAgent can present this list to the user the user profile must be available. Therefore the InterfaceAgent demands the user profile from the AgentManager. The ProfileHandlerAgent (being attached to the ProfileHandlerService) provides the user profile so that the InterfaceAgent can now present the list of headings to the user for selection.

Now it's the turn of the user to select items from this list. This selection arrives at the InterfaceAgent who gathers the desired information via the InformationRetrievalAgent and its corresponding InformationRetrievalService. Before the information can be presented to the user the AgentManager has to check authorisation issues and like in the paragraph above the user profile is necessary to meet the user's desires. Finally the InterfaceAgent presents the gathered content to the user.

6.1.3 Use case UserIdentification

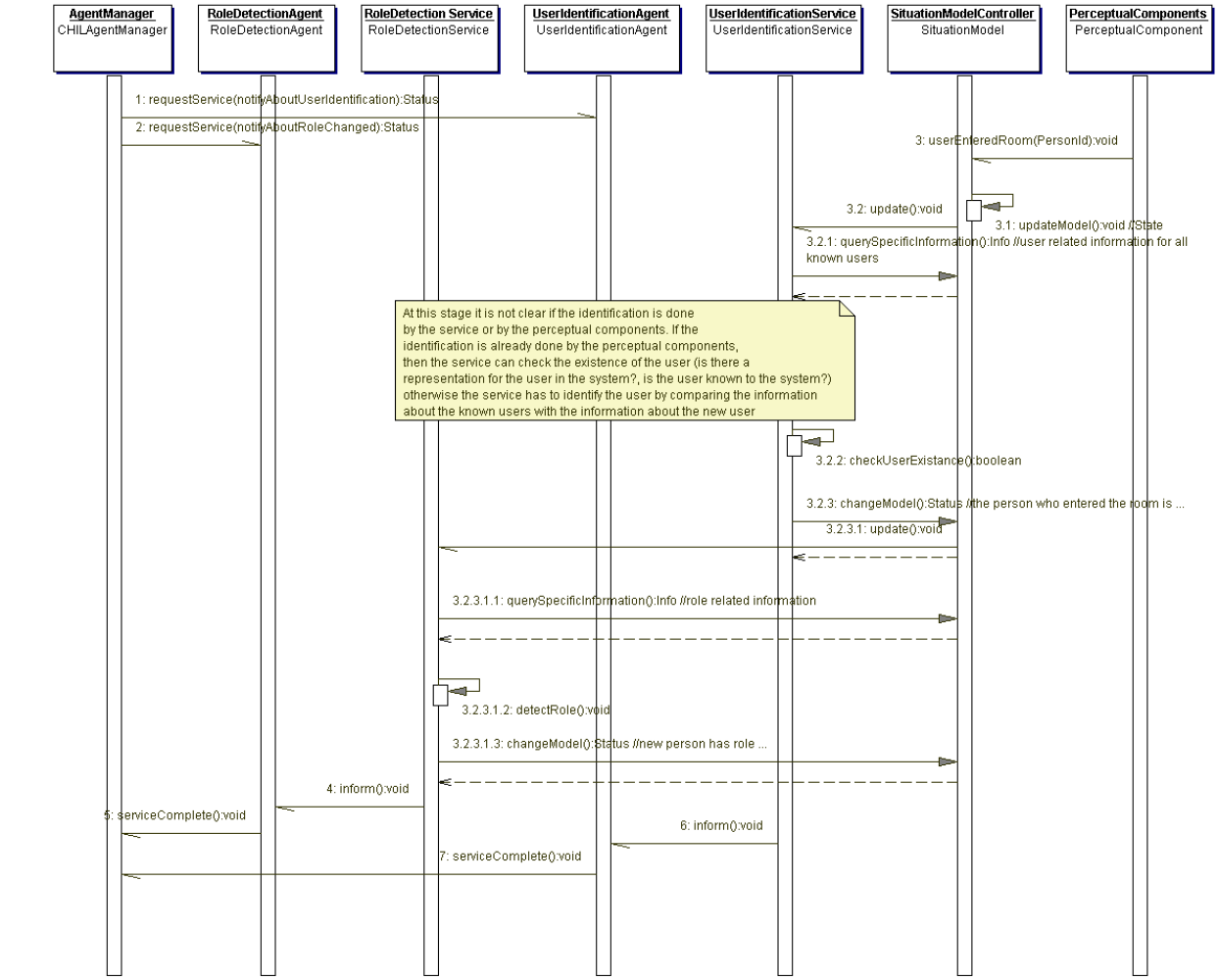

Figure 6-4: Sequence diagram for use case UserIdentification.

This sequence diagram shows the elementary service "UserIdentification" in conjunction with the service "RoleDetection".

Possible notifications to the other users that may result from the event "person enters room" are not shown in this diagram because they belong to other use cases like NotificationAbout-ParticipantListChange, which are not yet covered by the current document.

In the first two steps of this use case The AgentManager sends requests for notification about UserIdentification and RoleChange to the appropriate Services.

When a user enters the CHIL room, the perceptual components inform the SituationModel-Controller by calling the method userEnteredRoom. The SituationModelController updates its model to reflect the current situation and informs the UserIdentificationService about this change by calling its update method. There might be other parties interested in this notification as well.

Now it’s up to the UserIdentificationService to determine the identity of the new person. As mentioned in the note in the diagram above, at this stage of the CHIL project it is not clear if the identification is done by the service or by the perceptual components. If the identification

is already done by the perceptual components, then the service can check the existence of the user (is there a representation for the user in the system?, is the user known to the system?) otherwise the service has to identify the user by comparing the information about the known users with the information about the new user. After this check the result is sent to the SituationModelController by calling the changeModel method.

After sending the result to the SituationModelController the UserIdentificationService inform its corresponding agent about the new situation. The UserIdentificationAgent is now able to inform the AgentManager by sending a serviceComplete message.

The RoleDetectionService, which is also interested in notifications about UserIdentification, receives an update call when the SituationModelController updates its model. Before the RoleDetectionService can detect the role of the new person it has to gather some role related information from the SituationModelController by using the querySpecificInformation method. When all necessary information is available the RoleDetectionService can determine the role of the user and send this result back to the SituationModelController by using the changeModel method. As a last step the RoleDetectionService informs its corresponding agent about the role detection. The RoleDetectionAgent can now inform the AgentManager by sending a serviceComplete message.

6.1.4 User Front End

The user interface layer contains the *InterfaceAgent* class. An instance of this class acts as a personal assistant for a specific user and responds to requests by subscribing to the CHIL-services he/she wants to use.

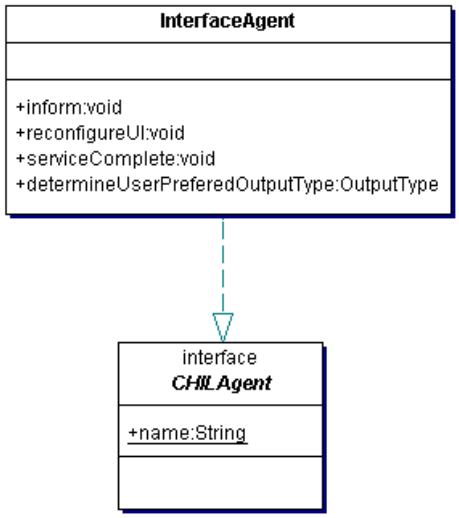

Figure 6-5:User Front End Class Diagram

The creation and destruction of interface agents may be maintained by a *LifecycleAgent*. The lifecycle agent creates an interface agent when a new user is recognised during entering the room. To facilitate this, the lifecycle agent must be subscribed to the *UserIdentificationService* (encapsulated by the *UserIdentificationAgent*). The lifecycle agent gets notified about new users through its *serviceComplete* function. If a user leaves the room, the lifecycle agent can shut down the users interface agent but there should be a delay because a user may only leave the room for short time, e.g. to have a short break or get some required papers. It must be discussed further if a lifecycle agent is required or if this functionality could also be provided by the *AgentManager*. It may also be possible that an interface agent lives outside of a CHIL room on a users laptop e.g. for providing summary information of a meeting.

If a newly created interface agent starts up it, first retrieves the current role of the user from the *SituationModel* to read the matching role profile from the *ProfileHandlerService* (encapsulated by the *ProfileHandlerAgent*). If the user is already identified and he/she has a user profile the interface agent blends it with the role profile otherwise the default settings of the role profile apply (see step 2 of the NotificationAboutAttentionLoss sequence diagram).

After that, the interface agent subscribes to the CHIL services that the user demands, by interpreting the blended user/role profile. For example a lecturer wants to be informed about an attention loss so its interface agent must subscribe to the *AttentionLossService*. Interface agents do not directly subscribe to a service but send a *solveProblem* request to the *AgentManager*. The *AgentManager* knows which agents (who encapsulate CHIL-services) can solve the demand, e.g. there may be different services that can solve a problem but some are already busy for other users.

During a scenario, the interface agent must take care about role changes of its user. To facilitate this, the interface agent must be subscribed to the *RoleDetectionService* (encapsulated by

the *RoleDetectionAgent*). If a role change occurs it must again read the role profile for the new role of the user, blend it with the user profile, subscribe to new services and unsubscribe from the no longer needed services.

If a CHIL-service comes up with some information for a user (through the interface agents its *serviceComplete* function) the interface agent must retrieve the users preferred way for notifications by querying the user/role profiles (in its *determineUserPreferredOutputType* function) and then use output components (like Targeted Audio) or a GUI on the users laptop/PDA.

6.1.5 Service Access and Control Layer

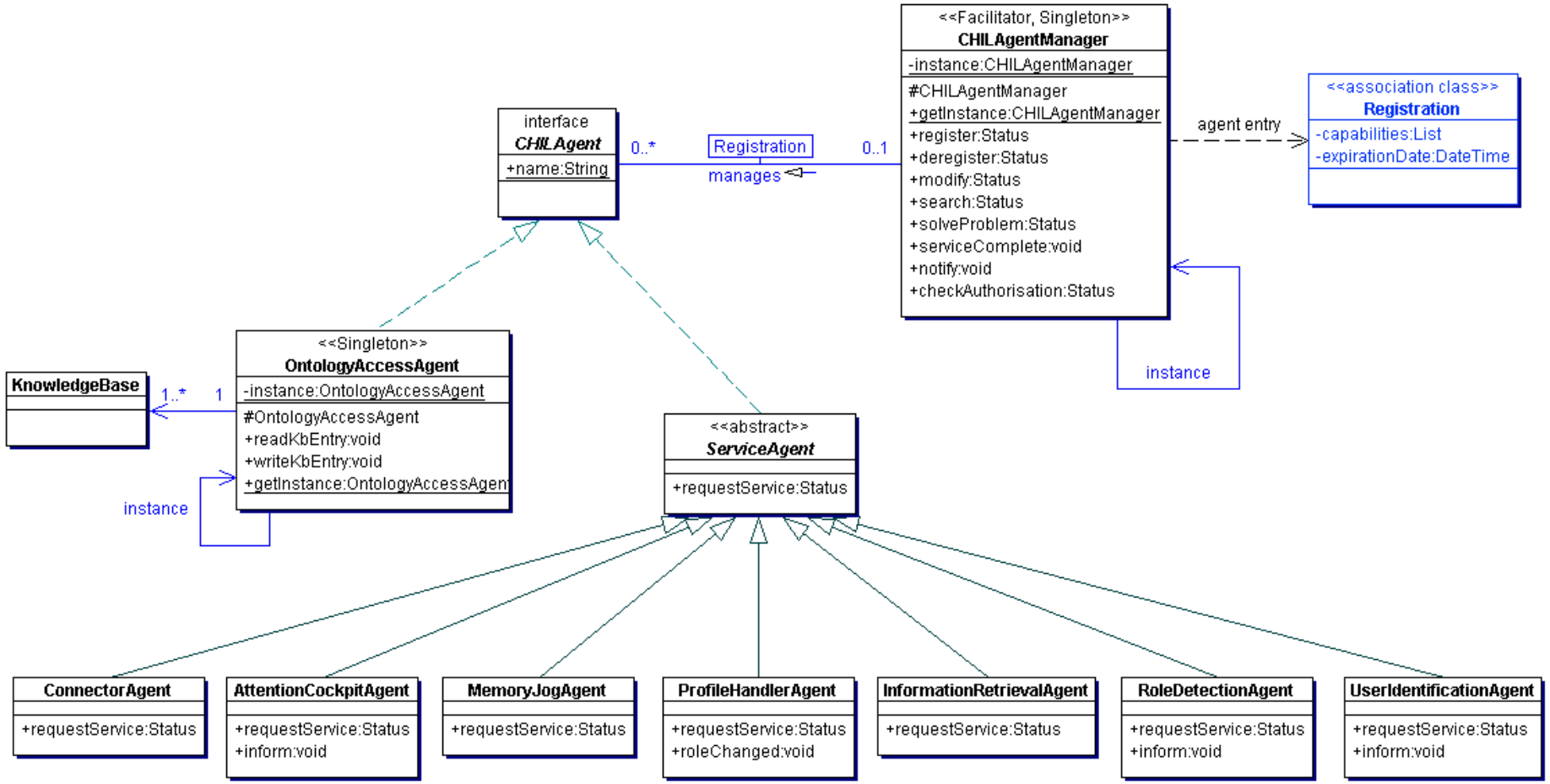

Figure 6-6: Service Access and Control Layer Class Diagram

6.1.5.1 CHILAgentManager

The CHILAgentManager acts as a facilitator for the agents in the Service Access and Control Layer. It is implemented using the “singleton” pattern. It manages the agent’s registration. Each registration of an agent consists of a list of its capabilities and an expiration date. The Agent Manager is also responsible for requesting the services provided by the different service agents, fusing the results and passing the fused results to the interface agent, which in turn will present the results to the user. The AgentManager accesses the ontology via the OntologyAccessAgent.

6.1.5.2 OntologyAccessAgent

The task of the OntologyAccessAgent is to make the ontology stored in a KnowledgeBase available to the other agents in the system. It is implemented using the “singleton” pattern.

6.1.5.3 ServiceAgents (MemoryJogAgent, ProfileHandlerAgent, InformationRetrievalAgent, AttentionCockpitAgent, ConnectorAgent)

There are several ServiceAgents like the MemoryJogAgent or the AttentionCockpitAgent. They act as a wrapper for the underlying services and offer them to potential clients in the agent system.

Service agents register at the AgentManager (kind of advertising). By registering they publish their capabilities to the other agents in the system.

Service invocation has normally started with a problem that a user wants to be solved. The interface agent allocated to the user takes the problem description and passes it as a solveProblem() request to the AgentManager.

The AgentManager which knows the capabilities of all the service agents via their registration, determines which agents capabilities best fits the problem, eventually decomposes the problem in subproblems and requests the services provided by the service agents in order to solve the subproblem.

The ServiceAgent is now responsible for dealing with the service request by using the corresponding service.

6.2 Services Layer

In the scope of the CHIL, smart room every computing device will be capable of providing services. These services include soft services (e.g., computational capabilities), as well as physical services relating to mobile phones, computer peripherals. We call these services provided in the scope of individual devices "elementary services".

The service-modelling layer provides the foundation for implementing CHIL non-obtrusive services such as the *Memory Jog*, the *Attention Cockpit* and the *Connector*. These services can be viewed as composite services comprising a group of elementary services (for example, searching information within a database or providing output through a TTS interface), that are triggered based on appropriate control logic. Note that the control logic should implement the triggering functionality supporting the non-intrusive nature of the CHIL services.

Therefore, a CHIL service can be defined as a set of service invocations, based on appropriate control/triggering mechanisms. The CHIL architecture provides the infrastructure enabling:

- Implementation, deployment and invocation of elementary services
- Aggregation of elementary services into CHIL Services
- Implementation of the control logic

As far as the control logic is concerned, the CHIL architecture does not specify the exact control algorithm. This is an application specific issue, to be defined in conjunction with the Services group (WP3). Rather the CHIL architecture will provide the means for implementing application specific control algorithms.

A service-oriented architecture (e.g., based on Web Services) will be employed to facilitate the deployment and implementation of elementary services. At this level the Service object (or interface) will be the main object abstracting an elementary service. This will provide a mechanism such that other distributed software object can discover this service and interact with it. All (elementary) services (soft-services or provided by computing devices) extend this class. Elementary services will provide the means for offering human assistance in the CHIL room, including the input/output procedures. Representative examples are:

- A TTS Service for outputting audio,
- A Target Audio Service for outputting audio to a particular person/participant,
- A Summarization Service providing summary of a portion of a speech / talk,
- A Person Information Retrieval (IR) Service providing information on a participant.

Several of these services rely on perceptual interfaces. Therefore, the corresponding service objects can be implemented through accessing directly the perceptual components layer. Alternatively, wrappers of these components can be implemented at the service layer. These wrappers should simply implement the Service interface and deal with service discovery and invocation aspects.

CHIL services comprise one or more elementary services along with a control logic guiding the invocation of services (how, when, in which order). Therefore, CHIL services feature a one-to-many association relationship to elementary services. A CHIL Service would get feature handles, bindings, and references to elementary services so that the latter can be invoked. Invocations will be performed in a distributed manner using a distributed programming model (e.g., Java RMI, Web Services, XML-RPC or an Agent Communication Language).

The control logic will encode *if-then-else* statements through accessing:

- Components of the situation modelling layer to reason about situations and action to be taken (i.e. the if part of the control logic)
- Elementary services or components of the perceptual layer to execute actions/services targeting human assistance.

With respect to the implementation of the control logic we envisage one early implementation based on a simple event based mechanism, and a more complex implementation relying on a rule engine and Ontologies. Both simple events mechanisms and more complex rule based mechanisms will implement a reactive layer activated when particular situations are recognized.

In the sequel we elaborate on each on of those two design choices.

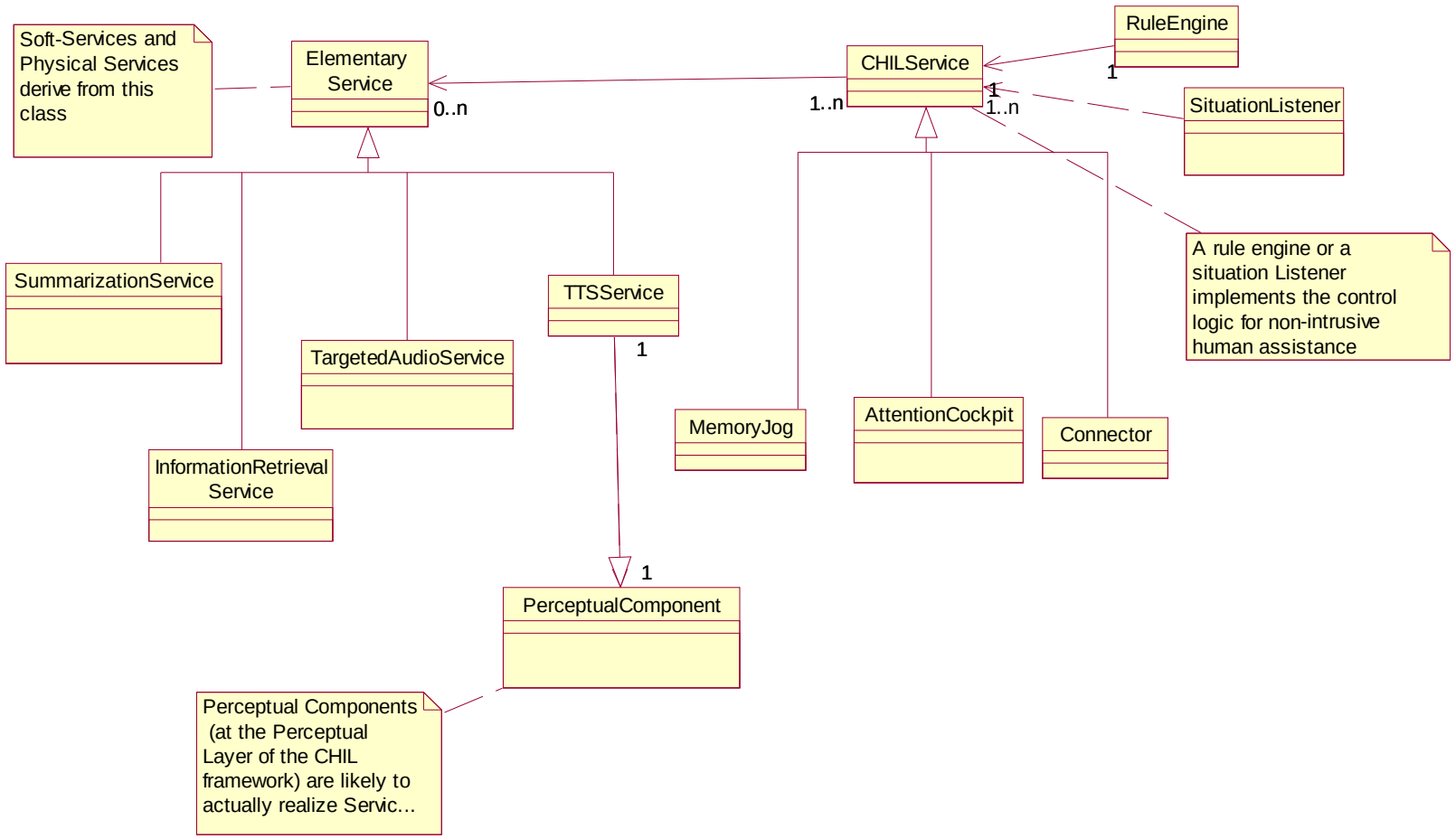

Figure 6-7: Service Layer Classes

SituationListener

The `SituationListener` class or interface provides an abstraction of an event based triggering mechanism, along with a callback model for handling different situations. The implementation mechanism for the events is still an open issue the following alternatives can be envisaged:

- Subscribing the CHIL Services object to the Situation Identification engines (i.e. components of the modelling layer).
- Implementing a global event mechanism, through maintaining an event queue similar to the approach taken in [3].

Callback functions implemented in the `SituationListeners` will then provide the required application specific control logic, through access elementary service capabilities.

RuleEngine

In order to improve the sophistication, scalability and extensibility of the CHIL services the control logic should be implemented within a *rule based inference engine (or "rule engine").* A rule engine may be viewed as an interpreter for condition-action pairs. The if/then statements that are interpreted are called rules. The condition portion of rules contains a set of tests. The action part specifies a sequence of actions that should be executed when all of the of the tests have been satisfied [15]. Rule based inference engines are often implemented using the RETE-matching algorithm. Rule based inference is well suited for implementing situation-modelling processes using Ontologies [16].

Basic service layer classes are depicted in Figure 6-7.

6.2.1 User Profile Specification

The user is described by a set of attributes. These attributes should include all CHIL relevant user data such as access rights, personality, body characteristics etc.

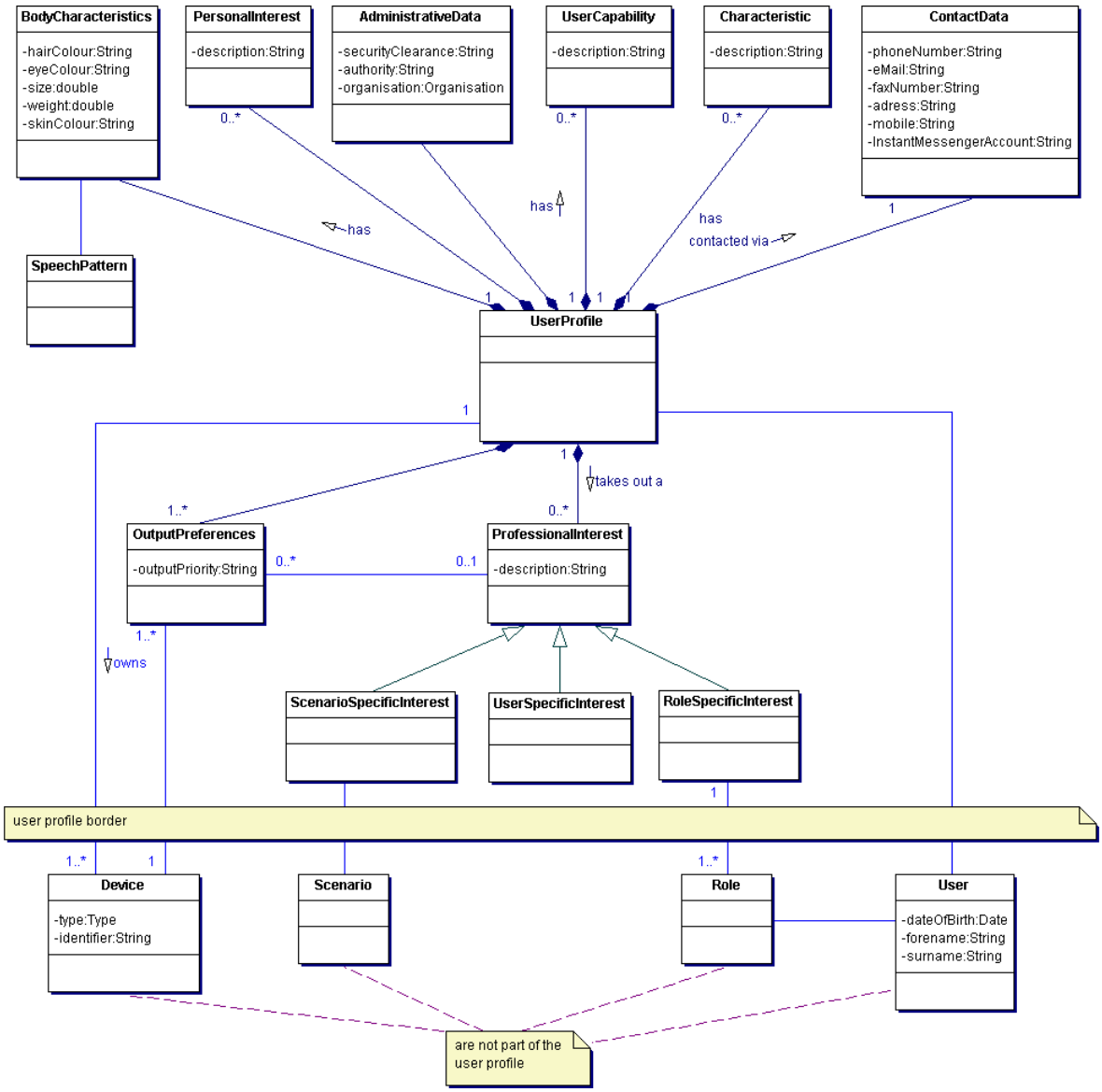

Figure 6-8:User profile specification

UserProfile: This is a package of all attributes that provides relevant information about the user.

ContactData: Describes how the user can be contacted.

OutputPreferences: Contains a description of the ranking order and how a user wants to be notified about something, on a specific device.

ProfessionalInterest: Superclass for all interests that may automatically lead to service subscriptions. Subclasses are RoleSpecificInterest, UserSpecificInterest and ScenarioSpeci-

ficInterest, which are meant to be used to specify interests for subscriptions in detail and depending on certain roles and scenarios. An example might be that while being assigned lecturer in a meeting, the user has the RoleSpecificInterest of being notified about attention loss. Such an interest would automatically lead to a subscription of the appropriate Service.

PersonalInterest: The above-described interests do not necessarily lead to a service-subscription (e.g. a subscription to every Greek goal shot in the Euro 2004 final). These interests describe additional user-specific interests.

BodyCharacteristics: Describe relevant physical attributes of the user (size, skin colour, speech pattern, …).

AdministrativeData: These are the user's administrative data in an organisation, for example his access rights.

UserCapability and **Characteristics**: These two attributes capture aspects of the user's personality, such as languages that he might speak.

6.2.2 ProfileHandler Service

The ProfileHandler service is used to administer a user profile. It is meant to read and write user specific attributes depending on the requested operation. The service can also handle more general requests for information, like "return all speech patterns available for the user". Another task is the handling of resource (information or device) specific user access. Every resource has a set of rules defining access to the resource itself. The access is granted if the user's role or his AdministrativeData match the resource's admission demands.

Additionally, the ProfileHandler service can be seen as a privacy guard of the user information. This aspect of profile handling seems to be very important, concerning the extremely sensible data in the profile.

6.2.3 InformationRetrieval Service

The InformationRetrieval Service provides information about a given topic. A retrieval job starts with a call to the services with a list of keywords as parameters. The service then begins with the search in all available data sources. To increase the quality of the search results computational measures are used. In this context the term quality means best fitting or covering information about the given keywords.

6.2.4 DeviceIntegration Service

The DeviceIntegration Service offers the possibility of integrating all devices in the area covered by the local CHIL-system. A method of communication between the device and the CHIL-system, one-way or bi-directional, is a necessity for linking up the device (e.g. WLAN, Bluetooth, Infrared, …). Through its perceptual components, the CHIL-system identifies a new device. Information about this device is then passed to the DeviceIntegration Service. In cooperation with data from the user profile (beeper number, etc.) the device is assigned to a user. After this the integration is started. This part of the Service is appropriate for devices that do not allow software installation, or are only located on the receiving end (e.g. beepers, video projector, …). Another way to integrate a device into the system is by user-initiated notification. This way is used to integrate a device into the CHIL-system by equipping it with a version of the current CHIL-software.

The actual integration consists of a user-to-device assignment and an update to the situation model. User-to-device assignments are performed with information from the user profile (phone number, …) but are also influenced by knowledge about users role and device locations.

6.2.5 UserIdentification Service

The UserIdentification service identifies users by their body characteristics (speech, face recognition, appearance), which have already been matched with users known to the system by the perceptual components. This service gathers information from different perceptual components such as a voice recognition component or a face recognition component. Even if the information is incomplete, e.g. a user entering the room does not say a word, the service then tries to determines the user who's profile best matches the given identification information. It also has to have a threshold of uncertainty. This threshold is used to distinguish between a new user to the system and the best matched user profile.

6.3 Situation Modelling Layer

The situation modelling layer provides the CHIL services with a description of the current state of the CHIL environment, and provides detection of events that are necessary to initiate or terminate service actions. A situation is a state description of the environment expressed in terms of entities (actuators and props) and their properties. Situation models determine the entities to observe, the properties to measure and the events to detect, and thus specify the selection and configuration of perceptual components.

A situation is a kind of state description composed of a conjunction of predicates. Predicates are truth functions that can take on logical or probabilistic values. Situations are defined in terms of an assignment of observed entities to "roles", the properties of the entities assigned to roles, and the relations (relative properties) of the entities playing roles. Thus the basic component of a situation model is an entity.

An entity is a correlated set of observed properties. Entities are detected and tracked by perceptual components as specified by the situation model. Entities may be actuators or props. Actuators are entities that are capable of spontaneous action that is of spontaneous changes in state. In most cases, actuators will be people, but mechanical or electrical devices for which actions must be detected may also be considered as actuators. Props are entities that are static, i.e. do not act.

Entities have numerical attributes that describe as position, orientation, size, configuration or external appearance. These may be used to compute relations. A relation is a predicate (truth) function computed over the attributes of one or more entities. Relations may be represented by Boolean or probabilistic truth-values. A unary relation computes a truth function over an attribute of a single entity. N-Array relations compute truth functions over sets of N entities.

Each situation is defined in terms of a set of roles and relations. The concept of role is an important (but subtle) tool for simplifying the network of situations. It is common to discover a collection of situations for an output state that have the same configuration of relations, but where the identity of one or more entities is varied. A role serves as a "variable" for the entities to which the relations are applied, thus allowing an equivalent set of situations to have the same representation.

A role is played by an entity that can pass an acceptance test for the role. In that case, it is said that the entity can play or adopt the role for that situation. In our framework, the relations that define a situation are defined with respect to roles, and applied to entities that pass the test for the relevant roles. A change in the assignment of an entity to a role can be an important source of events for CHIL services.

A situation model describes activity using a network of situations. Such a model specifies the entities, properties and relations that must be observed to provide a CHIL service. Changes in individual or relative properties of specified entities correspond to events that signal a change in situation. Such events can be used to trigger actions by the system.

For example, in a group discussion, at any instant, one person plays the "role" of the speaker while the other persons play the role of "listeners". Dynamically assigning a person to the role of "speaker" makes it possible to select perceptual component to acquire images and sound of the current speaker. Detecting a change in roles allows the system to reconfigure the video and audio acquisition systems.

Entities and roles are not bijective sets. One or more entities may play a role. A role may be played by one or several entities. The assignment of entities to roles may (and often will) change dynamically. Such changes provide the basis for an important class of events: role-events. Role events signal a change in assignment of an entity to a role, rather than a change in situation.

Each situation is configuration of entities (actuators or props), relations (predicates computed over entities), and a list of adjacent situations that are directly accessible from the current situation. A situation model determines the configuration of processes necessary to detect and observe the entities that can play the roles and the relations between roles that must be observed.

Roles and relations allow us to specify a context model as a kind of "script" for activity in an environment. However, unlike theatre, the script for a context is not necessarily linear. Context scripts are networks of situations where a change in situations is determined based on relations between roles.

6.3.1 Overview of layer architecture

Situation models rely on a few infrastructure classes that manage the set of entities, roles and relations within the context model. These are shown in Figure 6-9.

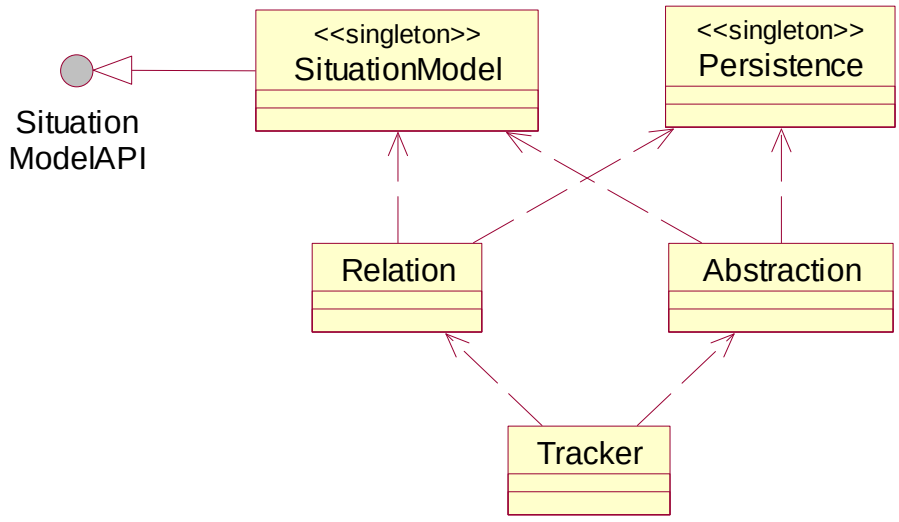

Figure 6-9 Situation Modelling architectural classes.

The roles of these classes are as follows:

- Abstraction models entities. Examples include *Person*, *Whiteboard*, *Table* and *Meeting*.
- Relation models a predicate (truth function) computed over one or more entity. Examples include At-Door(E1), In-Front-Of(E1, E2), Sitting(E1).
- Roles designate the specific entities within the scene and associate them to potential actions and events. Examples are *Lecturer*, *Audience*

History is maintained by recording the evolution of the situation within the network of situations so that chronological evolution can be queried.

The SituationModel is the main point of entry, enabling adding, querying and listing of objects in the model. It also enables searching for a particular type of objects, as for example looking for the current list of person objects in the situation model.

Both Abstractions and Relations contain a set of attributes. These attributes may have an associated probability, and will vary in time.

All the objects that will be modelled will have to satisfy the API of their base class. With the current set of use cases, we envision the following hierarchy:

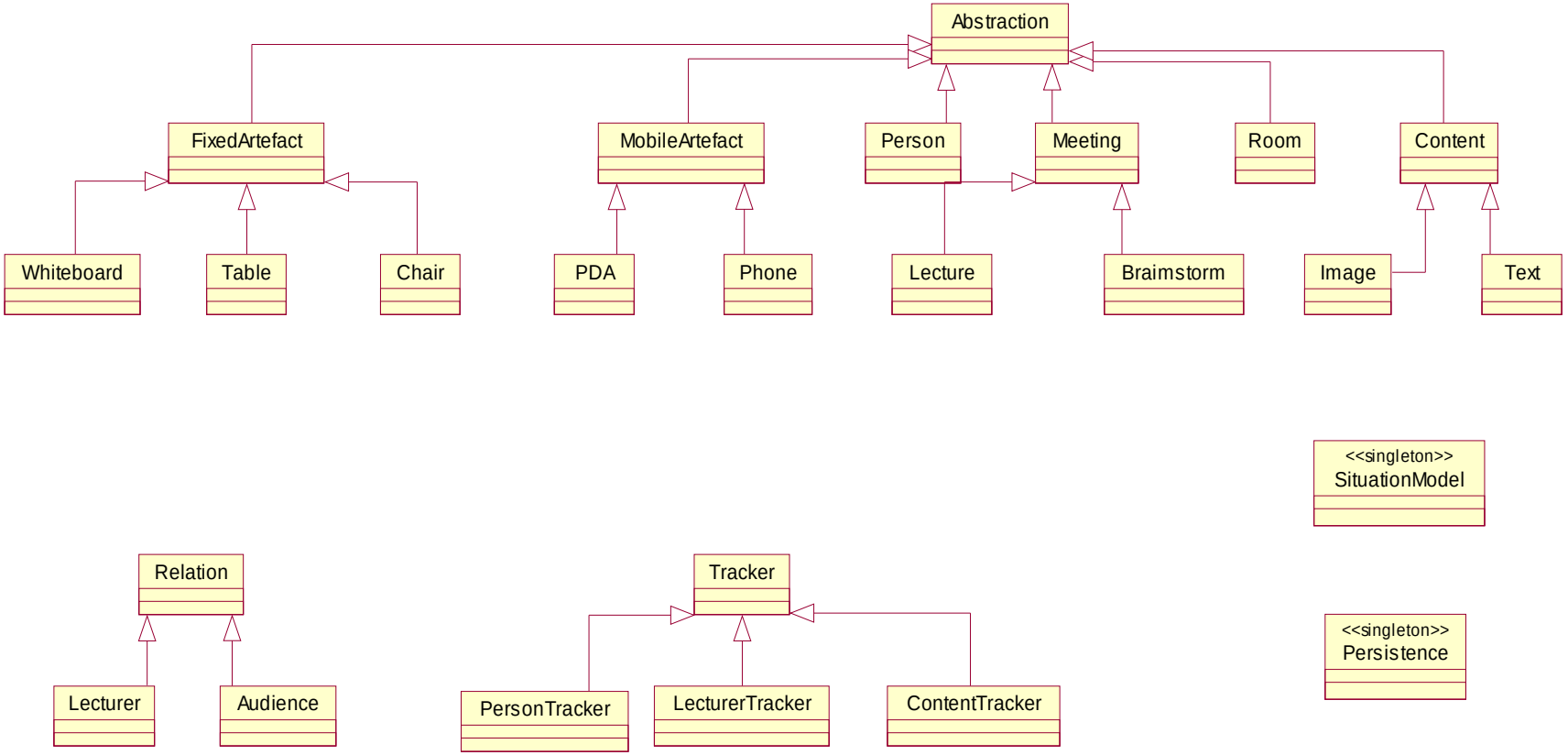

Figure 6-10: Modelled hierarchy of objects.

Here is a rough explanation of how the Person package would be handled in this framework. A CHIL service would select a situation model required for observing the CHIL environment. The situation model would launch and configure a perceptual component for detecting persons. When a person is detected, the situation model would launch a person tracker, and signal the event to the CHIL service.

The PersonTracker registers itself with the BodyTracker perceptual component from the layer below. As soon as there is an event from the BodyTracker saying there is a new body entering the room, the PersonTracker adds, through the SituationModel's API, a new Person abstraction into the model. This Person abstraction will itself register with several perceptual components so as to enable it to resolve the identity of the person, the location, and/or other aspects, which are accessible through its interface.

If the CHIL service requires the identity of persons, then the situation model may launch one or more perceptual components for person recognition. The identity of a person can be associated with an observed (tracked) person by some form of correlation. This could, for example, be spatial co-occurrence of the detection zone for person tracking and the observed area for face recognition. It could, alternatively, depend on temporal correlation. For example a process that detects mouth movements may correlate with identification via voice recognition.

If the person does not speak and is not turned towards a face-recognition camera, that the identity detection might not reach a sufficient level of confidence. Whenever recognition is achieved, events are transmitted to the situation model so that person identity can be correlated with a tracked entity.

Other perceptual components may register to be notified when the identity of the person is known with a certain degree of confidence. One example would be a meeting abstraction, registering with the PersonTracker to know the number of attendants, and registering with all persons to be notified about their identity. In case of identity detection delay, the display of meeting information will at least be able to tell that there is a new person, but not yet who it is.

Each objects is updated based on the information from the underlying perceptual components layer ("user moved out of the room", "someone came in", "person XY moved to whiteboard and started presenting") or possibly from its upper layer, as the service may know that a user is interacting with the PDE through the web-server, but this fact is not established yet through other means de to occlusions, bad lighting conditions or bad location in room.

As a support for monitoring, we assume there will be some logic needed that functions as some kind of context consistency checker on the data models represented in this layer (pre-conditions, post-conditions) – e.g. "number of people in the room" versus "number of people in meeting". These consistency-checking logics will be spread in the different objects, so as to implement them where most appropriate.

6.3.2 Eventing and polling

Every object in the framework implements an API for registering to attribute change events, as well as methods for reading these attributes. The basic object will provide events for attribute changes, and object activity (active/inactive). This API is embodied in the Abstraction and Relation interfaces.

Output of information is not yet well defined, but there may be abstractions that present an API to actually change some property. This may be the illumination in a room, or sending a targeted message to a particular user.

6.3.3 Extensibility

It is seen as important that new types of objects may be added seamlessly into the situation model. A new package typically will depend on perceptual components for detection and tracking, which will register to existing sources of events, and a type of abstraction or relation.

The object types, which are added, are heavily service dependent, as some situation information is collected only if a service needs it (e.g. audience attention level), so that particular attention is given to having a proper extensibility API for the situation modelling engine.

6.3.4 History Tracking

An important part of the situation model is its ability of tracking changes in time. This part is vital to recover information about the sequence of lecturers in a meeting, the links to their contents, or tracking the list of goals or agenda items.

The persistence layer models that aspect of history tracking, and also provides the API for recovering the history of a particular object or attribute, or giving the situation of all objects at a given point in time. History tracks situations, and thus records the assignment of entities to roles, as well as the relations concerning entities.

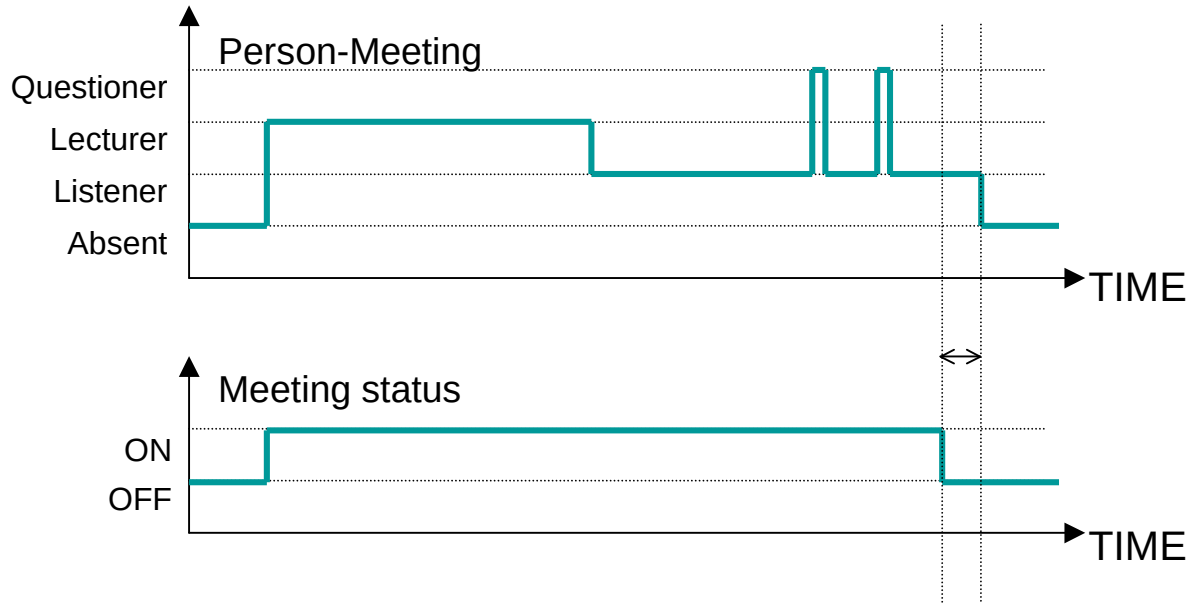

Figure 6-11: History tracking

The person was the first lecturer of the meeting, then was listening to another lecturer (not shown), asked two short questions, then spent another few minutes in the room after the meeting ended. The situation history process must be able to recover both the relation history and the status (attribute) history.

6.3.5 Start-up of system

Beat first glance it appears desirable to discover all relevant entities within the scene automatically in an unsupervised way. However, relevance depends on the task, and thus on the currently running CHIL service, and the situation model that this service requires. An important part of situation modelling process is determining the set of entities and relations that are required for each situation model. In early versions of the CHIL system, this choice will be made "off-line". In later versions, we will experiment with automatic acquisition processes for this information.

As a situation model is initialised, each of the relevant entities within the scene must be discovered. This will require development of appropriate detection procedures for initialisation.

6.4 Perceptual Components Layer

Perceptual components layer provides interpretation of data streams coming from various audio and video sources represented by the underlying layer of Logical Sensors to the upper layers of Situation Modelling and Services, and transfers requests for output generation from these upper layers to the layer of Logical Sensors.

6.4.1 Class Architecture

At the top of the object hierarchy of the Perceptual Components layer, there is the `PerceptualComponent` abstract object. This object includes attributes and operations that are common to all perceptual component objects and implements a set of interfaces described in chapter *Perceptual Component APIs*.

The next level in the hierarchy distinguish between a set of components which provide interpretation of data streams coming from various audio and video sources of the underlying layer of Logical Sensors (input components) and a set of components which generate output requested by the Situation Modelling layer and/or by the Services layer (output components). The objects are named `InputPComponent` and `OutputPComponent` for input and for output perceptual components respectively.

Input perceptual components are grouped according to their modality as descendants of VisualPComponent, AcousticPComponent, or AudioVisualPComponent. These components have a direct connection to the AudioSensor and/or VideoSensor objects from the Logical Sensor layer. Further extension to other types of input perceptual components (such as event-producing sensors like open-door detector) is represented by the object named OtherInputPComponent.

The hierarchy of perceptual component classes is shown in *Figure 6-12.*

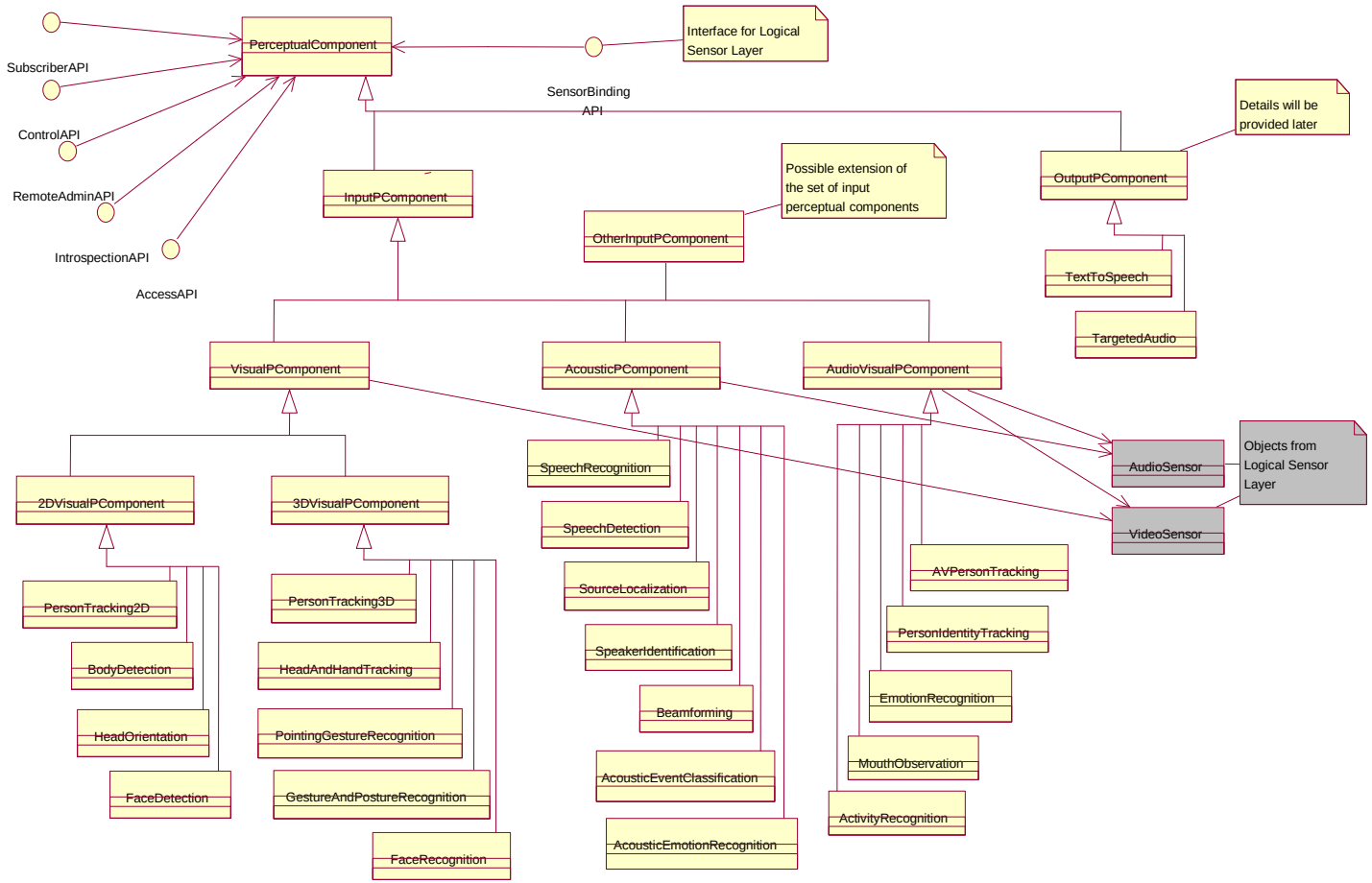

Figure 6-12: Hierarchy of Perceptual Component Classes

6.4.2 Proposed Perceptual Components

Perceptual components to be plugged into CHIL architecture are at the higher level divided into two categories: input and output components.

Proposed input perceptual components are the following:

- Visual perceptual components – 2D
  - Person localization and tracking (2D) [INRIA, ITC-irst]
  - Body detection [INRIA, ITC-irst]
  - Head orientation [INRIA, UniKarl/ISL]
  - Face detection and recognition [UniKarl/ISL, CMU, RESIT-AIT]
- Visual perceptual components – 3D
  - Person tracking (3D) [UniKarl/ISL]
  - Gesture/posture recognition [UniKarl/ISL, ITC-irst]
  - Head & hand tracking using stereo cameras [UniKarl/ISL]
  - Pointing gesture recognition using stereo cameras [UniKarl/ISL]
- Acoustic perceptual components
  - Speech recognition (including far-field) [UniKarl/ISL]
  - Source localization [INRIA, UniKarl/ISL]
  - Speech detection [INRIA, UniKarl/ISL]
  - Speaker identification [CMU]
  - Acoustic emotion recognition [CMU]
  - Acoustic event classification [CMU]
  - Beamforming [UniKarl/ISL]
- Audio-visual perceptual components
  - A/V person tracking [INRIA, UniKarl/ISL]
  - Person identity tracking [UniKarl/ISL]
  - Activity recognition [INRIA, UniKarl/ISL]
  - AVSR - mouth (lips) observation [INRIA]
  - Emotion recognition [KTH]

Output perceptual components are:

- Multimodal Speech Synthesis [KTH]
- Targeted Audio [INRIA, KTH, Daimler]

6.4.3 Perceptual Component APIs

Perceptual component interfaces provided to Situation Modelling layer and Services Layer are the following:

- **Subscriber API** – allows CHIL components to subscribe to the recognition events generated by a particular engine. It also lets the client components to subscribe for the stream of raw data (using the abstractions at the underlying Logical Sensor level).
- **Control API** – allows changing parameters or profiles for the recognition task (language, resolution, prototypes, etc.).
- **Remote Administration API** – allows component monitoring and remote management. This API will provide means for controlling the lifecycle of components, so that they can be invoked, suspended, updated, and reconfigured at runtime. Also, it will allow probing the status of components, as well as explicit support for debugging and tracing component.
- **Introspection API** – provides information on component versioning, its current interface version, mandatory and optional functionality, and offered semantics.
- **Access API** – provides functionality for allocation and release of a given resource.

An interface for the underlying layer of Logical Sensors is:

- **Sensor Binding API** – provides connectivity to the set of audio and video sensors represented by the Logical Sensors layer.

Generally, Perceptual Component interfaces can be either local APIs or remote (XML-based) APIs:

- Local API – directly accessing methods of particular object implemented in a standard language such as Java, C++, Python, Perl, etc. The first-cut will be Java.
- Remote API – XML-based, access to the standalone engines, some of them can be implemented as Web services.

6.5 Logical Sensors and Actuators Layer

6.5.1 Logical Sensors

Software Objects residing at the Logical Sensor Layer provide abstractions of the various sensors engaged in the CHIL sensing infrastructure. These abstractions will provide the means for controlling sensors, as well as for exploiting / consuming the data that they can provide.

At the top of the object hierarchy of the Logical Sensor Layer, there is the LogicalSensor abstract object (or interface) providing the most abstract representation of a sensor. This object includes attributes and operations that are common to all sensors regardless of the physical data, type and vendor. Such attributes and operations are those relating to the sensor description and its control functionality (e.g., type, description, “start()”, “stop()”, “kill()”).

The next level in the hierarchy defines object pertaining to particular types of sensors. Two major types of sensors are installed and user in the smart rooms of the CHIL partners [12]: audio and video sensors. As a result, two objects are specified corresponding to each of the types, namely `AudioSensor` and `VideoSensor`. Note that while these sensors derive

from the most general sensor object, they still constitute representation of Logical Sensors given that they remain independent of particular equipment vendors and types of Audio and Video sensors.

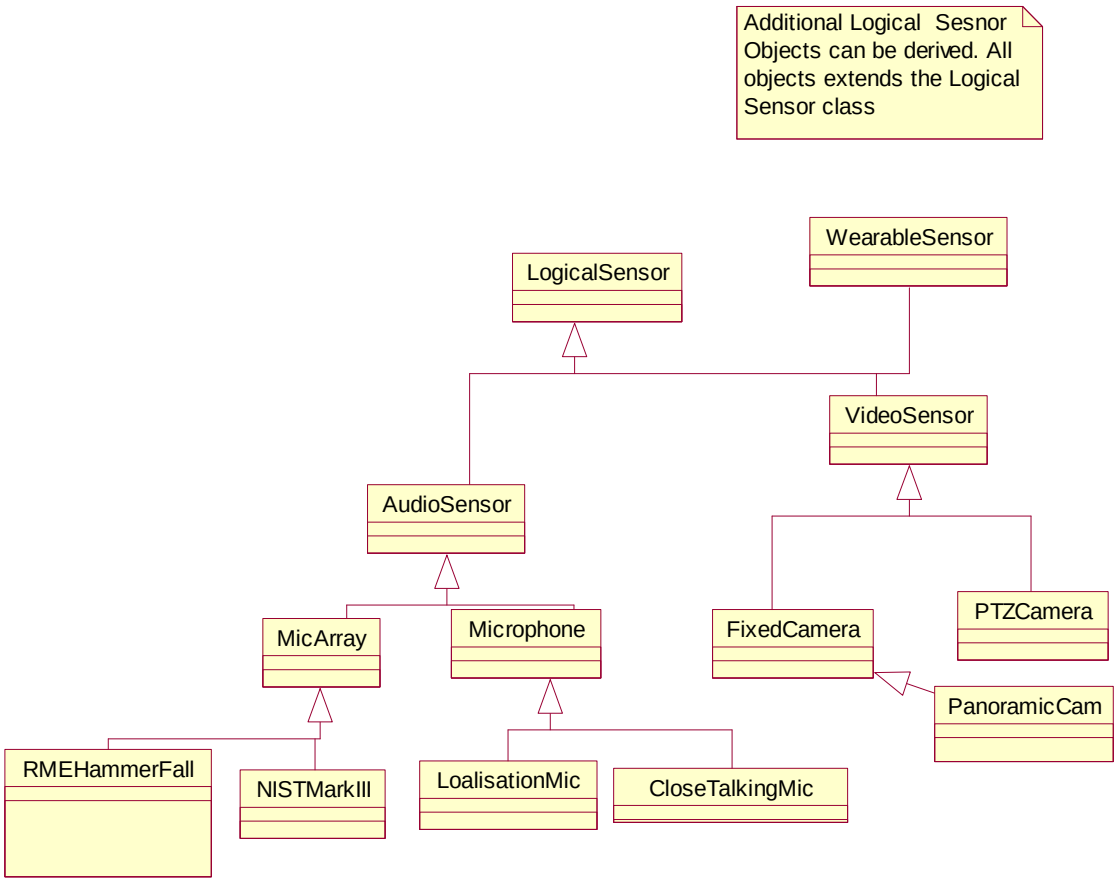

Figure 6-13: Basic Logical Sensor Layer Classes

Apart from the `AudioSensor` and `VideoSensor` objects, other objects can be devised at the same level to represent other types of sensors (e.g., `WearableSensor`, `TemparatureSensor`). These additional logical sensors do not seem expedient for implementing the early prototype CHIL services. It is no accident that the minimum sensor configuration for the CHIL intelligent spaces does not suggest sensors other than cameras and microphones. However, developers will be free to extend the `LogicalSensor` object towards extending the CHIL space with additional sensing functionality.

Generalizations of the `AudioSensor` and `VideoSensor` object provide more detailed representations of audio and video sensors. These generalizations are defined to capture sensor details pertaining to the various sensor types. `AudioSensor` can be extended to producing the `Microphone` and `MicArray` objects encapsulating behaviour pertaining to microphones and microphone arrays respectively. Accordingly, extensions of the `VideoSensor` object (e.g., FixedCamera, `PanoramicCamera`, `PTZCamera`) will result in more specific

representations of video sensors comprising functionality pertaining to the various camera types.

Following these definitions of `LogicalSensor` encapsulating behaviours tailored to specific sensor types, developers can further extend these objects to produce vendor specific classes that capture vendor specific behaviour and characteristics. These allow CHIL components to take advantage of vendor specific extensions.

Logical sensor layer objects will interface to perceptual, situation modelling and service components through allowing access to sensor streams. From an implementation perspective, logical sensor objects might communicate with NIST client software towards interfacing with sensor hardware (e.g., to control the sensor or capture streams). NIST Smart Flow clients can be wrapped within logical sensor classes through the `NSFSWrapper` objects, and accordingly used to implement the actual communication with low-level sensor control functionalities (e.g., capturing data, starting/stopping a sensor).

The hierarchy of logical sensor objects is depicted in Figure 6-13.

6.5.2 Logical Actuators

The description of these objects is an open issue yet.

6.6 Control/Metadata layer

In the CHIL architecture framework, the Control/Metadata layer (CML) mediates between the Low-Level Distributed Data Transfer layer (LDTL) and the Logical Sensors and Actuators layer (LSAL). The main purpose of this layer is to provide abstractions to the contents of the concrete data of the LDTL such that the LSAL can build its logical sensors on the abstract data.

6.6.1 Data and Control Abstraction: Interface to LDTL and LSAL

The NIST Smart Flow System is designed to handle continuous high-volume streams of data. Such streams origin on the low-level distributed data transfer architecture level of the CHIL architecture framework, mainly from hardware sensors such as microphone arrays or video cameras.

For the logical sensors level, the control/metadata level must provide a metadata view onto and stream control over the continuous high-volume streams. Metadata and control are realized through low-volume, typically non-continuous streams of data.

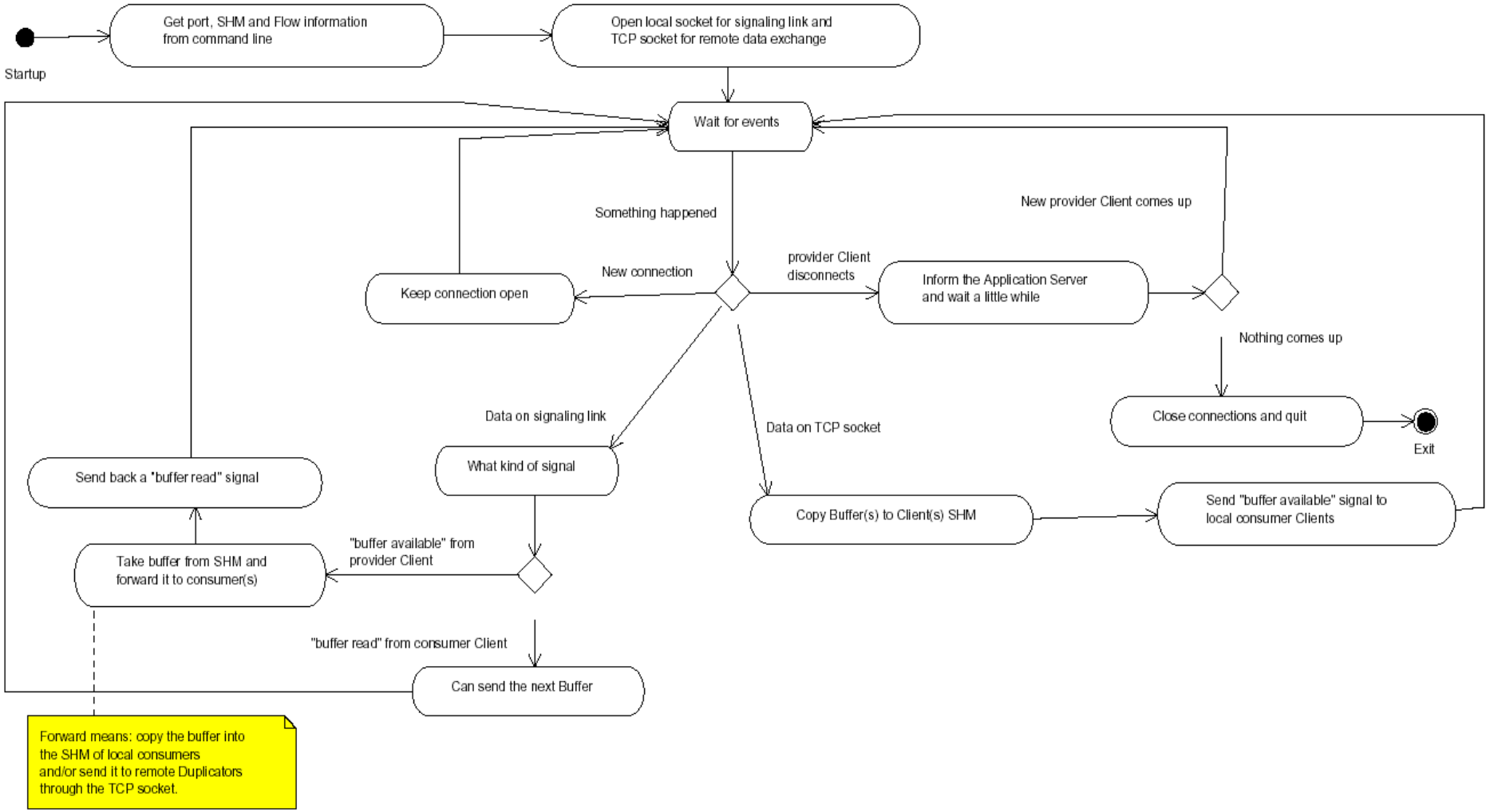

Figure 6-14: Duplicator Activity Diagram

The NIST Data Flow System Version 2 (NDFS-II) will be suitable to handle non-continuous streams (personal information from Cedrick Rochet on the ISL/IPD meeting on June 03, 2004). That is, NDFS-II can handle event-like data (such as control and metadata in the CHIL architecture). Still, a simple wrapper around NDFS-II on the control/metadata level is desirable in order to not directly expose NDFS-II to the logical sensors level. This wrapper will be essential for eventually replacing NDFS-II with a different low-level distributed data transfer system in the (far) future.

Some software components on the logical sensors level need direct access to the high-volume data streams. However, it is not desirable to build another distributed data transfer system for the upper levels that competes with NDFS-II. Rather, the wrapper around NDFS-II on the control/metadata architecture level should provide handles for providing direct access to NDFS-II streams without exposing internals of the NDFS-II system itself.

6.6.2 Resource Limitation Aspects

Care has to be taken when software components on the logical sensors level or above subscribe/unsubscribe direct access to high-volume streams of the NDFS-II via handles of the NDFS-II wrapper.

Due to limitations regarding network or system bus bandwidth, lavish dynamic subscription may endanger proper functionality of the components on the lower levels and thereby break the whole CHIL system. For example, requesting additional services, such as a user searching for a particular event in a video recording, may exhaust the total bandwidth and thereby endanger proper function of the core system functionality. Another example is a user copying a video recording to her/his notebook.

CHIL partners may want to define a static set of critical flows that minimally must be granted to work properly in order to guarantee core functionality of the smart room.

6.6.3 Exception Event Abstraction: Impact of Hardware Device Failures

In case a hardware device fails, exceptions from the hardware driver on the LDTL MUST be forwarded to the LSAL. This is necessary, because the upper levels in the CHIL architecture framework must be able to dynamically reconfigure the whole system in order to maximize availability of the smart room functionality even in the presence of hardware failures. Exceptions may be forwarded in an abstracted form, i.e. the detailedness of error messages may be reduced for the benefit of a better classification of errors that better fits the needs of the logical sensors level (e.g. classification into permanent or transient hardware failure rather than a concrete device's error message).

6.7 Low-Level Distributed Data Transfer layer

The primary functionality of low level distributed middleware components is to manage media flows originating from the various sensors and accordingly to make them available for processing by perceptual components. As flows are transmitted from sensors to perceptual components within a distributed system, these middleware components ensure that sensor input quality and throughput are the highest possible. To accomplish this, flow management components must implement high-throughput data transfer between end-points. Note that low level middleware components need to export appropriate interfaces to the sensors and perceptual components, in order to communicate with the sensors and perceptual interface. It is also

desirable that other context-aware components (e.g., residing at the situation modelling layer) are capable of accessing individual flows, to allow processing low-level sensor input at higher layers as well.

The CHIL project will employ the flow management capabilities of the NIST Smart Flow System (NSFS) [7] middleware, developed in the scope of the NIST Smart Space project. Reusing the NSFS middleware provides a solution that will accelerate production of early prototypes. As a consequence, middleware components defined in this paragraph take into account the design and flow management functionality of the NSFS. However, the CHIL low-level middleware will feature individuality, without being coupled to the NSFS. Practically, this implies that the design offers a possibility for replacing NSFS with pure CHIL low-level middleware components, without essentially affecting the functionality of the low-level middleware services. Adopting a major part of the NIST SmartFlow system design has the following advantages:

- Minimizes the design and implementation work required in the scope of a potential future replacement of the NSFS, towards migrating to a pure CHIL middleware solution.
- Minimizes the effort required to migrate technology components to a pure CHIL middleware solution.

Indeed, the following Low Level Distributed Middleware focuses mainly on the NIST Smart Flow System (NSFS) and on its successor, the NIST Data Flow System Version 2 (NDFS-II). Note that the design and development of the NDFS-II has already begun, but most likely it won't be finalized before the end of 2004 and will certainly need more time to fix bugs and bring it to a stable and usable state. Several members of the CHIL WP2 will follow closely (and some will participate in) the development of the NDFS-II, in order to design a system that fits our needs and to speed up its development. In the meantime, the NSFS will be deployed and extended with functionality needed to facilitate the work of developers (messaging, synchronized flow between clients, dynamic sized buffers).

6.7.1 NIST Smart Flow System

The current version of the NIST Smart Flow System (NSFS) is deployed by several CHIL participants. It is based on design approaches that include a very flexible flow history and a reference count garbage collection system for the flow history queue. These queue histories have multiple references from the various clients, and all clients have to free the history storage before it is actually freed for reuse. This behaviour reduces the robustness to faults, since the queue history can fill up and block further processing by all of the client nodes that use the flow when any single client using the flow fails, or cannot keep up with the input data rate. This reduces the scalability of the system. NSFS also imposes a very complex management problem for the shared memory subsystem, and a bottleneck through the server that handles the network I/O for the flow exchange between hosts.

6.7.1.1 NSFS Components

The NSFS has the following main components:

- *Client*: An executable program that uses the NSFS client library to consume and/or provide Flows, and process the Flows' data.
- *Flow*: A buffered data stream connecting two Clients and providing distributed data transport. These Flows have a type (e.g.: "Audio"), a name (e.g.: "Microphone Array") and a group.

- *Buffer*: The base unit of data exchanged among Clients via Flows when they emit data to, or consume data from a Flow. Each Flow has its own buffer characteristics, in particular the data type being transmitted and the maximum size that a buffer can have.
- *Host Server (sfd)*: An executable program that handles the distribution of the Flows between the Clients on a single host or on other hosts of the network, and conducts control functions using special control protocol.
- *Control Centre*: It carries out the connections among Flows and contains the list of the Flows, subscriptions, components, and all meta-information relative to Flows, starts Host Servers and Clients.
- *Host or System*: A computer participating in the NSFS by running a Host Server and some number of Clients.

There is one and only one Control Centre running in the system. Each host participating in the NSFS runs one Host Server, zero or one Flow provider Client per Flow and zero or more consumer Client per Flow. Host Servers are connected to each other and to the Control Centre by TCP sockets. Flow data and control information between hosts is sent through this socket. The Clients are connected only to the local Host Server through a local socket, but they are never connected directly to other Clients on the same host, or to Host Servers and Clients on other hosts or to the Control Centre. Control and meta-information is exchanged between a Client and the local Host Server through this socket. The Host Server maintains a single Flow History Queue for each Flow, i.e. the Clients do not have their own copies of the buffers: the consumer Clients of the same Flow on the same host get only read-only access to this common Flow History Queue.

6.7.1.2 C++ interface to the Smart Flow library

The functionality of the NSFS is available through a client library, which provides functions to connect to the system, create and subscribe to Flows, send and receive data from other Clients, etc. This library is written in C. Since some CHIL partners have requested to write their clients in the Perceptual Components layer using the NSFS (through other layers) in C++, we developed a C++ interface for the NSFS client library. This interface covers all functions and most of the data types and constants of the original C library. It is useful in particular for C++ developers as long as NDFS-II is not yet available. The class diagram of the interface is shown on Figure 6-15.

6.7.1.3 Messaging

Clients send and receive messages from the local Host Server, but these messages are only used to organize the cooperation between them (e.g.: to create or subscribe to a Flow, to indicate that a new buffer is available, or the Client exits, etc.). These messages are sent and received by the Smart Flow library internally, and they are not available from the Clients. This facility will be extended to provide an additional channel for communication between any Clients of the NSFS by additional library functions. Since we have Flows for high-bandwidth data exchange, and messages are small and rare, and these communication channels should be independent from Flows, they will not use Shared Memory but the socket connection between the Client and the local Host Server. When a Host Server gets such a message from one of its Clients, it forwards the message to the recipient Client through the local socket connection, if the Client is local, or to the Host Server running on the recipient Client's host, using the same

TCP socket used for other types of communication or data exchange. The messages are buffered by the Host Servers until the recipient Client reads them.

Messages can be sent or received either synchronously or asynchronously. As a result, a Client may block until the next message arrives, or it may check to determine if any new message have arrived, or it may send a message and continue its work immediately, or block and wait for the response message.

These messages can be used by Client developers for remotely controlling the behaviour of Clients on the other end of a Flow as via a remote procedure call protocol, e.g.: one consumer Client may say to the provider Client “Hey, I’m ready, you can send your data”.

Note, that this is just a facility to send messages to other Clients; neither the message format, nor its contents or its meaning is defined by the NSFS.

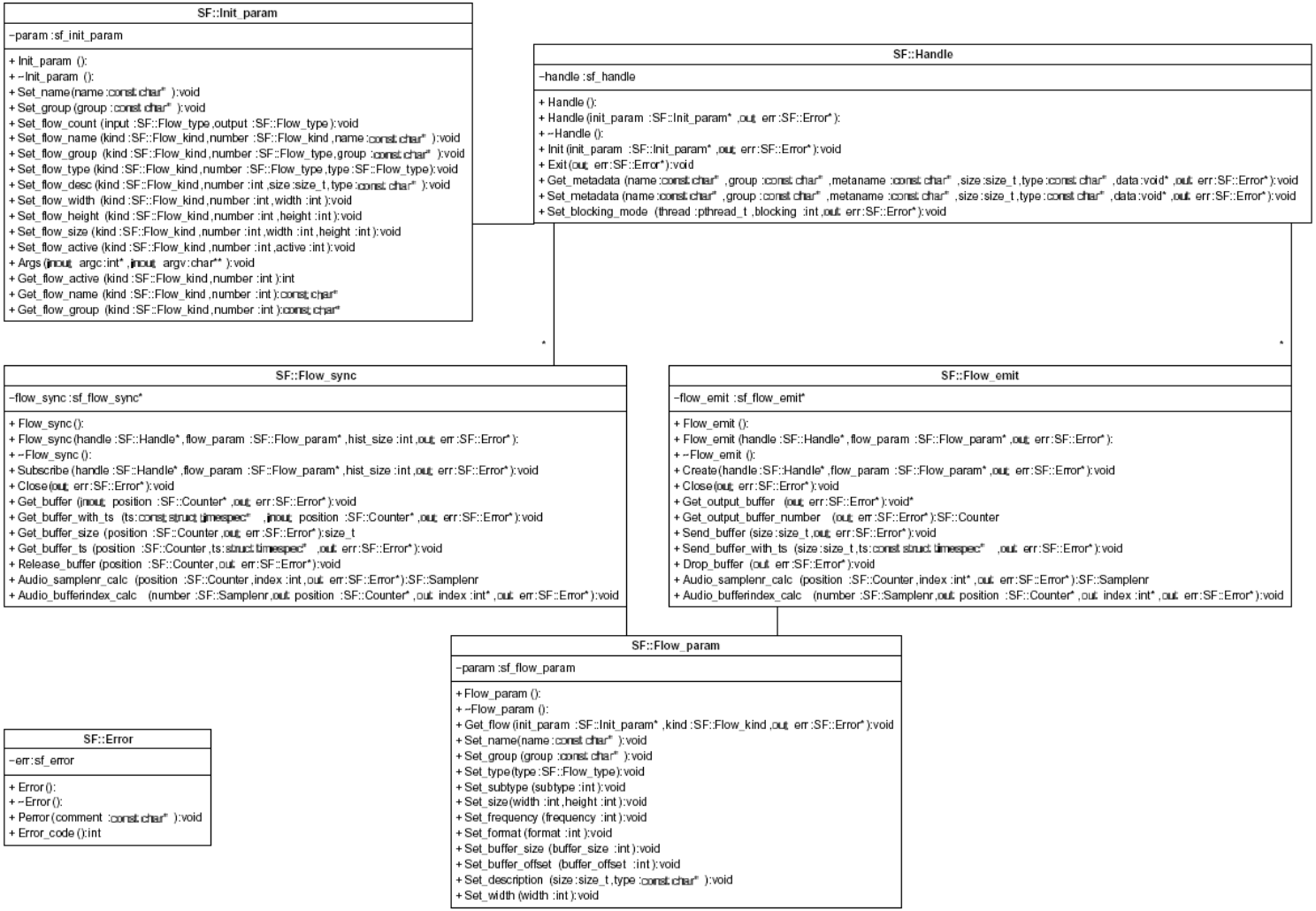

Figure 6-15: Class diagram of the C++ interface of the Smart Flow client library

6.7.1.4 Synchronized Flow between Clients

The provider and consumer Clients of a Flow are not synchronized: a provider Client sends Buffers to a consumer Client as soon as the data is available, and a consumer Client asks for a new Buffer as soon as it has processed the previous one. If the consumer Client is not fast enough, it can lose data. This behaviour is acceptable, if the system is on-line and data comes from sensors and real-time conditions must be met. However, this behaviour makes development of Clients more difficult, if data is read from hard-drives and if there are no real-time requirements. For the latter case, new types of flows will be developed whose Clients are synchronized with each other, i.e. the provider Client sends buffers only when all consumer Clients have free slots in their Buffer History Queue. The Host Server(s) performs all the work needed for synchronization when delivering Buffers; the synchronization will be fully transparent to the Clients.

6.7.2 NIST Data Flow System Version 2[5]

The NIST Data Flow System Version 2 (NDFS-II) addresses design flaws and weaknesses found in the NSFS. Its architecture is designed to replace the existing NSFS, but enhance its scalability, cross-platform compatibility, fault-tolerance and general robustness under load.

The NDFS-II will consist of a class library that client nodes can use to access and communicate with other (most probably remote) clients. The clients use this library to create and/or attach to data flows, and send messages to other clients. They must indicate which data flow(s) they need and which ones they provide (if any). The flows are specified with name and instance number, and are transparent with respect to network location (of course limited by bandwidth and network latency).

To achieve this, each computer participating in the system runs a host server daemon (sfd), an application server daemon, and one or more duplicators. Socket connections are opened between servers on different hosts to allow each server to maintain a full knowledge of the current state of the data flow system network of which the host is part.

6.7.2.1 NDFS-II components

The NDFS-II has the following main components:

- *Client*: An executable program that uses the NDFS-II client library to consume and/or provide Flows and process the Flows' data. There are also special clients that do not use flows, but provide control over the NDFS-II control network or display services such as the Control Centre.
- *Flow*: A buffered data stream connecting two Clients and providing distributed data transport. These flows have a type (e.g.: "Audio"), a name (e.g.: "Microphone Array") and an instance number. They can also carry additional information to Clients (e.g.: bits=24, rate=22050, channels=64).
- *Duplicator*: An executable program that handles the distribution of the flow between the Clients on a single host or on other hosts of the network.

[5] This section is mainly based on a comment draft about the functional architecture of the NDFS-II.

- *Buffer*: The base unit of data exchanged among Clients via Flows when they emit data to, or consume data from a Flow. Some kind of Flows can have their own Buffer characteristics, in particular the data type being transmitted and the maximum size that a buffer can have, but we plan to add Flows with more flexible characteristics, e.g.: dynamic sized buffers.
- *Client Message*: Clients can send messages to other Clients attached to the same Flow (or maybe to any other Clients as well). This facility can be used e.g. to remote control the behaviour of provider Clients by consumer Clients, but not instead of Flow data exchange. For more details about messaging see Section 6.7.1.3.
- *Application*: An Application is a Flow domain in which the Clients run. This allows the execution of several NDFS-II Flow graphs independently on the same physical computers without logical collisions.
- *Application Server*: The main server that centralizes all the information for a given Application. The information is in particular what and where are the Clients on the network and which Flows they provide or consume.
- *Host Server*: It is just a "gathering" server that lists the Application Servers on its host and conducts the NDFS-II control functions using the protocol transported via XML-RPC. (It acts as a "name server" for the Applications.)
- *Shared Memory (SHM)*: This facility is used by a given host to exchange data between Duplicators and Clients.
- *Host or System*: A computer participating in the NDFS-II graph by running a Host Server, an Application Server, and some number of Duplicators and Clients.

The operational relationship among these components is shown in Figure 6-16.

6.7.2.2 Host and Application Servers

The Host and Application Servers coordinate the data transport through the Flows to the Clients. There are two major communication protocols that make up the NDFS-II: the metadata or control interface among the Servers, and the data interfaces among Clients. The metadata or control interface is implemented by a set of RPC transactions (in particular using XML-RPC), and the data interface by buffered Flows.

One Host Server runs on each physical host. This server listens on a publicly known TCP port for both local and remote connections. It is created around a single select() statement and thus exchanges messages synchronously. It is the central point of contact for transmitting metadata or control across the NDFS-II Applications.

Zero or more Application Servers operate on each host. These servers maintain the state of their Application within each running host, including list of Flows, Clients, and other flow graph metadata.

The Host and Application Servers are based on a Remote Procedure Call mechanism. Each server listens on an open socket for incoming requests. This request has a name and various numbers of parameters. This request in turn invokes a method within the server. When the work is done (including other RPC to other servers as well, if needed), a response message is built and sent back to the caller. The RPC is transported via XML-RPC, but since the interface is decoupled from the implementation, on could replace this mechanism by others,

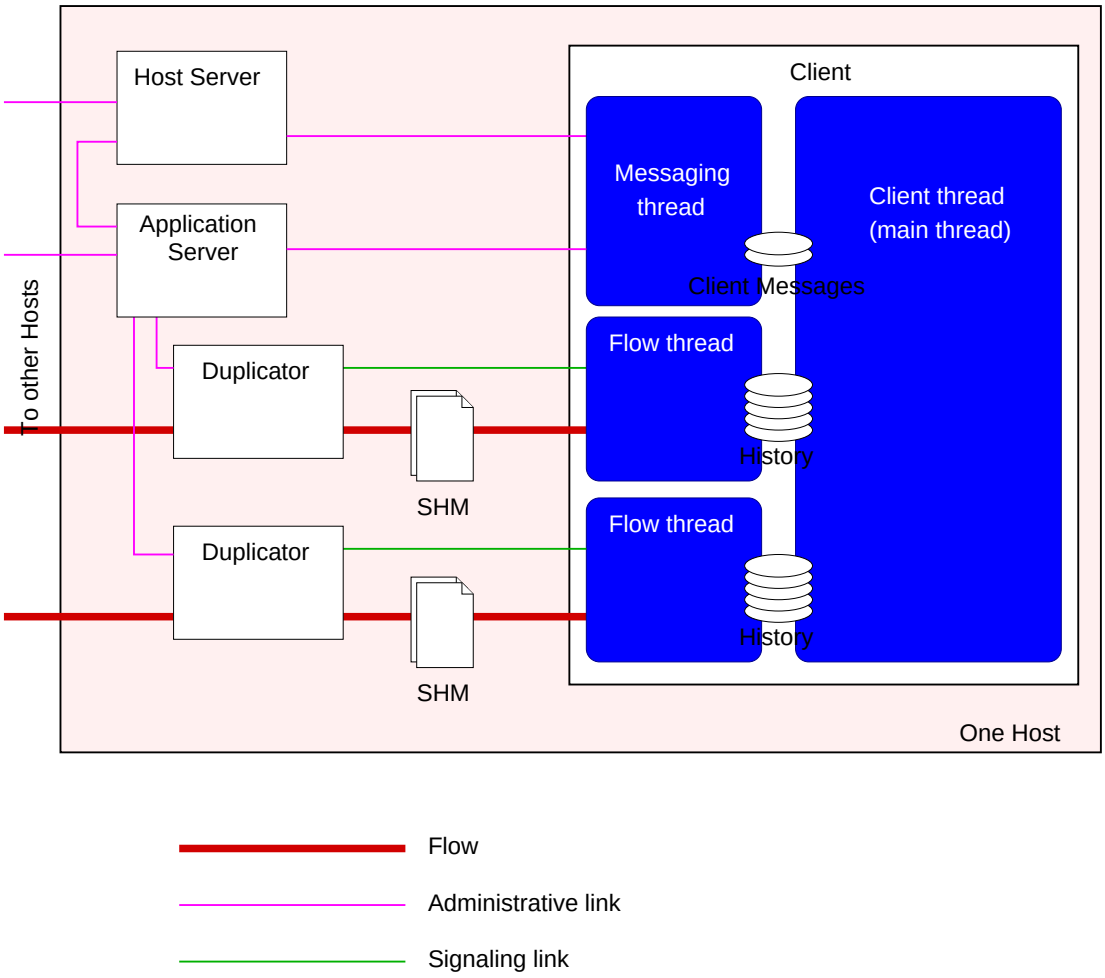

Figure 6-16. Operational relationship among the components of the NDFS-II.

including Unix RPC or Corba. The RPC calls handled by Host Servers, Application Servers or Clients are subject to further discussion.

When an Application Server starts, it does the following:

- It creates a list of Hosts, a list of Flows, and a list of Clients (all these lists are empty at startup).
- It sends an RPC message to the Host Server, advertising its application membership and port number.
- It contacts the other Application Servers by using a discovery mechanism. This mechanism is at first a simple file containing the hostname strings, but other methods and protocols will be developed in the future (e.g.: via network Broadcast, DNS TEXT records, UPNP).
- It opens a TCP socket and waits for connections.
- When a Client connects, the Application Server enters it in the peer list.
- When a Client asks for a Flow (either in provider or in consumer mode), the Application Server first looks in its Flow list to see if this Flow is already served either locally or remotely. If yes, it returns the information about the Flow and about the Duplicator serving the Flow (e.g.: local socket and SHM) to the Client. If not, it

spawns a new Duplicator, passing the information about the Flow to it. Then the Application Server informs all other connected Application Servers about the new Flow creation. Finally, the Duplicator contact information is returned to the Client.
- When another Application Server connects, a replication session is initiated so that all Application Servers know the information concerning all others. This is done to speed up the Application Server in deciding if a Flow exists or not.
- If a link is closed, the Application Server immediately removes all references to this peer from the lists, and forwards this information to the remote Application Servers.

6.7.2.3 Clients

Clients perform the actual work in the data Flow. They access the input Flows, execute algorithms, and transform the data and place it in the output Flow buffers. There are many control options that can be accessed through the API.

Each Client runs three types of threads: one messaging thread, one or more flow threads and one client (main) thread[6]. The first two types of threads are provided and managed by the client library; they are transparent for the Client developers. On Client startup, the main thread is started, which initialises the Client, creates the other threads (the messaging thread just after startup and the flow threads when the client creates or subscribes to Flow(s)), connects to Application and Host Servers, and performs the actual work. The messaging thread is responsible for exchanging control and Client Messages between the main thread and the Host Server through the administrative link using a remote procedure call protocol. Client Messages are buffered by the messaging thread until the main thread reads them. Each Flow has its own flow thread(s), each of them being responsible for communicating with the Duplicator(s) (i.e. handle the signalling link, and exchange Buffers through Shared Memory), and manage the Flow History Queue(s).

When a Client starts it does the following:
- It attempts to connect to the local Application Server using TCP socket connection. If it does not succeed, it tries again until timeout.
- Once connected, the Client uses RPC to identify itself to the Application Server. The Application Server registers it in its application table.
- Authentication procedure is conducted, if required.
- It transmits its input and output Flow requirements to the Application Server through RPC. The Application Server can disagree, for example if the same Flow is already provided elsewhere in the system. If it agrees, it returns information about the local socket to the Client, that will be used to communicate with the Duplicator for this Flow (signalling link), as well as the SHM information for the Flow.
- The Client connects to the Duplicator(s).

[6] Of course a Client can have multiple client threads, but the developer must take care of these threads.

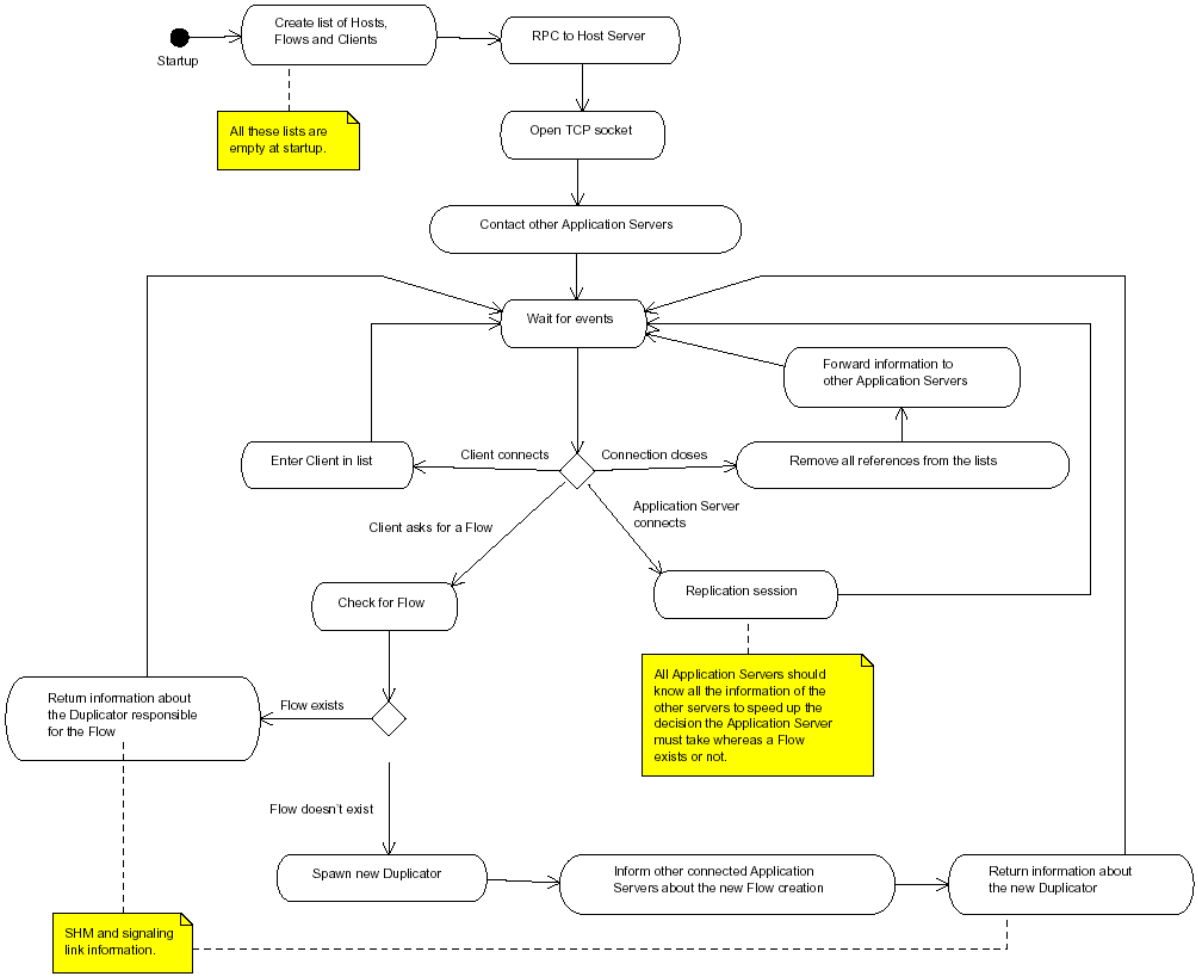

Figure 6-17. Application Server Activity Diagram

- The Client starts using the Shared Memory segment and the signalling link to exchange Buffers.
  - If the Client is a consumer Client, it waits on the signalling link. When a Buffer is available in the Shared Memory, the Duplicator sends a Buffer number byte to the Client. The Client (or more accurately: the flow thread of the Client) reads this Buffer, places it into its Flow History Queue, and returns a "buffer read" indicator to the Duplicator. If the Client is not consuming the data for whatever reason, it must return this "buffer read" indicator anyway.
  - If the Client is a provider client, it copies the Buffer to the Shared Memory and then sends a "buffer available" indicator. The Duplicator takes this Buffer, and returns a "buffer read" indicator. When time comes to copy another buffer, the Client can copy it into the next available Shared Memory buffer. If none is available, it waits on the signalling link for the next "buffer read" indicator, and it can then reuse the buffer. The buffers are always used in order, without skips, to simplify the design and to avoid buffer locking management.

- In any Client, if a "`read()`" returns -1 or 0, then something went wrong and the case must be dealt with. Most likely, however, the other end has just closed the connection and exited.
- When a Client wants to quit, it closes all connections and exits.

6.7.2.4 Data Flows

The data flows are the means of data transport and distributed processing. They are buffered to offer rudimentary Quality of Service (QoS) by allowing continuous smooth playback of audio and video, at the expense of some latency. In order to decouple the Clients from one another, the Flow History Queues are duplicated in each Client, and the data is copied to queues local to each Client. This means that each Client has its own writeable copy of the data contrary to the NSFS Clients, where thy Clients have only read only copies.

The provider and consumer Clients of the same Flow are currently not synchronized to each other, i.e. a provider Client sends its Buffers as fast, as it can, and the consumer Client can lose Buffers, if it's not fast enough. However, each Flow will have an attribute, which specifies, that the provider Client and the Consumer Client(s) will be synchronized, i.e. the provider Client sends its next Buffer only after received the acknowledgement from the consumer Client(s) about receiving the previous Buffer. This facility can be realized using internal messages between Clients or through special cooperation of Duplicators and Host Servers.

There is one Duplicator per Flow on each host. The Duplicator is responsible for flow distribution to other hosts as well as on the local host.

There is zero or one provider Client per Flow and zero or more consumer Client per Flow. This Client is connected to a Duplicator and to a Host Server on the same host, but never directly to other Clients or Servers on other hosts.

On a given host, flow data are exchanged by Shared Memory (SHM). Implementation is chosen as a "`mapped()`"-file without backing store, as this kind of shared memory is available on most platforms. Synchronization is made over local connections with the responsible Duplicator for this Flow (using semaphores should be avoided here, to simplify cross-platform compatibility and avoid dead-locking if a Client crashes unexpectedly).

When a Duplicator starts, it does the following:

- Using its command-line parameters, it knows the port it must open, the Shared Memory information, and the Flow it is supposed to serve.
- It opens a local socket in listening mode for the signalling link and a TCP socket in either listening or connecting mode, depending on the direction of the Flow.
- It waits for Client connections.
- If a consumer Client connects, it keeps the connection open and waits.
- If the provider Client connects with the local link, then it starts to forward Buffers to consumers, and collecting their acknowledgements.
- If a remote provider Client connects, the Buffers from the remote connection start to come in, the Duplicator is now provider of this Flow for this node and starts copying Buffers to the Shared Memory, sending signals to the consumer Clients via the opened connections.

- If the provider Client is local, and remote consumer Clients (actually not the Client itself but the Duplicators of remote Clients) have connected, then it starts to act as a client for this node, and sends Buffers to the remote peer(s) when they arrive from the Shared Memory and signalling link.
- When the provider Client is disconnected (a "`read()`" from the Client returns -1 or 0), the Duplicator informs the Application Server and waits for a little while. Another provider Client may start, and this way the current consumer Clients may starve, but not die. If nothing comes up, the Duplicator closes its connection(s) and quits.

6.7.2.5 Security

The Secure Socket Layer (SSL) can be used to create encrypted data flow application graphs that effectively operate within their own Virtual Private Network (VPN).

- Can be at the server port level for Host and Application Servers
- Flow level security is provided by the Duplicators using SSL
- All control channel can be encrypted
- Control Centre reads selected application map with application pass phrase
- Needs an Application specific pass phrase to access and control the Application Servers
- One key for each Application and one for the Host Server Network

6.8 CHIL Utilities

CHIL utilities provide general system internal information through the CHIL Utility object/interface, a wrapper that in turn provides the specific utility objects/interfaces. Initially, only GlobalTime is defined.

6.8.1 Global timing

GlobalTime is used for tracking system latency, quality of service, and to decide if an event is too delayed to warrant a response. In CHIL, an estimated global time is achieved by having all components retrieve the system's concept of *now* from the CHIL utilities at start-up. Any time-stamps sent by the process are relative to this global CHIL time. The GlobalTime object/interface, provide methods (e.g. now()) for any given component to access the best *now* available. GlobalTime is configurable for time resolution and format, and should be able to provide time in any time data type defined in the CHIL architecture framework.

Should a component for some reason fail to retrieve a global time at initialisation, it is permitted to continue. However, it is expected to:

1. Clearly mark all time-stamps as *unsynchronised* or *local*.
2. Repeat the attempt to retrieve global time at regular intervals.

Methods for comparing the local time of a component with the global time, comparing two global times, and formatting time stamps should follow a standard API, but should be implemented in each component, since it makes little sense to let the task of tracking latency rely heavily on network transactions. The methods may, however, be provided by CHIL utilities in the form of for example a library.

6.9 Ontology

CHIL Ontologies will be modelled using the Web Ontology Language (OWL) [17]. In order to get the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time) OWL DL will be used. OWL DL is so named due to its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL.

Ontological annotations will be used to define a manifest that elucidates a component's functionality based on ontological concepts and common classification schemes. Moreover, manifests give information about how to syntactically access a component. A component's grounding information may be used to automatically generate interfacing client code (e.g. wrapper code to make a perceptual component available to intelligent software agents). Contribution manifests are to be generated automatically from metadata contained in source code- and class files. Since ontological statements are to be machine interpretable it is necessary to agree on a common syntax to serialize ontological information. The normative exchange syntax for OWL is RDF/XML. The CHIL software environment does not impose any restrictions on the tools that may be used to work with CHIL Ontologies as long as these tools process OWL RDF/XML compliant source code. However, it may improve productivity if all CHIL partners involved in ontology engineering tasks agreed on one tool. The open-source development environment for Ontologies and knowledge-based systems Protégé [18] and its OWL plug-in from Stanford University is suggested as the reference tool for ontology engineering in course of the CHIL project. The description logic reasoning system RACER [19] is suggested as a reference inference engine for query answering over CHIL Ontologies. Figure 6-18 depicts how ontology- and software engineering tools interact in the CHIL software environment.

CHIL Ontologies will constitute the knowledge base used by a reasoner (e.g. *Racer*) for inferring and query answering. A knowledge acquisition system (e.g. *Protégé*) will be used for ontology editing. Annotated source files present one-to-one functional implementations of ontological concepts. Typed relations among ontological concepts will be preserved in those source files. *Eclipse* [20] has an extensible software development workbench will be augmented to process annotated source files and to make use of additional reasoner support.

At runtime, ontological manifests of annotated executables will be used for dynamic discovery of components. Learning algorithms will make use of additional ontological information in order to constitute a new working set of annotated executables that may perform better. Since contributions to the CHIL software environment may be written in different languages interfacing client code is to be generated automatically from either components' metadata or from manifest files.

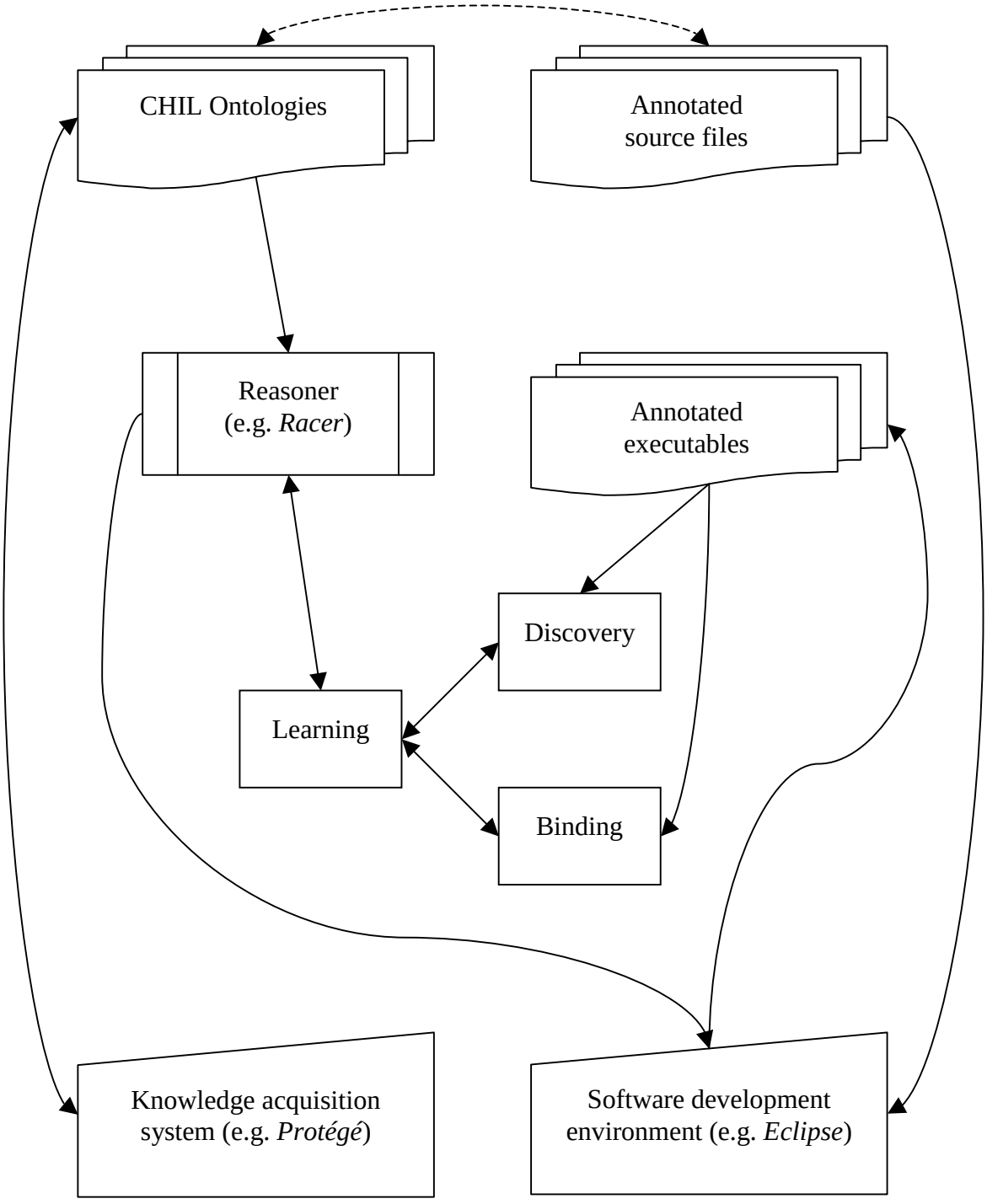

Figure 6-18: CHIL Semantic Software System

7 Open Issues

7.1 General Issues

User/Device-Mapping

Fraunhofer IITB question(s):

1. How is the mapping between a user and a personal device solved? Example: A user boots his notebook. The notebook connects to the CHIL system (via WLAN). How can the CHIL system identify the user of the notebook? Does the notebook send personal information to the CHIL system on login?
2. Is a personal device mapped to exactly one user, can a device be shared between users or can a user have more than one device?
3. Are users and devices tracked, i.e. can devices be swapped between users?

IBM answer(s):

The user will have to login to CHIL, either under an enrolled name (the CHIL would then activate his/her profile) or as anonymous (stateless connection, default profile). That should suffice at least as the first cut, because it also allows to users to move across different client machines.

User Identification

Fraunhofer IITB question(s):

1. Where does the actual user identification take place? Is it done by a service (User Identification Service) or is it part of the Perceptual Components?
2. Where is the reference data for the user identification (e.g. voice pattern, user characteristics, etc.) stored? Does every perceptual component store this information locally or is there a common database, for example in the Situation Modelling layer?

KTH answer(s):

It seems that any component needing to access a user model (be it a voice pattern or user preferences) should do so through the same interface. This interface could be connected to an array of local databases provided by other components, or centralised (and allow components to store data), or a mix of both, with an option to get external data thrown in. Using a common interface provides a way of postponing that decision, however. It would also allow partners to code local test databases with the "right" interface, so that they can be re-used.

INRIA answer(s):

The primary tasks of the situation-modelling layer are

- to inform the service about the current state of the environment, in particular, generating events when the situation changes - only events relevant to the service should be generated -,
- to specify the minimal set of perceptual components for the service.

Static world knowledge may be needed in the situation recognition process. The addition of a history module to the situation-modelling layer assures that it is possible to obtain a trace of past situations, and of past roles and relations of entities (particularly actuators).

User Interfaces

RESIT/AIT proposal(s):

User interfaces should define a scheme for interfacing to services, rather than a set of specific ("hard-coded") UIs. As far as UIs are concerned, a look at the iCrafter framework developed in the scope of Stanford iRoom project (http://iwork.stanford.edu/), enabling automatic service specific UI generation, is useful.

Perceptual Component Interfaces

IBM question(s):

1. Is it "OK" to rigorously divide visual components into 2D and 3D?
2. Are the acoustic, visual, etc. models viewed as being inside of the perceptual components wrapper – or should they be stored elsewhere and accessible through a dedicated API?
3. Should output component belong to this layer?

User Front-end

KTH proposal(s):

We believe that in order to decide in which manner to notify or communicate with users, the user front end should not only be allowed to access the user preference, but also the context of the situation. Fort example, an audio notification may be inappropriate even if the user prefers it in general, and some notifications may be relevant to all participants in a meeting, in which case an audio message to all may be more efficient than sending text messages to those who prefer that and voice to others. Furthermore, one of the efforts at KTH involves finding appropriate moments for the system to interrupt the meeting/lecture - this would be a service that delivers OkToInterrupt events or responds to OkToInterrupt queries. Users preferring spoken notifications may want the user interface to utilise such functionality as well, so that they aren't interrupted in their speech, for example.

Ontology

RESIT/AIT question(s):

1. How far Ontologies are relevant for all layers of the framework?
2. How far dynamic relationships may be represented with Ontologies as well?

Miscellaneous

IBM question(s):

1. Is a mechanism needed to prevent cycles in event chains?
2. At which layer resides the interface to personal calendars, personal info, room occupation calendars, etc?
3. Are User Profiles assumed to be provided at upper layers? Look at use cases involving notifications.
4. Scheduled meeting and detected meeting: How do we match them?
5. At what levels of CHIL hierarchy happens the fusion of speaker location and speaker id?

7.2 Special Issues

Chapter 2.11, 2.12

RESIT/AIT proposal(s):

We should pledge documentation (and external documentation in particular) just for the final prototype system to be delivered in CHIL.

UKA/IPD proposal(s):

Define data formats. We may want to choose XML. An XSD schema may be defined for each documentation category.
Define presentation formats. We may want to support HTML and PDF as presentation formats. XSLT programs could transform XML based documentation material into arbitrary presentation formats. Textinfo may be an option.

KTH proposal(s):

If we choose XML, we may want to have a look at DocBook (http://www.docbook.org/) Benefits are that it is an open standard, it's extensible, there are quite a number of programming related elements predefined (it is originally intended for man page style documentation), etc.), and Norman Walsh has coded a set of XSLT documents transforming DocBook to HTML as well as PDF and PS (the latter two are not such an easy task in reality - XSL-FO isn't all that user friendly).

Chapter 2.13

UKA/IPD proposal(s):

Consensus on the WP2 meeting in Karlsruhe was to defer the decision of whether to use a bug tracking system to the Paris meeting. This includes defining requirements to a prospective CHIL bug-tracking tool (e.g. Web based, automatic reporting of user computer environment). Are we going to use a COTS solution?

KTH proposal(s):

One way of building up a set of test cases gradually is to code a test case every time one fixes a bug. The test case should test for the fixed bug.

Chapter 6.1

How will agents be created and supervised during runtime?

Chapter 6.2

How will services be created and supervised during runtime?

Chapter 6.3

IBM proposal(s):

The Section needs to be synchronized with Section 3.4.2.4.
The discovery of entities (supervised/unsupervised) is an open question and needs to be discussed.
A Taxonomy for describing entities, roles, relationships, and situations is needed to improve the clarity of text.

Chapter 6.5.2

The description of Logical Actuators is missing.

8 Annex

8.1 References

[1] Bürkle, Axel et al.:
Functional Requirements.
CHIL-WP2-FunctionalRequirements-V1.0-2004-07-14-PU, 14-July-2004

[2] Phillips, Brenton A.:
Metaglue: A programming language for multi-agent systems.
Meng, dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Cambridge, MA. 1999

[3] Johanson, Brad and Fox, Armando:
The Event Heap:
A Coordination Infrastructure for Interactive Workspaces.
Proc. of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA), 2002

[4] Gray, Matthew et al.:
Hive: Distributed agents for networking things.
IEEE Concurrency 8(2): pp. 24-33, April-June 2000

[5] ***Microsoft Easy Living Project.***
http://research.microsoft.com/easyliving/

[6] Abowd, Gregory D. et al.:
A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications.
Human-Computer Interaction 16, 2001

[7] ***The NIST Smart Flow System***.
http://www.nist.gov/smartspace/toolChest/nsfs/

[8] Baeg, Soon C. et al.:
An Open Agent Architecture.
Proceedings of the AAAI Spring Symposium Series on Software Agents (AAAI Technical Report SS-94-03), pp. 1-8, Palo Alto, CA, AAAI. March 21-23, 1994

[9] Garlan, David et al.:
Project Aura: Towards distraction-free pervasive computing.
IEEE Pervasive Computing, pp. 22–31, 2002

[10] ***Facilitating Agent for Multi-cultural Communication.***
IST FAME Project (IST-2000-28323)

[11] ***The NIST Smart Spaces Project.***
http://www.nist.gov/smartspace/smartSpaces/

[12] Stiefelhagen, Rainer et al.:
Initial Specification of the Sensor Setup.
CHIL-ProposedSensorSetup-V1.3-2004-03-02(-CC), 02-March-2004

[13] ***Computers in the Human Interaction Loop.***
Annex I – "Description of Work", 20-October-2003

[14] Kleindienst, Honza:
Thoughts on CHIL-Architecture.
Grenoble, March 23rd 2004, 1st architecture meeting

[15] Forgy, C.L.:
RETE: A fast algorithm for the many pattern/many object pattern match problem.
Artificial Intelligence, Volume 19, Number 1, 1982

[16] ***Java Rule Engine API™ JSR-94***.
Java Community Process Specification, http://java.sun.com/jcp/

[17] ***W3C Web-Ontology (WebOnt) Working Group***
http://www.w3.org/2001/sw/WebOnt/

[18] **Protégé knowledge acquisition system**
http://protege.stanford.edu/

[19] **RACER: Renamed ABox and Concept Expression Reasoner**
http://www.sts.tu-harburg.de/~r.f.moeller/racer/

[20] **Eclipse Foundation**
http://www.eclipse.org/

8.2 Table list

8.3 Figure list